



ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

DIPARTIMENTO DI INFORMATICA - SCIENZA E INGEGNERIA - DISI

CORSO DI LAUREA IN
INGEGNERIA INFORMATICA

**EdgeCV4Safety:
SOLUZIONE DI EDGE VISION PER
SAFETY IN INDUSTRIA 5.0**

Relatore

Chiar.mo Prof. Ing. Paolo Bellavista

Tesi di laurea di

Correlatore

Prof. Ing. Riccardo Venanzi

Matteo Fontolan

Sessione Dicembre 2025

Anno Accademico 2024/2025

Per Aspera Ad Astra

A te papà, esempio, che nonostante tutte le avversità sei arrivato qui, per me

Abstract

Nell'ambiente industriale moderno si fa largo uso di dispositivi robotici e macchinari. Sebbene questi siano fondamentali per la produttività e l'efficienza, possono essere potenzialmente pericolosi per gli operatori. La possibilità di controllare o pilotare tali dispositivi rende concepibile la realizzazione di automazioni utili non solo agli scopi lavorativi per cui sono fondamentali, ma anche mirate alla sicurezza.

Attualmente gli apparecchi industriali sono spesso circondati da strutture fisiche o barriere dedicate a confinarne i movimenti, prevenendo così avvicinamenti incauti e pericolosi, persino letali, da parte del personale. Queste strutture occupano uno spazio operativo considerevole, limitando l'utilizzo proficuo di aree altrimenti sfruttabili. Inoltre, spesso si rivelano scomode, d'intralcio e costose, venendo solitamente realizzate su misura per ogni singolo macchinario.

Questo lavoro di tesi si propone di creare un sistema autonomo per il monitoraggio video di aree di lavoro con macchinari e robot, sfruttando telecamere. Tale sistema deve essere in grado di rilevare la presenza e la posizione dell'uomo all'interno della zona controllata ed alterare conseguentemente il comportamento dei dispositivi di interesse sulla base della vicinanza alle aree di rischio.

I vantaggi principali di questo approccio impattano positivamente sui costi e sugli ingombri delle strutture di sicurezza e sull'indipendenza dalla specifica tipologia di macchinario, conferendo la possibilità di gestire in modo centralizzato più dispositivi contemporaneamente.

Indice delle pagine

1. Introduzione.....	1
1.1 Contesto generale	2
1.2 Obiettivi.....	3
2. Related works e tecnologie	5
2.1 Related Works	7
2.2 Tecnologie	9
2.2.1 Cloud ed Edge Continuum.....	10
2.2.2 TSN	12
2.2.3 Docker.....	16
2.2.4 Object Detection.....	18
2.2.5 Depth Estimation	21
3. Il progetto EdgeCV4Safety	25
3.1 Architettura e scelte tecnologiche	25
3.1.1 Aravis	27
3.1.2 YOLO	28
3.1.3 YOLO-3D - Depth Anything v2	31
3.1.4 UniDepth.....	33
3.1.5 RTDE.....	35
3.1.7 PyTorch	37
3.1.8 Open Neural Network Exchange (ONNX).....	37
3.2 Implementazione	39
3.2.1 Implementazione Vision Submodule	41
3.2.2 Implementazione Controller Submodule	50
4. Deployment e test.....	59
4.1 Telecamera Basler GigE.....	60
4.2 Nvidia Jetson AGX Orin	61
4.3 Advantech UNO	63
4.4 Braccio Universal Robots	64

4.5 Switch TSN Cisco IE 4000.....	64
4.6 Deployment Jetson - Immagine e Container Docker.....	65
4.7 CUDA.....	69
4.8 TensorRT	70
4.9 Deployment Advantech	72
4.10 Test	73
4.10.1 Test Vision Submodule	74
4.10.2 Test Controller Submodule	88
5. Conclusioni e sviluppi futuri.....	93
Bibliografia	97

1. Introduzione

In ambito industriale con ogni nuova rivoluzione si è da sempre assistito ad un cambio di paradigma e alle conseguenze che questo comportava. Anche in tempi più recenti si sono susseguite due innovazioni industriali: la quarta e la quinta rivoluzione. Entrambe hanno portato enormi cambiamenti, non solo per quanto riguarda la tecnica, ma anche per il ruolo che rivestono tecnologia, uomo e risorse naturali nel mondo moderno[1].

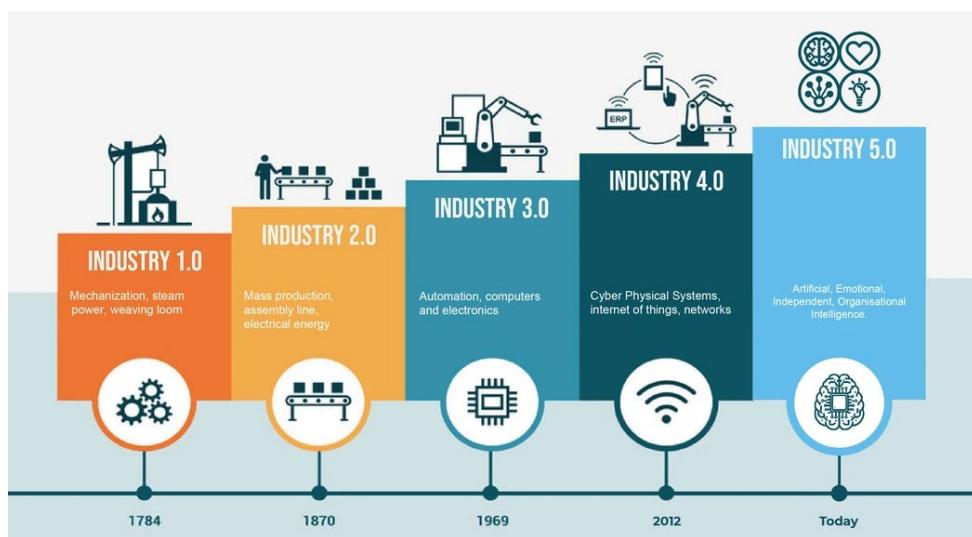


Figura 1 - Rivoluzioni industriali[2]

La prima è l'industria 4.0, nata in Germania nel 2011, da subito caratterizzata dalla presenza massiccia e trainante della tecnologia. Lo scopo principale di questa rivoluzione risiede nella creazione di sistemi di produzione *cyber-fisici* (*Cyber Physical Production Systems*) per la creazione di fabbriche intelligenti (*smart factories*). In questi ambienti la connessione in rete intelligente di macchine, processi e sistemi ha l'obiettivo di rendere la produzione flessibile, efficiente e personalizzata su larga scala. Tale connessione è standardizzata tramite il *Reference Architecture Model Industrie 4.0* (RAMI 4.0).

Le nuove tecnologie che si presentano come pilastri dell'Industria 4.0 sono: *Big Data*, robot autonomi, *Industrial Internet of Things*, *Cybersecurity*, *Edge-Cloud Continuum*, produzione additiva, realtà aumentata e simulazione.

Nel 2021, formalmente, la Commissione Europea ha introdotto l'industria 5.0, come iniziativa con lo scopo di riconoscere il potere dell'industria nel raggiungere obiettivi sociali più ampi, come sostenibilità e benessere dei lavoratori, ponendo la produzione al servizio dell'umanità nel rispetto dei limiti del pianeta. I valori fondamentali su cui questa nuovo paradigma si fonda sono centralità umana, che pone enfasi su salute fisica e mentale dei lavoratori, su benessere e tutela dei diritti fondamentali. Altri aspetti fondamentali sono la sostenibilità, che mira a favorire l'economia circolare, e la resilienza, per ottenere robustezza nella produzione industriale, permettendo di affrontare interruzioni e crisi importanti.

Fondamentalmente le tecnologie impiegate dall'industria 5.0 sono le medesime della 4.0, correggendone l'adozione in favore dei nuovi obiettivi e integrandone di nuove, come tecnologie bio-ispirate, materiali intelligenti ed energie rinnovabili. Centrali diventano però l'efficientamento energetico e la personalizzazione delle interazioni uomo-macchina. La vera differenza tra le due rivoluzioni, che sono una il completamento e l'estensione dell'altra, risiede principalmente nell'importanza che viene data alla centralità umana e alla sostenibilità. Nell'industria 4.0 queste sono di secondaria rilevanza, mentre nella 5.0 ne sono il fulcro[3-5].

1.1 Contesto generale

La progressione dei nuovi paradigmi industriali e delle moderne tecnologie ha contribuito alla diffusione di macchinari e dispositivi sempre più avanzati, sia dal punto di vista delle funzionalità e delle *performance*, che dei requisiti di sicurezza che questi devono rispettare, soprattutto nel rispetto dei valori dell'industria 5.0.

L'avanzamento apportato dalla quarta e dalla quinta rivoluzione industriale è notevole anche nel contesto delle condizioni lavorative. Queste tecnologie, in costante evoluzione, mirano a raggiungere nuovi traguardi di velocità ed efficienza, ma tale progresso può esporre i lavoratori a maggiori rischi.

Parallelamente all'automazione di azioni e operazioni produttive, si aprono opportunità significative per migliorare o innovare la sicurezza dei lavoratori. Tutto ciò è supportato dalla presa di piede di dispositivi *IoT* (*Internet of Things*) e degli approcci di *Edge-Cloud Continuum*.

Attualmente spesso i macchinari sono circondati da strutture fisiche e barriere protettive. Questi elementi implicano costi notevoli poiché sono specifici per ogni tipologia di macchinario e risultano poco flessibili. Anche l'ingombro che comportano è rilevante e questo riduce ulteriormente lo spazio operativo disponibile, creando dei disagi o rendendo l'ottimizzazione degli spazi circoscritta. In aggiunta, la presenza di questi elementi protettivi potrebbe limitare o addirittura non consentire l'interazione e la collaborazione con i macchinari (*cobot*).

1.2 Obiettivi

L'obiettivo principale di questo progetto di tesi è quello di realizzare un'architettura per un sistema capace di gestire il comportamento di macchinari industriali moderni a fini di sicurezza, impiegando un *pool* di risorse definito, adatto anche ad altri scopi e potenzialmente scalabile in maniera agevole. Uno dei fini è che il costo (eventuale) di tali risorse possa essere contenuto e ammortizzato facilmente, in particolare rispetto all'acquisto di strutture fisiche o altri sistemi più onerosi che sono oggigiorno preponderanti. Tale sistema, inoltre, deve essere fortemente flessibile e facilmente adattabile a funzionare e gestire dispositivi e macchinari differenti da quelli usati in questa dissertazione.

2. Related works e tecnologie

Nel panorama industriale, gli approcci e la ricerca atti a migliorare la sicurezza nell'ambiente di lavoro o comunque a prevenire incidenti ed errori sono diversi e, sempre più spesso, integrano sistemi di intelligenza artificiale e *machine learning*. Le scelte possibili sono molteplici e ciascuna presenta vantaggi e svantaggi. A seconda del caso d'uso specifico e delle esigenze, determinate soluzioni si rivelano più mirate ed appropriate di altre.

I pericoli per la sicurezza in ambito industriale sono diversi. Questi spaziano dal carico cognitivo allo stress mentale, dall'ergonomia ai cyberattacchi, ma il rischio più concreto e di interesse allo scopo della dissertazione riguarda le collisioni[6].

Le misure da adottare per la sicurezza del lavoratore nell'ambiente industriale non vengono prese unicamente a posteriori dall'azienda che li impiega, ma già durante la progettazione di macchinari e sistemi robotici cercando di mantenere il focus sull'uomo. Infatti, i produttori si pongono di realizzare questi dispositivi con un design mirato a limitare possibili rischi e pericoli, nei limiti delle funzioni operative che questi apparecchi devono avere.

In alcuni casi, nella fattispecie dei robot, in ambito di cooperazione uomo-macchina spesso i produttori inseriscono specifiche modalità di collaborazione[7].

1. La *Safety-related Monitored Stop* (da specifiche ISO10218-2 e ISO/TS 15066) permette l'accesso alla stessa area di lavoro anche per l'uomo, ma in momenti diversi. In particolare, il robot si ferma forzatamente prima che il collaboratore entri nell'area condivisa e rimane fermo fino al suo allontanamento.
2. L'*Hand Guiding* è impiegato per fini di riposizionamento o riprogrammazione dei percorsi del robot, permettendo agli operatori di muovere fisicamente i dispositivi in maniera sicura. Nello specifico il robot è in grado di fermarsi e di muoversi ad una bassa velocità.

3. La *Speed and Separation Monitoring* favorisce la collaborazione simultanea nella medesima area di lavoro attraverso il controllo *real time* della velocità e dei movimenti nello spazio del robot, il quale deve regolarli ed eventualmente fermarsi (passando alla *SMS*) per non mettere in pericolo il collaboratore.
4. La *Power and Force Limiting* è usata quando il robot e l'operatore possono avere contatti intenzionali e non. In questo caso il robot deve essere adeguatamente predisposto per essere sicuro al contatto (specifiche *ISO/TS 15066*), dissipando ad esempio la forza di collisione.

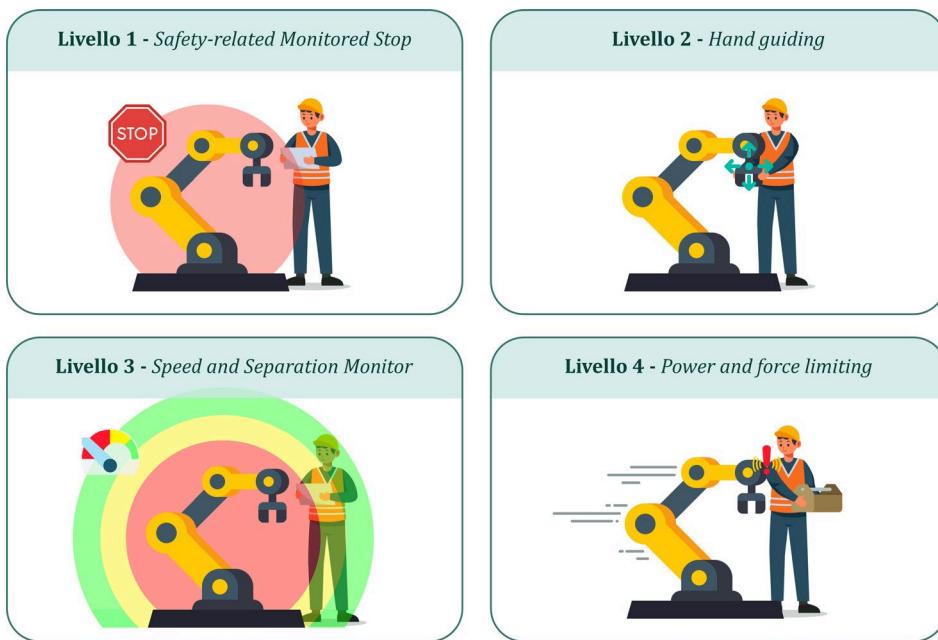


Figura 2 - Modalità di collaborazione uomo-macchina

La complessità di progettazione e sviluppo incrementa il costo di questi prodotti, rendendoli spesso poco graditi agli *stakeholder*. In aggiunta le normative sembrano spingere per un approccio tradizionale, che mira ad eliminare il rischio, limitando la velocità e la forza dei macchinari, sacrificando così efficienza e flessibilità.

Le suddette modalità possono essere comandate o implementate in sistemi che impiegano telecamere e sensori per monitorare in ogni momento lavoratori e macchine e controllarne il comportamento. Il prezzo per un sistema di questo tipo è una elevata complessità ed un costo non indifferente in attrezzatura

ulteriore rispetto al macchinario stesso, con il vantaggio non indifferente di avere maggiore elasticità.

Nuovi approcci puntano a massimizzare l'efficacia del processo, sfruttando sistemi di Intelligenza Artificiale non solo per il controllo dei prodotti della lavorazione, ma anche per migliorare l'efficienza e il ritmo produttivo, delegando mansioni pericolose o ripetitive ai robot.

2.1 Related Works

Diversi studiosi hanno realizzato ricerche ed articoli riguardo il tema della sicurezza in ambito di industria 4.0 e 5.0, esaminando e sperimentando soluzioni più o meno innovative. In particolare, la tendenza degli studi risulta verso un impiego dinamico e adattivo dell'apporto dei robot nella collaborazione uomo-macchina associati all'introduzione di nuove tecnologie come l'Intelligenza Artificiale.

Per superare gli approcci ormai obsoleti dei sistemi di sicurezza statici, è stata proposta l'idea di adottare un sistema dinamico che consenta a uomini e robot di adattare la misura di sicurezza impiegata durante l'operatività, in base al compito specifico. Questo approccio è chiamato *Deliberative Safety*[7]. Per attuare tale strategia vengono definiti due tipi di indicatori (*Key Performance Indicators*):

- Indicatore di flessibilità, per valutare le esigenze di un'operazione. Il robot deve poter ripianificare le proprie azioni, mantenendo la maggior velocità possibile per rimanere efficiente senza risultare pericoloso per il collaboratore (suo concorrente).
- Indicatore di implementazione, per valutare la complessità di una misura di sicurezza sulla base del suo costo e della sua difficoltà di implementazione.

In particolare, gli ideatori di questa strategia hanno individuato una tassonomia di misure di sicurezza tra cui il sistema può statuire. Queste sono: sicurezza perimetrale, a zone, reattiva, attiva e pianificata.

La sicurezza perimetrale (*Perimetral safety*) prevede che il robot si fermi quando l'umano entra nell'area. È semplice, ma non consente collaborazione.

Nella sicurezza a zone (*Zone safety*), invece, l'area di lavoro viene divisa in zone. Il robot deve rallentare progressivamente man mano che il lavoratore si avvicina.

La sicurezza reattiva (*Reactive safety*) sancisce che il robot si muova lentamente e si fermi al contatto (limitazione di forza e potenza). Questo permette alta concorrenza ma è inefficiente.

Nella sicurezza attiva (*Active safety*) vengono utilizzati sensori avanzati (visione, algoritmi predittivi) per evitare attivamente le collisioni modificando la propria traiettoria. Risulta molto flessibile ma estremamente complessa da implementare.

Nella sicurezza pianificata (*Planned safety*), uomo e robot pianificano congiuntamente la sequenza delle azioni prima della loro esecuzione. L'umano sa esattamente cosa farà il robot, permettendogli di lavorare in prossimità a velocità maggiori.

Questo sistema risulta vantaggioso poiché permette al robot di operare in modalità sicurezza a zone quando è isolato, per poi passare a sicurezza reattiva qualora l'operatore dovesse intervenire per un errore, o ancora alla sicurezza pianificata o attiva per la collaborazione. In questo modo l'interazione è sicura e si sfruttano maggiormente i benefici dell'impiego di questi macchinari.

Altri studi[1, 4, 5, 8] mirano ad enfatizzare come sia più vantaggioso sfruttare le macchine, le nuove tecnologie e l'Intelligenza Artificiale non come sostituti dell'uomo, bensì come strumenti di supporto per potenziare le sue capacità, alleggerirlo dai compiti ripetitivi e meramente fisici e al contempo migliorarne la sicurezza.

L'uomo, per sua natura, possiede doti peculiari e insostituibili, quali *expertise* e creatività, che devono rimanere centrali. Una collaborazione con la tecnologia come sua serva, può solo che giovare dell'apporto che questa può fornire. Il risultato si traduce in un potenziamento generale, frutto dei vantaggi di entrambi, una maggiore efficienza e una qualità migliore del lavoro e della vita.

Quindi l'obiettivo è quello di integrare il lavoro umano con quello robotico, in modo da avvalersi dei punti di forza di ambo le parti.

Siffatto approccio, assieme ad una progettazione ottimale delle interfacce uomo-macchina (*HMI*) e alla trasparenza e spiegabilità¹ (*explainability*) dei sistemi IA, è in grado di favorire la fluidità della cooperazione e migliorare la fiducia in questi sistemi, soprattutto con focus sulla quinta rivoluzione industriale.

2.2 Tecnologie

Questo progetto mira a costruire un sistema quanto più generale e adattabile, garantendo al contempo la sicurezza e l'incolumità dell'uomo.

Per giungere all'obiettivo sono senza dubbio necessari diversi requisiti:

- Supporto al *Real Time*: il sistema dovrà essere responsivo ed agire in tempi molto brevi, quindi è necessario adoperare tecnologie hardware e software adatte. A questo proposito, si è deciso di affidarsi allo standard *Ethernet* accoppiato al *set* di estensioni *TSN* (*Time-Sensitive Network*) per gestire il traffico di rete più sensibile. Questi si rendono necessari per la scelta di sfruttare l'approccio dell'*edge computing*.
- Rapidità di elaborazione: per soddisfare i requisiti di funzionamento in tempo reale nell'elaborazione delle immagini, si è scelto di impiegare algoritmi di *Machine Learning* specifici per l'*object detection* e la *depth estimation*, ottimizzandoli con una conversione nel formato ONNX.
- Genericità nei confronti dell'hardware: per mantenere l'infrastruttura il più indipendente possibile dallo strato fisico si è preferito adoperare risorse generali e adatte a supportare dispositivi differenti. Si utilizzeranno in particolare progetti *Open Source* come *Aravis*, per l'integrazione con telecamere industriali, e *Docker*, come substrato su cui

¹ Per spiegabilità o *explainability*, nei sistemi di Intelligenza Artificiale, si intende la capacità di un sistema di rendere comprensibili ed interpretabili i processi decisionali ed i risultati prodotti, sia per gli sviluppatori che per gli utenti finali, ossia i lavoratori in questo contesto. La spiegabilità permette di ottenere maggiore fiducia e trasparenza, aiuta il debug e il miglioramento generale del sistema.

eseguire i programmi al fine di slegarsi ulteriormente dall'architettura sottostante, rendendo il tutto replicabile. Inoltre, anche la conversione dei modelli di Intelligenza Artificiale nel formato ONNX favorisce questo disaccoppiamento.

2.2.1 Cloud ed Edge Continuum

Negli ultimi decenni la sempre maggiore esigenza di eseguire operazioni su hardware costoso ed oneroso da mantenere, ha spinto verso la costituzione di centri di computazione *off-site*, raggiungibili via Internet, con costo ridotto basato sull'uso di risorse *on demand*. Questo approccio, il Cloud, presenta molti vantaggi, soprattutto per le organizzazioni più piccole, che non dispongono di figure e capitale sufficiente a mantenere strutture simili da sole. Tuttavia, data la continua diffusione di dispositivi *IoT*, l'aumento della mole di dati da scambiare e memorizzare e delle operazioni da compiere, unite alle esigenze di rapida disponibilità dei risultati, si trova un ostacolo nella tradizionale soluzione Cloud. In particolare, il collo di bottiglia principale è rappresentato dalla rete. Viene da sé che data la natura basata su Internet del Cloud, le latenze possono facilmente aumentare e rendere impossibile il *real time*, oltre alla necessità di disporre di una connessione ad Internet stabile. L'enorme volume di dati inviati potrebbe congestionare facilmente la rete, senza contare i problemi correlati a privacy e sicurezza di trasporto e conservazione dei dati.

Per affrontare queste significative problematiche, l'industria ha adottato un modello ibrido: il *Continuum Cloud-Edge*. Il principio è la distribuzione della capacità di calcolo su tutta la rete (non solo *off-site* sul Cloud), così da poter utilizzare le risorse più adeguate sulla base delle esigenze e in modo dinamico. Il Cloud rimane la scelta ottima per eseguire compiti non urgenti e per memorizzare grandi quantità di dati a cui non è indispensabile accedere rapidamente. L'*edge*, ossia le risorse collocate al confine della rete locale, rappresenta la soluzione ideale per il calcolo e lo stoccaggio di dati fortemente dipendenti da prestazioni *real time*.

L'architettura del *Cloud-Edge Continuum* è suddivisa in più livelli:

- i. *Tier 1*, composto dai dispositivi IoT sulla rete locale, ossia i sensori e gli attuatori che raccolgono e generano i dati.
- ii. *Tier 2*, la parte *Edge*, al confine con Internet, dotata di elevate capacità di calcolo, che la rendono perfetta per essere destinata a computazioni locali che necessitano di rapide risposte oppure della pre-elaborazione dei dati per essere inviate al Cloud, al fine di diminuire il traffico e la congestione verso l'esterno.
- iii. *Tier 3*, rappresentato dal Cloud, accessibile via Internet, di cui fanno parte tipicamente smisurati *data center* che dispongono di risorse computazionali enormi e di grandi quantità di spazio per l'archiviazione a lungo termine.

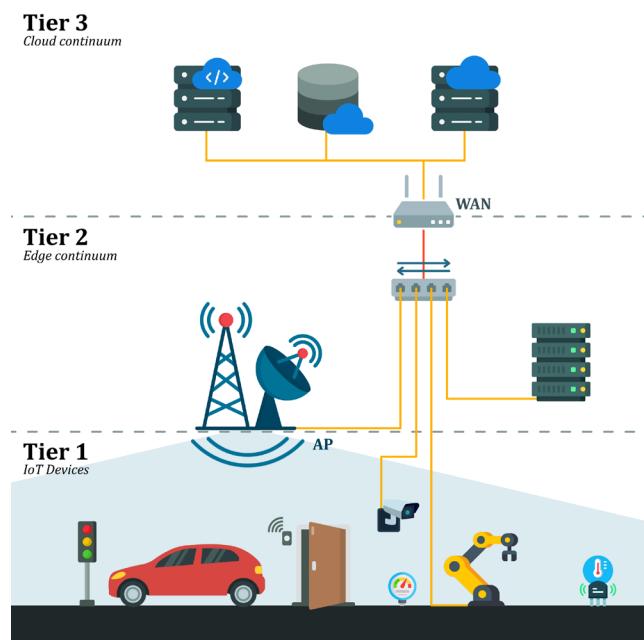


Figura 3 – Architettura a tre livelli Edge-Cloud Continuum

Sulla base del contesto, ossia di informazioni riguardanti la situazione attuale (come livelli di utilizzo delle risorse, latenza, velocità di connessione, posizione), viene scelta dinamicamente la destinazione su cui elaborare o stoccare i dati[9].

2.2.2 TSN

TSN (*Time-Sensitive Networking*) è una raccolta di standard, emendamenti e progetti, sviluppati dal *Task Group IEEE 802.1*, i quali operano al livello 2 della pila OSI. Lo scopo principale di TSN è migliorare ed estendere gli standard del protocollo *Ethernet* (*i.e.* IEEE 802.1 e IEEE 802.3) per garantire maggiore affidabilità e supporto al *real time*.

La peculiarità di TSN è l'aggiunta di priorità differenti ai messaggi. Per la realizzazione di questo sistema è necessario uno switch centrale in grado di sfruttare un meccanismo a *gate driver*: ad ogni porta vengono associate un numero di code pari alle diverse priorità, in cui andranno collocati i pacchetti corrispondenti. Questo permette di evadere prima i messaggi con priorità più alte e permettere una gestione più granulare del traffico.

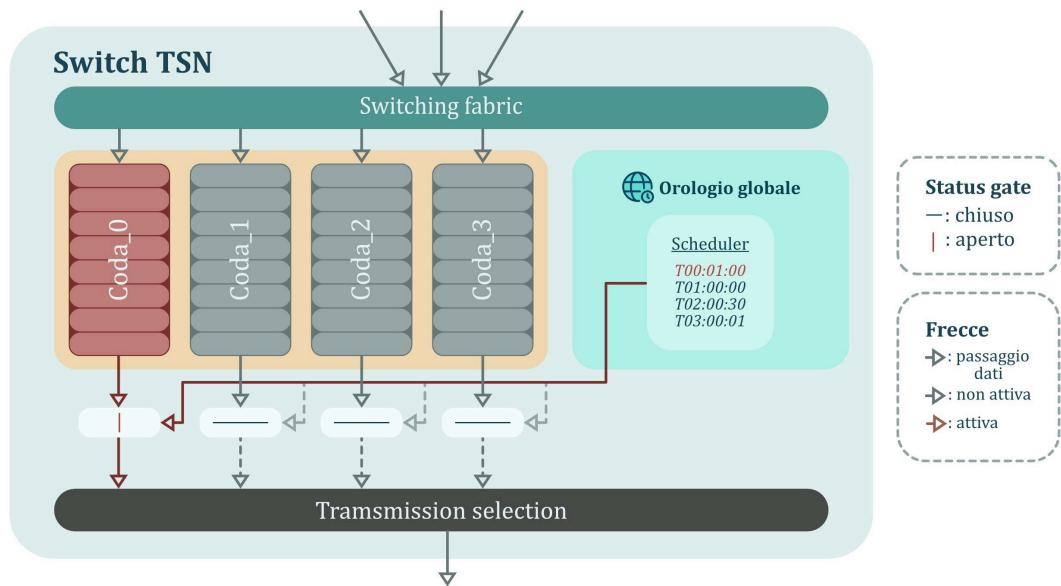


Figura 4 - TSN switch con code multiple

I dispositivi che inviano i flussi TSN sono detti *talker*, mentre quelli che li ricevono sono chiamati *listener*. Il *Central Network Controller* (CNC), tipicamente collocato all'interno dello switch, *on-premises*, si occupa di coordinare il traffico, calcolare le rotte e scandire gli orari di tutti i flussi. L'interfaccia utente del sistema è la *Centralized User Configuration* (CUC), ossia un applicativo che permette di stabilire i requisiti dei flussi che il CNC deve rispettare.

La raccolta di standard TSN è modulare, pertanto ogni progettista può selezionare quelli necessari al suo scopo ed escluderne altri.

Le macro-caratteristiche principali di TSN sono quattro: sincronizzazione temporale, latenza, affidabilità e gestione delle risorse.

Per la sincronizzazione temporale gli standard chiave sono IEEE 802.1AS e la sua revisione (AS-Rev). Questi definiscono il protocollo gPTP (*Generalized Precision Time Protocol*), evoluzione dello standard IEEE 1588. Lo scopo è sincronizzare con precisione gli orologi di tutti i dispositivi (*switch* e stazioni finali) sulla rete. In generale, il funzionamento si basa su una gerarchia di sincronizzazione ad albero, in cui in alto si trova un *Grandmaster* che funge da riferimento temporale, mentre gli altri dispositivi si comportano come *Clock Slaves*. La revisione introduce miglioramenti per la tolleranza ai guasti.

La latenza ha il fine di garantire che i messaggi critici giungano a destinazione entro una scadenza determinata. Per realizzare questo comportamento esistono diversi meccanismi di modellazione del traffico (*shapes*). Gli standard che rientrano in questa categoria sono molteplici.

IEEE 802.1Qbv - *Time-Aware Shaper* (TAS), introduce il concetto di *gate* per ogni coda di uscita. Questi *gate* si aprono e si chiudono seguendo una pianificazione temporale definita. In questo modo il traffico relativo ad ogni priorità può circolare nella propria finestra temporale, senza che si mescolino messaggi diversi. Questo richiede un'entità di configurazione di rete centralizzata (il CNC) e una sincronizzazione temporale precisa in tutta la rete. IEEE 802.1Qbu e 802.3br - *Frame Preemption*, permette a *frame* ad alta priorità di interrompere la trasmissione (*preemption*) di *frame* di più bassa priorità in corso. Un *frame* a bassa priorità è quindi suddiviso in frammenti, la cui trasmissione riprende dopo che il *frame* ad alta priorità è stato instradato. Così facendo la latenza dei messaggi critici cala notevolmente.

IEEE 802.1Qav - *Credit-Based Shaper* (CBS), si basa su un sistema di crediti. Questi vengono guadagnati da ogni coda mentre questa è in attesa e consumati durante la trasmissione. Un *frame* può essere inviato solo se il credito è positivo. Questa estensione è stata pensata per flussi multimediali (audio/video) con lo

scopo di regolare la larghezza di banda assegnata a determinate classi di traffico e cercare di ottenere una certa equità nella distribuzione delle risorse, limitando anche il *bursting*.

IEEE 802.1Qch - *Cyclic Queuing and Forwarding* (CQF), mira a semplificare la progettazione degli switch, impiegando due code che operano in maniera ciclica. In particolare, ogni *frame* ricevuto durante un ciclo temporale verrà sempre inoltrato in quello successivo. Pertanto, la latenza che si viene a creare è deterministica e prevedibile, in quanto è basata sulla durata del ciclo temporale e sulla lunghezza del percorso.

IEEE 802.1Qcr - *Asynchronous Traffic Shaping* (ATS), funziona in modo asincrono, rimodellando i flussi di traffico ad ogni *hop* per smussare le raffiche e dare priorità al traffico urgente tramite uno *scheduler* apposito. A differenza di TAS non richiede una sincronizzazione temporale globale precisa, data la caratteristica asincrona.

L'affidabilità (*Reliability*) ha lo scopo di garantire la consegna dei pacchetti anche in caso di guasti. Al fine TSN prevede vari standard.

IEEE 802.1CB - *Frame Replication and Elimination for Reliability* (FRER), si basa sulla creazione di copie dei *frame* critici alla sorgente, inoltrandoli attraverso percorsi disgiunti. Il dispositivo di destinazione, oppure uno *switch* intermedio, elimina i duplicati inoltrando solo la prima copia ricevuta, garantendo quindi la consegna, anche se non tutte le copie dovessero arrivare.

IEEE 802.1Qca - *Path Control and Reservation* (PCR), è lo standard necessario a stabilire, controllare e riservare esplicitamente i percorsi multipli disgiunti necessari per il funzionamento del FRER.

IEEE 802.1Qci - *Per-Stream Filtering and Policing* (PSFP), punta a migliorare affidabilità e sicurezza filtrando e monitorando il traffico per ogni flusso. In questo modo è possibile bloccare trasmissioni non autorizzate o non corrette, proteggendo la rete da attacchi o dispositivi malfunzionanti.

Nella gestione delle risorse è definito come configurare e gestire l'intera rete TSN. Anche qui si trovano diversi standard differenti.

IEEE 802.1Qcc - *Enhancements to Stream Reservation Protocol (SRP)*, definisce i modelli per la configurazione delle reti TSN (centralizzato, distribuito o ibrido), introduce le entità centrali responsabili del calcolo dei percorsi e della pianificazione (CNC).

IEEE 802.1Qcp - *YANG Data Model*, dota di un modello dati standardizzato (YANG) per la rappresentazione di configurazione e stato dei dispositivi TSN. In questo modo l'interfaccia è universale perché standardizzata e indipendente dal produttore hardware[10].



Figura 5 - Moduli TSN

La configurazione della rete TSN si compone di diversi passi (non sempre eseguiti nel seguente ordine):

- i. Scoperta della Topologia della rete per mezzo del CNC, utilizzando il protocollo *Link Layer Discovery* (LLDP) partendo da un determinato dispositivo, detto *seed*.
- ii. Definizione tramite CUC, da parte di un addetto, dei flussi necessari con i rispettivi requisiti di latenza e la dimensione massima dei pacchetti. Queste informazioni sono inviate al CNC tramite API REST.
- iii. Scheduling da parte del CNC dei percorsi e pianificazione temporale ottimale per ogni flusso, il che garantisce l'assenza di conflitti.
- iv. Distribuzione della pianificazione dal CNC verso ogni switch della porzione di competenza. Il CUC si occupa di programmare *talker* e *listener* con i rispettivi orari di invio e ricezione[11].

2.2.3 Docker

Docker è un potente strumento per la virtualizzazione leggera di risorse su macchine Linux *host* basato sul concetto di container. La peculiarità di questo approccio è il suo irrisorio costo della virtualizzazione.

Nei sistemi tradizionali basati su macchine virtuali vengono replicati interi sistemi operativi, su cui poi vengono eseguite le specifiche applicazioni. In Docker, invece, il *kernel* del sistema operativo della macchina *host* viene condiviso tra le unità, i container. Questo permette di alleggerire notevolmente il carico, favorendo l'efficienza e limitando il consumo di risorse, pur mantenendo i vantaggi di encapsulamento delle applicazioni, isolamento e portabilità tipici della virtualizzazione tradizionale.

Gli elementi caratteristici dell'ecosistema Docker sono: immagine, container e Dockerfile.

Un'immagine è un pacchetto immutabile, portatile e leggero che incapsula un'applicazione e le sue dipendenze. Si tratta sostanzialmente di un *template* statico. La particolarità di questo oggetto è la sua struttura del *file system* a strati *read-only*. Questa struttura a livelli viene impiegata per ottimizzare lo spazio usato, permettendo il riuso di *layer* comuni a immagini diverse. Gli strati sono creati sulla base delle istruzioni contenute nel Dockerfile.

Un container è un'istanza eseguibile e isolata di un'immagine. Il container aggiunge un substrato scrivibile sopra quelli dell'immagine statica. In questo substrato vengono inseriti tutti i cambiamenti apportati all'immagine di partenza a *runtime*. Questo permette di avviare più container con la stessa immagine di partenza senza duplicati, memorizzando unicamente gli strati che le differenziano. Eliminando un container, si perde, di fatto, lo strato scrivibile e dinamico.

All'atto di avvio di un container si possono operare diverse scelte circa la configurazione del sistema. Per quanto riguarda la rete, di norma, viene creata un'interfaccia apposita per Docker, ma è possibile anche condividere quella dell'*host*. Possono essere conferiti permessi più o meno elevati e può anche

essere deciso di rendere il container persistente, così da mantenerlo anche in caso di arresti anomali che ne comporterebbero l'eliminazione. In caso di container persistente è possibile anche renderlo riavviabile automaticamente.

Un Dockerfile è un file di testo che contiene le istruzioni su cui si basa la costruzione di un'immagine. Le istruzioni disponibili sono utili per indicare una specifica immagine esistente da cui partire (**FROM**), copiare file da macchina *host* a container (**COPY**), eseguire comandi *shell* (**RUN**). Ogni istruzione va a costituire uno dei livelli di sola lettura dell'immagine che permette di creare. Queste istruzioni sono standard e, per questo, risultato coerenti e riproducibili su qualsiasi macchina analoga.

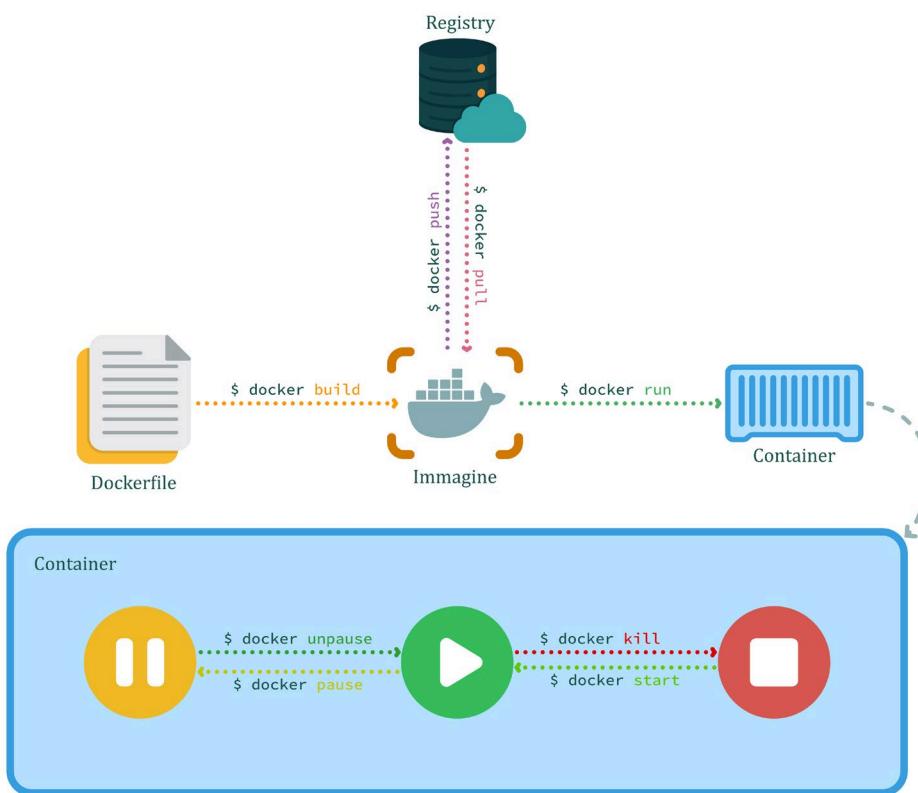


Figura 6 - Flusso comandi Docker

È presente un sistema di distribuzione di immagini Docker composto da due entità: *Registry* e *Repository*.

Il *Registry* è un *hub* centralizzato di archiviazione e distribuzione per le immagini. In particolare, esiste un *registry* pubblico, Docker Hub, che ospita una

vasta libreria di immagini precostruite, ma esistono anche *registry* privati utilizzati dalle aziende per archiviare immagini proprietarie in modo sicuro.

Ogni *registry* è composto di una o più *repository*, ossia raccolte di immagini Docker correlate, tipicamente versioni diverse della stessa applicazione.

Docker adotta un'architettura Client-Server, in cui il server è `dockerd`, un demone, con il compito di rimanere in attesa di comandi API inviati dal client, gestire gli oggetti Docker (container, immagini, volumi e reti) ed eseguire azioni principalmente sulla base delle istruzioni che riceve. Nel server risiedono quindi tutte le logiche e le meccaniche fondamentali di Docker. Il client Docker è `docker`, interfaccia a riga di comando (CLI) con cui l'utente interagisce inserendo i comandi da inviare al server. Di fatto quindi il client si occupa unicamente di tradurre i comandi ricevuti dall'utente in richieste API e mandarle al server Docker, facendo da intermediario[12].

Spesso il client e il server risiedono sulla medesima macchina *host*.

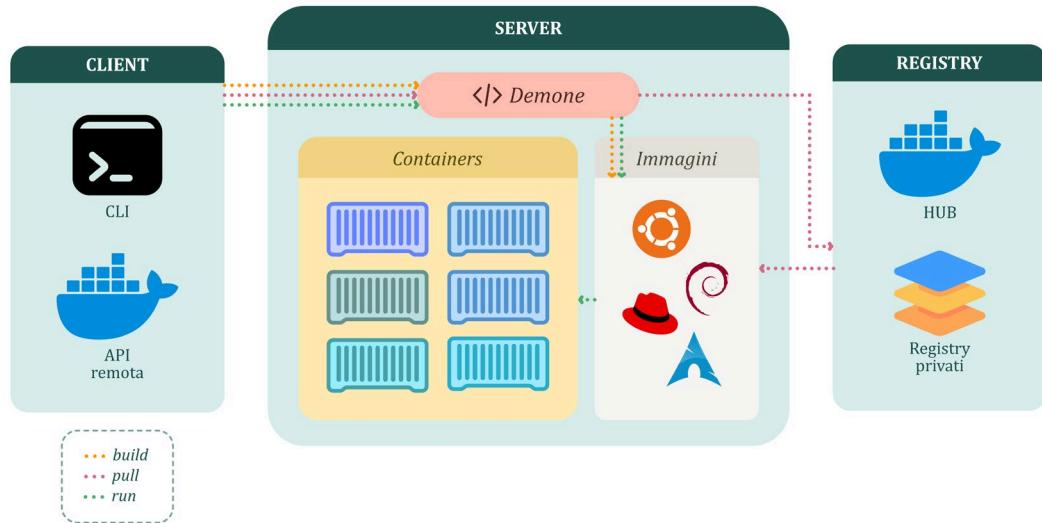


Figura 7 - Architettura Docker

2.2.4 Object Detection

L'*object detection* è una sottoclasse o compito della *computer vision* (CV), ossia della disciplina scientifica che si occupa di sviluppare metodologie algoritmiche e modelli computazionali volti ad ottenere una comprensione ad alto livello delle immagini e dei video digitali. Il suo obiettivo primario è replicare,

automatizzare e potenziare le funzioni della vista umana (percezione, elaborazione e interpretazione)[13]. L'*object detection* punta in particolare a risolvere alcuni sottoproblemi della *computer vision*: classificazione e localizzazione di oggetti. Cerca quindi di individuare ogni istanza di un oggetto presente in un'immagine o video, identificandone la classe di appartenenza e localizzando tale istanza in un'area rettangolare dello spazio che la contiene interamente, detta *bounding box*[14]. Questo compito differisce da altre branche della CV con cui potrebbe essere confusa, come l'*image classification*, che prevede unicamente l'individuazione della classe di un unico oggetto per immagine, e l'*instance segmentation*, che include sia classificazione che localizzazione, ma aggiunge la complessità di delineare esattamente il contorno di ciascun oggetto.

Il problema che affronta l'*object detection* era sentito già ben prima dell'avvento dell'Intelligenza Artificiale e del *machine learning*, tanto che fino al 2014 esistevano algoritmi atti al medesimo scopo basati su caratteristiche ingegnerizzate manualmente dagli sviluppatori. Alcuni dei principali sono *Viola-Jones Detector* (2001), primo algoritmo in grado di rilevare volti umani in tempo reale, e *Deformable Part-Based Model* (2008), in cui un oggetto è visto come insieme di parti deformabili che lo costituiscono, il che permette il suo riconoscimento tramite l'individuazione delle sue parti costitutive.

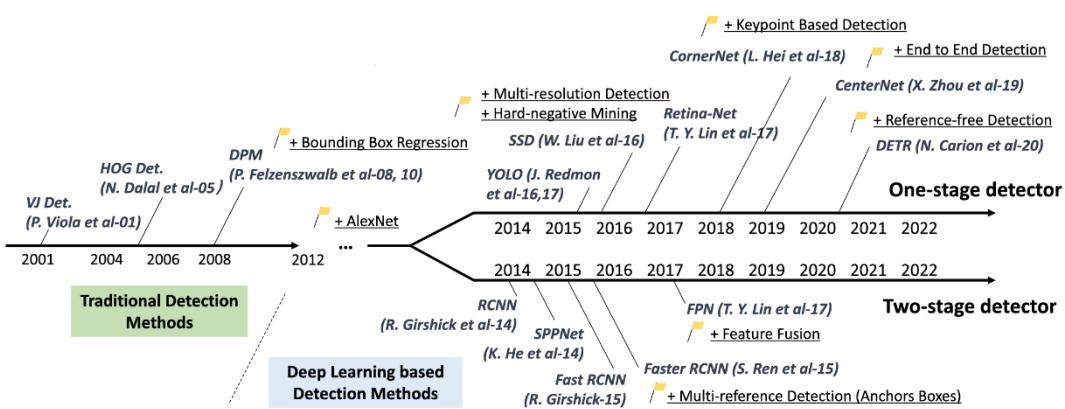


Figura 8 – Pietre miliari dell’Object Detection[15]

Successivamente è sopraggiunta una svolta: le reti neurali convoluzionali (CNN). Si tratta di reti neurali in grado di apprendere automaticamente le

caratteristiche rilevanti direttamente dai dati, permettendo il surclassamento delle *performance* rispetto alle soluzioni precedenti. In questo periodo si sono venute a creare due famiglie di rilevatori basati su CNN: rilevatori a due stadi (*Two-Stage Detectors*) e i rilevatori a stadio singolo (*One-Stage Detectors*). I rilevatori a due stadi prevedono due fasi, ossia la proposta delle regioni e poi la loro classificazione e rifinitura. Nella prima fase la rete genera delle proposte di regione, ovvero riquadri con alta probabilità di contenere un qualche oggetto, indipendentemente dalla sua classe, il cui obiettivo è assicurarsi di non perdere alcun potenziale oggetto. Nella seconda fase, invece, vengono analizzate tutte le proposte e a ciascuna viene assegnata un'etichetta (*label*) specifica in cui sono riportate le proprie informazioni di identificazione, come classe e posizione. Infine, vengono corrette le coordinate dei riquadri, scremando il numero dei risultati che identificherebbero gli stessi oggetti. Di questa tipologia fanno parte le R-CNN, le successive *fast* e *faster* R-CNN e le *Feature Pyramid Networks*. I rilevatori a singolo stadio trattano il rilevamento come un unico problema di regressione e classificazione. L'immagine viene passata attraverso la rete una sola volta, e l'*output* è una predizione diretta di classi degli oggetti e coordinate delle *bounding box*. Di questa categoria fanno parte ad esempio YOLO, *Single Shot MultiBox Detector* e RetinaNet. I rilevatori a singolo stadio sono molto più veloci e quindi adatti a contesti *real time*, ma meno accurati rispetto a quelli a due stadi.

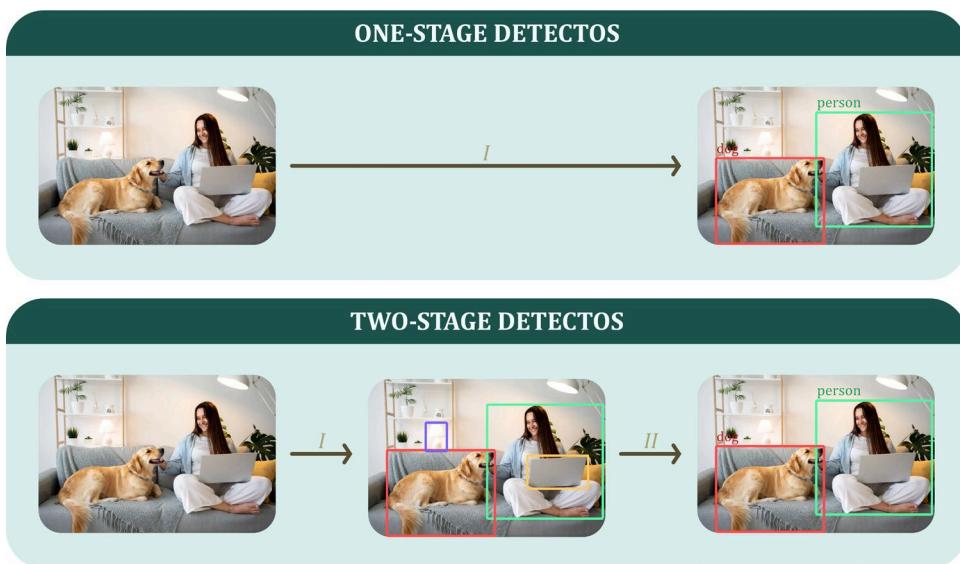


Figura 9 - One- e Two-Stage Detectors

Sebbene si siano fatti enormi progressi in termini di velocità di elaborazione, è possibile ricorrere ad ulteriori ottimizzazioni che impattano positivamente sui tempi di computazione a scapito della precisione. Adottare il *Network Pruning*, ossia la rimozione dei pesi non necessari, o la quantizzazione, ovvero l'impiego di rappresentazioni numeriche di dimensioni minori, permette di rispettare requisiti di tempo più stringenti. Altre ottimizzazioni si possono ottenere operando sul calcolo numerico, quindi impiegando l'*Integral Image* o la trasformata di Fourier (FFT) per accelerare le operazioni di convoluzione[15].

2.2.5 Depth Estimation

La *depth estimation* tratta la stima della profondità di una scena, ossia la predizione del calcolo della distanza di ogni punto della scena dal dispositivo che ha scattato la fotografia. Questo è un compito fondamentale della CV. La versione monoculare, quindi ottenuta da un solo punto di vista, è particolarmente impegnativa perché cerca di dedurre informazioni 3D da una singola immagine 2D, problema intrinsecamente mal posto dato che un'intera dimensione (la profondità) viene persa durante la proiezione dell'immagine.

L'evoluzione di questa tecnologia segue una linea simile a quella dell'*object detection*. Infatti, inizia nel 2009 con metodi basati sullo sfruttamento di elementi che suggeriscono la profondità, come ombre, messa a fuoco e punti di fuga. Questo approccio è fortemente inefficiente e funziona unicamente in scene molto controllate. Successivamente, con l'avvento del *machine learning*, sono state realizzate soluzioni ingegnerizzate manualmente e modelli probabilistici più robusti ma complessi e ancora lenti. In seguito, con il *deep learning* e le reti neurali, c'è stata una vera rivoluzione nel campo, che ha portato ad alta accuratezza, praticità ed applicabilità a scene eterogenee e complesse.

I modelli fondamentali di *deep learning* sono vari. Nelle CNN l'architettura tipica prevede un *encoder* che comprime le immagini in *feature* e un *decoder* che le usa per ricostruire una mappa di profondità. Le RNN (*Recurrent Neural Network*) sono impiegate per analizzare sequenze video grazie alla capacità di apprendere caratteristiche temporali tra i fotogrammi, risultando

estremamente utili per stimare la profondità nei video. Invece le GAN (*Generative Adversarial Network*) sono usate per generare mappe di profondità più realistiche e definite, riducendo l'effetto di sfocatura tipico di altri modelli.

I modelli di *depth estimation* si categorizzano sulla base del metodo di addestramento oppure per tipo di compito.

Per quanto riguarda l'apprendimento, questo può essere supervisionato, non supervisionato, oppure semi-supervisionato. L'apprendimento supervisionato prevede che l'addestramento avvenga tramite un *dataset* di immagini RGB in cui a ciascuna è associata la corrispondente mappa di profondità *ground truth* (GT), ossia basata su misurazioni accurate della profondità, ottenute con sensori LiDAR o fotocamere stereo. Lo svantaggio è il laborioso e oneroso dispendio necessario alla raccolta di mappe altamente precise.

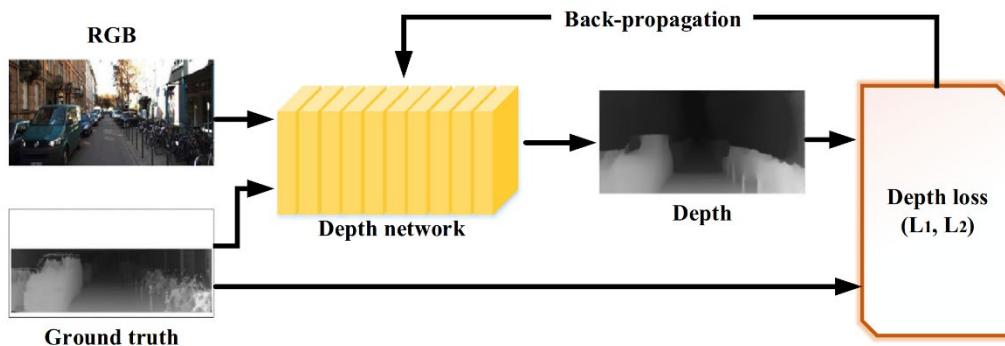


Figura 10 - Apprendimento supervisionato[16]

Per arginare gli svantaggi dell'apprendimento supervisionato, sono stati realizzati modelli con apprendimento non supervisionato, in cui, invece delle mappe di profondità GT, vengono sfruttati i vincoli geometrici delle scene. Qui esistono due differenti metodi: *stereo matching* e *monocular sequences*. Nello *stereo matching* la rete viene addestrata con coppie di immagini stereo da cui impara a riconoscere e predire le disparità (quindi la profondità) e utilizza queste differenze per ricostruire una delle immagini della coppia minimizzando la differenza tra quella ricostruita e quella reale. Nella *monocular sequences*, la rete viene addestrata con video monoculari, imparando a predire simultaneamente la profondità di un fotogramma e il movimento della

telecamera tra *frame* consecutivi. L'approccio non supervisionato è generalmente inferiore nei risultati e soffre di problemi di ambiguità di scala.

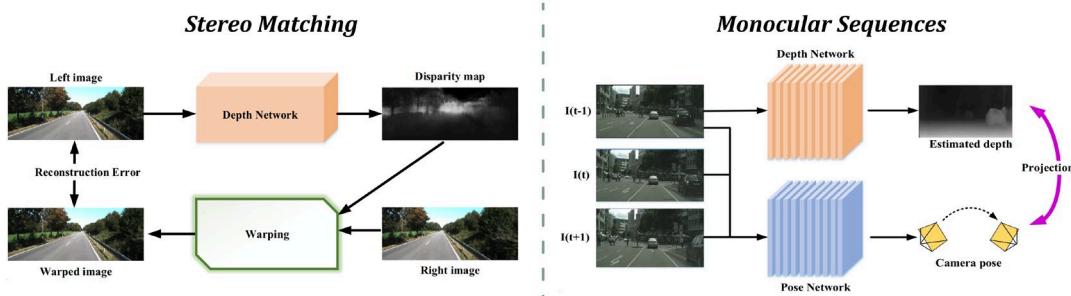


Figura 11 - Monocular Sequences e Stereo Matching[16]

Soppesando vantaggi e svantaggi dei procedimenti supervisionato e non, è stato realizzato un metodo ibrido, il semi-supervisionato, che prevede di ridurre l'impiego di dati GT, eseguendo l'addestramento con dati sintetici, affinandolo poi su dati reali e usando i dati forniti da sensori LiDAR per migliorare le stime.

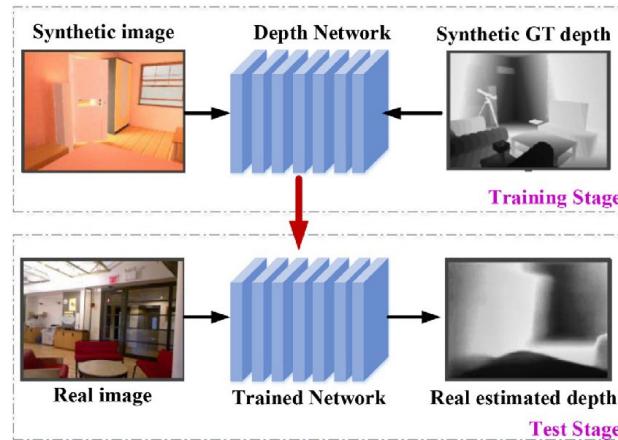


Figura 12 - Apprendimento semi-supervisionato[16]

La tassonomia per tipo di compito si basa sull'impegno che dovrà essere fatto del modello. Nel *Single-Task Learning* è previsto che la rete venga addestrata ad eseguire un unico compito, ossia predire la mappa di profondità, mentre nel *Multi-Task Learning* la rete è addestrata per eseguire più compiti contemporaneamente, partendo dal presupposto che questi compiti differenti si aiutino a vicenda. In altre parole, si fa leva sull'idea che imparare il contesto della scena possa migliorare la stima della profondità, come ad esempio combinando la *depth estimation* con *segmentation* oppure con il *tracking* del movimento della camera e dei pixel[16].

2. Related works e tecnologie

Gli algoritmi di *depth estimation* sono tipicamente in grado di stimare distanze relative tra gli elementi nella scena, ma esistono anche sistemi, detti metrici, capaci di fornire risultati assoluti che seguono la scala metrica. La difficoltà di questi ultimi si insinua nel fatto che le immagini sono manipolate dai parametri fisici della telecamera che cattura la scena, pertanto ne risultano fotografie alterate che dipendono da quei parametri, di cui il più importante è forse la lunghezza focale.



Figura 13 - Differenze risultanti da diversi parametri di una fotocamera[17]

3. Il progetto EdgeCV4Safety

Il comportamento che il sistema deve realizzare richiede che venga monitorata un'area di lavoro e che, all'entrata di una o più persone in questo spazio, vengano adottate delle opportune misure basate sulla posizione degli individui. Queste misure devono influenzare il comportamento dei macchinari di interesse nell'area.

L'architettura da concretizzare si compone di diversi elementi, ciascuno con il proprio scopo preciso. Questi si interfacciano tra loro e insieme adempiono all'obiettivo finale: conferire maggiore sicurezza del lavoratore che agisce all'interno dell'ambiente, permettendo la rimozione delle barriere fisiche attorno ai macchinari di interesse. Tali caratteristiche di compartimentazione conferiscono flessibilità, modularità e scalabilità al sistema.

3.1 Architettura e scelte tecnologiche

Per delineare l'architettura si possono individuare alcune funzioni essenziali e poi procedere effettuando le scelte migliori.

Innanzitutto, va monitorata un'area di lavoro. Per farlo, mantenendo l'architettura il più elastica e semplice, si è optato per utilizzare una telecamera atta ad essere collocata ovunque in maniera flessibile, pur mantenendo *performance* adeguate allo scopo. La scelta è ricaduta quindi su una telecamera industriale compatibile con la famiglia di protocolli TSN. Questo dettaglio conferisce il vantaggio ulteriore di disporre di un'interfaccia di connessione standard, che permette anche di coprire lunghe distanze in modo stabile e responsivo.

Per il monitoraggio effettivo dello spazio lavorativo, non è sufficiente la mera acquisizione delle immagini, ma è necessario eseguire delle elaborazioni sui dati raccolti dalla telecamera. Tali computazioni devono portare a riconoscere la presenza di una o più persone. Qui entra in gioco la *computer vision*. Si è optato quindi di utilizzare un modello di *object detection* rapido ed efficiente. Oltre a questo, si ritiene necessario anche riuscire a stimare le distanze delle

3. Il progetto EdgeCV4Safety

persone individuate, per ottenere dei comportamenti più granulari. Allo scopo si farà uso di modelli di *depth estimation* metrici ad alte prestazioni. Per il supporto al *real time*, le elaborazioni con questi modelli devo essere eseguite su un calcolatore potente, dotato di componenti adatte, nella fattispecie di una scheda video (GPU) dedicata.

Per orchestrare il traffico TSN della telecamera e farlo pervenire al computer dedicato all'elaborazione dei risultati, si rende indispensabile disporre di hardware adeguato, ossia di uno *switch* in grado di rispettare le specifiche dettate dalla famiglia dei protocolli TSN.

Infine, sulla base dei risultati ottenuti dalle computazioni è essenziale applicare opportune contromisure sui dispositivi e macchinari di interesse. Perciò, i risultati acquisiti vanno interpretati e, di conseguenza, inviate le nuove direttive ai rispettivi macchinari.

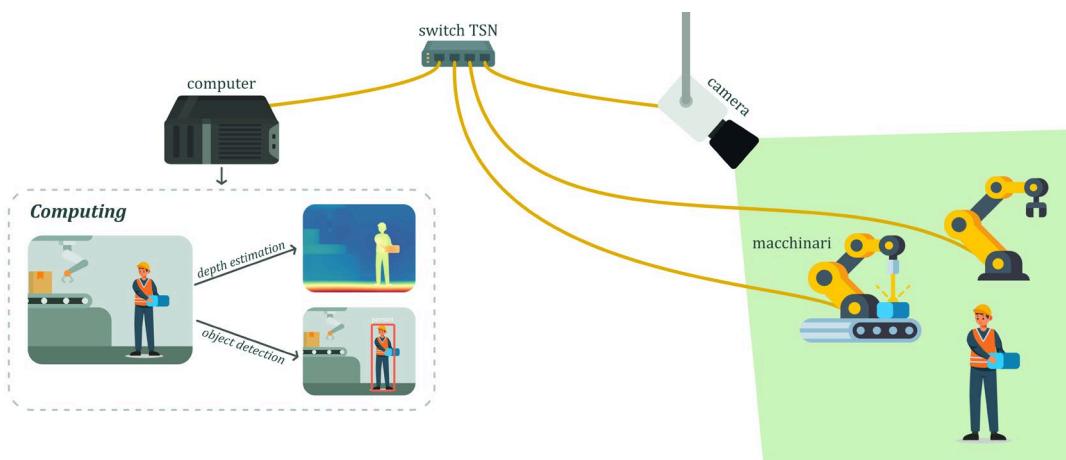


Figura 14 - Architettura generale del sistema

Nell'architettura finora descritta, rappresentata in *Figura 14*, lo switch funge da collante per tutti gli elementi, mettendoli in comunicazione e orchestrando i flussi di dati prioritizzandoli. Il nodo con GPU è il cuore dell'architettura, in quanto deve eseguire le elaborazioni richieste, stabilire il comportamento che i macchinari devono assumere e comunicare loro le nuove disposizioni. Per alleggerire il peso che graverebbe sul nodo con GPU, ma anche per garantire maggiore modularità, si può pensare di delegare le decisioni ad un ulteriore nodo, il quale non ha bisogno di hardware particolarmente prestante. A causa delle necessità di *real time* della natura del caso d'uso, è imprescindibile che i

nodi impiegati per le computazioni siano situati in loco, sfruttando quindi il principio dell'*edge continuum* per evitare l'*overhead* che introdurrebbe una comunicazione esterna. Questo, assieme all'utilizzo di TSN e di modelli di IA performanti in formato ONNX dovrebbe garantire adeguate prestazioni, idonee al *real time*.

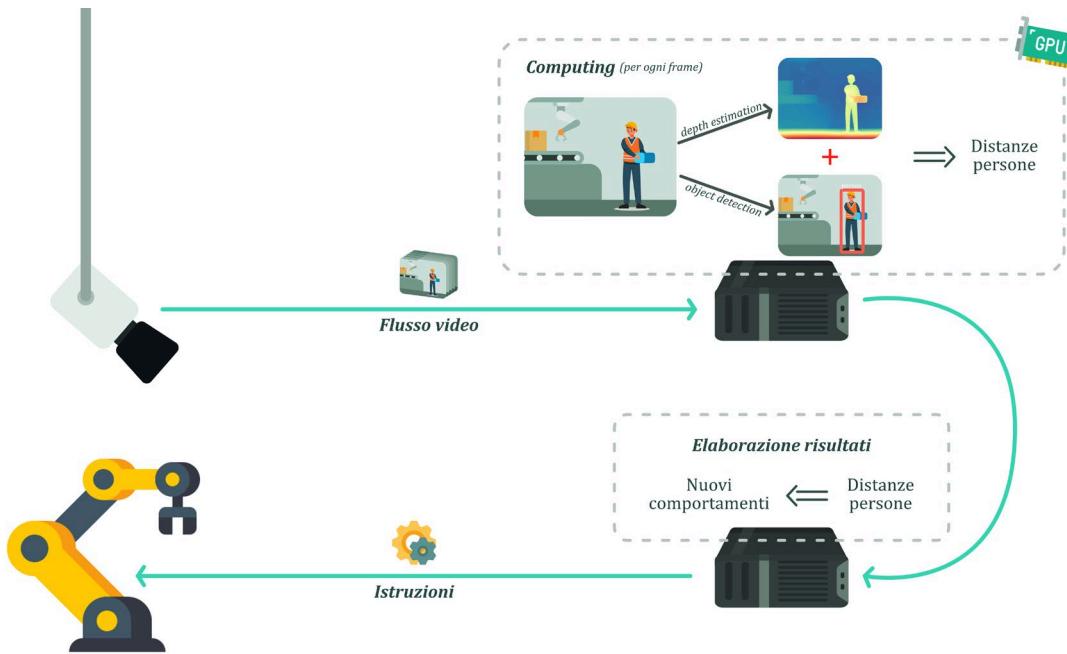


Figura 15 - Flusso del funzionamento dell'architettura

3.1.1 Aravis

Aravis è una libreria per la configurazione, il controllo e la gestione del flusso video di telecamere digitali industriali ad alte prestazioni. Si tratta di un progetto *open source* in grado di comandare una varietà elevata di dispositivi, comportandosi da intermediario tra un computer ed una o più telecamere.

La problematica principale affrontata da questa libreria riguarda la gestione delle telecamere industriali, le quali richiedono protocolli di comunicazione specifici per garantire velocità, bassa latenza e controllo su ogni parametro, a differenza delle webcam *consumer plug-and-play*.

Aravis supporta due dei protocolli più importanti ed utilizzati in questo ambito:

- GigE Vision, standard per la trasmissione video e il controllo su rete Ethernet, tipicamente Gigabit Ethernet, ma ormai diffuso anche per

versioni successive. Il vantaggio principale è la possibilità di usare connessioni cablate molto lunghe con infrastrutture di rete regolari[18].

- USB3 Vision, simile a GigE Vision, ma basato su USB 3.0, il che rende la larghezza di banda generalmente più elevata e una connessione più diretta[19].

Il funzionamento di Aravis si basa sul supporto di GenICam (*Generic Interface for Cameras*), uno standard che descrive le funzionalità di una telecamera tramite un file XML esposto da ciascun dispositivo che lo supporta[20]. Aravis sostanzialmente sfrutta tale file XML, facendo da intermediario e fornendo un'API unificata ed indipendente dallo specifico dispositivo con cui comunica[21].

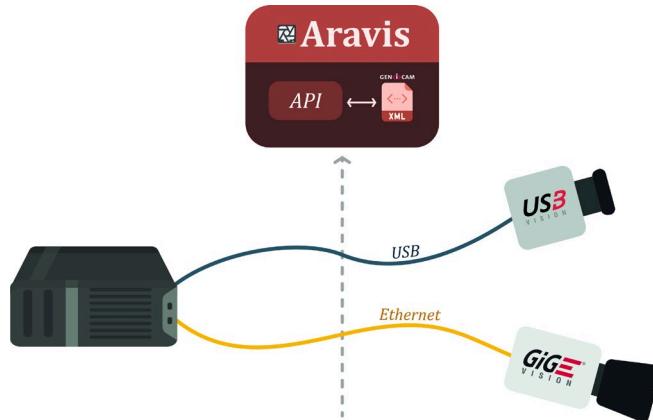


Figura 16 - Schema funzionamento Aravis

Nell'architettura del progetto Aravis è fondamentale per gestire la comunicazione con la telecamera in uso prescindendo dal modello e disponendo comunque della possibilità di modificare i parametri dei dispositivi in maniera indipendente.

3.1.2 YOLO

YOLO è un sistema di *computer vision* per *object detection* in tempo reale, basato sul rilevamento a stadio singolo (*one-stage*), come suggerisce l'acronimo del nome *You Only Look Once*. Essendo un rilevatore a stadio singolo, la sua caratteristica principale è proprio la rapidità.

YOLO è realizzato per computare anche su dispositivi con potenza limitata (quali smartphone, droni, telecamere e dispositivi *edge* in generale). L'uso che viene fatto di questo modello spazia dal controllo dei prodotti industriali, con lo scopo di individuare difetti o imperfezioni, alla videosorveglianza. Per gli obiettivi di questo lavoro, verrà impiegato in quanto stato dell'arte per l'*object detection real time*.

Questo modello tratta il rilevamento come un unico problema di regressione che, partendo da un'immagine, identifica le classi degli oggetti tramite *bounding box*, ovvero rettangoli che delimitano l'area in cui si trova un determinato oggetto.

Il flusso che segue YOLO prevede innanzitutto l'acquisizione in input di un'immagine di dimensione fissa, solitamente a bassa risoluzione. Successivamente, l'immagine viene suddivisa in sotto-porzioni tramite una griglia di dimensione $S \times S$. Ogni cella della griglia è responsabile di rilevare gli oggetti il cui centro geometrico ricade al suo interno. Per ciascuna cella, vengono predetti:

- *Bounding box*, rappresentate dalle coordinate x e y e da altezza h e larghezza w del rettangolo.
- Confidenza, ossia un punteggio indice della sicurezza della predizione. In particolare, della probabilità che ci sia realmente un oggetto nella *box* e quanto sia accurata la *box* predetta rispetto all'oggetto reale, calcolando il rapporto tra l'area di sovrapposizione e l'area totale delle due *box*, ossia l'*Intersection over Union (IoU)*.

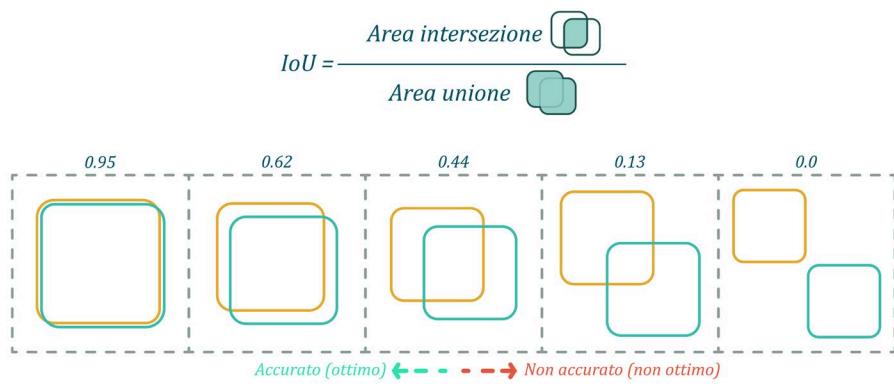


Figura 17- Intersection over Union

3. Il progetto EdgeCV4Safety

- Probabilità e classe, per ogni box viene indicata una classe di oggetti (come ad esempio persona, cane oppure difetto, graffio) a cui viene predetto appartenga il contenuto con la relativa probabilità.

L'output grezzo è composto da un numero n di box, riportate come tupla $(x, y, w, h, \text{confidenza})$ con relativa classe e probabilità. Tipicamente questo insieme viene ripulito, soprattutto perché con questo flusso possono crearsi *bounding box* sovrapposte per lo stesso oggetto. Ciò viene fatto tramite un algoritmo di pulizia, il *Non-Maximum Suppression* (NMS). Questo prevede un ciclo di fasi che si ripete finché non rimangono solamente le box migliori e non sovrapposte per ogni istanza di oggetto individuato, eliminando via via quelle con confidenza minore o che si sovrappongono significativamente. In questo modo vengono eliminate tutte le box che predicono lo stesso oggetto (IoU sovrapposto), mantenendo solo la migliore per ognuno.

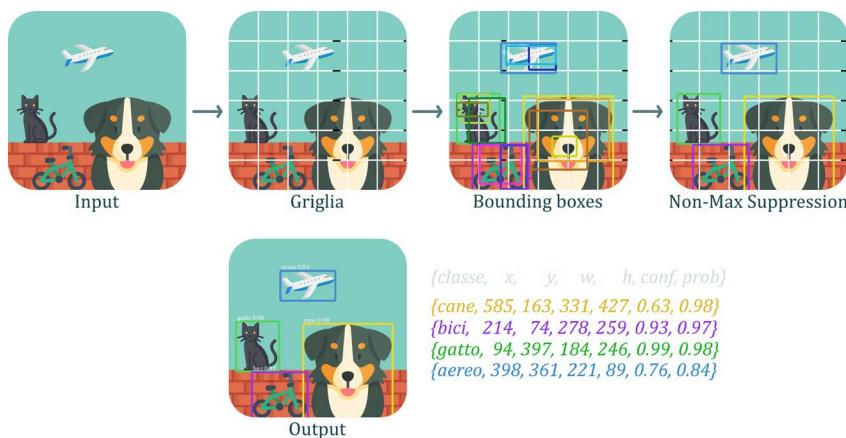


Figura 18 - Flusso di processamento di un'immagine in YOLO

Il risultato è un'immagine con rettangoli chiari, ognuno associato ad un'istanza di una classe e ad un punteggio di confidenza.

Nel corso degli anni, a partire dal 2015, sono state rilasciate diverse versioni di YOLO. Ciascuna nuova uscita apportava migliorie sotto aspetti particolari.

Le prime versioni si sono focalizzate principalmente sulla velocità a scapito dell'accuratezza, frangente su cui sono stati realizzati miglioramenti nelle versioni successive, grazie all'integrazione di meccanismi sofisticati che hanno perfezionato soprattutto il riconoscimento di oggetti di piccole dimensioni. Nel corso degli anni ci sono stati drastici progressi anche sotto l'aspetto

ingegneristico, integrando miglioramenti sulle tecniche di *deep learning*, che si sono tradotti in migliori *performance*. L'avanzamento più significativo è stato il passaggio da un *framework* di ricerca ad un ecosistema industriale: PyTorch. I vantaggi che questo ha implicato si sono ripercossi in particolare nella facilità d'uso, modifica ed implementazione e nell'integrazione su larga scala.

Le diverse versioni posseggono sotto-versioni che differiscono per le dimensioni. La scelta della versione comporta benefici che si spostano più verso la velocità o l'accuratezza, in base alle risorse hardware, ai vincoli progettuali e alle esigenze[22].

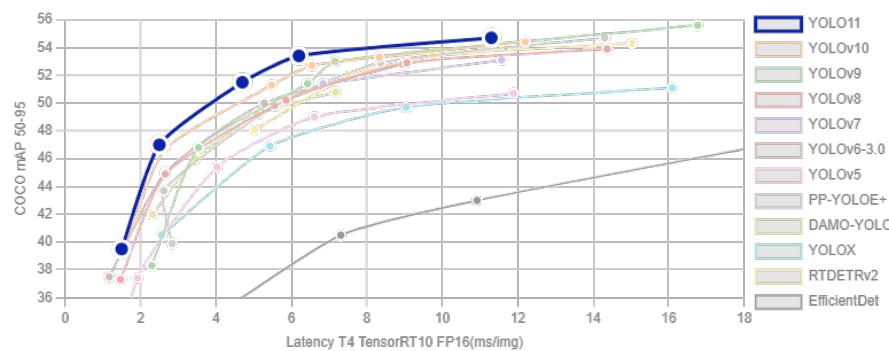


Figura 19 - Comparazione tempi di computazione di YOLO nelle versioni per ogni peso[23]

3.1.3 YOLO-3D - Depth Anything v2

YOLO-3D è un progetto che combina due modelli: YOLO, per il riconoscimento degli oggetti, e Depth Anything v2, per la *depth estimation* monoculare metrica (MMDE).

Il flusso di questo modello prevede:

- i. *Object detection* tramite YOLO11, in cui vengono individuati gli oggetti presenti nella scena con le rispettive *bounding box* 2D.
- ii. *Depth estimation* operata da Depth Anything v2, che genera una mappa di profondità.
- iii. *3D box estimation*, che combina i dati delle *box* 2D ottenute dall'output di YOLO11 per stimare le *box* 3D.
- iv. Visualizzazione, che prevede l'*output* con le *bounding box* 3D e una visione aerea per comprendere meglio come sono disposti gli oggetti lungo la profondità[24].

3. Il progetto EdgeCV4Safety

La problematica principale che affronta Depth Anything v2 è la granularità delle profondità individuate. Infatti, spesso gli oggetti (o parti di essi) troppo sottili o complessi non vengono rilevati correttamente, ma vengono esclusi o viene fatta confusione, sfociando in risultati del tutto imprecisi. Per arginare il problema e ottenere *performance* comunque elevate si ricorre alla *Knowledge Distillation*. Innanzitutto, si crea un modello *teacher* a partire dal modello di *computer vision* più potente esistente (attualmente DINOv2-Giant). Si addestra questo modello con immagini sintetiche, così da renderlo in grado di predire dettagli fini e gestire geometrie complesse, ma non abituato alle immagini reali.

In secondo luogo, il modello *teacher* analizza un *dataset* enorme di immagini reali e per ognuna predice una mappa di profondità e le corrispondenti etichette, fortemente accurate. Tutti questi dati vanno a creare un nuovo *dataset* a cui ad immagini reali corrispondono risultati estremamente precisi. Infine, il nuovo *dataset* viene impiegato per addestrare i modelli effettivi di Depth Anything v2, i quali assimilano robustezza, diversità e finezza caratterizzanti del *teacher*, rendendoli però più leggeri e veloci[25]. Si tratta infatti un modello semi-supervisionato.

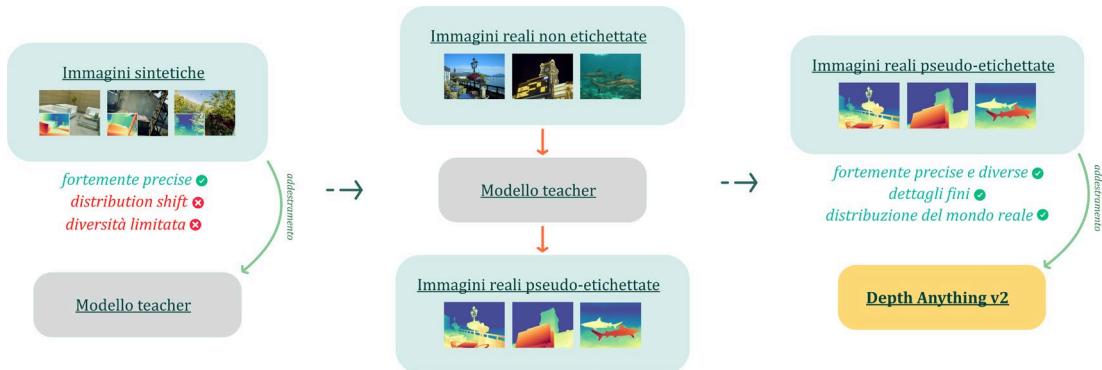


Figura 20 - Flusso di addestramento Depth Anything v2

Depth Anything V2 è l'evoluzione del modello V1 ed offre una vastità di modelli già addestrati sia per ambienti interni che esterni, sia in grado di calcolare distanze metriche che relative, ciascuno disponibile in numerosi formati di dimensioni, quindi con *performance* differenti[26].

La *pipeline* YOLO-3D, risultante dall'unione di YOLO con Depth Anything V2, appare estremamente adatta all'elaborazione richiesta per gli obiettivi di

questo lavoro. Dalle informazioni che risultano per ogni *frame* di un flusso video è quindi possibile triangolare la distanza (stimata) da un determinato punto noto ad un oggetto individuato.

3.1.4 UniDepth

UniDepth è un modello di *depth estimation* monoculare e metrico (MMDE), progettato per essere universale. L'obiettivo è creare un sistema in grado di ricostruire una scena 3D in unità metriche a partire da una singola immagine, senza alcuna informazione aggiuntiva (come i parametri intrinseci della fotocamera).

L'approccio che viene adottato da questo modello è innovativo sotto diversi aspetti.

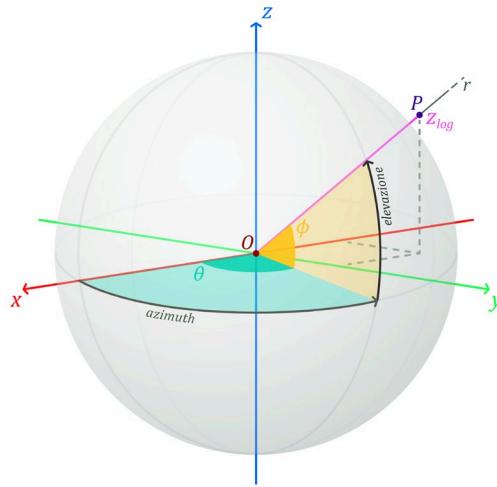


Figura 21 - Coordinate pseudo-sferiche e coordinate cartesiane

Innanzitutto, invece di predire direttamente le coordinate cartesiane (x, y, z) di un punto 3D, UniDepth calcola le sue coordinate in uno spazio pseudo-sferico (θ, ϕ, z_{log}) . θ e ϕ rappresentano due angoli che definiscono la direzione del raggio della fotocamera che parte dal centro ottico e passa per il punto 3D, mentre z_{log} rappresenta la distanza lungo quel raggio. Sostanzialmente gli angoli rappresentano rispettivamente θ l'*azimuth*, ossia l'angolo orizzontale, e ϕ l'*elevazione*, cioè l'angolo verticale. Il vantaggio che introduce questo approccio sta nel fatto che le caratteristiche della fotocamera rappresentate dai due angoli e la distanza sono disaccoppiate ed ortogonali, pertanto l'errore di

3. Il progetto EdgeCV4Safety

stima della profondità non influenza in alcun modo l'apprendimento degli angoli e viceversa. Al contrario, le tradizionali coordinate cartesiane sono fortemente dipendenti l'una dall'altra.

Il modello prevede anche un modulo adibito alla stima dei parametri della fotocamera. Questo prende in input le *feature* dall'immagine e predice una rappresentazione della fotocamera tramite la coppia di angoli (θ, ϕ) per ogni pixel dell'immagine. La predizione così generata viene usata come *prompt* per il modulo di profondità per la scala e la prospettiva della scena, alleggerendo il carico sul modulo della profondità e rendendo modulari i compiti di ciascuno.

Al fine di migliorare ulteriormente disaccoppiamento e robustezza, è stata introdotta anche la *Geometric Invariance Loss*, ossia, durante l'addestramento, per ogni immagine vengono create due viste differenti applicando trasformazioni geometriche (come zoom e traslazioni), simulando la stessa scena ripresa da due fotocamere diverse. La rete, quindi, processa entrambe le viste ottenute generando per ciascuna le caratteristiche di profondità interne, condizionate dalle stime delle rispettive fotocamere, le quali devono essere coerenti tra loro. L'obiettivo è insegnare al modello che la struttura 3D sottostante deve rimanere invariata indipendentemente dal punto di vista.

L'architettura è quindi composta da un *encoder*, realizzato da CNN o *Transformer*, che astrae le caratteristiche dall'immagine, un modulo fotocamera che riceve tali caratteristiche e predice la mappa dei raggi (θ, ϕ) e un modulo di profondità che riceve le caratteristiche e viene guidato dalla stima della fotocamera per predire la mappa di profondità z . L'output finale risulta dalla combinazione dei due moduli.

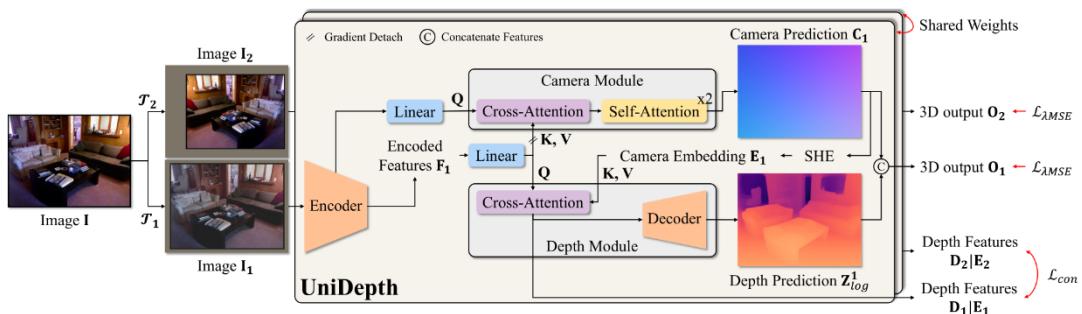


Figura 22 - Modello architettonale UniDepth[27]

Il modello viene addestrato con *dataset* enormi con scene eterogenee che spaziano dalla guida, ad ambienti interni. Questa diversità fa sì che l'apprendimento sia generalizzabile, quindi adatto a diversi contesti[27].

La struttura di UniDepth lo rende ideale ad essere accoppiato a YOLO per ottenere risultati di stima precisi.

3.1.5 RTDE

Nel panorama industriale esistono diversi protocolli e sistemi per la comunicazione tra macchine e computer. Alcuni sono standard e *open source*, altri privati e riservati a dispositivi proprietari. Tipicamente i vantaggi delle soluzioni proprietarie si traducono in maggiore ottimizzazione ed efficienza poiché sono costruite attorno e in sinergia con l'hardware specifico. Questo è il caso di RTDE (*Real-Time Data Exchange*), protocollo realizzato da Universal Robots per i propri robot.

RTDE è un'interfaccia di comunicazione ad alte prestazioni che consente uno scambio di dati bidirezionale e sincrono tra il controller di un robot, che agisce da server, e applicativi client residenti su macchine esterne.

Il fulcro del controllo in tempo reale risiede nella capacità di scambiare flussi di dati a frequenza elevata e costante. Solitamente tale frequenza è impostabile fino a 125 *Hz* o addirittura fino a 500 *Hz* per le ultimissime serie (come nei bracci che verranno usati nella trattazione), che si traduce in una trasmissione di pacchetti di dati ogni 2 *ms*. Questa bassa latenza garantisce reattività e movimenti fluidi.

All'interno del sistema del robot esistono dei registri virtuali che vengono utilizzati per scrivere o leggere dati. Questi possono rappresentare lo stato corrente del robot (variabili di stato lette dal client, definite come *output*), oppure lo stato che questo deve assumere (valori scritti dal client, definiti come *input*). I registri e le variabili di stato possono contenere tipi di dato differenti e possono essere accessibili solo in lettura o in lettura e scrittura, principalmente sulla base della loro funzione.

I pacchetti viaggiano simultaneamente in maniera sincrona tramite TCP/IP in entrambe le direzioni: da e verso il robot. La trasmissione è regolata dalla frequenza impostata per il protocollo. I pacchetti sono strutturati tramite un file di configurazione XML, in cui viene stabilito quali dati verranno scambiati specificatamente per ogni direzione. I file di configurazione, chiamati *recipe*, su client e server devono essere coerenti tra loro. Tipicamente, i pacchetti provenienti dal server riportano lo stato attuale del robot (per esempio posizione dei giunti e velocità), mentre quelli provenienti dal client rappresentano parametri desiderati che esso deve assumere, tramite l'indicazione dei valori da impostare nei rispettivi registri di input.

RTDE opera in collaborazione stretta con programmi specifici scritti in URScript (linguaggio proprietario per gli *script* dei robot di UR) caricati sul robot stesso e messi in esecuzione in maniera sequenziale. Ogni script agisce come un interprete a basso livello, leggendo i valori dai registri di *input* e passandoli a comandi di movimento in tempo reale. In questo modo, i movimenti del robot possono essere alterati dinamicamente tramite i pacchetti che vengono inviati dal client al server[28].

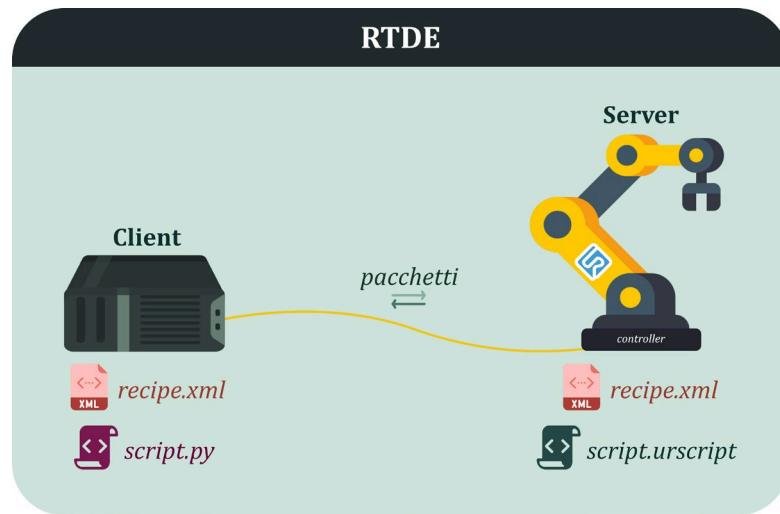


Figura 23 - Elementi di RTDE

La scelta di questo protocollo è dettata dai robot disponibili per i test. Alternative meno vincolanti si possono trovare nel protocollo Modbus e nella famiglia OPC.

3.1.7 PyTorch

Inizialmente creato come *wrapper* Python, PyTorch riutilizza gran parte del codice a basso livello della libreria Torch. È stato sviluppato e promosso principalmente da Facebook (Meta) ed è diventato rapidamente di ampio utilizzo, offrendo un'alternativa più dinamica a *framework* come Theano e TensorFlow. La sua caratteristica distintiva sono i grafi computazionali dinamici: il grafo delle operazioni viene costruito progressivamente durante l'esecuzione del codice. Questo approccio non solo semplifica notevolmente il debugging, ma si rivela estremamente flessibile. Oltre a questo, PyTorch ha da subito incluso il supporto per ONNX e per TorchScript, un *runtime* che permette di eseguire i modelli in ambienti ad alte prestazioni.

PyTorch è strutturato su tre livelli principali di astrazione: *Tensor*, *Variable* e *Module*. *Tensor* è l'unità fondamentale dei dati, un *array N*-dimensionale simile a quelli di NumPy, che può quindi essere eseguito su GPU accelerando drasticamente i calcoli. *Variable* è un nodo del grafo computazionale che incapsula un *Tensor* e ha la capacità di memorizzare i dati e il gradiente ad esso associato. Nelle ultime versioni *Variable* e *Tensor* sono state unificate, ma il concetto rimane il medesimo. Infine, *Module* rappresenta un *layer* di una rete neurale (o la rete intera) e ha il compito di contenere lo stato e i pesi apprendibili del modello, oltre alla logica di calcolo[29].

PyTorch è estremamente potente e adottato nell'ambito della *computer vision*. Questo successo deriva dalla capacità di eseguire *layer* convoluzionali in modo rapido su GPU con notevole precisione. Tant'è che i modelli impiegati nel progetto (YOLO, UniDepth v2, Depth Anything v2) si basano tutti sul *framework* PyTorch[30–32].

3.1.8 Open Neural Network Exchange (ONNX)

Open Neural Network Exchange (ONNX) nasce e si afferma come standard fondamentale grazie alla sempre più effettiva transizione verso il paradigma dell'*edge AI*, in particolare a causa delle significative sfide che emergono

3. Il progetto EdgeCV4Safety

nell'implementazione di modelli di IA, soprattutto di grandi dimensioni, in dispositivi con risorse limitate.

ONNX affronta il problema dell'eterogeneità degli ecosistemi di *machine learning*, consentendo ai modelli addestrati in un *framework* (come PyTorch o TensorFlow) di essere facilmente implementati su diverse piattaforme con uno sforzo ridotto.

ONNX agisce come una rappresentazione intermedia universale per i modelli di *machine learning*. La sua architettura si basa su due componenti principali: *Model Proto* e *Computational Graph*. *Model Proto* è una riproduzione serializzata dell'architettura del modello, mentre *Computational Graph* è sostanzialmente un grafo che descrive in dettaglio le operazioni e il flusso di dati del modello. Questo approccio strutturato garantisce che l'inferenza sia coerente su diversi ambienti. Il componente chiave dell'ecosistema è l'*ONNX Runtime* (ORT), un motore di inferenza ad alte prestazioni progettato per eseguire modelli ONNX in modo efficiente su un'ampia gamma di hardware. Grazie a un'interfaccia modulare chiamata *Execution Provider* (EP), ORT può sfruttare le librerie hardware specifiche per ottimizzare le prestazioni su diverse architetture. Ad esempio, per i dispositivi dotati di GPU NVIDIA, ORT supporta al meglio le loro caratteristiche tramite gli EP CUDA e TensorRT.

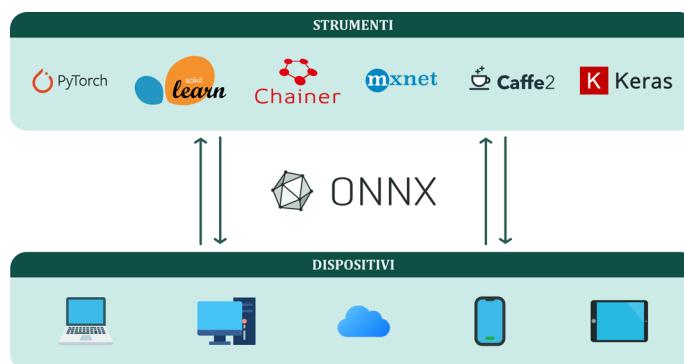


Figura 24 - ONNX come intermediario

Per gli scenari *mobile* ed *edge*, esiste una versione specializzata, ONNX Runtime Mobile, che offre dimensioni binarie ridotte e la possibilità di effettuare l'addestramento direttamente sul dispositivo, preservando la privacy.

Per adattare modelli di IA complessi a dispositivi con risorse limitate, sono necessarie tecniche di ottimizzazione aggressive, che migliorano significativamente le *performance* a scapito di accuratezza e precisione, talvolta anche in maniera pesante. Si può ricorrere alla compressione del modello, tramite tecniche quali la quantizzazione, il *pruning* o la *knowledge distillation*. Altre ottimizzazioni, che tende ad applicare l'ONNX *Runtime* stesso, si possono ottenere agendo a livello di grafo. Alcune di queste sono: *Constant Folding*, *Dead Code Elimination* e *Operator Fusion*. Il *Constant Folding* prevede il pre-calcolo di parti di grafo che dipendono da input costanti, in modo da ottimizzare l'esecuzione. La *Dead Code Elimination* rimuove le operazioni che non influenzano l'*output* finale del modello, eliminando *overhead* superflui. L'*Operator Fusion*, invece, unisce più operazioni sequenziali in un unico nucleo più efficiente, riducendo l'*overhead*. Ulteriori ottimizzazioni possono essere effettuate considerando le caratteristiche specifiche dell'hardware di destinazione (come NPU o DSP). Il *Neural Architecture Search* (NAS) è una tecnica che automatizza la progettazione di architetture di reti neurali, cercando configurazioni che bilanciano accuratezza con metriche specifiche dell'hardware come latenza e consumo di memoria.

ONNX si configura come una scelta strategica e potente per l'architettura proposta. Grazie al suo *runtime*, migliora le *performance* sfruttando acceleratori hardware specifici e si adatta dinamicamente agli ambienti di esecuzione tramite gli EP. Il suo vantaggio fondamentale è l'abbattimento delle dipendenze dai *framework* di origine, disaccoppiando la fase di addestramento da quella di deployment. YOLO, UniDepth e Depth Anything possono essere convertiti in formato ONNX facilmente dato che sono basate su PyTorch.

3.2 Implementazione

Per mantenere traccia dei cambiamenti, gestire gli sviluppi, conservare una versione consistente del progetto e corredarla della documentazione relativa è stata creata una *repository* GitHub, EdgeCV4Safety[33], resa pubblica al termine dello studio.

3. Il progetto EdgeCV4Safety

Tale *repository* funge da contenitore per due ulteriori *repository*, utilizzate come sotto-moduli (*git submodules*), nelle quali sono state pensate, progettate e realizzate le architetture generali EdgeCV4Safety-Vision[34] e EdgeCV4Safety-Controller[35].

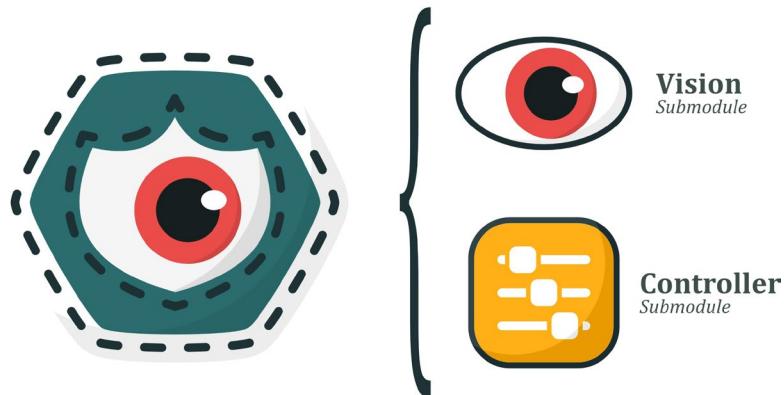


Figura 25 – Sotto-moduli EdgeCV4Safety

Ogni sotto-*repository* è dotata della rispettiva documentazione che ne illustra caratteristiche, funzionamento, installazione e configurazione.

La parte di Vision si occupa essenzialmente di impostare i parametri di configurazione, avviare lo *stream* video della telecamera ed eseguire *object detection* e *depth estimation*, inviando i risultati ottenuti al Controller.

Il sotto-modulo Controller invece monitora una o tutte le interfacce di rete in attesa gli pervengano gli esiti inviati dal sotto-modulo Vision. Una volta ricevuto un risultato, questo viene elaborato e vengono mandate le indicazioni al robot di interesse tramite la libreria del protocollo RTDE.

L'intera implementazione è stata realizzata in Python per le sue caratteristiche di rapida prototipazione, ampia adozione in campo IA e indipendenza dalla piattaforma. Sebbene questo linguaggio, di alto livello, possa introdurre *overhead* maggiori rispetto ad altri (quali C o C++), appoggiandosi all'ambiente ONNX e in generale su una molteplicità di librerie scritte in C++, quell'*overhead* sostanzialmente si annulla.

3.2.1 Implementazione Vision Submodule

Il sotto-modulo Vision implementa il sistema effettivo di *computer vision real time*. La *pipeline*, illustrata in *Figura 26*, prevede diversi passi, partendo dalla configurazione iniziale, passando per l'avvio e la ricezione dello *stream* video, proseguendo con *object detection* e *depth estimation* e terminando con l'invio dei risultati verso il nodo Controller scelto.

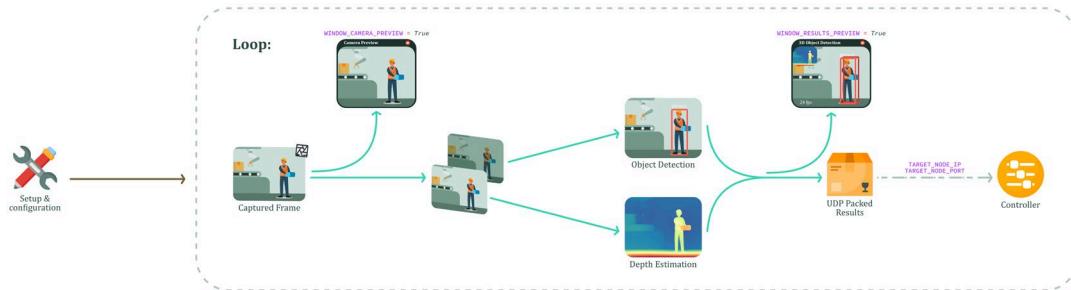


Figura 26 - Flusso di funzionamento sotto-modulo Vision

L'implementazione di questa *pipeline* fa uso delle tecnologie di cui finora discusso. La struttura si suddivide in diversi moduli. Il modulo principale è il file `run_cv.py` che contiene il ciclo di acquisizione, coordinazione dei modelli, eventuale visualizzazione e invio dei risultati. In questo file sono importati i moduli relativi alla CV. In particolare, è presente un modulo dedicato al modello di *object detection*, `detection_model.py`, due moduli intercambiabili per la *depth estimation*, `depth_model_depthanything.py` e `depth_model_unidepth.py`, infine `bbox3d_utils.py`, dalla *repository* di YOLO-3D[24], che contiene funzioni di utilità dedicate alla creazione delle *bounding box* 3D.

Lo script `run_cv.py` prevede una prima sezione in cui vengono dichiarate delle costanti di configurazione per i diversi aspetti (*Figura 27* e *Figura 28*).

3. Il progetto EdgeCV4Safety

```
40 # Models settings
41 DEPTH_MODEL_CHOICE = "unidepth" # Depth estimation model: "unidepth", "depthanything"
42 DEPTH_MODEL_SIZE = "small" * Depth model size: "small", "base", "large"
43 YOLO_MODEL_SIZE = "nano" # YOLOv1 model size: "nano", "small", "medium", "large",
    "extra"
44
45 # Detection settings
46 CONF_THRESHOLD = 0.5 # Confidence threshold for object detection
47 IOU_THRESHOLD = 0.4 # You threshold for NMS
48 CLASSES = [0] # Filter by class, e.g., [0, 1, 2] for specific CLASSES, None for disable
49
50 # Feature toggles
51 ENABLE_BEV = False # Enable Bird's Eye View visualization
52 ENABLE_PSEUDO_3D = True # Enable pseudo-3D calculation
53
```

Figura 27 - Parametri di configurazione computer vision (*run_cv.py*)

```
53 # Preview enabled/disabled
54 WINDOW_CAMERA_PREVIEW = False # Show camera preview window
55 WINDOW_RESULTS_PREVIEW = False # Show results window
56
57 # Camera settings
58 CAMERA_IP = '192.168.37.150' # None for aravis auto-choice (first found)
59 CAMERA_FRAME_RATE = 22
60 CAMERA_PIXEL_FORMAT = Aravis.PIXEL_FORMAT_BAYER_RG_8
61 CAMERA_GAIN = 30.0
62 CAMERA_AUTO_EXPOSURE = True
63 CAMERA_EXPOSURE_TIME = 8000
64 CAMERA_BUFFER_TIMEOUT = 200000
65 CAMERA_IMAGE_ROTATION_ANGLE = 0 # @ to disable
66 CAMERA_ROI_HEIGHT = 0 4 8 to disable
67 CAMERA_ROT_WIDTH = 6 ° @ to disable
68
69 # Camera infos (Set to 0 to use distances from camera)
70 CAMERA_HEIGHT_FROM_GROUND = 1.7
71 CAMERA_DISTANCE_FROM_FIXED_OBJECT = 2
72
73 # Output Target Node settings
74 TARGET_NODE_IP = '192.168.37.58'
75 TARGET_NODE_PORT = 13750
```

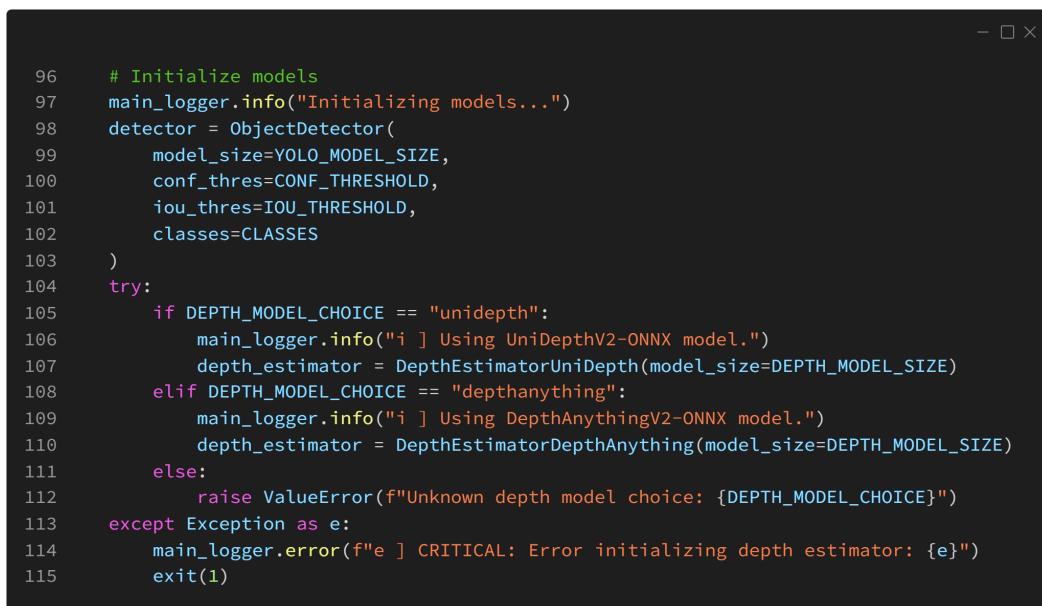
Figura 28 - Parametri di configurazione di comunicazione e stream (*run_cv.py*)

Il cuore di tutto il funzionamento si trova nel metodo `main`, in cui per prima cosa viene creata la *socket UDP* (Figura 29) per mezzo della quale potrà avvenire la comunicazione dei risultati al nodo di destinazione.

```
86 # Socket UDP for data sending to Target Node
87 try:
88     udp_client_socket = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
89     main_logger.info(f"i ] UDP Socket created to send data to {TARGET_NODE_IP}:
    {TARGET_NODE_PORT}")
90 except socket.error as e:
91     main_logger.info(f"e ] Error in creating UDP Socket: {e}")
92     udp_client_socket = None
```

Figura 29 - Creazione socket UDP (*run_cv.py*)

Il passo successivo prevede la creazione di opportune istanze di classi per i rispettivi compiti di *computer vision*. A tal proposito, nei file `depth_model_depthanything.py` e `depth_model_unidepth.py` sono definite due classi intercambiabili per la *depth estimation* con Depth Anything v2 e con UniDepth v2: `DepthEstimatorDepthAnything` e `DepthEstimatorUniDepth` rispettivamente. Affianco a queste esiste una classe analoga per l'*object detection* tramite YOLO11, la cui dichiarazione è collocata all'interno del file `detection_model.py`. Sulla base dei parametri definiti in testa al file, vengono quindi creati opportunamente gli oggetti corretti per ciascun compito che sono poi configurati internamente grazie al metodo di `init` (*Figura 30*).



```

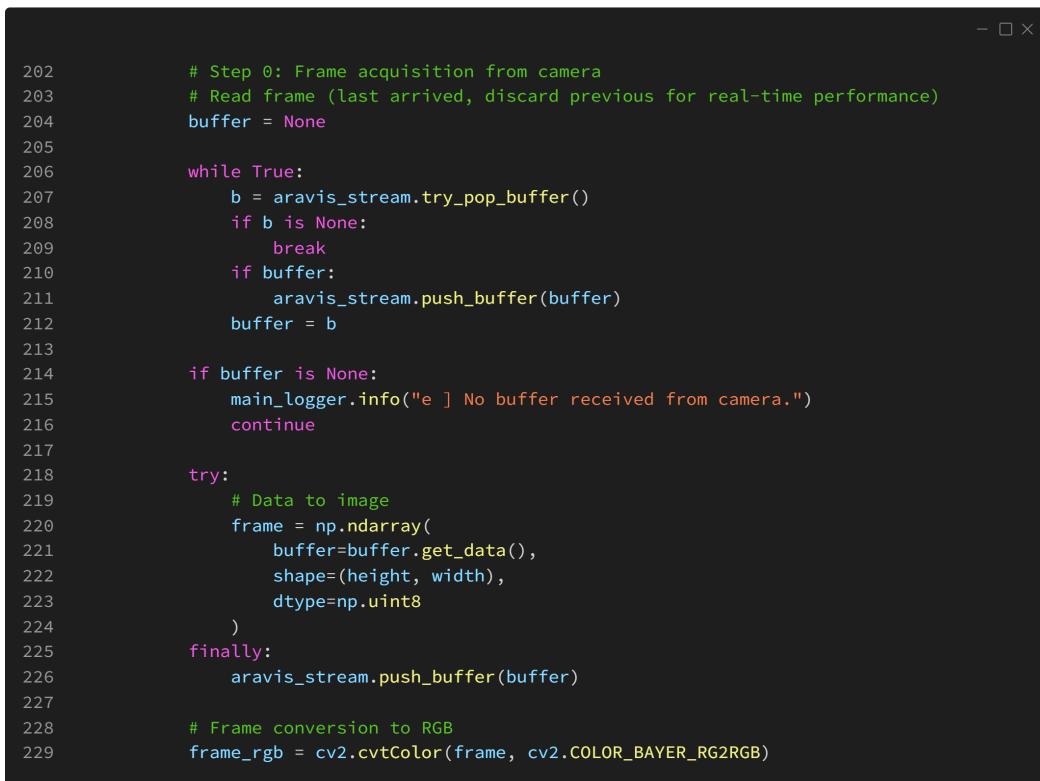
96     # Initialize models
97     main_logger.info("Initializing models...")
98     detector = ObjectDetector(
99         model_size=YOLO_MODEL_SIZE,
100        conf_thres=CONF_THRESHOLD,
101        iou_thres=IOU_THRESHOLD,
102        classes=CLASSES
103    )
104    try:
105        if DEPTH_MODEL_CHOICE == "unidepth":
106            main_logger.info("i ] Using UniDepthV2-ONNX model.")
107            depth_estimator = DepthEstimatorUniDepth(model_size=DEPTH_MODEL_SIZE)
108        elif DEPTH_MODEL_CHOICE == "depthanything":
109            main_logger.info("i ] Using DepthAnythingV2-ONNX model.")
110            depth_estimator = DepthEstimatorDepthAnything(model_size=DEPTH_MODEL_SIZE)
111        else:
112            raise ValueError(f"Unknown depth model choice: {DEPTH_MODEL_CHOICE}")
113    except Exception as e:
114        main_logger.error(f"e ] CRITICAL: Error initializing depth estimator: {e}")
115        exit(1)

```

Figura 30 - Creazione oggetti associati ai modelli di computer vision (run_cv.py)

In seguito, vengono create le finestre di visualizzazione in caso siano abilitate, creati gli oggetti per la visualizzazione della vista aerea e per il calcolo delle *bounding box* 3D, creati e impostati la telecamera e lo *stream* di Aravis. Poi si trova il *loop* di esecuzione del programma, che si ripete finché non ne è richiesta l'interruzione. In questo ciclo, inizialmente viene recuperato l'ultimo *frame* ricevuto da Aravis dalla telecamera industriale, scartando gli eventuali precedenti presenti nella coda, non lavorati, al fine di migliorare le prestazioni *real time*, evitando di elaborare immagini datate e non più coerenti con quello che sta succedendo nell'area di interesse. Dato che le immagini possono essere

acquisite dalla telecamera in formato di colore dei pixel diverso da quello RGB richiesto in *input* dai modelli di *computer vision*, i dati ricevuti dalla telecamera vengono opportunamente convertiti prima di essere elaborati. Questa conversione si rende necessaria soprattutto in ragione del fatto che alcuni formati di colore sono più leggeri di altri, quindi permettono di risparmiare banda e tipicamente anche di condividere un numero maggiore di *frame*. Ad esempio, il formato Bayer RG porta con sé l'informazione di un solo colore per pixel (invece dei tre di RGB), comportando un risparmio di banda di due terzi nel caso specifico. Per le fasi del processamento vengono anche create delle copie del *frame* originale.



```
202     # Step 0: Frame acquisition from camera
203     # Read frame (last arrived, discard previous for real-time performance)
204     buffer = None
205
206     while True:
207         b = aravis_stream.try_pop_buffer()
208         if b is None:
209             break
210         if buffer:
211             aravis_stream.push_buffer(buffer)
212         buffer = b
213
214         if buffer is None:
215             main_logger.info("e ] No buffer received from camera.")
216             continue
217
218         try:
219             # Data to image
220             frame = np.ndarray(
221                 buffer=buffer.get_data(),
222                 shape=(height, width),
223                 dtype=np.uint8
224             )
225         finally:
226             aravis_stream.push_buffer(buffer)
227
228         # Frame conversion to RGB
229         frame_rgb = cv2.cvtColor(frame, cv2.COLOR_BAYER_RG2RGB)
```

Figura 31 - Ricezione dati da telecamera Aravis e conversione in RGB (*run_cv.py*)

Segue la fase di *object detection* con il modello di YOLO precaricato all'istanziazione dell'oggetto **ObjectDetector**. Si ottiene quindi come risultato l'insieme di tutte le *bounding box* 2D rilevate, ciascuna corredata da classe, confidenza e coordinate.

3. Il progetto EdgeCV4Safety

```
239         # Step 1: Object Detection (YOLO-ONNX)
240         try:
241             detection_frame, detections = detector.detect(detection_frame)
242         except Exception as e:
243             main_logger.info(f"e ] Error during object detection: {e}")
244             detections = []
245             cv2.putText(detection_frame, "Detection Error", (10, 60),
246                         cv2.FONT_HERSHEY_SIMPLEX, 0.7, (0, 0, 255), 2)
```

Figura 32 - Object detection (*run_cv.py*)

Poi è il turno della *depth estimation* con il modello selezionato (*Figura 33*) nelle fasi precedenti, eseguita mediante l'oggetto `DepthEstimator*` istanziato. Si acquisisce come esito della computazione la mappa di profondità (*depth map*) del *frame* fornito in *input*, in cui ad ogni *pixel* è associata una distanza, da intendersi come segmento telecamera-punto della scena.

```
248         # Step 2: Depth Estimation (UnidepthV2-ONNX or DepthAnythingV2-ONNX)
249         try:
250             depth_map = depth_estimator.estimate_depth(original_frame)
251             depth_colored = depth_estimator.colorize_depth(depth_map)
252         except Exception as e:
253             main_logger.info(f"Error during depth estimation: {e}")
254             # Create a dummy depth map
255             depth_map = np.zeros((height, width), dtype=np.float32)
256             depth_colored = np.zeros((height, width, 3), dtype=np.uint8)
257             cv2.putText(depth_colored, "Depth Error", (10, 60),
258                         cv2.FONT_HERSHEY_SIMPLEX, 0.7, (0, 0, 255), 2)
```

Figura 33 - Depth estimation (*run_cv.py*)

Dopo la *depth estimation*, si passa alla stima delle profondità per ogni oggetto (*Figura 34*), che avviene in maniera differente in base alla classe. Per persone e animali, si è ritenuto corretto e sensato sfruttare unicamente il valore del punto centrale, mentre per tutte le altre classi si esegue una media nella regione della *bounding box*, irrilevante per il nostro caso di studio, dato che verrà filtrata unicamente la classe `person`. In questa fase vengono anche stimate le *bounding box* 3D (*Figura 34*).

3. Il progetto EdgeCV4Safety

```
260      # Step 3: 3D Bounding Box Estimation
261      boxes_3d = []
262      active_ids = []
263
264      for detection in detections:
265          try:
266              bbox, score, class_id, obj_id = detection
267
268              # Get class name
269              class_name = detector.get_class_names()[class_id]
270
271              # Get depth in the region of the bounding box
272              # Try different methods for depth estimation
273              if class_name.lower() in ['person', 'cat', 'dog']:
274                  # For people and animals, use the center point depth
275                  center_x = int((bbox[0] + bbox[2]) / 2)
276                  center_y = int((bbox[1] + bbox[3]) / 2)
277                  depth_value = depth_estimator.get_depth_at_point(depth_map, center_x, center_y)
278                  depth_method = 'center'
279              else:
280                  # For other objects, use the median depth in the region
281                  depth_value = depth_estimator.get_depth_in_region(depth_map, bbox, method='median')
282                  depth_method = 'median'
283
```

Figura 34 - Calcolo della profondità per ogni classe (run_cv.py)

```
284      # Create a simplified 3D box representation
285      box_3d = {
286          'bbox_2d': bbox,
287          'depth_value': depth_value,
288          'depth_method': depth_method,
289          'class_name': class_name,
290          'object_id': obj_id,
291          'score': score
292      }
293      boxes_3d.append(box_3d)
294      # Keep track of active IDs for tracker cleanup
295      if obj_id is not None:
296          active_ids.append(obj_id)
297      except Exception as e:
298          main_logger.info(f"Error processing detection: {e}")
299          continue
300
```

Figura 35 - Calcolo bounding box 3D (run_cv.py)

Si procede con operazioni legate alla visualizzazione dei risultati (se abilitata) e alla selezione delle distanze. Per lo sviluppo dell'architettura illustrata, si è optato di preferire unicamente la distanza minore individuata, poiché il nuovo comportamento prevede la modifica della velocità in base alla distanza delle persone rilevante. Per questo motivo il comportamento è dipendente unicamente dalla più vicina rilevata. Tale distanza viene prima manipolata, impiegando il teorema di Pitagora per triangolare la lontananza da un punto, posto davanti alla telecamera, parallelamente al terreno. Per fare ciò, si scorrono i risultati e al contempo si aggiungono, se la visualizzazione è abilitata, le box 3D calcolate in precedenza al *frame* finale.

3. Il progetto EdgeCV4Safety

```
303     # Step 4: Visualization & Distance selection
304     # Draw boxes on the result frame && data send
305     min_depth_value = float('inf')
306     for box_3d in boxes_3d:
307         try:
308             act_distance = abs((math.sqrt(pow(box_3d['depth_value'], 2) -
309                                     pow(CAMERA_HEIGHT_FROM_GROUND, 2))) -
310                                     CAMERA_DISTANCE_FROM_FIXED_OBJECT) # distance
311             main_logger.info(f"r ] Detected {box_3d['class_name']} ({box_3d['score']:.2f}) at
312                                     depth {act_distance:.2f} m.")
313             if act_distance < min_depth_value:
314                 min_depth_value = act_distance
315             if WINDOW_RESULTS_PREVIEW:
316                 # Determine color based on class
317                 class_name = box_3d['class_name'].lower()
318                 ...
319
320                 # Draw box with depth information
321                 result_frame = bbox3d_estimator.draw_box_3d(result_frame, box_3d, color=color)
322             except Exception as e:
323                 main_logger.info(f"Error drawing box: {e}")
324             continue
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
```

Figura 36 - Calcoli per la visualizzazione e selezione della distanza triangolata (*run_cv.py*)

L'ultimo passo rilevante prevede l'invio dei dati selezionati via UDP al nodo Controller, a patto che la *socket* creata all'inizio sia disponibile (Figura 36).

```
334     # Step 5: Send data via UDP if socket is available
335     if udp_client_socket:
336         try:
337             packed_distance = struct.pack('<f', float(min_depth_value))
338             udp_client_socket.sendto(packed_distance, (TARGET_NODE_IP, TARGET_NODE_PORT))
339             main_logger.info(f"Minimum distance '{min_depth_value:.2f}' m send via UDP to
340                               {TARGET_NODE_IP}:{TARGET_NODE_PORT}")
341         except socket.error as e:
342             main_logger.info(f"e ] Error while sending UDP packet: {e}")
343         else:
344             main_logger.info("e ] UDP Socket not initialized. Impossible to send data.")
```

Figura 37 - Invio dati al nodo Controller tramite socket UDP (*run_cv.py*)

La funzione `main` termina con le ultime operazioni relative alla visualizzazione (se abilitata), le quali prevedono l'aggiornamento di una o più finestre con i nuovi dati, e al calcolo del *frame rate*. L'output che viene visualizzato nella finestra dei risultati è riportato in Figura 38 dove sono presenti in basso il *frame rate* e in alto a sinistra la mappa di profondità, oltre alle possibili *bounding box* 3D individuanti le persone rilevate.

L'esecuzione termina con la chiamata alla funzione `main`, che avvia la *pipeline* discussa.

3. Il progetto EdgeCV4Safety



Figura 38 - Output finestra dei risultati (bounding box + depth estimation).
A sinistra l'output di Depth Anything v2 small, a destra quello di UniDepth v2 small.

Gli altri moduli del comparto Vision, ad eccezione di `bbox3d_utils`, incorporano la logica intera dei singoli modelli ONNX contenuti nella cartella `models/`. In particolare, i tre moduli mappano alle diverse dimensioni il percorso del modello specifico e carico il *provider* ONNX migliore disponibile per il proprio compito. Nell'ordine il primo è TensorRT (GPU Nvidia e Jetson), poi CUDA (GPU Nvidia), DirectML (Windows) e infine CPU. I primi eseguono su GPU e con ottimizzazioni via via meno pesanti, per arrivare all'ultimo livello di *fallback*, ossia la CPU.

```
38      # Logic to automatically select the best available provider for object detection
39      available_providers = ort.get_available_providers()
40      provider_options = None
41
42      if 'TensorrtExecutionProvider' in available_providers:
43          yolo_logger.info("Using TensorRT Execution Provider.")
44          provider = 'TensorrtExecutionProvider'
45          cache_path = os.path.join(os.path.dirname(__file__), "trt_cache_yolo")
46          if not os.path.exists(cache_path):
47              os.makedirs(cache_path)
48          provider_options = [{
49              'trt_engine_cache_enable': True,
50              'trt_engine_cache_path': cache_path,
51          }]
52      elif 'CUDAExecutionProvider' in available_providers:
53          yolo_logger.info("Using CUDA Execution Provider.")
54          provider = 'CUDAExecutionProvider'
55      elif 'DmlExecutionProvider' in available_providers:
56          yolo_logger.info("Using DirectML Execution Provider (for Windows AMD/Intel GPU).")
57          provider = 'DmlExecutionProvider'
58      else:
59          yolo_logger.info("No specialized GPU provider found. Using CPU Execution Provider.")
60          provider = 'CPUExecutionProvider'
```

Figura 39 - Scelta del miglior provider per l'esecuzione (`detection_model.py`).
Nota: per i `depth_model_*.py` valgono implementazioni del tutto analoghe.

Tutti e tre i moduli poi condividono diversi metodi che ciascuno implementa in maniera dedicata. Questi prevedono l'inizializzazione, il pre- e post-

3. Il progetto EdgeCV4Safety

processamento, il processamento stesso (`detect` ed `estimate_depth`) e metodi specifici di utilità, come quelli per colorare la mappa di profondità o per il calcolo della distanza per una *bounding box*. Le funzioni di ciascun modulo sono riportate in *Figura 40*, *Figura 41* e *Figura 42*.

```
15 class ObjectDetector:  
16     """  
17         Object detection using a YOLOv11 ONNX model, optimized with the best  
18         available ONNX Runtime Execution Provider for the current hardware.  
19     """  
20     def __init__(self, model_size='small', conf_thres=0.25, iou_thres=0.45, classes=None, provider=None): ...  
21     # Load COCO class names  
22     def _load_coco_classes(self): ...  
23     # Preprocess the input image  
24     def _preprocess(self, image): ...  
25     # Postprocess the model output  
26     def _postprocess(self, output, ratio, pad_y, pad_x): ...  
27     # Main detection method  
28     def detect(self, image): ...  
29     def get_class_names(self): ...
```

Figura 40 - Struttura classe ObjectDetector (detection_model.py)

```
38 class DepthEstimatorUniDepth:  
39     """  
40         Depth estimation using UniDepth V2 ONNX model, optimized with the best  
41         available ONNX Runtime Execution Provider.  
42     """  
43     def __init__(self, model_size='small', provider=None): ...  
44     # Create default camera intrinsics  
45     def _create_default_intrinsics(self, h, w): ...  
46     # Preprocess the input image  
47     def _preprocess(self, image): ...  
48     # Estimate depth for the input image  
49     def estimate_depth(self, image): ...  
50     # Colorize the depth map for visualization  
51     def colorize_depth(self, depth_map, cmap=cv2.COLORMAP_INFERNO): ...  
52     # Get depth value at a specific point  
53     def get_depth_at_point(self, depth_map, x, y): ...  
54     # Get depth statistics in a bounding box region  
55     def get_depth_in_region(self, depth_map, bbox, method='median'): ...
```

Figura 41 - Struttura classe DepthEstimatorUniDepthONNX (depth_model_unidepth.py)

```

15  class DepthEstimatorDepthAnything:
16      """
17          Depth estimation using Depth Anything v2 ONNX model, optimized with the best
18          available ONNX Runtime Execution Provider for the current hardware.
19      """
20  >     def __init__(self, model_size='small', provider=None): ...
21
22      # Preprocess the input image
23  >     def _preprocess(self, image): ...
24
25      # Estimate depth for the input image
26  >     def estimate_depth(self, image): ...
27
28      # Colorize the depth map for visualization
29  >     def colorize_depth(self, depth_map, cmap=cv2.COLORMAP_INFERNO): ...
30
31      # Get depth value at a specific point
32  >     def get_depth_at_point(self, depth_map, x, y): ...
33
34      # Get depth statistics in a bounding box region
35  >     def get_depth_in_region(self, depth_map, bbox, method='median'): ...
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116

```

Figura 42- Struttura classe DepthEstimatorDepth AnythingONNX (depth_model_depthanything.py)

Il *submodule* Vision richiede l'installazione dell'ONNX Runtime più adeguato, di Aravis 0.10 e Python almeno 3.8. Sono necessarie poi le librerie Python `filterpy`, `numpy`, `opencv_contrib_python`, `PyGObject` e `scipy`.

3.2.2 Implementazione Controller Submodule

Il sotto-modulo Controller rappresenta il fulcro di applicazione delle direttive del sistema ai robot. Il suo funzionamento si basa principalmente sulla libreria RTDE di Universal Robots[36], tuttavia questa potrebbe essere tranquillamente sostituita o affiancata ad altri protocolli di comunicazione industriale quali Modbus o la famiglia OPC. Il Controller riceve in input i pacchetti inviati dal sotto-modulo Vision, i quali contengono i dati di distanza. Tali dati sono analizzati per la scelta del nuovo comportamento da inviare ad un robot tramite la libreria RTDE.

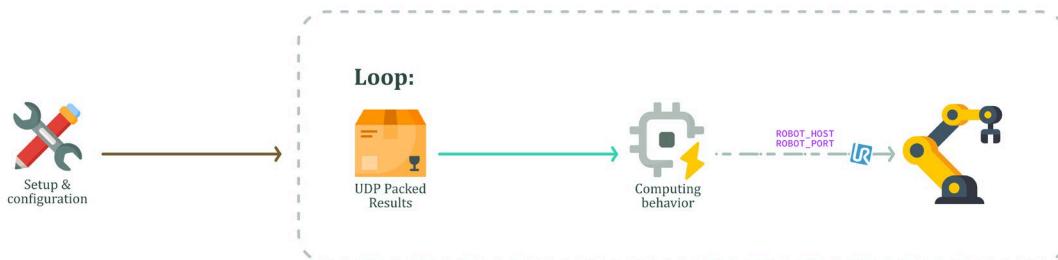


Figura 43 - Flusso di funzionamento sotto-modulo Controller

La struttura di questo sotto-modulo prevede uno *script* principale che orchestra ogni cosa, sfruttando la parallelizzazione in più processi e *thread*. Lo *script* principale è rappresentato dal file `SpeedControllerUDP.py` o da `SlowedSpeedControllerUDP.py`, i quali eseguono i medesimi ruoli ma differiscono per le modalità e la tempestività con cui riflettono sul robot il comportamento calcolato dai risultati ricevuti. Inizialmente è prevista una serie di costanti in cui inserire indirizzo IP del robot, porta di RTDE, percorso al file di configurazione della comunicazione RTDE e frequenza di comunicazione con il robot (*Figura 44*). Possono essere poi stabilite delle fasce di distanza a cui associare dei rispettivi valori di velocità (*Figura 45*). Tali fasce vengono poi usate in una funzione apposita per il calcolo della corrispondente (*Figura 46*).

```

22 ROBOT_HOST = "10.4.1.87" # Must correspond to the robot IP
23 ROBOT_PORT = 30004 # Must correspond to the robot RTDE port (default 30004)
24 CONFIG_XML = './recipe.xml' # Make sure this is the correct path to your recipe.xml
25 RTDE_FREQUENCY = 30 # Hz

```

*Figura 44 - Costanti di configurazione RTDE (*SpeedControllerUDP.py)*

```

27 # --- Speed thresholds and params ---
28 VELOCITY_ZONE_1      = 0.0 # % [0,1]
29 VELOCITY_ZONE_2      = 0.0 # % [0,1]
30 VELOCITY_ZONE_3      = 1.0 # % [0,1]
31 VELOCITY_ZONE_4      = 1.0 # % [0,1]
32 VELOCITY_ZONE_5      = 1.0 # % [0,1]
33
34 ZONE_1_END_DISTANCE = 1.5 # m
35 ZONE_2_END_DISTANCE = 2.1 # m
36 ZONE_3_END_DISTANCE = 2.6 # m
37 ZONE_4_END_DISTANCE = 3.5 # m

```

*Figura 45 - Fasce di velocità (*SpeedControllerUDP.py)*

```

48 def calculate_speed_fraction(distance: float) -> float:
49     """
50     Calculate the speed fraction based on the distance.
51     """
52     if distance < 0:
53         return VELOCITY_ZONE_5
54     elif distance < ZONE_1_END_DISTANCE:
55         return VELOCITY_ZONE_1
56     elif ZONE_1_END_DISTANCE <= distance < ZONE_2_END_DISTANCE:
57         return VELOCITY_ZONE_2
58     elif ZONE_2_END_DISTANCE <= distance < ZONE_3_END_DISTANCE:
59         return VELOCITY_ZONE_3
60     elif ZONE_3_END_DISTANCE <= distance < ZONE_4_END_DISTANCE:
61         return VELOCITY_ZONE_4
62     else:
63         return VELOCITY_ZONE_5

```

*Figura 46 - Funzione per il calcolo della velocità (*SpeedControllerUDP.py)*

Nel `main` è contenuta la logica di avvio e gestione del sistema. All'inizio viene avviato un processo figlio deputato alla gestione di tutta la parte di comunicazione UDP con il nodo Vision. Per farlo viene invocato un altro *script*, `udp_listener.py`, in cui è presente l'implementazione del comportamento desiderato. La comunicazione tra il processo principale e questo processo figlio avviene tramite *pipe*, meccanismo di comunicazione unidirezionale con politica FIFO, che cattura tutto quello che il figlio carica su standard *output*. La lettura avviene riga per riga (`bufsize=1`) non appena disponibile.

```
212     script_dir = os.path.dirname(os.path.abspath(__file__))
213     udp_script_path = os.path.join(script_dir, "udp_listener.py")
219     udp_process = subprocess.Popen(
220         [sys.executable, udp_script_path],
221         stdout=subprocess.PIPE,
222         stderr=subprocess.PIPE,
223         text=True,
224         bufsize=1, # Changed from 0 to 1 for line buffering
225         preexec_fn=os.setsid
226     )
```

Figura 47 - Avvio processo UDP `udp_listener.py` nel thread principale (*`SpeedControllerUDP.py`)

Gestire questa logica di comunicazione separatamente permette di ottimizzare le tempistiche, evitando ritardi dovuti all'esecuzione di altri compiti. La scelta di lanciare questa parte come processo separato conferisce robustezza, tolleranza ai guasti, modularità ed indipendenza. Il vantaggio maggiore in realtà giunge dal fatto che Python permette l'esecuzione del *bytecode* ad un solo *thread* per volta in un singolo processo, in questo modo viene meno il parallelismo. Avendo, invece, due processi distinti si ha un vero parallelismo, in quanto ognuno possiede il proprio interprete e la propria memoria, quindi ciascuno può essere eseguito da un diverso *core* della CPU.

Poi viene avviato un *thread* dedicato alla comunicazione RTDE con il robot (*Figura 48*), che quindi entra in concorrenza con il *thread* che esegue il `main` all'interno del processo principale. Al *thread* RTDE viene passato uno `stop_event`, oggetto che servirà per segnalargli quando terminare. Questi compiti di comunicazione sono affidati ad un *thread* dedicato del processo principale per isolare le operazioni lente e bloccanti della rete in modo che

queste non influiscano sulla ricezione che il *thread* principale opera dal processo UDP.

```
208     stop_event = threading.Event()
209     rtde_thread = threading.Thread(target=run_rtde_controller, args=(stop_event,), daemon=True)
210     rtde_thread.start()
```

Figura 48 - Avvio thread RTDE nel thread principale (*SpeedControllerUDP.py)

Nel **main** segue il ciclo principale, in cui il *thread* attende eventi su tre canali differenti adottando una gestione di I/O multipli efficiente senza bloccare l'esecuzione, tramite **select**. Se questa rileva una nuova riga proveniente dal processo UDP, il *thread* principale la legge e se inizia per “**DISTANCE:**” ne estrae il valore numerico inserendolo in una coda *thread-safe*, accessibile quindi al *thread* RTDE.

```
245     try:
246         while not stop_event.is_set():
247             ready_fds = poller.poll(100) # Timeout increased to 100ms to reduce aggressive polling
248
249             for fd, event in ready_fds:
250                 if fd == udp_process.stdout.fileno():
251                     line = udp_process.stdout.readline()
252                     if line:
253                         if line.startswith("DISTANCE:"):
254                             try:
255                                 received_distance = float(line.strip().split(':')[1])
256                                 distance_queue.put(received_distance)
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
```

Figura 49 - Gestione canali tramite select e distanze nel thread principale (*SpeedControllerUDP.py)

Viene monitorato anche l'input da tastiera per controllare che venga inserito “**q**” (Figura 50). Nel caso venga rilevato il processo termina e di conseguenza vengono arrestati i due *thread* e il processo.

```
271             elif fd == sys.stdin.fileno():
272                 line = sys.stdin.readline().strip()
273                 if line == 'q':
274                     logging.info("User requested interruption. Closing...")
275                     stop_event.set()
```

Figura 50 - Gestione terminazione programma del thread principale (*SpeedControllerUDP.py)

In questo caso di terminazione da parte dell'utente, mediante il rilevamento della pressione del tasto “**q**”, il blocco **finally** assicura la chiusura ordinata delle risorse e dei processi. Viene, infatti, inviato il segnale di terminazione al processo UDP, che viene ucciso (*kill*) in caso non venga ricevuta risposta in tempi utili. Infine, si attende che il *thread* RTDE concluda il suo ciclo, disconnettendosi dal robot e terminando in maniera pulita.

3. Il progetto EdgeCV4Safety

```
291     finally:
292         logging.info("Waiting for threads and processes to terminate...")
293         time.sleep(0.1) # Brief pause to avoid minor race conditions
294
295         # Handle UDP subprocess termination
296         if udp_process and udp_process.poll() is None:
297             logging.info("Sending termination signal to UDP subprocess...")
298             udp_process.terminate() # Send SIGTERM
299             try:
300                 udp_process.wait(timeout=2) # Wait for graceful termination
301             except subprocess.TimeoutExpired:
302                 logging.info("UDP subprocess did not terminate gracefully, killing it.")
303                 udp_process.kill() # Send SIGKILL
304
305         # Wait for RTDE thread termination
306         rtde_thread.join(timeout=5)
307         if rtde_thread.is_alive():
308             logging.info("RTDE thread did not terminate gracefully within the timeout.")
```

Figura 51 - Blocco *finally* conclusivo del thread principale (*SpeedControllerUDP.py)

Il *thread* RTDE è costituito principalmente da cicli innestati. Nel primo ciclo interno viene tentata la connessione al robot in maniera continua (*Figura 52*). Questo fa sì che in caso di disconnessioni, la connessione venga ripristinata il prima possibile e che, in caso questo si protraggia, si continui a tentare, segnalando al contempo le anomalie.

```
74     # The outer loop handles RTDE reconnection attempts
75     while not stop_event.is_set():
76         try:
77             logging.info("[RTDE_TX] Attempting to connect to UR robot...")
78             con = rtde.RTDE(ROBOT_HOST, ROBOT_PORT)
79
80             # --- RTDE CONNECTION LOOP ---
81             retries = 0
82             MAX_RETRIES = 5 # Increased retries to avoid immediate failure
83             while not con.is_connected() and not stop_event.is_set():
84                 try:
85                     except Exception as e:
86                         logging.info(f"[RTDE_TX] .info error in RTDE thread: {e}. Retrying full sequence in 5 seconds.", exc_info=True)
87                         time.sleep(5) # Pause before retrying the connection after a generic error
88                 finally:
89                     # Ensure a clean disconnection in any case
90                     if con and con.is_connected():
91                         try:
92                             logging.info("[RTDE_TX] Sending send_pause before disconnection.")
93                             con.send_pause() # Important to release controls
94                             logging.info("[RTDE_TX] RTDE disconnection in progress.")
95                             con.disconnect()
96                             logging.info("[RTDE_TX] Connessione RTDE disconnessa con successo.")
```

Figura 52 – Ciclo di riconnessione del thread RTDE (*SpeedControllerUDP.py)

Successivamente è presente un altro ciclo (*Figura 53*) interno al ciclo principale, il quale ad ogni iterazione riceve lo stato del robot, svuota la coda delle distanze per ottenere l'ultimo valore disponibile, evitando così di reagire a dati vecchi (in caso il thread fosse più lento del produttore di dati), calcola la velocità tramite l'apposita funzione (*Figura 46*) e invia al robot il comando solo se la velocità è cambiata rispetto all'ultimo valore inviatogli. La funzione per il calcolo della velocità prende in *input* una distanza e restituisce un valore tra 0.0

e 1.0 che rappresenta la frazione della velocità massima consentita da impostare sul robot. Basandosi sulla velocità precedentemente inviata è possibile non tenere traccia dell'ultima distanza ricevuta ed evitare di inviare costantemente valori finché tale parametro non debba realmente cambiare.

```

140     # --- MAIN RTDE COMMUNICATION LOOP ---
141     while not stop_event.is_set() and con.is_connected():
142         state = con.receive() # Receive a state packet from the robot
143         if state:
144             try:
145                 while not distance_queue.empty(): # Empty the queue to get only the latest value
146                     current_distance = distance_queue.get_nowait()
147             > except Empty:
148                 new_speed_fraction = calculate_speed_fraction(current_distance)
149                 if input_data and hasattr(input_data, 'speed_slider_fraction') and \
150                     new_speed_fraction != previous_speed_fraction:
151
152                     input_data.speed_slider_fraction = new_speed_fraction
153                     con.send(input_data) # <--- SEND THE SPEED SLIDER
154                     previous_speed_fraction = new_speed_fraction # Update the value for the next comparison
155

```

Figura 53 - Logica di invio dati al robot del thread RTDE (*SpeedControllerUDP.py)

Il processo UDP viene lanciato in modo che questo esegua il contenuto del file `udp_listener.py`. Questo innanzitutto prevede la creazione dell'oggetto di comunicazione, ossia una *socket* UDP, coerentemente con il *submodule* Vision. Dopo la creazione dell'oggetto viene impostata l'opzione di riuso di indirizzo e porta associati alla *socket*, in modo da poterli recuperare in caso di riavvio. Viene anche aumentata la dimensione del *buffer* di ricezione per evitare che il sistema operativo scarti i pacchetti se questi arrivano in raffiche veloci e lo *script* non le legge istantaneamente.

```

42     logging.info("Creating UDP socket...")
43     sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
44     sock.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
45
46     sock.setsockopt(socket.SOL_SOCKET, socket.SO_RCVBUF, 1024 * 1024) # 1 MB

```

Figura 54 - Creazione ed impostazione socket UDP nel processo UDP (`udp_listener.py`)

Successivamente avviene il *bind* della *socket* ad indirizzo e porta specificate in testa al file stesso (Figura 55). In seguito, il processo entra nel ciclo principale dello *script*, in cui innanzitutto si mette in ascolto con la *socket* in maniera bloccante.

```

28     LISTEN_IP = '192.168.37.50' # TARGET_NODE_IP or 0.0.0.0 to listen on all interfaces
29     LISTEN_PORT = 13750 # TARGET_NODE_PORT

```

Figura 55 - Parametri di connessione per la socket (`udp_listener.py`)

3. Il progetto EdgeCV4Safety

```
61     try:
62         sock.bind((LISTEN_IP, LISTEN_PORT))
63         logging.info(f"UDP listener listening on {LISTEN_IP}:{LISTEN_PORT}. Waiting for packets...")
64     except socket.error as e:
65         # Critical error. If bind fails, the program cannot function.
66         logging.info(f"Bind ERROR: Unable to bind to {LISTEN_IP}:{LISTEN_PORT}: {e}")
67         sys.exit(1) # Exit with an error code that the parent can detect
68
69     # Main reception loop
70     while True: # The loop breaks when the parent process terminates it.
71         try:
72             data, addr = sock.recvfrom(1024) # Maximum buffer size
```

Figura 56 - Inizio ciclo principale processo UDP (*udp_listener.py*)

Una volta sbloccato dalla chiamata, quindi ricevuto un pacchetto, il processo provvede a controllare che questo abbia esattamente la dimensione di un dato *float*, scartando automaticamente pacchetti errati e successivamente ne estrae il contenuto convertendolo in *float*. Estratto il dato provvede a scriverlo su standard *output* posponendo la stringa “**DISTANCE:**”, come si aspetta il processo principale. Per mandare il dato istantaneamente viene forzato lo svuotamento immediato del *buffer*, anche perché il sistema operativo potrebbe attendere di raccogliere più dati prima di inviarli.

```
75         if len(data) == struct.calcsize('<f'):
76             # Decompress the bytes into a float value (little-endian)
77             received_distance = struct.unpack('<f', data)[0]
78
79             sys.stdout.write(f"Distance:{received_distance}\n")
80             sys.stdout.flush() # Essential to ensure immediate sending
```

Figura 57 - Spacchettamento del messaggio e scrittura su *stdout* (*udp_listener.py*)

Infine, al termine del programma, viene chiusa correttamente la *socket*, rilasciando le risorse allocate.

```
95     finally:
96         # Resource cleanup
97         if sock:
98             sock.close()
```

Figura 58 - Pulizia delle risorse (*udp_listener.py*)

L’implementazione illustrata è condivisa dai due suddetti file ***SpeedControllerUDP.py**, la differenza risiede nel fatto che **SpeedControllerUDP.py** apporta istantaneamente i cambiamenti di velocità robot (coerentemente con quanto detto), mentre la sua versione **Slowed** lo fa in maniera rallentata, filtrando eventuali distanze spurie o frutto di errori di previsione. Ciò avviene attendendo di ricevere un numero sufficiente di

distanze di una fascia differente rispetto alla corrente, per poi applicare effettivamente i cambiamenti. In particolare, sono state realizzate due condizioni: una più permissiva per il cambio verso fasce più critiche, ossia in cui la velocità deve diminuire, e una più rigorosa, in cui sono richieste più ricezioni per aumentare la velocità. Questo approccio senza dubbio aumenta la tolleranza a falsi risultati, ma può implicare una riduzione dell'efficienza, a causa del più lento aumento di velocità. Allo stesso tempo potrebbe migliorare anche la sicurezza grazie allo scongiuro di movimenti bruschi e improvvisi.

```
168     if new_speed_fraction > LAST_SPEED RECEIVED:  
169         CURR_TIMES_HIGH += 1  
170         if(CURR_TIMES_HIGH < MIN_TIMES_HIGH):  
171             new_speed_fraction = LAST_SPEED RECEIVED  
172             CHANGED = False  
173         else:  
174             CURR_TIMES_HIGH = 0  
175             LAST_SPEED RECEIVED = new_speed_fraction  
176             CHANGED = True  
177         else:  
178             if new_speed_fraction < LAST_SPEED RECEIVED:  
179                 CURR_TIMES_LOW += 1  
180                 if(CURR_TIMES_LOW < MIN_TIMES_LOW):  
181                     new_speed_fraction = LAST_SPEED RECEIVED  
182                     CHANGED = False  
183                 else:  
184                     CURR_TIMES_LOW = 0  
185                     LAST_SPEED RECEIVED = new_speed_fraction  
186                     CHANGED = True
```

Figura 59 - Variazione di velocità rallentata (SlowedSpeedControllerUDP.py)

Il sotto-modulo Controller non ha particolari dipendenze da rispettare oltre a disporre di Python, dato che i moduli della libreria RTDE sono già stati inseriti nella directory `rtde/`.

4. Deployment e test

Il deployment è stato effettuato con risorse aderenti all'architettura delineata in precedenza. Ogni dispositivo e macchinario preso in oggetto è stato messo a disposizione da *BI-REX (Big Data Innovation and Research Excellence)*, *Competence Center* nazionale situato a Bologna, istituito dal Ministero delle Imprese e del *Made in Italy*[37]. *BI-REX* si occupa di molteplici attività di ricerca e spesso collabora con *Alma Mater Studiorum*, come in questo caso, per fornire strumentazioni e dispositivi particolari, fondamentali per progetti, esperimenti e prove di questo genere.

Nello specifico, l'architettura particolare adottata ha previsto l'utilizzo di una telecamera industriale Basler GigE, un nodo Nvidia Jetson AGX Orin, un nodo Advantech, uno switch TSN Cisco e un braccio Universal Robots. L'utilizzo di due nodi è totalmente arbitrario, ma contribuisce a rafforzare la modularità del sistema.

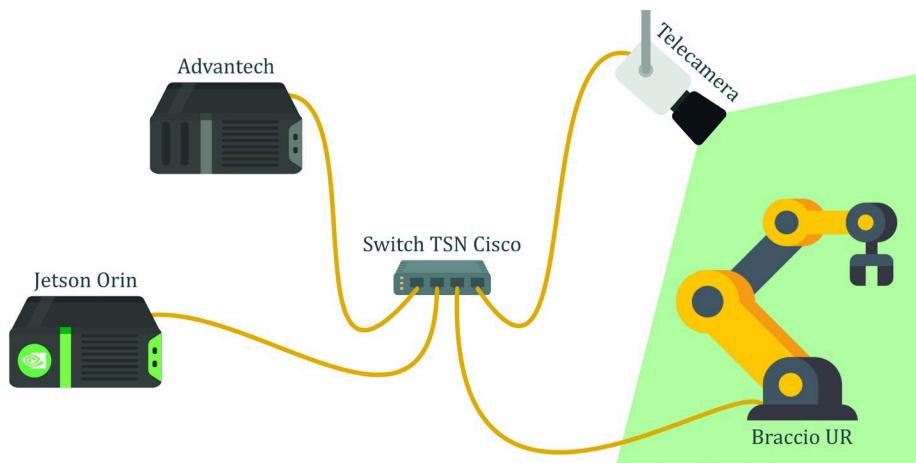


Figura 60 - Architettura del progetto

In questo contesto, tutti gli elementi sono collegati allo *switch* e i due nodi si dividono i compiti: il Jetson Nvidia, dotato di GPU, sarà adibito ad eseguire le operazioni previste dal *submodule Vision*, mentre l'altro nodo, l'Advantech, quelle del *submodule Controller*, che si interfacerà con il braccio. Il flusso del funzionamento è rappresentato in *Figura 61*.

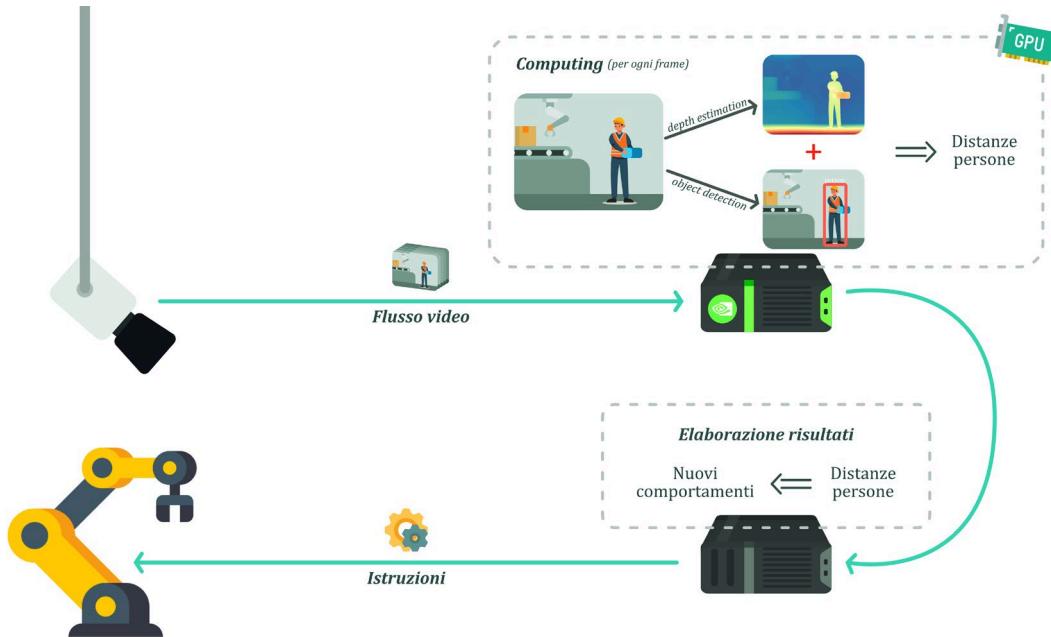


Figura 61 - Flusso del funzionamento dell'architettura del progetto

4.1 Telecamera Basler GigE

La telecamera impiegata è nello specifico una Basler ace 2 (a2a2448-23gcBAS). Questa telecamera supporta lo standard GigE Vision su *Fast* e *Gigabit* Ethernet. È dotata di un sensore Sony in grado di catturare video fino a 23 *fps* con supporto a diversi formati pixel (RGB, BGR, mono, Bayer, YcBcr422), con una risoluzione di 2448 *px* × 2048 *px* a 5 *MP* con possibilità di ridurre l'area di interesse (*ROI*) e di modificare tutti i parametri video. Inoltre, può essere alimentata tramite cavo Ethernet PoE (*Power over Ethernet*) snellendo ulteriormente l'installazione e conferendole maggiore flessibilità[38].



Figura 62 - Basler ace 2 GigE[38]

Nel deployment, illustrato in *Figura 63*, la telecamera è stata montata su un supporto stampato in 3D, in cima ad un'asta all'altezza di 2.3 *m*, inclinata verso il basso e arretrata rispetto al braccio robotico di 1 *m*. In questo modo è possibile monitorare l'area antistante al braccio.



Figura 63 - Deployment Telecamera

4.2 Nvidia Jetson AGX Orin

Nvidia Jetson AGX Orin è una piattaforma di calcolo pensata e progettata per applicazioni varie a livello *edge*. Questa piattaforma ha rappresentato un enorme salto alla sua uscita, in particolare per le sue prestazioni applicate all’Intelligenza Artificiale in rapporto soprattutto ai bassi consumi (30 – 40W), risultando decisamente più potente rispetto alle generazioni precedenti.



Figura 64 - Jetson ADLINK DLAP-411-Orin[39] e deployment

Il modello usato nel deployment è un Jetson ADLINK DLAP-411-Orin. Tale sistema è dotato di un SoC (*System-on-Chip*), in cui sono inseriti i componenti principali:

- GPU (*Graphical Processing Unit*) con architettura NVIDIA Ampere, che dispone di *Tensor Core* accelerati specializzati per il calcolo matriciale, essenziale per operazioni come il *deep learning*. Questa scheda può offrire *performance* fino a 275 *TOPS* (*Tera Operations Per Second*) con calcoli a precisione INT8.
- DLA (*Deep Learning Accelerator*), processori hardware efficienti a livello energetico, progettati per eseguire inferenze di reti neurali

convoluzionali (CNN). Lo scopo di questi componenti è alleggerire la GPU da compiti standard.

- CPU con architettura ARM 64-bit (aarch64), Cortex-A78AE nello specifico, che dispone di 8 core, ma fino a 12 in altre configurazioni.
- 32 GB (o 64 GB in altre versioni) di memoria RAM LPDDR5 con una larghezza di banda di più di 204 GB/s, fondamentale per limitare il collo di bottiglia con le altre componenti.
- 64 GB di archiviazione eMMC integrata, più rapida delle tradizionali memorie di archiviazione perché situata sullo stesso *chip* della CPU.
- Acceleratori video e di visione per gestire più flussi contemporaneamente e adempiere a compiti di *computer vision* e processamento di immagini, alleggerendo ulteriormente CPU e GPU.

L'architettura hardware generale del Jetson, composta dalle sopracitate componenti, è illustrata in *Figura 65*.

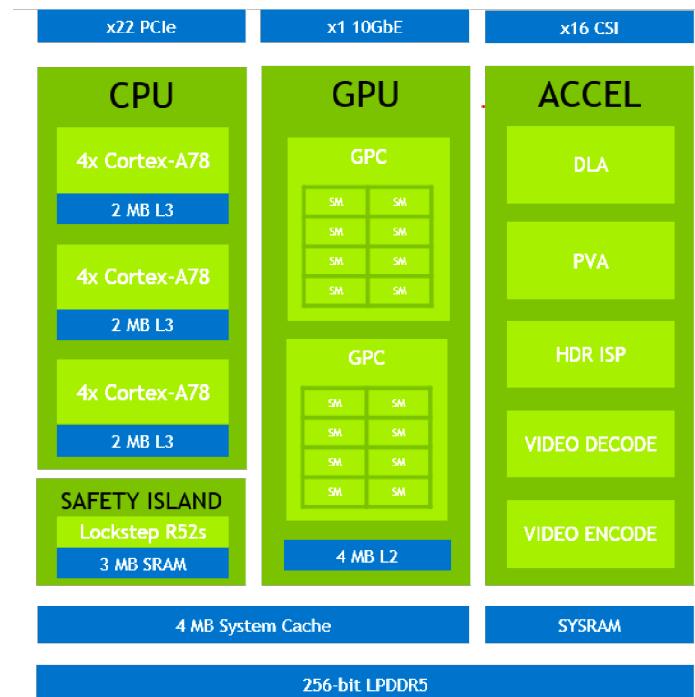


Figura 65 - Diagramma SoC Orin[40]
NOTA: il dispositivo utilizzato integra 2 cluster da 4 Core ciascuno (non 3).

Un altro elemento caratterizzante di questa macchina è il suo ecosistema software completo, chiamato *Jetpack*. Questo fornisce lo *stack* integrale, che comprende il *bootloader* ed il *kernel* Ubuntu, oltre alle librerie di alto livello.

Sono incluse anche librerie di accelerazione, tra cui CUDA. Inoltre, è nativamente supportato Docker, con strumenti di orchestrazione e immagini ufficiali predisposte con librerie e strumenti specifici per diversi casi d'uso. Sono disponibili anche svariati SDK (*Software Development Kit*) per applicazioni definite che ottimizzano le *pipeline*[40].

Il Jetson usato, collegato allo switch TSN, è stato aggiornato ad Ubuntu 22.04.5 LTS (*jammy*) come sistema operativo, con *kernel* 5.15.148-dlap411.1v4.0. La versione LTS del sistema garantisce stabilità a lungo termine, mentre quella del *kernel* è una versione specifica compilata per l'hardware caratteristico. La versione del *Jetpack* a cui è stata aggiornata la macchina è la 6.1, L4T R36.4.0. Per quanto riguarda le principali librerie, atte a sfruttare al meglio l'architettura della GPU NVIDIA, abbiamo CUDA *Toolkit* alla versione 12.6.68, cuDNN 9.3.0.75, TensorRT 10.3.0.30 e Vulkan 1.3.204.

4.3 Advantech UNO

Il nodo Advantech utilizzato, collegato allo *switch* TSN, è in particolare il modello UNO-148, che monta un processore Intel Core i5 1145G7E con *clock* di 2,60GHz, 3 livelli di *cache*, *core* ad alta efficienza e architettura Intel Tiger Lake-LP x86 a 64 bit. Questa CPU ha 4 *core* fisici e 8 *thread* logici, ottenuti sfruttando l'*Hyper-Threading* e una scheda video integrata Intel TigerLake-LP GT2. Inoltre, questa macchina è dotata di 32 GB di RAM e 512 GB di SSD. Lato software, il sistema operativo è Ubuntu 20.0.2 (*Focal Fossa*), con *kernel* 5.10.120.



Figura 66 - Advantech UNO-148

Sul nodo è stata installata la libreria Python RTDE v2.7.2[36] per la comunicazione con i robot UR e Python alla versione 3.12.3.

4.4 Braccio Universal Robots

Nei test eseguiti per la dissertazione sono stati usati diversi bracci robot, tutti prodotti da Universal Robots e in particolare appartenenti alla e-Series. Questi sono bracci *general-purpose* collaborativi (*cobot*), accomunati dall'utilizzo del protocollo RTDE per la comunicazione e l'interfacciamento tramite calcolatore. Grazie al protocollo in questione, è possibile utilizzare indipendentemente uno qualsiasi di questi bracci (o anche più di uno), a patto di conoscerne l'indirizzo IP. Questo garantisce l'intercambiabilità dell'hardware.

Tali robot dispongono di un'interfaccia di programmazione intuitiva, PolyScope, la quale permette configurazioni rapide. Inoltre, esiste un intero ecosistema, UR+, che permette di utilizzare sensori, pinze, software di terze parti certificati, che si integrano direttamente in PolyScope.

La e-Series è dotata anche di numerose funzioni di sicurezza configurabili, tra cui limiti di forza, velocità e potenza, oltre a piani di sicurezza virtuali per favorire la collaborazione[41].



Figura 67 - bracci Universal Robots e-Series[42]

4.5 Switch TSN Cisco IE 4000

Lo *switch* Cisco IE 4000, oltre alle normali funzionalità di *switch* Ethernet, come l'inoltro dei pacchetti e l'ottimizzazione della comunicazione, permettendo lo sfruttamento di tutta la banda disponibile in tutte le direzioni, supporta anche il protocollo TSN. Allo scopo è dotato di hardware dedicato, come un *chip* FPGA (*Field-Programmable Gate Array*), ossia un processore programmabile che consente di gestire le operazioni di temporizzazione e pianificazione richieste dalla famiglia di protocolli TSN.

Questo *switch* implementa gli standard IEEE 802.1AS per la sincronizzazione del tempo e IEEE 802.1Qbv per la pianificazione temporale dei pacchetti, in cui

sono definite le finestre temporali nelle quali vanno trasmessi esclusivamente i pacchetti critici[11].

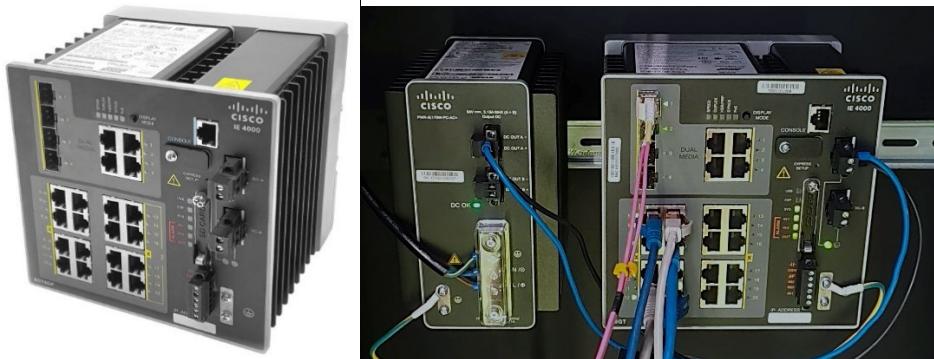


Figura 68 - Switch TSN Cisco IE 4000[11] e deployment

4.6 Deployment Jetson - Immagine e Container Docker

Per creare un ambiente di sviluppo robusto, performante e isolato in cui poter installare liberamente le dipendenze richieste, si è optato di utilizzare un'immagine Docker in cui poter convertire i modelli nel formato ONNX ed eseguire il sotto-modulo Vision.

È stata impiegata l'immagine Docker `dustynv/l4t-pytorch:r36.4.0`[43], una soluzione ottimizzata per l'architettura ARM64 dei dispositivi NVIDIA Jetson realizzata dall'ingegner Dustin Franklin (*dustynv*). Questa immagine, basata su L4T per JetPack 6.1, fornisce uno *stack* software completo e preconfigurato. Include PyTorch 2.4.0 con il suo ecosistema (TorchVision 0.19.0, TorchAudio 2.4.0) compilati con il supporto per l'accelerazione hardware NVIDIA. Sono preinstallate le librerie chiave dello stack NVIDIA:

- CUDA Toolkit 12.6.1 per la comunicazione diretta con la GPU e l'accelerazione dei calcoli.
- cuDNN 9.4.0.58 per l'ottimizzazione delle primitive di *deep learning*, come le convoluzioni.
- TensorRT 10.4.0 per massimizzare le *performance* di inferenza sull'hardware NVIDIA
- OpenCV compilata con il supporto CUDA abilitato, per delegare alla GPU le operazioni di *computer vision* e alleggerire il carico sulla CPU.

L'ambiente è completato da Python 3.10.12 e dalle sue librerie scientifiche essenziali come NumPy, SciPy e Pandas, oltre a *utility* di sviluppo come Git[44].

Per preparare l'ambiente di esecuzione del sotto-modulo Vision, è stata creata un'immagine Docker personalizzata, utilizzando la sopracitata immagine come base (*Figura 70*). Successivamente sono state eseguite una sequenza di operazioni, illustrate in *Figura 69*, per configurare tutte le dipendenze software necessarie. Il processo inizia con la compilazione e l'installazione della libreria Aravis dal codice sorgente. Questo passaggio include l'installazione delle dipendenze di *build* (Meson, CMake e Ninja), la clonazione della *repository* ufficiale, la compilazione e la pulizia dei sorgenti e degli strumenti non più necessari. Successivamente, viene clonato il sotto-modulo Vision. Infine, l'ambiente Python viene completato installando le due componenti chiave: l'ONNX *Runtime* 1.23.0 con supporto GPU tramite una *wheel* pre-compilata da NVIDIA, specifica per l'architettura aarch64 del Jetson, e le restanti librerie Python richieste dal progetto, gestite in modo standardizzato tramite il file **requirements.txt**.

```
1  #!/bin/bash
2
3  apt update && apt upgrade -y
4
5  apt install libxml2-dev libgl2.0-dev cmake libusb-1.0-0-dev gobject-introspection \
6    libgtk-3-dev gtk-doc-tools xsltproc libgstreamer1.0-dev \
7    libgstreamer-plugins-base1.0-dev libgstreamer-plugins-good1.0-dev \
8    libgirepository1.0-dev gettext
9  apt install meson ninja-build cmake
10
11 git clone https://github.com/AravisProject/aravis.git
12 cd aravis
13 meson setup build
14 cd build
15 ninja
16 ninja install
17 ldconfig
18 echo 'export GI_TYPELIB_PATH=/usr/local/lib/aarch64-linux-gnu/girepository-1.0/:$GI_TYPELIB_PATH' >> ~/.bashrc
19 source ~/.bashrc
20 cd ../..
21 rm -rf aravis
22
23 git clone https://github.com/EdgeCV4Safety/EdgeCV4Safety-Vision.git
24 cd EdgeCV4Safety-Vision
25 pip3 install -r requirements.txt
26
27 wget https://pypi.jetson-ai-lab.io/jp6/cu126/+f/4eb/e6a8902dc7708/onnxruntime_gpu-1.23.0-cp310-cp310-linux_aarch64.whl
28 pip3 install onnxruntime_gpu-1.23.0-cp310-cp310-linux_aarch64.whl
```

Figura 69 - Preparazione container Vision

```
1  FROM dustynv/l4t-pytorch:r36.4.0 (last pushed 1 year ago)
2
3  ENV DEBIAN_FRONTEND=noninteractive
4  ENV SHELL=/bin/bash
5  WORKDIR /POC
```

Figura 70 - Dockerfile per il deployment su Jetson Orin

L'immagine è stata costruita tramite il comando `docker build` da *shell* e taggata come `edgecv-vision`. La configurazione di avvio (Figura 71) è stata definita per garantire un funzionamento robusto e performante in un ambiente industriale. Il primo punto focale è stata la persistenza conferita al container per riavviarsi automaticamente in caso di errore o al riavvio del sistema (`--restart=unless-stopped`). È stato abilitato l'accesso completo a tutte le GPU (una) disponibili sull'*host* (`--runtime=nvidia` e `--gpus all`), requisito fondamentale. È stato anche deciso di permettere al container di condividere direttamente l'interfaccia di rete del sistema ospitante (`--network=host`) per minimizzare la latenza e semplificare la comunicazione, oltre che evitare eventuali limitazioni. Successivamente, sono stati concessi privilegi di sistema estesi (`--cap-add=SYS_ADMIN`) per consentire al software nel container di interagire a basso livello con l'hardware e le interfacce di rete. Il container è stato poi avviato in modalità *detached* (`-d`) con il nome `edgecv-vision` per una più facile gestione e riconoscimento.

```
$ docker build --no-cache -t edgecv-vision .
$ docker run --runtime=nvidia \
    --network=host \
    --cap-add=SYS_ADMIN \
    --restart=unless-stopped \
    -it --gpus all \
    -d --name EdgeCV4Safety \
    edgecv-vision
```

Figura 71 - Costruzione immagine e avvio container per il deployment

In seguito, è stato modificato a 9000 il valore della *Maximum Transmission Unit* (MTU) dell'interfaccia dell'*host*, affinché supportasse i *Jumbo Packets*, necessari per il trasporto di pacchetti di grandi dimensioni, come quelli provenienti dalla telecamera.

4. Deployment e test

```
$ CONN_NAME=$(nmcli connection show | grep eth0 | awk '{ print $1 }')"  
$ sudo nmcli connection modify "$CONN_NAME" ethernet.mtu 9000  
$ sudo nmcli connection up "$CONN_NAME"
```

Figura 72 - Impostazione a 9000 MTU (Jumbo Packets) su eth0

Infine, sono stati impostati i parametri cablati nello script principale (`run_cv.py`) necessari per l'esecuzione. Tralasciando dimensioni e modelli di *computer vision* che varieranno durante i test. I parametri di confidenza e *IoU* sono scelti per filtrare parte dei falsi positivi, mentre nell'*array* `CLASSES` è stato inserito unicamente lo 0 in quanto rappresentativo della classe *person*. Nel seguito è stata disabilitata la visualizzazione aerea, abilitando quella per lo pesudo-3D, disabilitando invece le finestre per ottimizzare le prestazioni. Sono stati poi inseriti i parametri desiderati per la configurazione della telecamera (come l'indirizzo IP e i settaggi video). Infine, sono stati inseriti posizione del braccio (relativa alla telecamera) e indirizzo IP e porta del nodo Controller per la comunicazione via UDP.

```
42 # Models settings  
43 DEPTH_MODEL_CHOICE = "unidepth" # Depth estimation model: "unidepth", "depthanything"  
44 DEPTH_MODEL_SIZE = "small" # Depth model size: "small", "base", "large"  
45 YOLO_MODEL_SIZE = "nano" # YOLOv1 model size: "nano", "small", "medium", "large", "extra"  
46  
47 # Detection settings  
48 CONF_THRESHOLD = 0.5 # Confidence threshold for object detection  
49 IOU_THRESHOLD = 0.4 # You threshold for NMS  
50 CLASSES = [0] # Filter by class, e.g., [0, 1, 2] for specific CLASSES, None for disable  
51  
52 # Feature toggles  
53 ENABLE_BEV = False # Enable Bird's Eye View visualization  
54 ENABLE_PSEUDO_3D = True # Enable pseudo-3D calculation  
55  
56 # Camera settings  
57 CAMERA_IP = '192.168.37.150' # None for aravis auto-choice (first found)  
58 CAMERA_FRAME_RATE = 22  
59 CAMERA_PIXEL_FORMAT = Aravis.PIXEL_FORMAT_BAYER_RG_8  
60 CAMERA_GAIN = 20.0  
61 CAMERA_AUTO_EXPOSURE = True  
62 CAMERA_EXPOSURE_TIME = 8000  
63 CAMERA_BUFFER_TIMEOUT = 200000  
64 CAMERA_IMAGE_ROTATION_ANGLE = 0 # 0 to disable  
65 CAMERA_ROI_HEIGHT = 0 # 0 to disable  
66 CAMERA_ROT_WIDTH = 0 # 0 to disable  
67 BUFFER_QUEUE_SIZE = CAMERA_FRAME_RATE * 2 # Number of buffers that can be stored  
68  
69 # Camera infos (Set to 0 to use distances from camera)  
70 CAMERA_HEIGHT_FROM_GROUND = 2.3  
71 CAMERA_DISTANCE_FROM_FIXED_OBJECT = 1  
72  
73 # Output Target Node settings  
74 TARGET_NODE_IP = '192.168.37.58'  
75 TARGET_NODE_PORT = 13750
```

Figura 73 - Parametri Vision (`run_cv.py`)

4.7 CUDA

CUDA (*Compute Unified Device Architecture*) è una piattaforma di calcolo parallelo e un modello di programmazione sviluppato da NVIDIA. Questo permette agli sviluppatori di utilizzare la potenza di calcolo delle GPU NVIDIA per accelerare i calcoli computazionali.

Prima di CUDA, le GPU erano utilizzate quasi esclusivamente per il *rendering* grafico. NVIDIA ha intuito che l'architettura massicciamente parallela delle GPU, composta da centinaia o migliaia di *core*, poteva essere sfruttata per risolvere problemi computazionalmente complessi in ambito scientifico e ingegneristico.

In CUDA, funzioni scritte in linguaggi di basso livello, come C o C++, che vengono eseguite in parallelo su un gran numero di *thread* sulla GPU sono chiamate *kernel*. Ogni *thread* esegue lo stesso codice, ma su dati diversi. I *thread* sono organizzati in blocchi e i blocchi in una griglia. Questa gerarchia permette agli sviluppatori di gestire in modo efficiente ed organizzato l'enorme parallelismo della GPU. Oltre al modello di programmazione, NVIDIA fornisce un ricco ecosistema di librerie ottimizzate per specifici domini, come cuDNN (*CUDA Deep Neural Network library*) per accelerare le operazioni primitive delle reti neurali profonde (quali convoluzioni e *pooling*) [45–47].

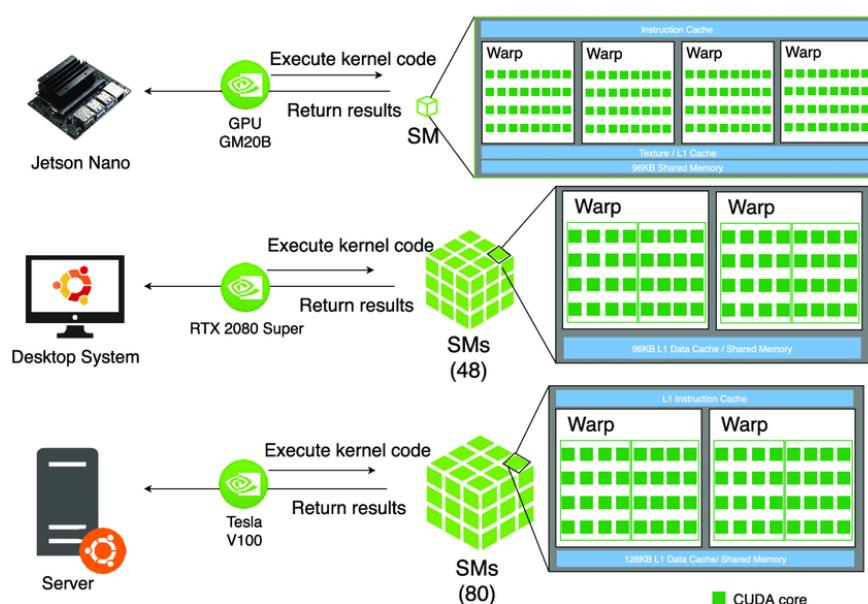


Figura 74 - Esecuzione di kernel CUDA su GPU NVIDIA su piattaforme diverse [48]

L'*Execution Provider* (EP) di CUDA per ONNX si affida alla libreria di inferenza cuDNN, la quale opera aggregando le computazioni della rete neurale in blocchi granulari. Questi blocchi possono essere singole operazioni, come una convoluzione, oppure operatori fusi che combinano più passaggi, ad esempio convoluzione, attivazione e normalizzazione in un'unica operazione.

Il vantaggio principale degli operatori fusi è la riduzione del traffico verso la memoria globale, che spesso rappresenta un collo di bottiglia per le operazioni computazionalmente leggere come le funzioni di attivazione.

Per garantire le massime prestazioni, i blocchi operativi ottimali vengono selezionati tramite euristiche specifiche per la GPU o attraverso una ricerca esaustiva. Quest'ultima viene eseguita solo durante la prima inferenza su un nuovo dispositivo, causando un leggero rallentamento iniziale. Questo però assicura che tutte le esecuzioni successive utilizzino l'implementazione più performante disponibile per ogni blocco[49].

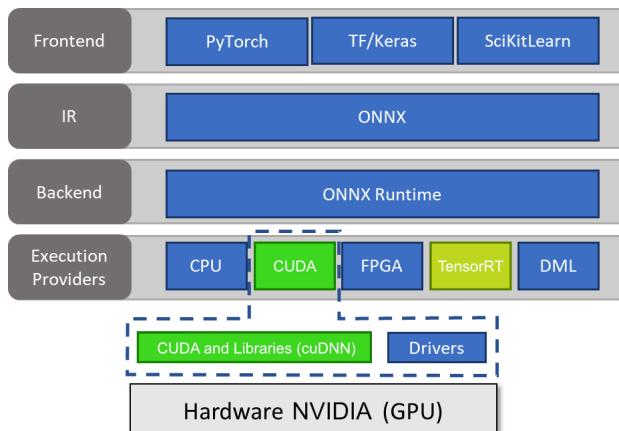


Figura 75 - Flusso CUDA e ONNX

4.8 TensorRT

TensorRT è un SDK (*Software Development Kit*) progettato specificamente per ottimizzare modelli di *deep learning* addestrati e accelerare l'inferenza su GPU NVIDIA.

Per l'ottimizzazione, TensorRT adotta diverse tecniche, quali *Precision Calibration*, *Layer and Tensor Fusion*, *Kernel Auto-Tuning* e *Dynamic Tensor Memory Management*. La *Precision Calibration* prevede la conversione dei

modelli dalla precisione standard a 32-bit (FP32) a precisioni inferiori e più veloci come 16-bit (FP16) o 8-bit (INT8), riducendo l'uso di memoria e sfruttando l'hardware specializzato (come i *Tensor Core*) senza una significativa perdita di accuratezza. Il *Layer and Tensor Fusion* combina più *layer* del modello in un'unica operazione (*kernel*), riducendo l'*overhead* di memoria e aumentando l'efficienza della GPU. Il *Kernel Auto-Tuning* prevede la selezione delle implementazioni di *kernel* CUDA più performanti per le specifiche piattaforma hardware e architettura del modello. Il *Dynamic Tensor Memory Management* ottimizza l'allocazione della memoria video (GPU), riducendo l'impronta complessiva[50].

Ottimizzando l'intero grafo ed eventualmente riordinando le operazioni, si può arrivare ad ottenere prestazioni migliori. I possibili percorsi di esecuzione vengono quindi valutati e il più performante viene selezionato e salvato come *engine*. L'*engine* non contiene unicamente l'elenco di operazioni di un grafo, ma anche i suoi pesi e ogni informazione necessaria all'esecuzione. Il processo di creazione di un *engine* può richiedere diverso tempo, a favore di prestazioni maggiori durante l'esecuzione. Un *engine* è persistente, pertanto può essere usato anche per esecuzioni successive. Nel progetto, infatti, gli *engine* TRT vengono mantenuti in una *directory* apposita, in modo da sfruttarli senza doverli ricreare ad ogni esecuzione.

Il flusso di lavoro generale parte da un modello pre-addestrato (in formati come PyTorch o ONNX) e lo trasforma in un *engine* TensorRT ottimizzato. Il percorso più efficiente e interoperabile è la conversione del modello in formato ONNX e la successiva ottimizzazione tramite TensorRT. La serie Jetson supporta TensorRT e questo flusso è proprio quello che è stato adottato per i test del sistema in cui TensorRT è stato usato come *Execution Provider*[49, 51].

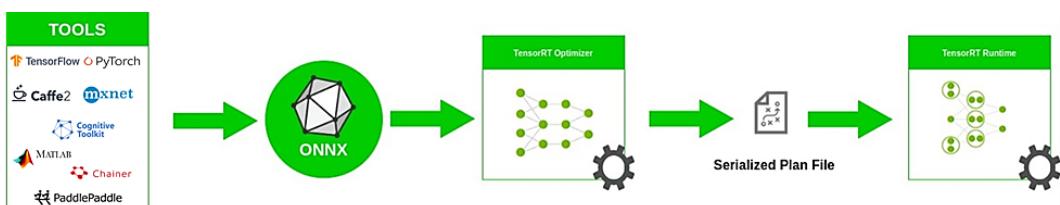


Figura 76 - Flusso TensorRT e ONNX[51]

4.9 Deployment Advantech

Sul nodo Advantech il deployment ha richiesto pochi e semplici passaggi. È bastato clonare il sotto-modulo Controller, al cui interno sono già presenti i file della libreria RTDE.

```
$ git clone https://github.com/EdgeCV4Safety/EdgeCV4Safety-Controller.git
```

Figura 77 - Clonazione repository Controller

Successivamente, sono stati configurati i parametri per la connessione, inserendo indirizzo IP e porta di robot e nodo Vision, oltre ai parametri riguardanti la frequenza di scambio dei messaggi con il robot e il file XML della *recipe*. Per la frequenza va scelto un valore maggiore del *frame rate* per essere certi i messaggi arrivino in ogni condizione, ma comunque minore o uguale alla frequenza massima supportata (125 o 500 Hz).

```
1 ROBOT_HOST = "10.4.1.87" # Robot IP
2 ROBOT_PORT = 30004 # Robot RTDE port (default 30004)
3 CONFIG_XML = './recipe.xml'
4 RTDE_FREQUENCY = 100 # Hz
```

Figura 78 - Parametri RTDE (SpeedControllerUDP.py)

```
1 LISTEN_IP = '192.168.37.50' # TARGET_NODE_IP or 0.0.0.0 to listen on all interfaces
2 LISTEN_PORT = 13750 # TARGET_NODE_PORT
```

Figura 79 - Parametri connessione UDP (udp_listener.py)

```
1 <?xml version="1.0"?>
2 <rtde_config>
3
4   <recipe key="in">
5     <field name="speed_slider_mask" type="UINT32"/>
6     <field name="speed_slider_fraction" type="DOUBLE"/>
7   </recipe>
8
9   <recipe key="out">
10    <field name="actual_TCP_speed" type="VECTOR6D"/>
11    <field name="target_TCP_speed" type="VECTOR6D"/>
12  </recipe>
13
14 </rtde_config>
```

Figura 80 - File di configurazione recipe.xml

Infine, è stato scritto il file XML contenente la ricetta per RTDE, ossia i dati da ricevere in *input* e da mandare in *output* ad ogni iterazione del ciclo scandito dalla frequenza scelta. Per l'obiettivo della modifica della velocità, in accordo con la documentazione[52], i registri da scrivere risultano lo *speed_slider_mask*, per abilitare la modifica della velocità, e lo *speed_slider_fraction*, per impostare il valore effettivo che il robot deve raggiungere. Mentre, per quanto riguarda i valori da leggere dal robot, si è scelto, come *feedback*, di leggere i valori di velocità attuale e *target* del punto centrale dell'utensile montato sul braccio (*actual_TCP_speed* e *target_TCP_speed*).

4.10 Test

Per la validazione empirica dell'architettura precedentemente descritta, è stata condotta una fase di *testing* sistematico utilizzando l'hardware trattato nei sottocapitoli precedenti, in particolare Advantech UNO-148 e Jetson AGX Orin con containerizzazione Docker con l'immagine `dustynv/l4t-pytorch:r36.4.0` opportunamente configurata. La metodologia si è basata sulla misurazione di metriche quantitative chiave, quali i tempi di esecuzione e il consumo di risorse computazionali (CPU, RAM e GPU) per ciascun nodo del sistema. L'analisi è stata condotta in maniera granulare, focalizzandosi sui due componenti principali: il sotto-modulo Vision e quello Controller. I risultati ottenuti, presentati nel seguito, permettono di effettuare un'analisi comparativa, evidenziando le differenze prestazionali, i punti di forza dell'implementazione e le eventuali aree di criticità.

Nell'analisi globale che segue, la latenza introdotta dalla rete è stata ritenuta trascurabile. Questa assunzione è validata dall'utilizzo di TSN, che assicura tempi di transito deterministici nell'ordine dei microsecondi, ordini di grandezza inferiori alla latenza di inferenza. Per ragioni analoghe, anche i tempi di attuazione del protocollo RTDE non sono stati considerati.

Tutte le misurazioni sono state effettuate a sistema a regime, escludendo i tempi di inizializzazione e *setup* dei modelli e dei *provider*.

4.10.1 Test Vision Submodule

I test condotti sul sotto-modulo Vision si sono concentrati in particolare sulle prestazioni e l'utilizzo di risorse con i diversi modelli di Intelligenza Artificiale che lo caratterizzano, convertiti opportunamente in formato ONNX. In particolare, i modelli soggetto delle seguenti indagini sono YOLO11, nelle sue diverse dimensioni (*nano, small, medium, large, extra*), e i due modelli di *depth estimation* Depth Anything v2 e UniDepth v2, anch'essi nelle loro diverse dimensioni (*small, base, large*). Per Depth Anything nella fattispecie, sono stati impiegati i modelli metrici addestrati con scene interne (*indoor*), mentre per tutti gli altri modelli, data la generalità dell'addestramento, sono stati impiegati i modelli tradizionali. La lista completa è riportata in *Figura 81*.

Tutti i test condotti per questo modulo e le metriche raccolte incorporano l'intero sistema, comprensivo di sistema operativo del Jetson, processi in *background* e containerizzazione con Docker.

depthAnything_v2_metric_indoor_base.onnx	389 MB
depthAnything_v2_metric_indoor_large.onnx	1.34 GB
depthAnything_v2_metric_indoor_small.onnx	99 MB
unidepthv2b.onnx	458 MB
unidepthv2l.onnx	1.42 GB
unidepthv2s.onnx	137 MB
yolo11l.onnx	102 MB
yolo11m.onnx	80.7 MB
yolo11n.onnx	10.7 MB
yolo11s.onnx	38.1 MB
yolo11x.onnx	228 MB

Figura 81 - Modelli e dimensioni utilizzati per i test[53]

La prima analisi condotta ha avuto come soggetto di studio il confronto delle *performance* dei singoli compiti di *computer vision*, al fine di isolarne le rispettive prestazioni. Sono quindi state comparate le singole dimensioni di ogni modello in termini di prestazioni con due diversi *Execution Providers*: CUDA e TensorRT.

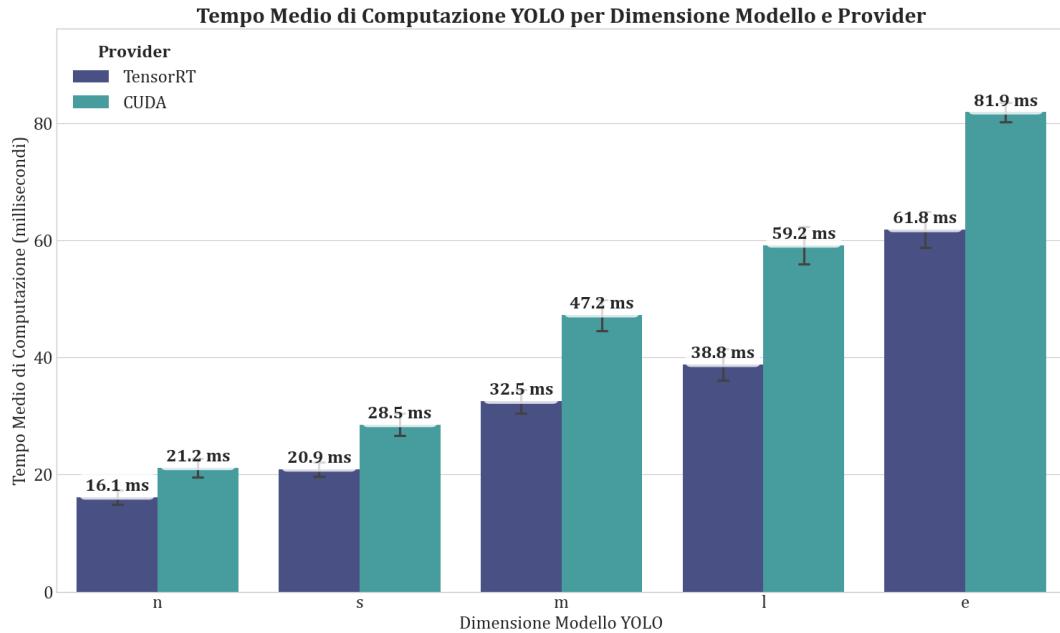


Figura 82 - Grafico comparativo tempi medi di elaborazione YOLO (CUDA vs TensorRT)

Il grafico in *Figura 82* presenta un'analisi comparativa dei tempi di inferenza per le diverse varianti del modello YOLO, eseguite con gli *Execution Provider* CUDA e TensorRT. Emerge chiaramente come l'impiego di TensorRT garantisca un significativo miglioramento prestazionale, quantificabile in un'accelerazione media di quasi il 30% rispetto all'EP CUDA. Tale divario è attribuibile al processo di ottimizzazione del grafo che TensorRT effettua durante la fase di costruzione del suo *engine*, che include la fusione di operatori e la selezione dei *kernel* più efficienti. Inoltre, una verifica cruciale condotta sugli *output* ha confermato che i risultati prodotti dai due *provider* sono del tutto consistenti. Pertanto, l'adozione di TensorRT si rivela una strategia efficace dal punto di vista computazionale, ma anche affidabile, rappresentando la scelta vantaggiosa per il deployment del modello. Osservando le barre di errore, si nota una varianza molto contenuta, che indica un'elevata stabilità e prevedibilità dei tempi di inferenza. Questa leggera oscillazione rispetto al tempo medio, sebbene minima, è un comportamento atteso in un sistema reale ed è attribuibile a una combinazione di fattori architetturali e del sistema operativo. A livello di sistema operativo, il *jitter* è introdotto dalle micro-interruzioni non deterministiche causate dallo *scheduler*, che gestisce i processi in *background*. A questo si aggiunge il comportamento dinamico

dell'architettura della GPU, la quale adatta costantemente le frequenze di *clock* per ottimizzare il bilanciamento tra *performance* e consumi. Infine, essendo il Jetson un SoC, si può verificare un'inevitabile contesa del *bus* di memoria condiviso tra CPU e GPU, che aggiunge un ulteriore *overhead*.

L'analisi prestazionale dei modelli di *depth estimation*, i cui dati sono graficati in *Figura 83*, rivela un netto e consistente vantaggio di UniDepth su DepthAnything in termini di latenza di esecuzione. Questa superiorità si manifesta in tutte le dimensioni di modello testate (*small*, *base* e *large*) e per entrambi gli *Execution Provider* (CUDA e TensorRT), confermando un'efficienza intrinsecamente maggiore dell'architettura di UniDepth. Tuttavia, il distacco si assottiglia all'aumentare delle dimensioni dei modelli, passando da un 23% per i modelli *small* ad un 5-7% per i modelli più pesanti. Anche per questi risultati, si osserva una bassa varianza, evidenziata dalle barre di errore, che indica stabilità e per cui valgono le medesime cause riportate per YOLO (*multitasking* del sistema operativo, frequenza di *clock* dinamica e concorrenza del *bus*).

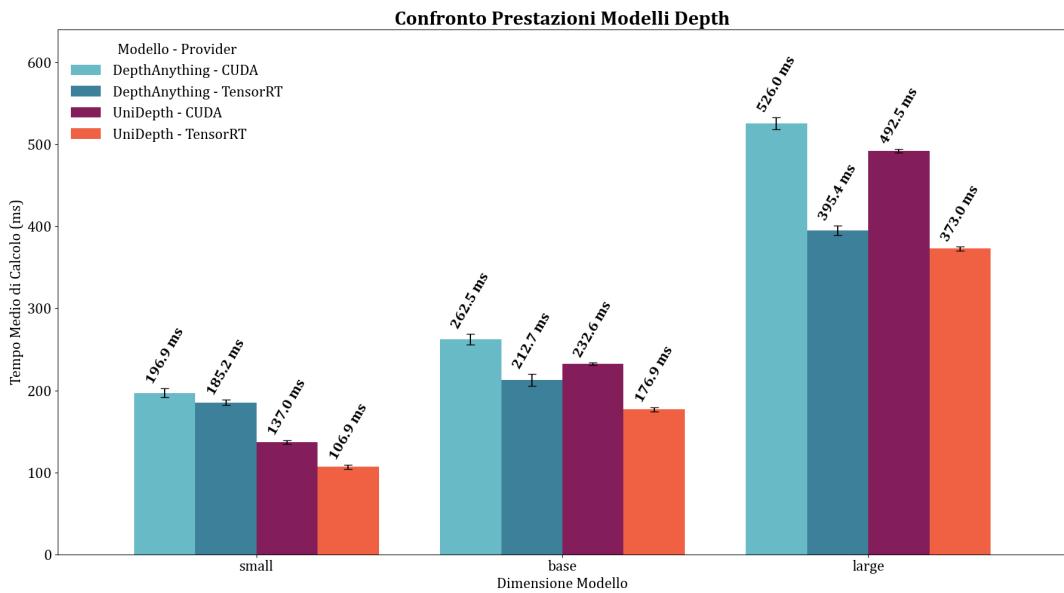


Figura 83 - Grafico comparativo tempi medi di elaborazione Depth Anything e UniDepth (CUDA vs TRT)

L'analisi incrociata con la qualità degli *output*, illustrata in *Figura 84* mediante mappe di profondità, introduce una criticità fondamentale. L'impiego dell'*Execution Provider* TensorRT, sebbene offra una significativa accelerazione, causa un degrado qualitativo inaccettabile per alcune varianti dei modelli. In

generale, per le dimensioni *small* e *base* di entrambi i modelli (con TensorRT) si osservano risultati inutilizzabili. In particolare, quelli di UniDepth si reputano completamente anomali, mentre quelli di Depth Anything, sebbene più sensati, sono caratterizzati da perdita di dettaglio, artefatti e aberrazioni, tali da renderli comunque non affidabili. È invece interessante notare come entrambi i modelli in versione *large* mostrino una maggiore resilienza alle ottimizzazioni di TensorRT. A differenza delle varianti più piccole, queste non subiscono un degrado catastrofico, ma producono un *output* di qualità comparabile alla versione CUDA e con tempi nettamente inferiori, risultando il 25% più rapidi. Si ipotizza che la causa delle anomalie riscontrate per le dimensioni minori risieda nelle ottimizzazioni aggressive di TensorRT, che minano la stabilità numerica dei complessi modelli di stima della profondità. Mentre, verosimilmente, la robustezza dei modelli *large*, dovuta al maggior numero di parametri, li renda meno sensibili agli errori di approssimazione introdotti da TensorRT.

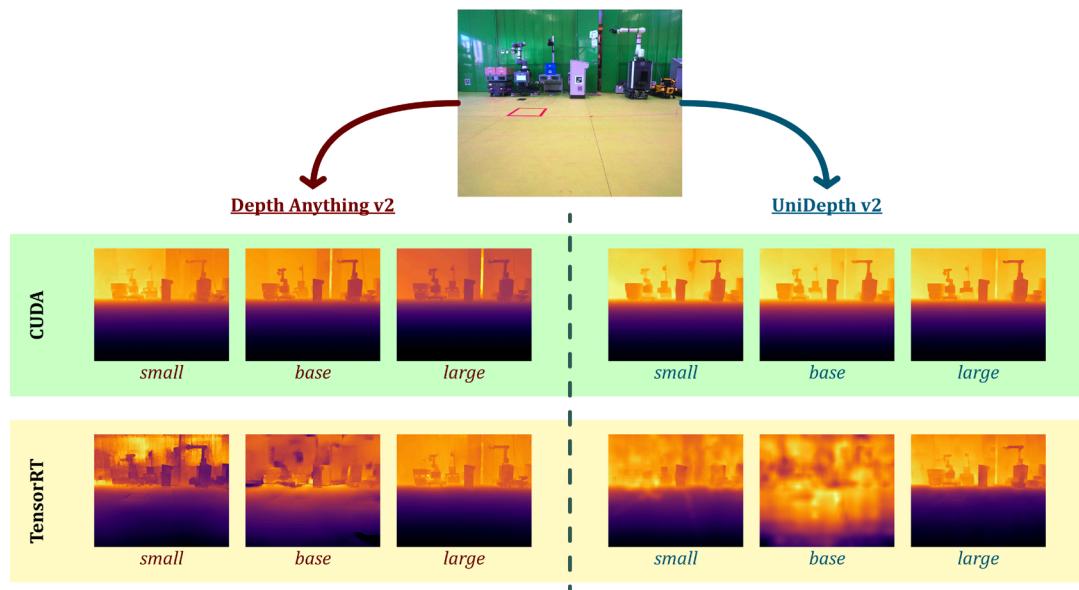


Figura 84 - Comparazione risultati Depth Anything e UniDepth con EP CUDA e TensorRT

Per una valutazione olistica del sotto-modulo, l'analisi è proseguita esaminando le *performance* di tre combinazioni di modelli, ciascuna rappresentativa di un diverso *trade-off* tra velocità e carico computazionale:

- La combinazione più leggera, che accoppia il modello YOLO *nano* con le varianti *small* dei modelli di stima della profondità, rappresentando l'opzione teoricamente più performante.
- La combinazione più pesante, che combina il modello YOLO *extra* con le varianti *large* dei modelli di profondità, costituendo l'opzione più esigente in termini di risorse.
- La combinazione bilanciata, che utilizza YOLO *large* con i modelli di profondità *small*. Questa configurazione è stata identificata come l'alternativa empiricamente più robusta, poiché le varianti più leggere di YOLO, pur essendo veloci, mostrano una tendenza a generare falsi positivi (ad esempio, classificando i bracci robotici come persone), mentre per i modelli di stima di profondità, i risultati risultano pressoché analoghi per tutte le dimensioni (con dettaglio maggiore all'aumentare delle dimensioni).

Per ciascuna configurazione sono state misurate le metriche di latenza e consumo di risorse con entrambi gli *Execution Provider* (CUDA e TensorRT). Sebbene, come riscontrato in precedenza, alcuni modelli di profondità non producano risultati utilizzabili, si è deciso di estrarne comunque le metriche a scopo comparativo.

Analizzando i risultati della configurazione leggera (YOLO *nano* + Depth *small*), tenendo conto dell'inaffidabilità dei dati di inferenza dei modelli di stima della profondità con TensorRT, si confermano due fondamentali risultati. Per prima cosa, la combinazione con UniDepth risulta mediamente più rapida di un abbondante 30% rispetto a quella con Depth Anything. In secondo luogo, si osserva nuovamente il guadagno prestazionale atteso derivante dall'utilizzo di TensorRT per la sola inferenza di YOLO. Pertanto, il *frame rate* massimo supportato da Depth Anything si attesta attorno ai 4 *fps*, mentre quello di UniDepth arriva quasi a 6 *fps*. Anche qui si osserva una varianza contenuta, come atteso dall'analisi dei due componenti (YOLO e Depth) separati, che ribadisce e conferma la stabilità del sistema.

4. Deployment e test

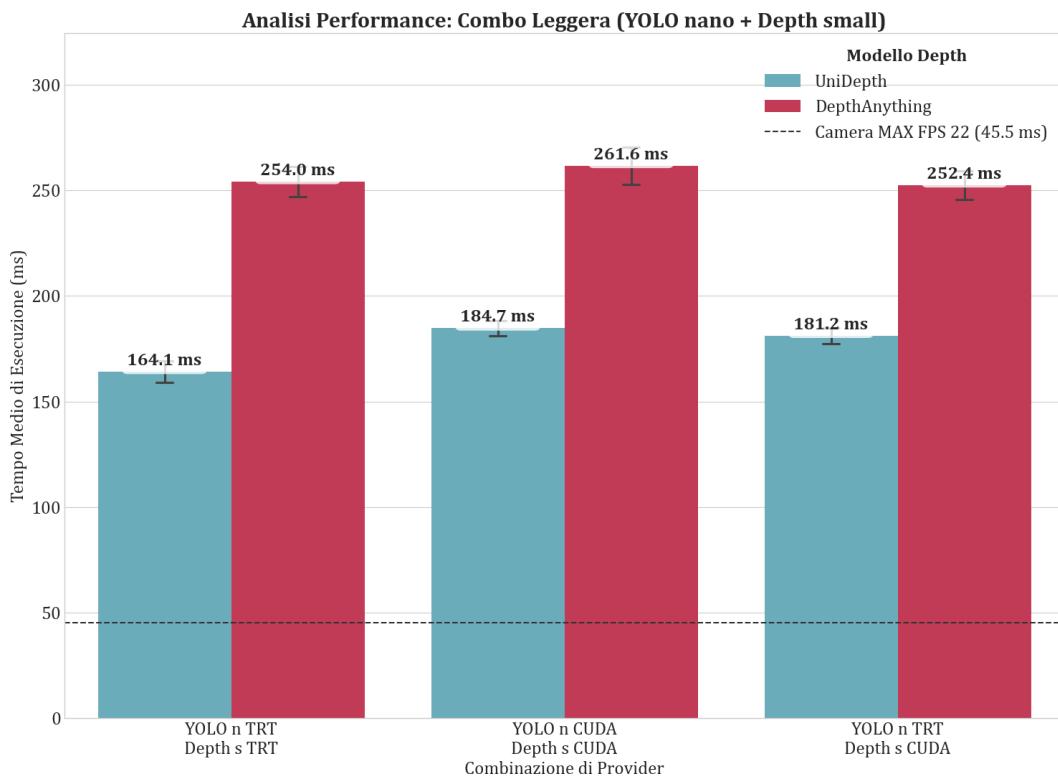


Figura 85 - Grafico comparativo tempi medi di elaborazione YOLO nano + Depth small

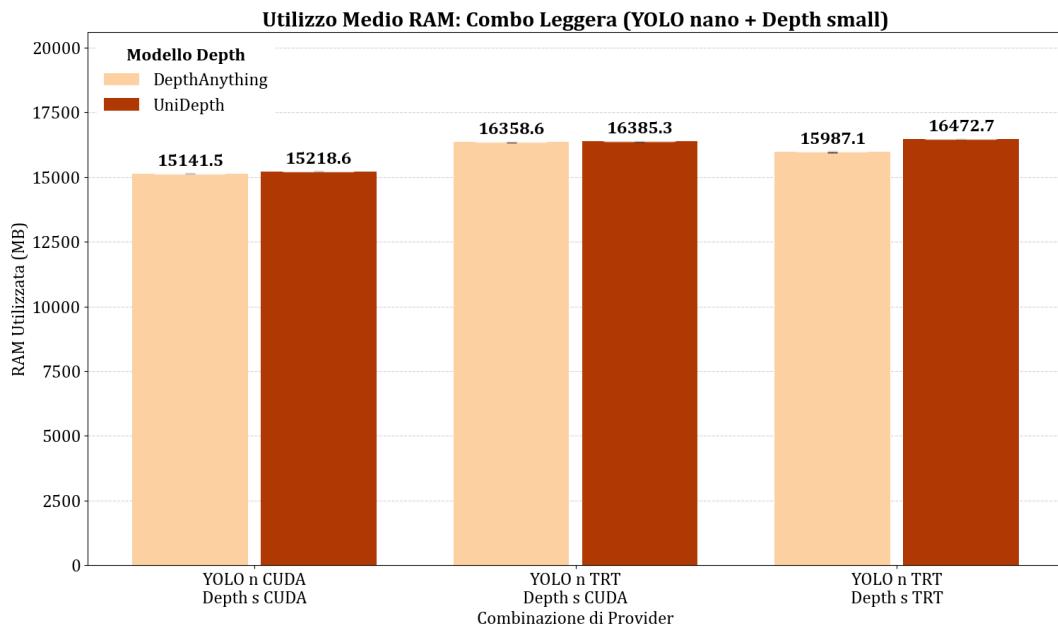


Figura 86 - Grafico comparativo utilizzo RAM per elaborazione YOLO nano + Depth small

L'analisi del consumo di risorse per la configurazione leggera fornisce un quadro dettagliato del comportamento del sistema (comprendendo sistema operativo, applicativi correlati e containerizzazione).

L'impronta di memoria del sistema, illustrata in *Figura 86*, si è rivelata stabile, come mostrato dalle barre di errore praticamente sovrapposte, attestandosi intorno ai 16 GB. Questo comportamento è tipico dei modelli di *deep learning*, che allocano la memoria necessaria per i pesi e i *buffer* di attivazione all'avvio. Si osserva un leggero incremento nell'utilizzo della RAM quando si impiega TensorRT, potenzialmente a causa dello spazio di lavoro aggiuntivo allocato dal suo *engine*.

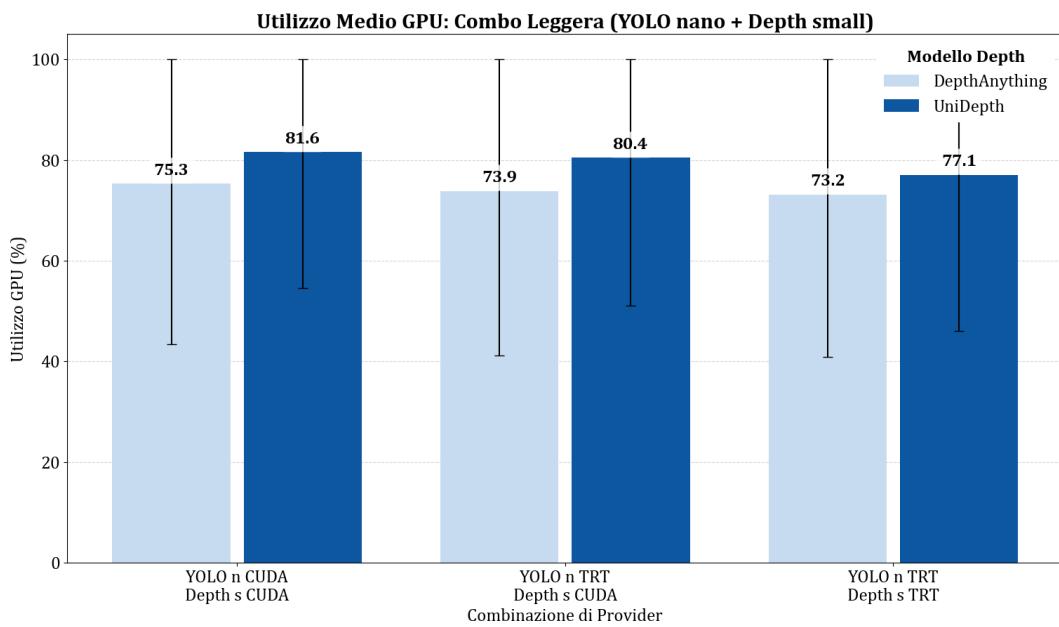


Figura 87 - Grafico comparativo utilizzo GPU per elaborazione YOLO nano + Depth small

L'analisi dell'utilizzo della GPU, riportata in *Figura 87*, è la più significativa e di interesse. La caratteristica saliente è l'ampia varianza, indicata dalle barre di errore, che conferma come la GPU non operi a un carico costante, ma alterni fasi di elaborazione intensa (vicine al 100%) a momenti di parziale inattività o attesa (attorno al 40-50%). Questi momenti di minor carico sulla GPU corrispondono alle fasi in cui la CPU è predominante nel ciclo, come durante il *pre-processing* del *frame* successivo o il *post-processing* del risultato precedente, prima che la nuova inferenza possa essere lanciata. Ciò è ulteriormente confermato dal valore medio, che si colloca verso il centro dell'intervallo delimitato dalle barre di errore. Nonostante questa variabilità, emerge un *pattern* chiaro: UniDepth mostra un utilizzo medio ($\approx 80\%$) e minimo più elevato rispetto a DepthAnything ($\approx 75\%$). Correlare questo dato con i tempi

di inferenza, notevolmente inferiori per UniDepth in questa configurazione, porta alla conclusione che UniDepth sembra in grado di saturare la GPU in modo più efficiente. Il maggior carico medio e minimo suggeriscono che il modello riduca i tempi morti tra le operazioni, massimizzando il *throughput* computazionale e completando il calcolo più rapidamente.

L'utilizzo della CPU, rappresentato in *Figura 88*, rimane su valori contenuti, mediamente intorno al 20-23%, senza significative variazioni tra le diverse combinazioni di *provider*. Questo dato conferma, come prevedibile, la natura GPU-bound della *pipeline*, dove la computazione è quasi interamente delegata all'acceleratore grafico. Il carico misurato è la somma di due componenti: un carico di base, dovuto al sistema operativo e alla containerizzazione, e un carico attivo generato dalle fasi non delegabili alla GPU (quali acquisizione, pre- e post-processing), la cui alternanza spiega la variabilità indicata dalle barre di errore.

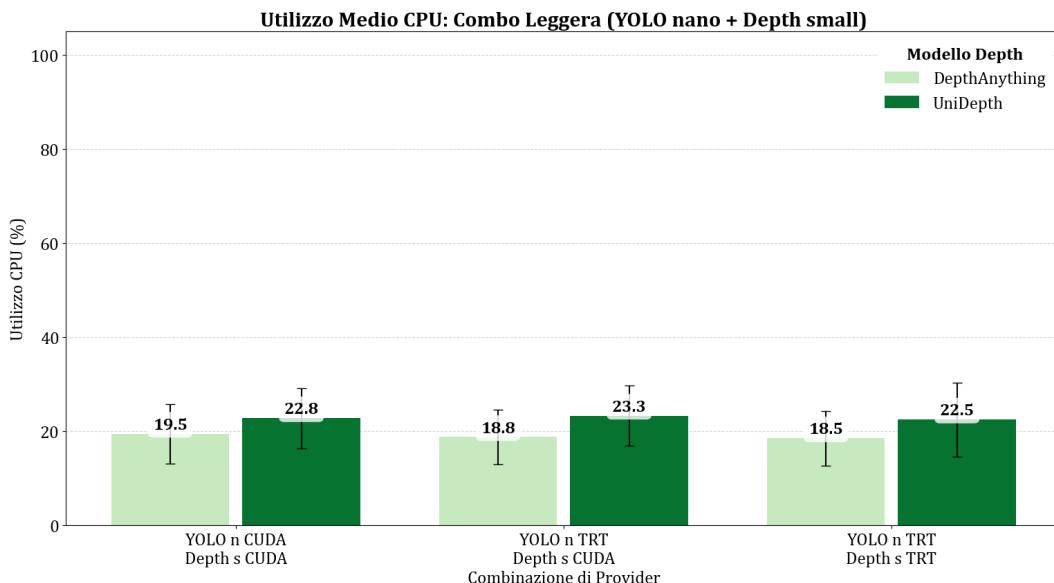


Figura 88 - Grafico comparativo utilizzo CPU per elaborazione YOLO nano + Depth small

Procedendo con l'analisi della configurazione a carico computazionale elevato (YOLO *extra* + Depth *large*), si riscontrano tendenze generali coerenti con quelle osservate per la configurazione leggera e con i risultati dei singoli modelli, sebbene con grandezze e implicazioni differenti.

In linea con i risultati prestazionali precedenti, UniDepth si conferma il modello più performante, ma il suo vantaggio su DepthAnything si riduce drasticamente,

passando a circa un 6-7%, margine notevolmente inferiore rispetto a quello osservato nella configurazione leggera. Questo suggerisce che l'efficienza di UniDepth scali meno efficacemente all'aumentare della complessità del modello. L'impatto di TensorRT deve essere analizzato a due livelli. La configurazione completamente basata su TensorRT (YOLO TRT + Depth TRT) è, come previsto, la più veloce in assoluto. Essendo i risultati dei modelli di profondità affidabili in questo contesto anche con TensorRT, si assiste ad un netto miglioramento delle *performance* con questo *provider*, concretizzabile in un circa 27%. I dati afferenti alle prestazioni, illustrati in *Figura 89*, mostrano quindi che il sistema è in grado di smaltire un carico di poco più di due immagini al secondo in maniera stabile, come segnalato dalla bassissima varianza (per cui valgono le medesime affermazioni discusse in precedenza).

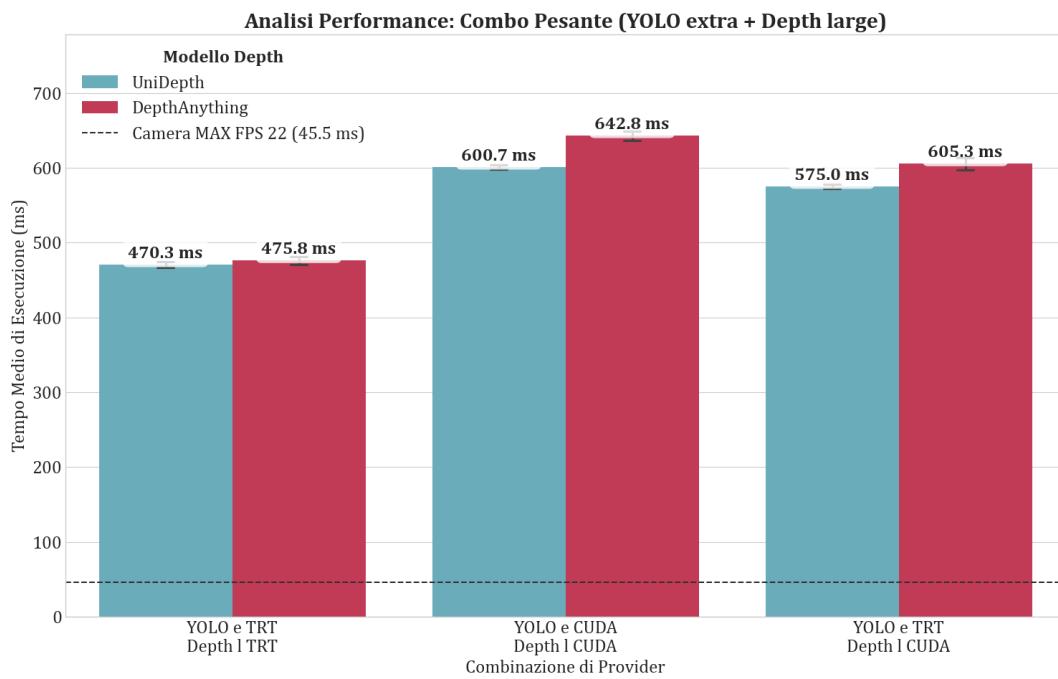


Figura 89 - Grafico comparativo tempi medi di elaborazione YOLO extra + Depth large

Si osserva poi un aumento significativo dell'impronta di memoria, illustrato in *Figura 90*, rispetto alla configurazione leggera, ma comunque ancora costante e stabile (varianza pressoché nulla). È particolarmente evidente l'impatto di TensorRT, infatti la sua adozione per il solo modello YOLO incrementa la RAM utilizzata di circa 1 GB, probabilmente a causa dello spazio di lavoro richiesto dall'*engine*. La configurazione totalmente basata su TensorRT mostra un

consumo che arriva a superare addirittura i 20 GB, confermando come le ottimizzazioni di TensorRT avvengano al costo di un maggiore utilizzo di memoria.

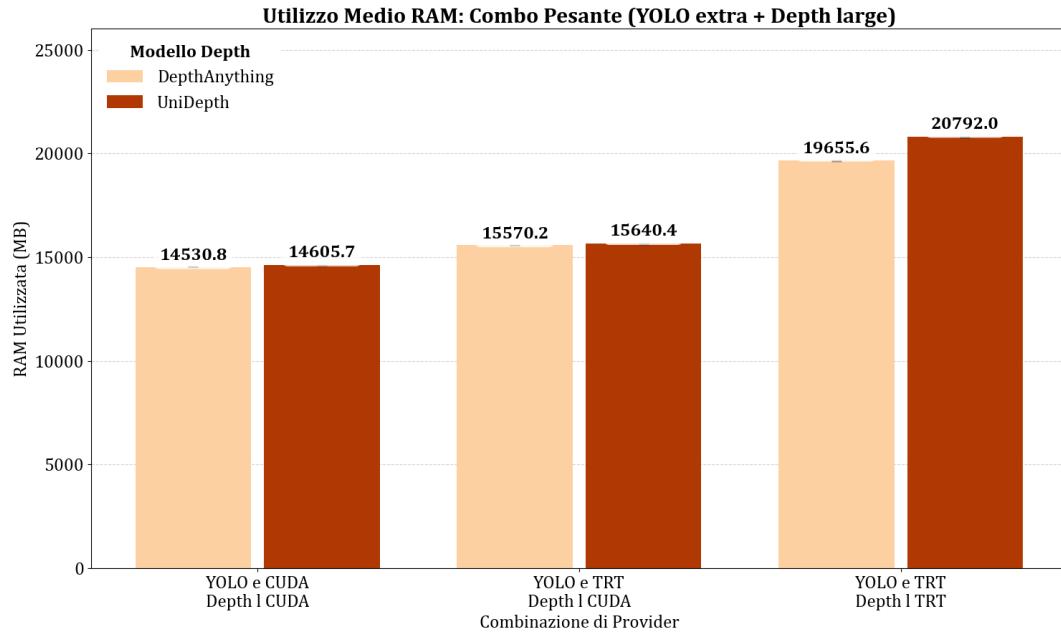


Figura 90 - Grafico comparativo utilizzo RAM per elaborazione YOLO extra + Depth large

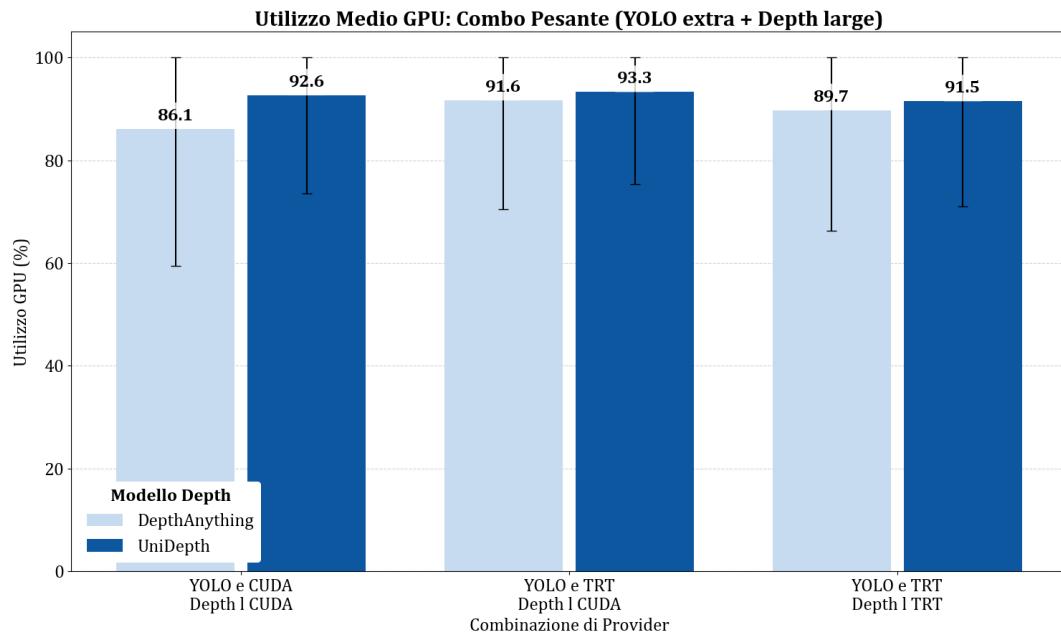


Figura 91 - Grafico comparativo utilizzo GPU per elaborazione YOLO extra + Depth large

L'utilizzo medio della GPU, illustrato in *Figura 91*, si attesta stabilmente sopra il 90% per quasi tutte le configurazioni, indicando che il sistema è fortemente *GPU-bound* e riportando le medesime caratteristiche generali discusse in

precedenza. La GPU opera costantemente vicino al suo limite di saturazione, il che spiega direttamente i lunghi tempi di esecuzione osservati.

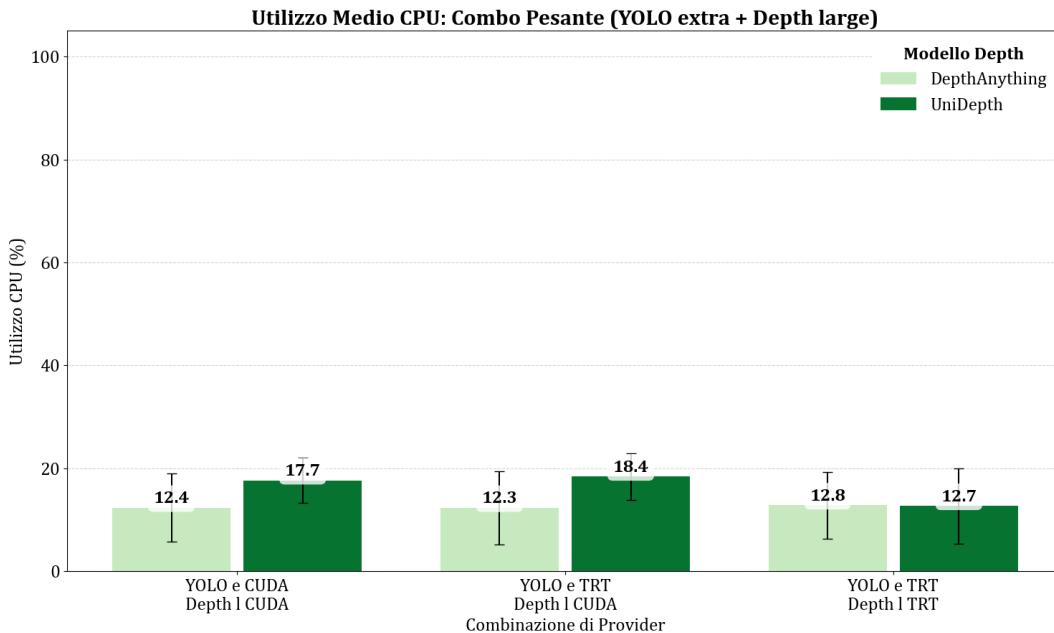


Figura 92 - Grafico comparativo utilizzo CPU per elaborazione YOLO extra + Depth large

Coerentemente con una *pipeline* fortemente basata sull'utilizzo della GPU, il carico sulla CPU rimane ancora marginale. Le variazioni osservate tra i diversi *provider*, i cui dati sono stati raccolti in *Figura 92*, sono minime e non indicano un impatto significativo, confermando che la CPU non costituisce in alcun modo un collo di bottiglia e che il suo utilizzo oscilla in maniera contenuta a causa dei fattori precedentemente esaminati. Si nota però un pattern apparentemente controidintuitivo, ossia che, rispetto alla configurazione leggera (*Figura 88*), l'utilizzo della CPU è minore. Si passa infatti da un 19-22% medio ad un 12-18% medio. Anche la varianza diminuisce e diminuiscono i suoi valori massimi e minimi. Questo fenomeno è in realtà un indicatore diretto del *throughput* del sistema. Con i modelli leggeri, l'elevato *frame rate* impone alla CPU di alternare rapidamente fasi di lavoro attivo (pre- e post-*processing*) a brevi periodi di attesa, generando continue fluttuazioni nel carico e, di conseguenza, una media e una varianza più alte. Al contrario, con i modelli pesanti, il collo di bottiglia della GPU diventa così predominante che la CPU trascorre la stragrande maggioranza del tempo in uno stato di attesa a basso consumo, quasi costante. Questa lunga fase di inattività forzata riduce l'impatto dei brevi picchi di lavoro,

risultando in un utilizzo medio più basso e in una varianza significativamente più contenuta, poiché i valori misurati si raggruppano attorno a questo stato di attesa a basso carico.

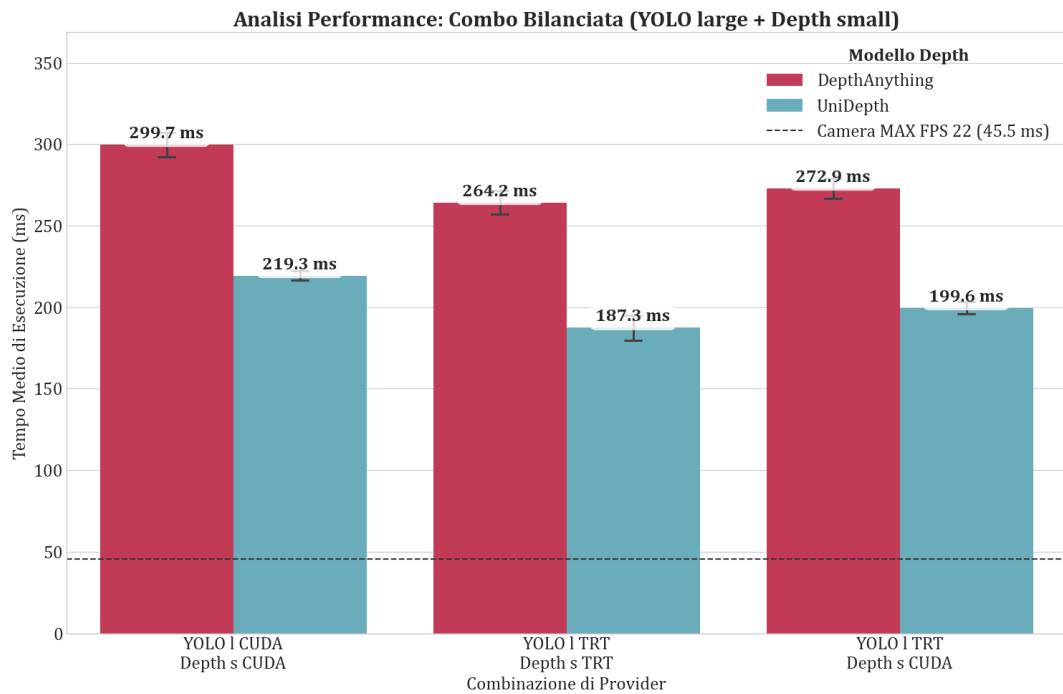


Figura 93 - Grafico comparativo tempi medi di elaborazione YOLO large + Depth small

Si avanza infine con l'analisi della configurazione identificata come il compromesso empirico ottimale, la quale accoppia un modello di *object detection* ad alta accuratezza (*YOLO large*) con i modelli di stima della profondità a minor carico computazionale (*Depth small*). I risultati prestazionali, illustrati in *Figura 93*, confermano le diverse tendenze chiave.

Coerentemente con le osservazioni precedenti, UniDepth mantiene un vantaggio prestazionale netto e significativo su Depth Anything. UniDepth è circa il 30% più veloce. Questo vantaggio si mantiene in tutte le combinazioni, provando che, per i modelli *small*, l'architettura è intrinsecamente più efficiente. Anche in questa configurazione si riafferma l'efficacia di utilizzare TensorRT come *Execution Provider* per il solo modello YOLO (dato che i risultati di Depth TRT sono inutilizzabili), che comporta miglioramenti fino al 10% circa. Il sistema si rivela stabile, come osservabile dalla ridotta varianza (per cui valgono le cause di cui sopra).

Complessivamente, quindi, UniDepth permette di elaborare quasi 5 *frame* al secondo, contro gli scarsi 4 di Depth Anything.

Andando ad esaminare le metriche di utilizzo delle risorse, queste rispecchiano esattamente il comportamento fino ad ora riscontrato e atteso.

L'utilizzo della RAM, mostrato in *Figura 94*, si attesta su un livello elevato ma stabile, con valori che oscillano tra i 14 GB e i 17 GB. Coerentemente con i test precedenti, l'adozione di TensorRT introduce un *overhead* di memoria quantificabile, verosimilmente dovuto allo spazio di lavoro e alla memorizzazione dell'*engine* ottimizzato e in generale l'utilizzo rimane costante.

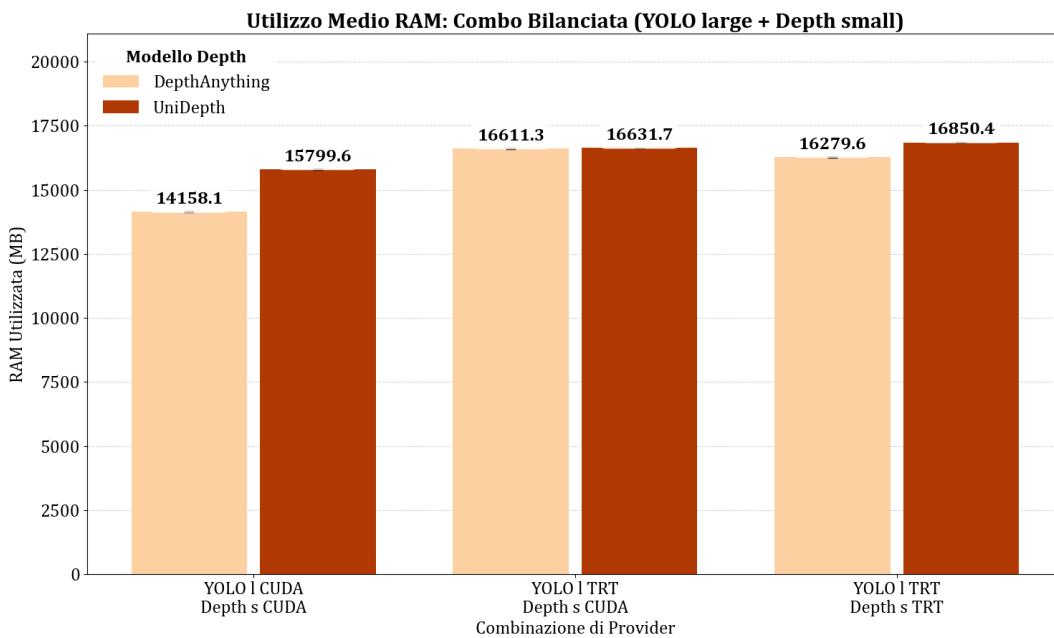


Figura 94 - Grafico comparativo utilizzo RAM per elaborazione YOLO large + Depth small

L'analisi del carico della GPU è la più indicativa ed è illustrata in *Figura 95*. I dati sono analoghi a quelli della configurazione leggera (e in linea con la configurazione pesante) e mostrano un sistema che, pur non essendo completamente saturo come nello scenario pesante, opera comunque a un regime di carico elevato, mediamente tra il 75% e l'84%, per cui valgono i ragionamenti sulla variazione del carico esposti in precedenza in questo contesto.

Si riconferma nuovamente la superiore efficienza di UniDepth nel saturare l'hardware grafico. Il suo utilizzo medio della GPU è costantemente superiore a quello di Depth Anything in tutte le combinazioni. Questa capacità di mantenere

la GPU più attiva permette di ridurre i tempi di inattività tra le operazioni e si traduce direttamente e in modo misurabile nei tempi di esecuzione inferiori osservati anche nelle analisi prestazionali precedenti.

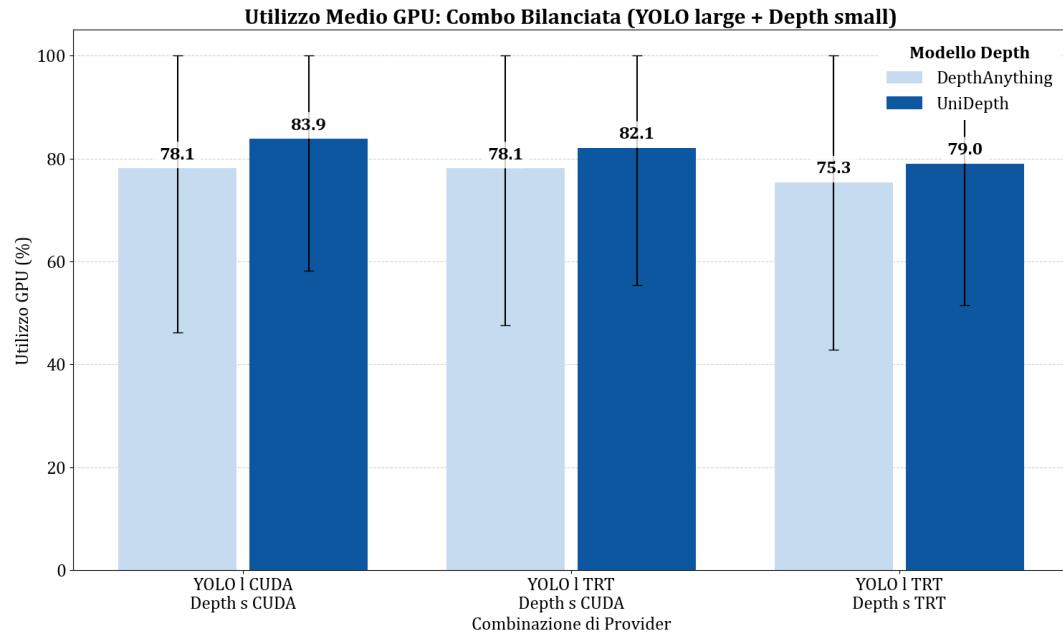


Figura 95 - Grafico comparativo utilizzo GPU per elaborazione YOLO large + Depth small

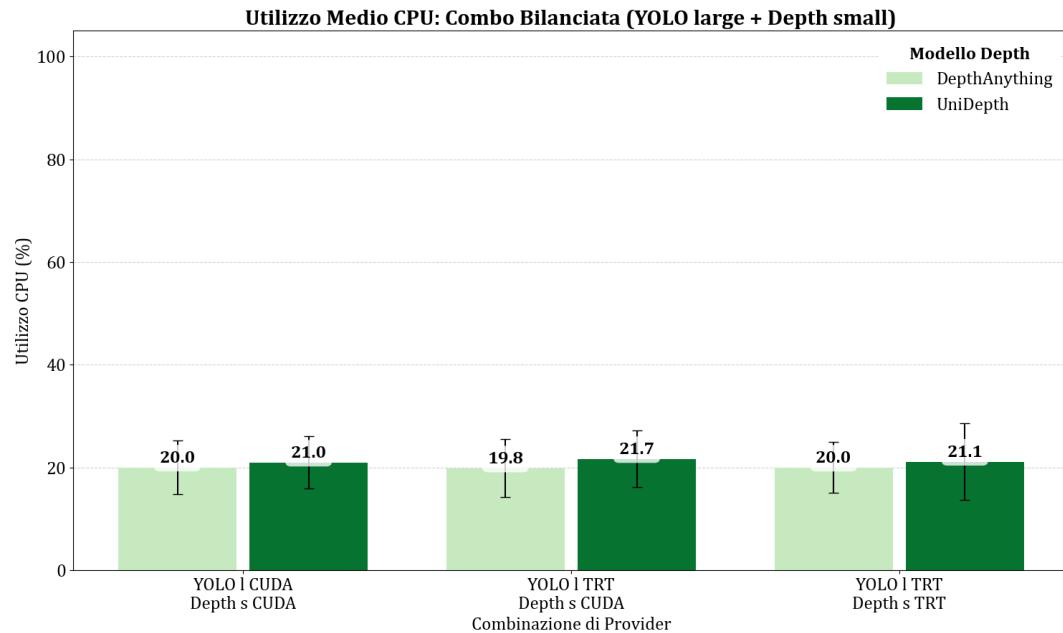


Figura 96 - Grafico comparativo utilizzo CPU per elaborazione YOLO large + Depth small

Il carico sulla CPU, le cui metriche sono riportate in Figura 96, rimane marginale, con un utilizzo medio stabile intorno al 20-22%. Analogamente allo scenario leggero si registra un utilizzo della CPU più elevato rispetto alla

configurazione pesante, a causa del *throughput* più elevato. Questi dati confermano ancora una volta, in modo inequivocabile, la natura GPU-*bound* della *pipeline*, in cui la CPU svolge un ruolo di supporto e non costituisce mai un collo di bottiglia prestazionale, alternando fasi di utilizzo maggiore a fasi più rilassate.

4.10.2 Test Controller Submodule

La valutazione del sotto-modulo Controller si è focalizzata sulla quantificazione del suo carico computazionale in funzione della frequenza di comunicazione del protocollo RTDE. Questo parametro introduce un *trade-off* fondamentale: frequenze elevate garantiscono una maggiore reattività e un controllo più fine, ma al costo di un incremento del traffico di rete e del potenziale congestionamento. Sono quindi state selezionate tre frequenze operative, tenendo conto dei limiti dell'hardware: 30 Hz e 100 Hz come opzioni a basso impatto, e 500 Hz come limite massimo supportato dai bracci utilizzati.

Tutte le metriche illustrate in questa sezione sono da attribuirsi alla sola esecuzione del sotto-modulo Controller.

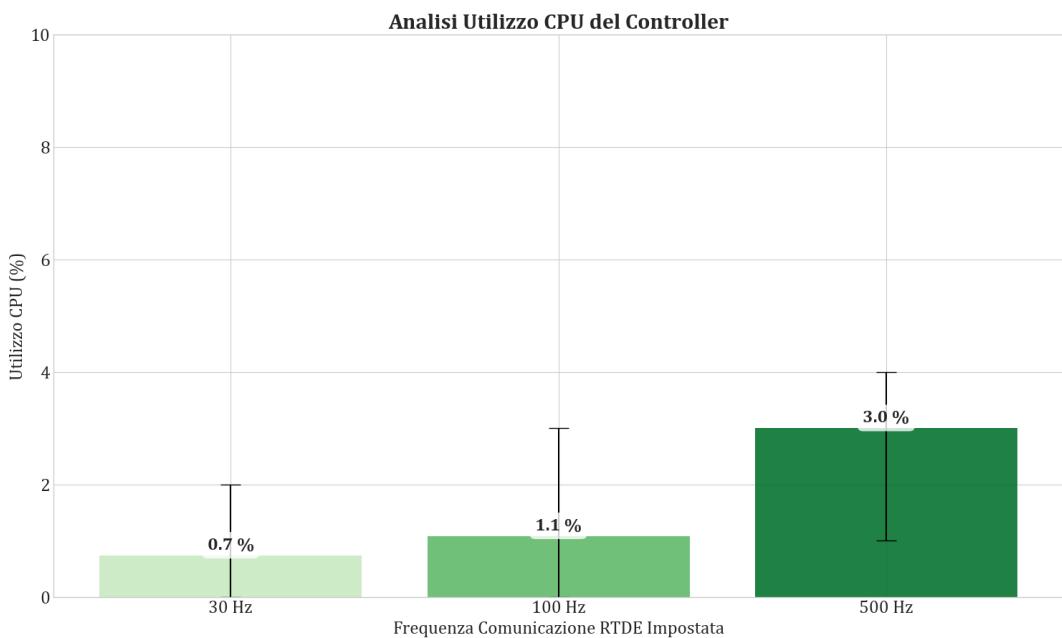


Figura 97 - Grafico comparativo utilizzo CPU nel Controller

I dati sull'utilizzo della CPU, illustrati in *Figura 97*, dimostrano che il modulo presenta un *overhead* computazionale estremamente contenuto. Si osserva una correlazione diretta e lineare tra la frequenza RTDE e il carico sulla CPU, che passa da uno 0.7% medio a 30 Hz fino a un 3.0% medio a 500 Hz.

Un aspetto significativo, evidenziato dalle ampie barre di errore, è l'elevata varianza nell'utilizzo della CPU. Questa eterogeneità è una conseguenza diretta dell'architettura software del Controller. Il *thread* principale, che gestisce la comunicazione RTDE, è caratterizzato da cicli di `sleep` per mantenere la frequenza impostata, risultando in periodi di quasi totale inattività. Contemporaneamente, l'altro processo (quello che gestisce la comunicazione UDP) e l'altro *thread* (quello che esegue il `main`) del modulo sono prevalentemente I/O-bound, infatti passano la maggior parte del tempo in attesa di messaggi. Questa natura asincrona e non computazionalmente intensiva spiega sia il basso utilizzo medio sia le fluttuazioni significative.

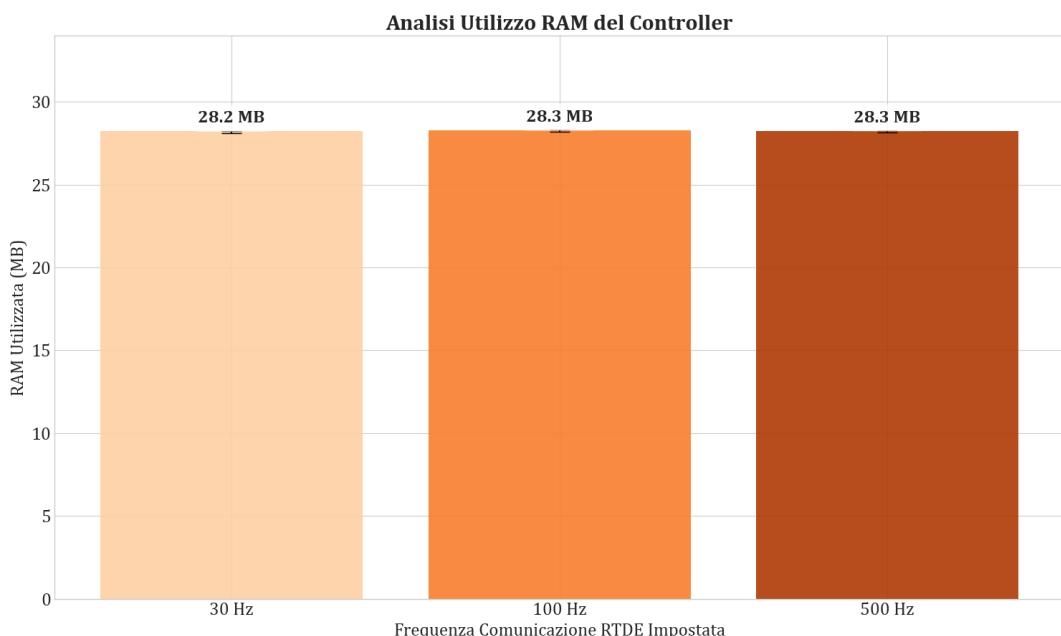


Figura 98- Grafico comparativo utilizzo RAM nel Controller

L'analisi dell'impronta di memoria del Controller, i cui dati sono riportati in *Figura 98*, rivela un consumo di RAM estremamente contenuto e, come previsto, sostanzialmente invariante al variare della frequenza di comunicazione RTDE. Questo comportamento è atteso e riflette la natura dell'applicazione. Infatti,

l'allocazione della memoria per le strutture dati, le variabili e le librerie avviene in fase di inizializzazione e non è dipendente dalla frequenza operativa. Con un valore medio di circa *28 MB*, il consumo è da considerarsi trascurabile, specialmente se posto in netto contrasto con i gigabyte richiesti dal sotto-modulo Vision.

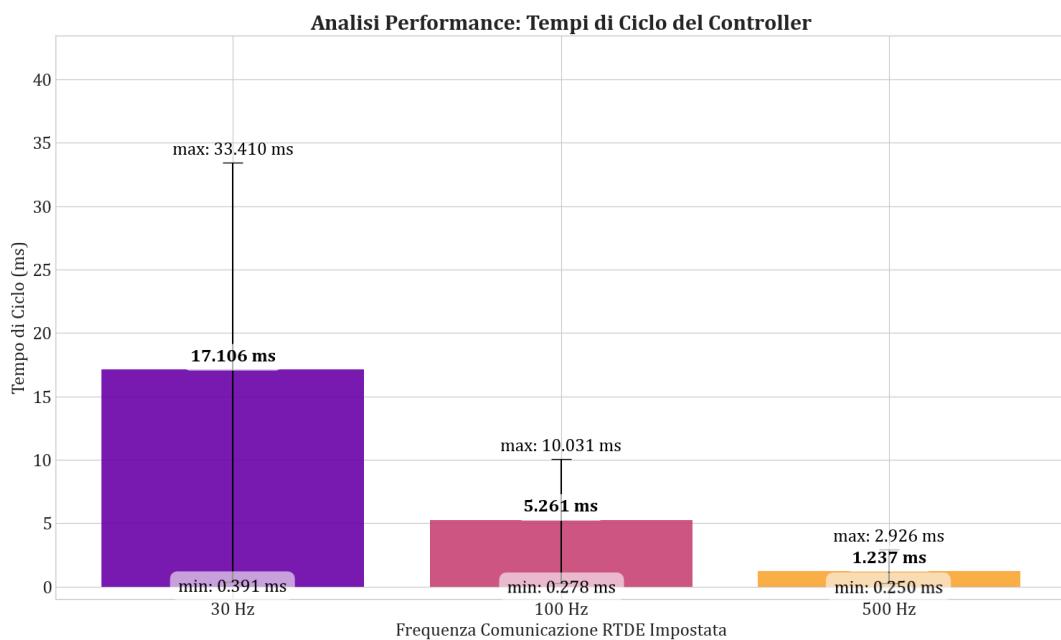


Figura 99 - Grafico comparativo tempi di esecuzione di un ciclo nel Controller

L'analisi delle *performance* del Controller, misurate come tempo di ciclo e illustrate in *Figura 99*, rivela che la latenza del sotto-modulo è interamente governata dal meccanismo di *throttling*, legato alla frequenza di comunicazione RTDE, e non dal carico computazionale.

I tempi di ciclo massimi osservati (*33.41 ms*, *10.03 ms*, *2.92 ms*) sono strettamente correlati al periodo teorico imposto dalla frequenza RTDE (rispettivamente *33.3 ms*, *10 ms*, *2 ms*). Questi picchi non rappresentano un'elaborazione lunga, ma il tempo totale del ciclo quando la funzione di `sleep` attende per quasi l'intera durata del periodo per mantenere la frequenza desiderata (caso peggiore). I tempi di ciclo minimi registrati, che si attestano tra *0.250 ms* e *0.391 ms*, rappresentano la reale velocità di esecuzione della logica del Controller. Questi valori sub-millisecondo dimostrano che il calcolo effettivo è estremamente rapido.

I valori medi indicano il comportamento tipico del sistema. Il fatto che siano inferiori al periodo *target* suggerisce che il sistema operativo e la logica di controllo del *loop* introducono del *jitter*, ma nel complesso il sistema rispetta la frequenza impostata.

5. Conclusioni e sviluppi futuri

In questa dissertazione è stata progettata, implementata e validata l'architettura di EdgeCV4Safety, un sistema innovativo di *edge vision* per la sicurezza in contesti di Industria 4.0 e 5.0. L'obiettivo primario era superare le limitazioni delle barriere fisiche tradizionali, proponendo una soluzione flessibile, scalabile e intelligente basata sulla percezione visiva per garantire la collaborazione sicura tra uomo e macchinari.

L'analisi empirica condotta ha permesso di trarre diverse conclusioni fondamentali. In primo luogo, l'architettura modulare proposta, suddivisa nei sotto-moduli Vision e Controller, si è dimostrata robusta, funzionale ed efficace. L'integrazione di tecnologie eterogenee come YOLO per l'*object detection*, i modelli UniDepth v2 e Depth Anything v2 per la stima della profondità, il formato ONNX per l'interoperabilità e il protocollo RTDE per il controllo del robot, ha validato la fattibilità dell'approccio. Il Controller, in particolare, ha dimostrato di avere un *overhead* computazionale e di memoria del tutto trascurabile. L'analisi delle *performance*, poi, ha rivelato che il sistema è fortemente *GPU-bound*, con il sotto-modulo di Vision che rappresenta il principale collo di bottiglia. Tra le varie configurazioni testate, quella bilanciata (YOLO *large* + Depth *small*) è emersa come il miglior compromesso tra accuratezza e velocità. UniDepth si è confermato il modello di stima della profondità più efficiente, spiccando in particolar modo nelle varianti a basso carico, grazie alla sua superiore capacità di saturare l'hardware grafico. Tuttavia, con una latenza media che si attesta intorno ai 200 ms, non sono stati raggiunti i vincoli effettivi previsti dal (*hard*) *real time*. Ciononostante, si ritiene che tale tempo di reazione sia ipoteticamente sufficiente per attivare protocolli di sicurezza non critici, come un arresto di emergenza di un robot in risposta a comportamenti anomali di un operatore.

Uno dei risultati più significativi è stata la scoperta dell'impatto negativo dell'*Execution Provider* TensorRT sui modelli di stima della profondità. Sebbene

garantisca un'accelerazione di circa il 30% per YOLO ed un 27% circa per i modelli di profondità (*large*), le sue ottimizzazioni aggressive si sono rivelate deleterie per l'integrità dei modelli più leggeri di *depth estimation*, producendo risultati corrotti e inaffidabili. Questa scoperta è cruciale, poiché evidenzia come un'ottimizzazione spinta, se non attentamente validata, possa compromettere la funzionalità di un sistema di sicurezza.

In definitiva, questo lavoro di tesi ha dimostrato con successo la validità di un sistema di sicurezza dinamico basato su *computer vision*. Le limitazioni prestazionali riscontrate non sono di natura concettuale, ma sono strettamente legate all'hardware impiegato, la piattaforma NVIDIA Jetson AGX Orin.

Le conclusioni tratte aprono la strada a numerosi e promettenti sviluppi futuri, volti a superare i limiti attuali e ad estendere le funzionalità del sistema.

La via più diretta per superare il *bottle neck* prestazionale è l'adozione di hardware di nuova generazione. Il nuovissimo NVIDIA Jetson AGX Thor, basato su architettura *Blackwell*, promette fino a 800 *TOPS* in INT8, offrendo un incremento prestazionale teorico di quasi 3 volte rispetto all'Orin. Ancora più importante, Thor introduce il supporto nativo per la precisione FP8. Questa potrebbe essere la soluzione alla criticità di TensorRT: l'FP8 offre una velocità paragonabile a INT8 ma con una maggiore robustezza numerica, che potrebbe preservare l'integrità dei complessi modelli di stima della profondità durante l'ottimizzazione. Questo permetterebbe di sfruttare pienamente l'accelerazione di TensorRT su tutta la pipeline[54, 55].

Oltre al passaggio ad hardware più prestante, si possono esplorare tecniche di ottimizzazione più aggressive ma controllate sia per il codice stesso, ad esempio utilizzando un linguaggio di più basso livello e apportando ottimizzazioni specifiche, ma anche relativamente ai modelli impiegati, come la quantizzazione o il *pruning* dei modelli, per ridurre ulteriormente la latenza e l'impronta di memoria sui dispositivi *edge*.

5. Conclusioni e sviluppi futuri

Per aumentare la robustezza del sistema e coprire aree di lavoro più ampie, uno sviluppo futuro cruciale è l'integrazione di *stream* video provenienti da più telecamere. La fusione dei dati 3D permetterebbe di gestire le occlusioni e di creare mappe di profondità più complete e affidabili dell'ambiente.

L'attuale Controller reagisce alla distanza istantanea, ma una logica più avanzata potrebbe implementare un tracciamento della traiettoria umana per prevederne i movimenti a breve termine. Questo permetterebbe al robot di reagire in modo più fluido e proattivo, rallentando gradualmente invece di fermarsi bruscamente.

Oltre a rilevare la presenza e la posizione, il sistema potrebbe essere esteso per riconoscere specifiche azioni o posture dell'operatore (quali persona a terra, movimenti anomali), abilitando risposte di sicurezza più contestualizzate.

In un'ottica più ampia, si potrebbero pensare integrazioni più raffinate e flessibili tra robot e comportamento umano, come ad esempio il cambio di *task* all'avvicinamento di un operaio o di una figura particolare (riconoscibile magari da un indumento o DPI particolare), oppure ancora l'adozione di determinate misure in base alle condizioni dinamiche in cui versa l'ambiente.

In conclusione, EdgeCV4Safety ha posto solide fondamenta. Gli sviluppi futuri qui delineati non solo promettono di superarne le attuali limitazioni, ma tracciano un percorso verso la realizzazione di sistemi di sicurezza per l'industria veramente intelligenti, adattivi e incentrati sull'uomo, in piena coerenza con i principi dell'Industria 5.0.

Bibliografia

- [1] Nahavandi S. Industry 5.0—A Human-Centric Solution. *Sustainability* 2019; 11: 4371.
- [2] Discovering the Promise of the Fifth Industrial Revolution in Education., <https://www.aiteachercourse.com/blog/discovering-the-promise-of-the-fifth-industrial-revolution-in-education> (accessed 11 November 2025).
- [3] Xu X, Lu Y, Vogel-Heuser B, et al. Industry 4.0 and Industry 5.0—Inception, conception and perception. *Journal of Manufacturing Systems* 2021; 61: 530–535.
- [4] Adel A. Future of industry 5.0 in society: human-centric solutions, challenges and prospective research areas. *J Cloud Comp* 2022; 11: 40.
- [5] Alves J, Lima TM, Gaspar PD. Is Industry 5.0 a Human-Centred Approach? A Systematic Review. *Processes* 2023; 11: 193.
- [6] Giallanza A, La Scalia G, Micale R, et al. Occupational health and safety issues in human-robot collaboration: State of the art and open challenges. *Safety Science* 2024; 169: 106313.
- [7] Hanna A, Larsson S, Götvall P-L, et al. Deliberative safety for industrial intelligent human–robot collaboration: Regulatory challenges and solutions for taking the next step towards industry 4.0. *Robotics and Computer-Integrated Manufacturing* 2022; 78: 102386.
- [8] Gabsi AEH. Integrating artificial intelligence in industry 4.0: insights, challenges, and future prospects—a literature review. *Ann Oper Res.* Epub ahead of print 8 May 2024. DOI: 10.1007/s10479-024-06012-6.
- [9] Silva DMAD, Sofia RC. A Discussion on Context-Awareness to Better Support the IoT Cloud/Edge Continuum. *IEEE Access* 2020; 8: 193686–193694.
- [10] Zhang T, Wang G, Xue C, et al. Time-Sensitive Networking (TSN) for Industrial Automation: Current Advances and Future Directions. *ACM Comput Surv* 2024; 57: 30:1-30:38.
- [11] TSN Cisco IE 4000 Switch - About Time-Sensitive Networking. *Cisco*, https://www.cisco.com/c/en/us/td/docs/switches/lan/cisco_ie4000/t_sn/b_tsn_ios_support/b_tsn_ios_support_chapter_01.html (accessed 15 October 2025).

- [12] Muzumdar P, Bhosale A, Basyal GP, et al. Navigating the Docker Ecosystem: A Comprehensive Taxonomy and Survey. *AJRCoS* 2024; 17: 42–61.
- [13] Computer Vision | IBM, <https://www.ibm.com/it-it/think/topics/computer-vision> (2021, accessed 16 October 2025).
- [14] Object Detection: Definizione, Modelli e Utilizzi | Ultralytics, <https://www.ultralytics.com/it/glossary/object-detection> (accessed 16 October 2025).
- [15] Zou Z, Chen K, Shi Z, et al. Object Detection in 20 Years: A Survey. *Proceedings of the IEEE* 2023; 111: 257–276.
- [16] Ming Y, Meng X, Fan C, et al. Deep learning for monocular depth estimation: A review. *Neurocomputing* 2021; 438: 14–33.
- [17] Yin W, Zhang C, Chen H, et al. Metric3D: Towards Zero-shot Metric 3D Prediction from A Single Image. pp. 9043–9053.
- [18] GigE Vision. *Automate*, <https://www.automate.org/vision/vision-standards/vision-standards-gige-vision> (accessed 15 October 2025).
- [19] USB3 Vision Standard. *Automate*, <https://www.automate.org/vision/vision-standards/usb3-vision-standard> (accessed 15 October 2025).
- [20] GenICam – EMVA, <https://www.emva.org/standards-technology/genicam/> (accessed 15 October 2025).
- [21] Aravis Project. Aravis, <https://github.com/AravisProject/aravis> (2025, accessed 15 October 2025).
- [22] Hussain M. YOLO-v1 to YOLO-v8, the Rise of YOLO and Its Complementary Nature toward Digital Manufacturing and Industrial Defect Detection. *Machines* 2023; 11: 677.
- [23] Ultralytics. YOLO11, <https://docs.ultralytics.com/it/models/yolo11> (accessed 15 October 2025).
- [24] Nielsen NH. niconielsen32/YOLO-3D, <https://github.com/niconielsen32/YOLO-3D> (2025, accessed 15 October 2025).
- [25] Yang L, Kang B, Huang Z, et al. Depth Anything V2. *Advances in Neural Information Processing Systems* 2024; 37: 21875–21911.
- [26] depth-anything (Depth Anything), <https://huggingface.co/depth-anything> (2025, accessed 19 October 2025).

- [27] Piccinelli L, Yang Y-H, Sakaridis C, et al. UniDepth: Universal Monocular Metric Depth Estimation. pp. 10106–10116.
- [28] Slepicka M, Helou J, Borrmann A. RTDE robot control integration for Fabrication Information Modeling. Chennai, India. Epub ahead of print 7 July 2023. DOI: 10.22260/ISARC2023/0016.
- [29] Imambi S, Prakash KB, Kanagachidambaresan GR. PyTorch. In: Prakash KB, Kanagachidambaresan GR (eds) *Programming with TensorFlow: Solution for Edge Computing Applications*. Cham: Springer International Publishing, pp. 87–104.
- [30] Piccinelli L. lpiccinelli-eth/UniDepth, <https://github.com/lpiccinelli-eth/UniDepth> (2025, accessed 26 October 2025).
- [31] ultralytics/ at main · tallal13/ultralytics. GitHub, <https://github.com/tallal13/ultralytics> (accessed 26 October 2025).
- [32] DepthAnything/Depth-Anything-V2, <https://github.com/DepthAnything/Depth-Anything-V2> (2025, accessed 26 October 2025).
- [33] EdgeCV4Safety/EdgeCV4Safety: AI-Driven Safety and Control for Human-Robot Collaboration in Industry 4.0/5.0, <https://github.com/EdgeCV4Safety/EdgeCV4Safety/tree/main> (accessed 29 October 2025).
- [34] EdgeCV4Safety/EdgeCV4Safety-Vision, <https://github.com/EdgeCV4Safety/EdgeCV4Safety-Vision> (2025, accessed 29 October 2025).
- [35] EdgeCV4Safety/EdgeCV4Safety-Controller, <https://github.com/EdgeCV4Safety/EdgeCV4Safety-Controller> (2025, accessed 29 October 2025).
- [36] UniversalRobots/RTDE_Python_Client_Library, https://github.com/UniversalRobots/RTDE_Python_Client_Library (2025, accessed 21 October 2025).
- [37] BI-REX - Chi Siamo. BI-REX, <https://bi-rex.it/chi-siamo/> (accessed 15 October 2025).
- [38] Basler. ace 2 a2A2448-23gcBAS | GigE Camera | Basler | Basler AG, <https://www.baslerweb.com/en/shop/a2a2448-23gcbas/> (accessed 15 October 2025).
- [39] DLAP-411-Orin | NVIDIA Jetson Edge AI Platform | ADLINK, https://www.adlinktech.com/Products/Deep_Learning_Accelerator_Pl

- tform_and_Server/Inference_Platform/DLAP-411-Orin (accessed 15 October 2025).
- [40] Karumbunathan LS. Nvidia jetson agx orin series. *A Giant Leap Forward for Robotics and Edge AI Applications Technical Brief*.
- [41] Automazione robotica collaborativa | Universal Robots Cobots, <https://www.universal-robots.com/it/> (accessed 15 October 2025).
- [42] e-Series, <https://www.universal-robots.com/it/prodotti/e-series/> (accessed 15 October 2025).
- [43] Image Layer Details - dustynv/l4t-pytorch:r36.4.0 | Docker Hub, <https://hub.docker.com/layers/dustynv/l4t-pytorch/r36.4.0/images/sha256-a05c85def9139c21014546451d3baab44052d7cabef854d937f163390bfd5201b> (accessed 5 November 2025).
- [44] Franklin D. Jetson Containers(Machine Learning Containers for Jetson and JetPack), <https://github.com/dusty-nv/jetson-containers> (2025, accessed 26 October 2025).
- [45] CUDA C++ Programming Guide — CUDA C++ Programming Guide, <https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html> (accessed 7 November 2025).
- [46] CUDA Deep Neural Network. *NVIDIA Developer*, <https://developer.nvidia.com/cudnn> (accessed 7 November 2025).
- [47] CUDA Toolkit Documentation 13.0 Update 2, <https://docs.nvidia.com/cuda/index.html> (accessed 7 November 2025).
- [48] FIGURE 1. CUDA Kernel Execution on the GPUs in Different Platforms. *ResearchGate*, https://www.researchgate.net/figure/CUDA-Kernel-Execution-on-the-GPUs-in-Different-Platforms_fig1_347824334 (accessed 7 November 2025).
- [49] End-to-End AI for NVIDIA-Based PCs: CUDA and TensorRT Execution Providers in ONNX Runtime. *NVIDIA Technical Blog*, <https://developer.nvidia.com/blog/end-to-end-ai-for-nvidia-based-pcs-cuda-and-tensorrt-execution-providers-in-onnx-runtime/> (2023, accessed 7 November 2025).
- [50] B GP, Marion Lincy G R, Rishekeeshan A, et al. Accelerating Native Inference Model Performance in Edge Devices using TensorRT. In: *2024 IEEE Recent Advances in Intelligent Computational Systems (RAICS)*, pp. 1–7.

- [51] Speeding Up Deep Learning Inference Using TensorFlow, ONNX, and NVIDIA TensorRT. *NVIDIA Technical Blog*, <https://developer.nvidia.com/blog/speeding-up-deep-learning-inference-using-tensorflow-onnx-and-tensorrt/> (2021, accessed 7 November 2025).
- [52] Real-Time Data Exchange (RTDE) Guide — PolyScope Tutorials documentation, <https://docs.universal-robots.com/tutorials/communication-protocol-tutorials/rtde-guide.html> (accessed 26 October 2025).
- [53] justwhitee/EdgeCV4Safety-Models at main, <https://huggingface.co/justwhitee/EdgeCV4Safety-Models/tree/main> (2025, accessed 7 November 2025).
- [54] NVIDIA Blackwell Platform Arrives to Power a New Era of Computing. *NVIDIA Newsroom*, <https://nvidianews.nvidia.com/news/nvidia-blackwell-platform-arrives-to-power-a-new-era-of-computing> (accessed 8 November 2025).
- [55] Kani A. NVIDIA DRIVE Thor Strikes AI Performance Balance, Uniting AV and Cockpit on a Single Computer. *NVIDIA Blog*, <https://blogs.nvidia.com/blog/drive-thor/> (2022, accessed 8 November 2025).

Voglio dedicare queste pagine a chi mi ha supportato nella scrittura di questa tesi e lungo tutto il percorso che mi ha portato fin qui, rendendo possibile questo traguardo.

Ringrazio il professor Bellavista, che mi ha dato spunti interessanti e stimolanti, facendomi fare nuove esperienze che saranno utilissime per il mio futuro.

Ringrazio il dottor Venanzi, che in questi mesi di tirocinio e tesi è stato sempre disponibile e pronto a darmi le indicazioni giuste in ogni momento.

Un ringraziamento speciale va al tutor Tazzioli, che mi ha affiancato e aiutato nelle ricerche e negli esperimenti con un entusiasmo, una pazienza e una curiosità che sono stati di grande ispirazione.

Ringrazio BI-REX e tutto il suo staff, per la grande opportunità (e la pazienza) che mi hanno offerto, ospitandomi per il tirocinio e permettendomi di svolgere tutti i test necessari.

Ringrazio i miei genitori, perché senza di loro non avrei mai potuto iniziare e concludere questo percorso. Grazie per il vostro supporto e per non aver mai smesso di credere in me. Grazie per tutti i sacrifici che avete fatto (e continuate a fare). Grazie perché in tutti questi anni avete sempre esaudito i miei desideri, accompagnandomi sulla mia strada, e avete investito in me, anche senza capire fino in fondo dove andasse ogni centesimo.

Un grazie particolare, però, va a te, papà. Perché nonostante tutto hai lottato per rimanere qui con noi, e so che, in ogni caso, ci saresti stato e ci sarai sempre.

Vi voglio bene.

Un ringraziamento speciale, sicuramente uno dei più importanti di tutti, va a te, "Saro". Tu, che hai vissuto questo percorso con me ogni giorno, seguendone ogni passo. Grazie per avermi sempre capito e incoraggiato. Sei stata e sei il mio posto sicuro. Sei la ragione per cui io stesso ci ho creduto davvero, soprattutto nei momenti bui.

Grazie di esserci sempre.

Ringraziamenti

Ringrazio la mia "tata", Mirco e anche le piccole Gioia e Cloe per la gioia e il calore sempre presente, per esserci stati superando la distanza e ogni imprevisto.

Ringrazio Alessio, che nonostante il poco tempo passato insieme, c'è sempre stato, dall'inizio e come sempre, come un vero amico.

Ringrazio i miei "colleghi" (e amici, circa), per avermi fatto uscire dalla mia bolla (meglio tardi che mai) e aver alleggerito le lunghe ore di lezione e studio con le loro risate. Se state leggendo, non pensateci neanche (sì, proprio a *quello*).

Infine, un ringraziamento a tutti coloro che, in un modo o nell'altro, spesso indirettamente, hanno contribuito a questo traguardo. Citando solo qualcuno: menzione d'onore al professor Chesani "dalla folta criniera", per avermi trasmesso da subito la curiosità e la passione per questo Corso; grazie ad Antonio, per aver reso il suo ristorante Sapporo il mio insostituibile posto felice in cui staccare e ricaricarmi nelle lunghe sessioni; menzione speciale per Gemini e ChatGPT per aver risposto (male) anche alle domande più futili; grazie anche a qualche sconosciuto youtuber indiano (e non) per i suoi provvidenziali video.

Grazie a tutti voi.

* Di certo non si ringraziano Tper (32 e 33 in particolare), Trenitalia, gli "accertamenti dell'autorità giudiziaria", Lord Farquaad, la celiachia e tutte le mie altre patologie.

