ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

SCUOLA DI SCIENZE

Dipartimento di Informatica - Scienza e Ingegneria

Corso di Laurea in Informatica

Embedded Object Detection su ESP32-S3: ottimizzazione e deployment per l'Edge

Relatore:
Prof. Luciano Bononi

Co-Relatore:
Dott. Giorgio Tsiotas

Tesi di laurea di:
Alessio Prato

ANNO ACCADEMICO 2024-2025

SESSIONE II

Abstract

Tradizionalmente, dispositivi a ridotta capacità computazionale come i microcontrollori sono stati impiegati principalmente per la raccolta e la trasmissione di dati, delegando l'elaborazione intensiva a computer o infrastrutture esterne. Questa tesi esplora la possibilità di eseguire l'inferenza di modelli di *deep learning* direttamente a bordo di un microcontrollore, nello specifico un ESP32-S3 dotato di camera integrata. L'obiettivo è quello di trasformare il microcontrollore da semplice nodo di acquisizione ad un possibile sistema autonomo di percezione, capace di rilevare oggetti in tempo reale senza dipendere da risorse esterne. A tal fine, viene presentato un flusso di lavoro che comprende la creazione del firmware per interagire con il microcontrollore e la sua camera, l'adattamento architetturale e il *porting* di un modello *state-of-the-art* di *Object Detection*, appartenente alla famiglia You Only Look Once. Attraverso tecniche come la quantizzazione, *model scaling* e *fine-tuning* per *domain adaptation*, è stato possibile bilanciare l'accuratezza del modello con le limitazioni computazionali dell'ambiente *embedded*. I risultati, valutati tramite metriche come l'*Average Precision* (AP) e il tempo di inferenza, mostrano la fattibilità di eseguire modelli di visione artificiale complessi direttamente sull'Edge, aprendo la strada a scenari di applicazione leggeri, indipendenti e a basso consumo.

Indice

Capitolo 1

Introduzione

Negli ultimi anni, campi come il *Deep Learning* e la *Computer Vision* hanno conosciuto una crescita straordinaria, sia dal punto di vista scientifico che economico, influenzando in modo sempre più profondo la vita delle persone. Il progresso tecnologico ha portato non solo alla nascita di modelli sempre più complessi e performanti, ma anche allo sviluppo di architetture più leggere ed efficienti, capaci di svolgere gli stessi compiti con un minor consumo di risorse computazionali ed energetiche.

Questa evoluzione ha favorito lo sviluppo di nuovi paradigmi come l'Edge AI, il settore dedicato all'esecuzione di modelli di intelligenza artificiale direttamente su dispositivi locali, senza la necessità di un'infrastruttura esterna. Le motivazioni che spingono verso questa direzione sono numerose. In primo luogo quello economico, poiché la possibilità di eseguire modelli anche su hardware a basso costo, come i microcontrollori, elimina molte barriere d'accesso, sia nella ricerca che nell'industria. In secondo luogo, l'indipendenza dalla rete e la riduzione della latenza, rendono i sistemi più affidabili e adatti a scenari *real-time*, grazie alla vicinanza fisica tra il sensore e l'unità di elaborazione. Un ulteriore vantaggio è rappresentato dalla *privacy* dei dati, che rimangono all'interno del dispositivo invece di essere trasmessi a server remoti. Infine, l'efficienza energetica costituisce un aspetto sempre più rilevante: l'esecuzione di modelli diret-

tamente in locale consente di ridurre drasticamente i consumi, rispetto ai grandi *data center* richiesti per il calcolo centralizzato.

In questo contesto, la sfida principale diventa quindi quella di rendere eseguibili modelli di *deep learning* complessi su hardware estremamente limitato, mantenendo un equilibrio tra accuratezza, prestazioni e costi computazionali.

Tra i tre capitoli che seguono, nel primo vengono presentati i concetti teorici e il contesto generale su cui si basa il lavoro. Nel secondo viene descritto l'hardware utilizzato, gli strumenti di sviluppo e il firmware realizzato per gestire il microcontrollore. Il terzo e ultimo capitolo è infine dedicato al *deployment* e all'ottimizzazione di un modello di *Object Detection* sul dispositivo, con una valutazione delle prestazioni ottenute.

Capitolo 2

Background

2.1 Deep Learning

L'intelligenza artificiale (AI) indica la capacità di una macchina di svolgere compiti che richiedono forme di ragionamento, percezione o decisione tipiche dell'intelligenza umana. Possiamo individuare un primo approccio classico all'intelligenza artificiale con la *Symbolic AI*, nella quale il comportamento di un sistema è descritto attraverso regole deterministiche. Al contrario, si parla di *Machine Learning* (ML) quando un sistema è in grado di apprendere automaticamente schemi o relazioni a partire dai dati, senza essere esplicitamente programmato per ogni singola operazione. Esempi classici di ML includono modelli come i *Decision Trees* e le *Support Vector Machines* (SVM). Il *Deep Learning* (DL), sottoinsieme del *Machine Learning*, utilizza reti neurali profonde, modelli composti da numerosi *hidden layers* interconnessi, ispirate in modo semplificato al funzionamento del cervello umano.

Attualmente, il *Deep Learning* rappresenta una delle più importanti conquiste dell'informatica moderna. A differenza dei metodi tradizionali, le reti neurali profonde apprendono direttamente dalla materia grezza dell'informazione, costruendo autonomamente rappresentazioni gerarchiche

sempre più astratte. La potenza del Deep Learning deriva proprio dalla profondità delle reti: la combinazione di molte trasformazioni non lineari permette di modellare relazioni estremamente complesse tra input e output, risolvendo problemi che fino a pochi anni fa erano considerati inaccessibili[1].

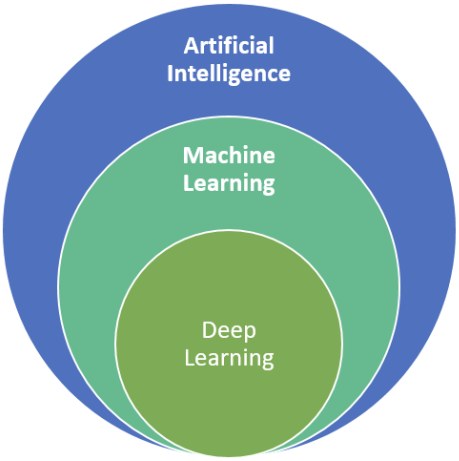


Figura 2.1: Relazione tra *Artificial Intelligence*, *Machine Learning* e *Deep Learning*

2.1.1 Reti Neurali Artificiali

Le reti neurali artificiali (*Artificial Neural Networks*) costituiscono il modello computazionale alla base del *Deep Learning*. Le unità fondamentali di queste reti sono i neuroni artificiali, astrazioni matematiche ispirate in modo semplificato ai neuroni biologici. Ciascuno di questi nodi della rete riceve in input un insieme di valori (solitamente rappresentati da un tensore), calcola una combinazione lineare degli input ponderata dai relativi pesi, aggiunge un termine di bias e applica infine una funzione di attivazione tipicamente non lineare. Formalmente, l'output di un neurone artificiale è definito come:

$$y = \phi\left(\sum_{i=1}^{d} w_i x_i + b\right) \tag{2.1}$$

dove:

- $x_1, x_2, \ldots, x_d$ sono gli input del neurone,
- $w_1, w_2, \ldots, w_d$ sono i pesi associati a ciascun input,
- b è il bias,
- ϕ è la funzione di attivazione.

I valori w_i e b di ogni neurone, costituiscono i parametri della rete, ovvero grandezze che vengono adattate e ottimizzate durante la fase di addestramento (*training*). In una rete neurale artificiale, i neuroni sono organizzati in *layer* (strati), generalmente classificabili in *input layer*, *hidden layer* e *output layer*. L'*input layer* riceve i dati in ingresso, gli *hidden layer* costituiscono i livelli intermedi responsabili dell'elaborazione delle rappresentazioni interne, mentre l'*output layer* produce la predizione finale.

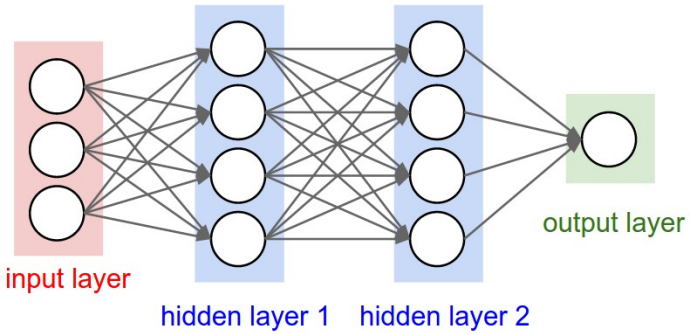


Figura 2.2: *Artificial Neural Network*

Si definisce Rete Neurale Profonda (*Deep Neural Network*) una rete che possiede più di un *hidden layer* ($n_{\text{hidden}} > 1$), in caso contrario si parla di Rete Neurale Superficiale (*Shallow Neural Network*). Le *Deep Neural Networks* sono i modelli che caratterizzano e rendono oggi possibile il *Deep Learning*.

2.1.2 Processo di apprendimento

L'apprendimento (*training*) rappresenta una fase cruciale nel funzionamento delle reti neurali. Esso consiste nell'esporre la rete a un ampio dataset, costituito da dati etichettati o non etichettati, con l'obiettivo di ottimizzare i parametri della rete, al fine di ridurre l'errore tra l'output predetto dal modello e il valore reale. Il processo di *training* si articola in molteplici epoche, ovvero cicli durante i quali la rete elabora l'intero *training set*. Un input x attraversa i vari layer della rete fino a raggiungere quello di output, in un processo denominato *forward pass*, producendo una predizione y.

L'errore, ossia la differenza tra la predizione y e il valore effettivo Y, viene quantificato mediante una *loss function*, la quale restituisce un valore da minimizzare. Successivamente, l'algoritmo di *backpropagation* calcola i gradienti di questa funzione di perdita rispetto ai parametri della rete, propagando l'errore all'indietro attraverso i vari *layer*. Tali gradienti, infine, sono poi utilizzati dall'algoritmo di ottimizzazione (*optimizer*) per aggiornare i valori dei parametri, secondo una regola di discesa del gradiente:

$$\theta^{(t+1)} = \theta^{(t)} - \eta \nabla_\theta \mathcal{L}(\theta^{(t)}) \tag{2.2}$$

dove:

- $\theta^{(t)}$ indica il vettore dei parametri alla t-esima iterazione (*batch*) di addestramento
- η è il learning rate, un iperparametro arbitrario che costituisce il passo di aggiornamento
- $\nabla_\theta \mathcal{L}(\theta^{(t)})$ è il gradiente della funzione di perdita rispetto ai parametri, calcolato nella configurazione corrente.

Parallelamente, durante o al termine di ogni epoca, le prestazioni del modello vengono valutate su un insieme di dati separato, detto *validation set*. Quest'ultimo non partecipa all'aggiornamento dei parametri, e

non viene "visto" dalla rete, ma serve a misurarne la capacità di generalizzazione su dati non osservati, in modo da monitorare fenomeni come l'*overfitting*, ossia l'eccessivo adattamento ai dati di addestramento.

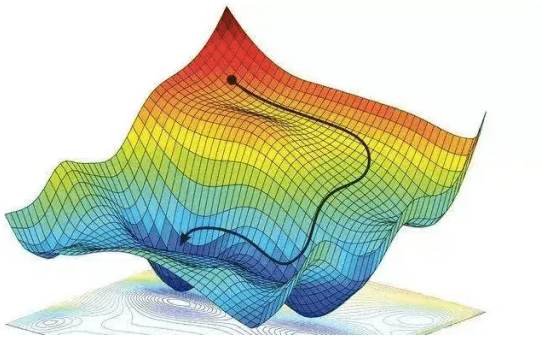


Figura 2.3: Discesa del Gradiente

In base a come è strutturato il dataset, parliamo di apprendimento supervisionato se i dati sono associati a etichette, oppure di apprendimento non supervisionato o semi-supervisionato in caso contrario. Queste diverse modalità influenzano profondamente la natura della *loss function* e il modo in cui il modello apprende dalle informazioni disponibili.

2.2 Computer Vision

La *computer vision* è un campo dell'intelligenza artificiale che si occupa di permettere ai computer di "vedere" il mondo, analizzando ed estraendo informazioni da immagini e video. Sebbene le sue origini risalgano a metodi di elaborazione delle immagini e tecniche geometriche indipendenti dal *machine learning*, negli ultimi anni le reti neurali profonde hanno rivoluzionato il settore, offrendo prestazioni senza precedenti in numerosi problemi di questo ambito.

Tra i principali problemi affrontati dalla *computer vision* moderna troviamo:

- **Image Classification**: assegnare ad un'intera immagine una o più etichette che ne descrivano il contenuto.

- **Object Detection**: individuare e localizzare oggetti di interesse all'intero di un'immagine tramite *bounding boxes*.
- **Image Segmentation**: classificare ogni singolo pixel all'interno di un'immagine, in base all'oggetto o regione a cui appartiene.
- **Pose Estimation** e **Keypoint Regression**: stimare la posizione e l'orientamento di strutture articolate (come il corpo umano) identificando punti chiave (*keypoints*) caratteristici.

2.2.1 Convolutional Neural Networks

Le reti neurali convoluzionali (*Convolutional Neural Networks*, CNN) sono reti neurali progettate per elaborare dati con struttura spaziale o temporale, come immagini, video o segnali [2]. Basate sull'operazione di convoluzione, che permette di estrarre automaticamente caratteristiche locali dal dato in input, le CNN costituiscono oggi la base di molte architetture moderne per la *computer vision*.

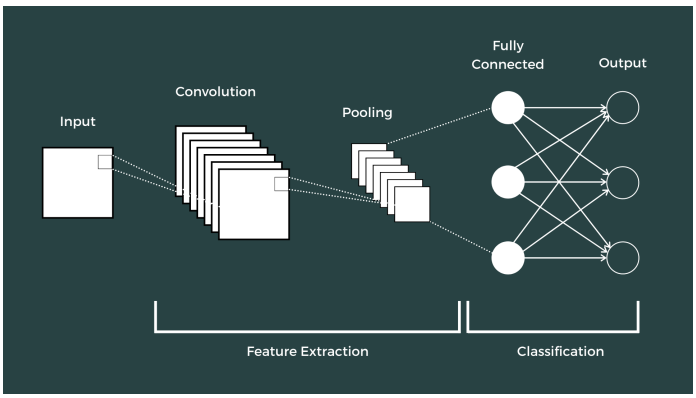


Figura 2.4: *Convolutional Neural Network*

I *layer* convoluzionali costituiscono l'elemento distintivo di questa architettura. Quando un input attraversa un layer convoluzionale, su di esso viene applicato un filtro (*kernel*), rappresentato da una piccola matrice di pesi che "scorre" sull'input con un determinato passo (*stride*). Questa operazione, detta convoluzione, consente di estrarre automaticamente

caratteristiche locali (*features*), come bordi, angoli o *texture* specifiche. L'output di tale operazione è una mappa di attivazione (*feature map*), che indica la presenza e la posizione di una specifica *feature* nell'input. Inoltre, per controllare le dimensioni spaziali dell'output e preservare le informazioni ai margini, è comune applicare il *padding*, consistente nell'aggiunta di pixel (solitamente con valore nullo) ai bordi dell'input.

La convoluzione 2D, nel contesto del *deep learning*, può essere formalmente descritta come:

$$(I * K)(x, y) = \sum_{i=0}^{m-1} \sum_{j=0}^{n-1} K(i, j) \cdot I(x + i, y + j) \tag{2.3}$$

dove I è l'immagine in input e K il *kernel* di dimensione m×n.

Un altro *layer* fondamentale è quello di *pooling*, che riduce progressivamente la risoluzione spaziale delle *feature maps* mantenendo le informazioni salienti (spesso scegliendo il valore massimo, *max pooling*) e diminuendo il carico computazionale dei layer successivi. Questa operazione aiuta anche a rendere il modello più robusto contro piccole variazioni o traslazioni nell'input.

Nella parte finale della rete, il *fully connected layer* (o *dense layer*) collega tutti i neuroni tra loro e aggrega le *features* estratte per produrre la predizione finale. Storicamente utilizzato nei modelli di classificazione, è oggi spesso sostituito o integrato da soluzioni più efficienti, come le *Fully Convolutional Networks*, che consentono al modello di accettare in input immagini di qualsiasi dimensione, aumentando la flessibilità e migliorando le prestazioni in termini di velocità.

2.2.2 You Only Look Once

Introdotto per la prima volta nel 2015 da Joseph Redmon, *You Only Look Once* (YOLO) [3] è una famiglia di architetture per l'*object detection* in tempo reale. Questa classe di modelli ha migliorato significa-

tivamente le performance in termini di tempo di inferenza rispetto agli approcci precedenti e, in versioni successive, è stata estesa anche ad altri compiti, come la segmentazione semantica. L'innovazione principale di YOLO, come suggerisce il nome, consiste nel prevedere simultaneamente *bounding boxes* e classi in un unico passaggio, mediante una rete neurale convoluzionale, trattando l'*object detection* come un problema di regressione.

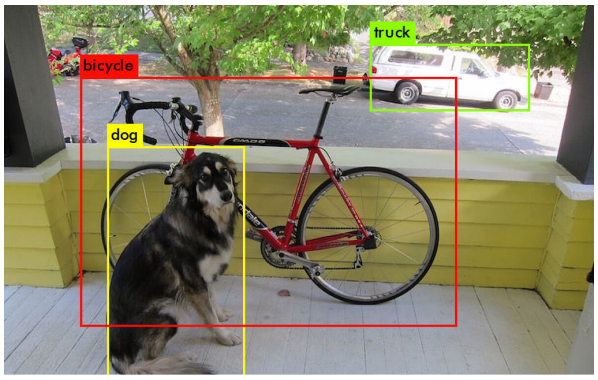


Figura 2.5: Bounding boxes predette da YOLOv1

Quando YOLO esegue un'inferenza, l'intera immagine viene analizzata in un unico passaggio, individuando le zone in cui è più probabile che si trovino gli oggetti di interesse. L'immagine viene divisa in una griglia S *x* S e, per ogni cella della griglia, vengono predette N *bounding boxes* (aree di forma rettangolare) con associate un *confidence score*, che indica quanto il modello è sicuro che la *bounding box* contenga un oggetto e sia posizionata correttamente, e le relative probabilità condizionali per ogni classe. Nonostante YOLO sia relativamente giovane, l'architettura ha subito negli ultimi anni numerose nuove versioni ottimizzate ed è oggi mantenuto dell'azienda *Ultralytics*.

2.3 Sistemi Embedded

Un sistema *embedded* è un sistema di calcolo integrato all'interno di un sistema più ampio, progettato per svolgere una o più funzioni specifiche. Diversamente dai computer di largo consumo, i sistemi *embedded*

sono tipicamente soggetti a vincoli di costo, consumo energetico, spazio, affidabilità e, quando richiesto, a vincoli *real-time* legati a latenza e determinismo. Grazie alla loro efficienza e affidabilità, i sistemi *embedded* trovano applicazione in numerosi settori, tra cui l'*automotive*, il medicale, l'industria e l'elettronica di consumo. Un tipico sistema *embedded* può essere costituito da un microcontrollore (*Microcontroller Unit*, MCU), un microprocessore o *System-on-Chip* (*Microprocessor Unit*, MPU / SoC), un *Digital Signal Processor* (DSP), una FPGA, oppure un computer a scheda singola (*Single Board Computer*, SBC), a seconda dei requisiti di progetto e del compito da svolgere.

2.3.1 Microcontrollori

I microcontrollori rappresentano spesso il cuore dei sistemi embedded. Questi chip tipicamente integrano su un unico substrato di silicio i principali componenti di un computer, come la CPU, la memoria volatile (RAM) e la memoria non volatile (Flash); in alcuni casi, tuttavia, parte della memoria può essere esterna. Gli MCU dispongono inoltre di numerose periferiche di input/output, come i GPIO (*General Purpose Input/Output*), e di interfacce e bus di comunicazione seriale come I^2C, SPI, UART, CAN, USB, Ethernet e altre, le quali consentono l'interazione con sensori, attuatori e altri dispositivi.

Su tali dispositivi può essere caricato (*flashato*) del codice, solitamente scritto in un linguaggio di programmazione come C, C++ o MicroPython per eseguire funzioni dedicate, e possono operare con o senza un sistema operativo. In particolare, nei contesti dove è necessario garantire tempi di risposta deterministici, vengono impiegati sistemi operativi real-time (*Real-Time Operating Systems*, RTOS). Un RTOS è un sistema operativo progettato per garantire che le operazioni vengano eseguite entro vincoli di tempo rigorosi e prevedibili. È impiegato soprattutto in contesti *safety-critical*, dove la correttezza del sistema non dipende solo dal risultato delle elaborazioni, ma anche dal momento in cui queste vengono completate. In questi sistemi operativi specifici, la creazione, la pianifi-

cazione e la terminazione dei processi avvengono in modo deterministico, assicurando tempi di risposta costanti e controllabili.

2.4 Edge AI

Con il termine *Edge AI* si indica l'esecuzione di operazioni di intelligenza artificiale, *machine learning* o *deep learning* direttamente su dispositivi locali (*on the edge*), ossia al bordo della rete. Tradizionalmente, si è sempre ritenuto che i modelli di *deep learning* richiedessero ingenti risorse computazionali per poter essere eseguiti in modo efficace ed efficiente. Oggi tale ipotesi rimane valida solo in parte: l'evoluzione delle reti neurali non ha mirato unicamente al miglioramento dell'accuratezza, ma anche all'ottimizzazione dell'efficienza, con l'obiettivo di ottenere prestazioni comparabili riducendo la complessità computazionale. Questa tendenza ha reso possibile l'esecuzione di inferenze di modelli di *deep learning* anche su dispositivi *embedded* [4]. Fino a pochi anni fa, microcontrollori e sistemi simili venivano impiegati esclusivamente per attività elementari, come la lettura di sensori o il controllo di attuatori, delegando i calcoli complessi a computer esterni o a infrastrutture *cloud*. Quando invece si punta a implementare modelli di *deep learning* su dispositivi con risorse estremamente limitate, si entra nell'ambito del *Tiny Machine Learning* (*TinyML*), una branca dell'Edge AI focalizzata sulla miniaturizzazione e ottimizzazione dei modelli per consentirne il *deploy* su hardware con capacità computazionali ridotte.

2.4.1 Tecniche di compressione

L'impiego di tecniche di compressione delle reti neurali è essenziale per rendere l'inferenza eseguibile su dispositivi con risorse limitate, migliorando l'efficienza computazionale e riducendo l'occupazione di memoria, con l'obiettivo di ridurre al minimo la perdita di accuratezza [5].

Scaling e Pruning

Entrambe le tecniche intervengono direttamente sulla struttura della rete neurale. Lo *scaling* architetturale agisce a livello di progettazione del modello, riducendone la larghezza (ad esempio, diminuendo il numero di canali nei *layer* convoluzionali), la profondità (riducendo il numero totale di *layer*) o, per i modelli di visione, l'*image size* dell'input *layer*. Il *pruning*, invece, consiste nella rimozione selettiva di connessioni, neuroni o filtri ritenuti poco significativi, con l'obiettivo di diminuire la complessità computazionale mantenendo prestazioni comparabili. Nonostante la sua efficacia, il pruning risulta più complesso da applicare in modelli già ben ottimizzati o con layer ricchi di dipendenze e non sequenziali.

Quantizzazione

La *quantizzazione* è una delle tecniche più diffuse per ridurre il peso e il costo computazionale di una rete neurale, cercando di minimizzare la perdita di accuratezza. Essa consiste nel rappresentare i parametri della rete, originariamente espressi in FLOAT32 (virgola mobile a singola precisione, 4 byte per valore), mediante formati numerici a minore precisione, come FLOAT16 o INT8. È anche possibile adottare una quantizzazione ibrida mantenendo, per esempio in FLOAT32, i parametri più sensibili per ridurre l'impatto sulla qualità del modello.

In base alla fase in cui viene applicata, la quantizzazione può assumere forme differenti:

- **Post-Training Quantization (PTQ):** applicata dopo l'addestramento del modello. Può essere:
  - *Statica*, quando effettuata prima del *deploy* del modello, attuando la quantizzazione di pesi e attivazioni tramite una fase di calibrazione su un *dataset* rappresentativo. Questo approccio consente di ridurre il tempo di inferenza e la memoria occupata, a fronte di possibili perdite di accuratezza.

  - *Dinamica*, quando le attivazioni vengono quantizzate e dequantizzate durante l'inferenza, riducendo la necessità di calibrazione preventiva e la perdita di accuratezza rispetto alla quantizzazione statica. Tuttavia, questo approccio offre benefici computazionali inferiori, a causa delle conversioni a *runtime*.

- **Quantization-Aware Training (QAT):** la quantizzazione è integrata direttamente nel processo di addestramento, permettendo alla rete di adattarsi alle approssimazioni introdotte e, in genere, di preservare meglio l'accuratezza a parità di *bit-width*.

Nel caso della rappresentazione in INT8, la più usata in contesti come il *deploy* su microcontrollori, i valori reali vengono mappati nell'intervallo $[-128, 127]$ mediante una trasformazione affine:

$$r = S(q - Z) \tag{2.4}$$

dove:

- r è il valore reale in FLOAT32,
- q è il valore quantizzato in INT8,
- S è il fattore di scala (*scaling factor*),
- Z è lo *zero-point*, ossia l'offset che rappresenta lo zero in forma quantizzata.

Durante la quantizzazione statica, i parametri S e Z vengono determinati per ogni layer attraverso un processo di calibrazione, basato su un dataset di input rappresentativo del dominio reale di utilizzo. La qualità di questo dataset risulta quindi cruciale per minimizzare la perdita di accuratezza complessiva del modello.

Capitolo 3

Architettura Hardware e Software del progetto

3.1 Architettura Hardware

L'hardware utilizzato in questo progetto consiste in una scheda di sviluppo personalizzata (*custom development board*), realizzata dall'azienda *DF Robot* e basata sul microcontrollore ESP32-S3 progettato da *Espressif Systems*. Questa piccola scheda da 42 x 42 mm, è stata scelta per la presenza di una camera integrata, per il supporto di accelerazione vettoriale, e per l'estensione della PSRAM, elementi cruciali per il raggiungimento degli obiettivi del progetto.

3.1.1 ESP32-S3

Il cuore di questa scheda, come anticipato, è il *system-on-chip* ESP32-S3, che integra un processore *dual-core* Xtensa LX7 a 32 bit, capace di raggiungere una frequenza di clock fino a 240 MHz. Il *chip* dispone di 512 KB di SRAM (*Static Random Access Memory*), una memoria ad alta velocità utilizzata principalmente per lo *stack* e l'*heap* del codice in esecuzione. In quest'ultima, sono inoltre presenti 16 KB di RTC RAM (*Real-Time Clock RAM*), una sezione di memoria dedicata al mantenimento dei dati durante la modalità di *deep sleep*. La *custom board* integra anche 8 MB di PSRAM (*Pseudo-static RAM*) esterna al *chip*, una soluzione economica ma essenziale per la gestione di grandi *buffer* di dati, come pesi e tensori dei modelli di *deep learning*.

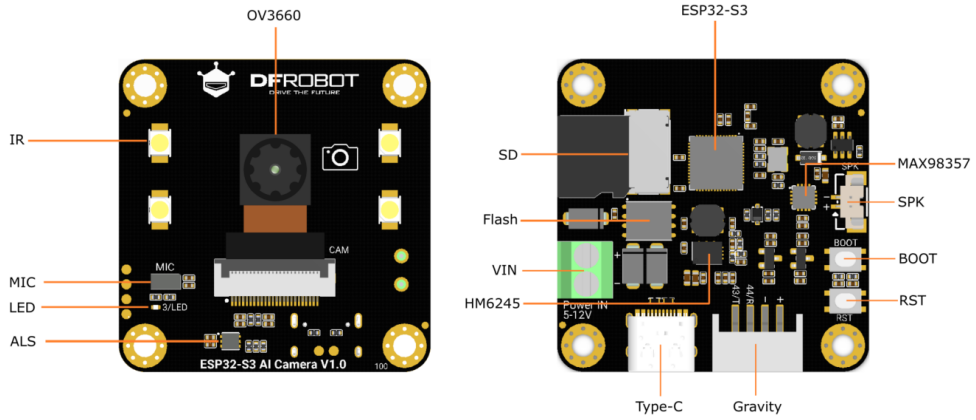


Figura 3.1: ESP32-S3 Custom Board

Per quanto concerne la connettività, il microcontrollore integra un modulo Wi-Fi 2.4 Ghz e un modulo Bluetooth Low Energy 5.0, che consentono la comunicazione *wireless* con altri dispositivi e reti locali. La scheda non contiene unità di calcolo dedicate all'inferenza, come NPU o TPU, ma il chip S3 integra un set esteso di istruzioni pensato per migliorare l'efficienza operativa di specifici algoritmi di *deep learning* e *digital signal processing* su numeri *fixed-point* [6]. Basato sul paradigma SIMD (*Single Instruction Multiple Data*), questo set di istruzioni supporta operazioni vettoriali a 8 bit, 16 bit e 32 bit, aumentando significativamente la velo-

cità di elaborazione parallela sui dati interi. In particolare, le istruzioni aritmetiche, come la moltiplicazione, lo *shifting* e l'accumulazione, possono eseguire operazioni e trasferimenti sui dati contemporaneamente, aumentando così l'efficienza di esecuzione delle singole istruzioni.

Tra le periferiche aggiuntive fornite dalla *custom board*, la più importante è la camera integrata OV3660, dotata di una risoluzione di 3 Megapixels e di un *Field of View* (FOV) di 160°. Ad essa sono affiancati inoltre due LED, di cui uno dedicato al flash. La scheda include anche una porta USB Type-C per il *flashing* del codice, un lettore di schede SD per l'espansione della memoria non volatile, e i bottoni di BOOT e RESET. È inoltre integrato nella scheda un microfono e un connettore *Gravity*, non utilizzati in questo progetto, ma potenzialmente utili per sviluppi futuri. Un aspetto limitante di questa scheda, è la disponibilità ridotta di GPIO liberi: la maggior parte di essi è riservata alla connessione con la camera, lasciando utilizzabili solo i quattro pin del connettore *Gravity* (due di alimentazione, uno di trasmissione TX e uno di ricezione RX). Ciò può rappresentare un vincolo qualora si vogliano collegare fisicamente diversi dispositivi esterni.

3.1.2 Configurazione e ottimizzazione hardware

Sono state apportate alcune modifiche alle impostazioni della scheda tramite il menu di configurazione accessibile da terminale, principalmente per soddisfare i requisiti del progetto e rimuovere alcune limitazioni di fabbrica. La prima modifica ha riguardato l'aumento della frequenza di *clock* della CPU, portata al valore massimo di 240 MHz per sfruttare appieno le potenzialità di calcolo del microcontrollore, inizialmente limitate ai 160 MHz, presumibilmente per motivi di consumo energetico o affidabilità. È stata poi abilitata la PSRAM, disattivata di default: in assenza di essa, l'esecuzione del modello sarebbe stata vincolata alla sola SRAM, più veloce ma di capacità molto inferiore, impedendo di allocare i tensori necessari all'inferenza. Inoltre, il tipo di interfaccia seriale utilizzata per il *monitor* è stato modificato da UART0 a USB CDC, consenten-

do l'invio di input da tastiera direttamente da terminale, e migliorando l'interazione con la scheda in fase di test. Infine, è stata sbloccata la memoria *flash*, estendendo lo spazio accessibile dai 4 MB iniziali ai 16 MB disponibili, così da poter memorizzare oltre al firmware, anche i file binari dei modelli.

3.2 Framework e Strumenti

Di seguito, vengono descritti gli strumenti, i *framework* e le librerie che hanno fatto parte dello sviluppo, dagli strumenti verticali sull'*embedded*, fino ai *framework* utilizzati per lo svolgimento degli esperimenti.

3.2.1 Toolchain Espressif

ESP-IDF

Per lo sviluppo del software da *flashare* sul microcontrollore è stato utilizzato ESP-IDF (*Espressif IoT Development Framework*), il framework ufficiale fornito da *Espressif Systems* per la programmazione dei loro microcontrollori[7]. Questo ambiente di sviluppo è stato preferito al più diffuso Arduino IDE per la maggiore granularità e controllo che offre sul comportamento dell'hardware, sulla gestione dei thread e per la compatibilità diretta con le librerie di inferenza.

ESP-IDF fornisce un sistema di *build* basato su CMake, Ninja e GCC, il quale consente lo sviluppo del firmware in linguaggi come C o C++. Include driver per la gestione delle periferiche integrate, in particolare per la fotocamera utilizzata nel progetto, e stack software per il supporto ai principali protocolli di connettività come Wi-Fi e Bluetooth. È dotato inoltre di un *component manager* per la gestione automatica delle dipendenze, ed è caratterizzato dall'essere basato su FreeRTOS.

FreeRTOS

FreeRTOS è un sistema operativo *real-time open source* progettato per microcontrollori e integrato nativamente all'interno di ESP-IDF [8]. Esso è costituito da un'implementazione estremamente leggera, scritta quasi interamente in C con pochissime porzioni in *assembly*, e consente di gestire in modo efficiente lo *scheduling* e la sincronizzazione dei *tasks*. Fornisce costrutti come semafori, *mutex* e code per la comunicazione tra processi concorrenti, oltre a meccanismi per la gestione della memoria, sia statica che dinamica.

Figura 3.2: Logo di FreeRTOS

ESP-NN, ESP-DL ed ESP-PPQ

ESP-NN è una libreria di basso livello sviluppata da Espressif, che rappresenta il motore di inferenza del chip. Fornisce primitive matematiche ottimizzate, come convoluzioni e *pooling*, implementate per sfruttare al meglio le istruzioni SIMD del processore. ESP-DL, basata su ESP-NN, è invece una libreria di livello più alto che gestisce la *pipeline* di inferenza dei modelli, orchestrando la sequenza dei *layer* e allocando dinamicamente i tensori e i *buffer* necessari all'esecuzione. ESP-PPQ completa la *toolchain*, fornendo gli strumenti per la quantizzazione post-addestramento di modelli di *deep learning* (in INT8 o INT16) e la loro esportazione nel formato binario .espdl, eseguibile direttamente da ESP-DL. Insieme, queste tre librerie hanno supportato in maniera efficace il *deploy* dei modelli sul dispositivo.

3.2.2 Ambiente di sperimentazione

Pytorch e Ultralytics

PyTorch è un *framework* di *deep learning open-source* per il linguaggio Python. È divenuto uno degli strumenti più popolari in ambito di ricerca e sviluppo grazie alla sua natura dinamica e flessibile, che lo rendono ideale per la prototipazione e la sperimentazione. È caratterizzato dall'uso di grafi computazionali dinamici, costruiti in tempo reale durante l'esecuzione del codice, il che permette un maggiore controllo e una più agevole fase di debug rispetto ad altri *framework* statici. Ultralytics, costruita sopra PyTorch, è una libreria che fornisce implementazioni pre-addestrate e completamente configurabili dei modelli You Only Look Once (YOLO), semplificando le fasi di addestramento, inferenza, valutazione ed esportazione.

Colab

Gli esperimenti di addestramento e fine-tuning sono stati condotti su Google Colaboratory (Colab), una piattaforma cloud che consente di eseguire *notebook* Jupyter direttamente da browser utilizzando macchine virtuali remote, anche con acceleratori hardware come GPU o TPU. Nel progetto è stato impiegato un runtime dotato di GPU NVIDIA Tesla T4, scelta per la sua buona disponibilità e per l'ottimo rapporto tra prestazioni e tempi di allocazione del *runtime*.

Roboflow

Roboflow è una piattaforma web che semplifica la gestione e la preparazione di dataset per compiti di computer vision. Fornisce strumenti grafici per l'etichettatura, l'*augmentation* e l'esportazione dei dataset nei formati più comuni, tra cui quello di YOLO.

3.3 Architettura Software

Il software sviluppato e *flashato* nel microcontrollore, adotta un'architettura modulare basata sul framework ESP-IDF, che prevede un punto di ingresso definito nel file `main.cpp` e una suddivisione del progetto in

componenti distinti. Ciascun componente, composto dalle proprie implementazioni (`.cpp`) e *headers* (`.h`), incapsula una funzionalità specifica del sistema. Nei paragrafi seguenti vengono descritti i principali componenti software, le *task* impiegate, la gestione della concorrenza di quest'ultime e le interfacce di test sviluppate.

3.3.1 Componenti ad alto livello

Main

Come anticipato, `main.cpp` rappresenta il punto di ingresso del firmware, tramite la funzione `app_main` riconosciuta dal framework di sviluppo. È il punto da cui l'esecuzione logica del programma ha inizio al momento dell'alimentazione della scheda. La prima operazione eseguita consiste nell'inizializzazione della NVS (*Non Volatile Storage*), ovvero la memoria *flash* del microcontrollore. Successivamente vengono create le task iniziali del sistema `cli_task` e `ai_task`, la cui implementazione è definita all'interno dello stesso file.

Camera

Il componente `Camera` implementa uno strato di astrazione hardware della fotocamera integrata nella scheda. Espone funzioni per l'inizializzazione, la deinizializzazione e l'acquisizione di immagini in formato JPEG, interfacciandosi direttamente con i driver forniti da ESP-IDF. Sono inoltre implementati metodi per ottenere informazioni sulla risoluzione attualmente impostata e per modificarla dinamicamente. Il componente supporta fino a 16 risoluzioni differenti, configurabili a runtime.

Web Server

Il componente `Web Server` implementa un server HTTP *embedded* che fornisce un'interfaccia di comunicazione accessibile da browser. Adotta un'architettura *RESTful* e definisce degli *endpoint* che espongono funzionalità di acquisizione fotografica, controllo della risoluzione e av-

vio dell'inferenza. In questo modo, consente di interagire con gli altri componenti del sistema attraverso semplici richieste HTTP.

Inference

Il componente `Inference` è responsabile della gestione dei modelli di *deep learning*, curandone l'inizializzazione e l'esecuzione della pipeline di inferenza, comprendente le fasi di preprocessing, processing e postprocessing.

Monitor

Il componente `Monitor` funge da wrapper per alcune funzioni di diagnostica fornite da ESP-IDF, permettendo di ottenere informazioni sullo stato della memoria RAM, sulle task attive, sull'utilizzo della CPU e sulle partizioni della memoria flash. Questo modulo è stato utilizzato principalmente a fini di monitoraggio e *debugging* durante lo sviluppo.

3.3.2 Task principali e gestione concorrenza

All'interno del sistema sono state create e implementate tre task principali, ciascuna con un ruolo specifico nella gestione delle funzionalità e della concorrenza:

- **cli_task**: creata all'avvio del sistema, tale task implementa l'interfaccia utente a riga di comando (CLI) per effettuare *debug* e ricezione di comandi da porta seriale, da parte dell'utente. Caratterizzata da bassa priorità, rimane in attesa di un input dall'utente.
- **webserver_task**: creata su richiesta dell'utente da CLI, è una *One-Shot Initialization Task*, dedicata all'inizializzazione asincrona del web server. Essa esegue sequenzialmente l'attesa della connessione WiFi, il caricamento dei modelli di *deep learning* in PSRAM, l'inizializzazione della fotocamera e l'avvio del server HTTP con registrazione degli *endpoint* REST. Al completamento della sequenza

di setup, la task termina liberando le risorse temporanee, delegando la gestione del server alla task *httpd* fornita da ESP-IDF.

- **ai_task**: anch'essa creata all'avvio del sistema, è una task permanente che implementa un pattern *Producer-Consumer* per l'esecuzione asincrona delle inferenze. Rimane in attesa su una *queue*, dalla quale attende un *frame* JPEG acquisito dalla camera e il tipo di modello da utilizzare. Un volta ricevuti, esegue l'inferenza in modo parallelo e non bloccante.

Oltre alla creazione, terminazione e sospensione dei task, sono stati impiegati costrutti di FreeRTOS per la gestione della concorrenza.

In particolare, essendo il *frame buffer* della camera una risorsa condivisa, sono stati utilizzati semafori di tipo *mutex* per garantire l'accesso mutualmente esclusivo ed evitare *race conditions*.

```
1  // Aspetta al massimo 5 secondi per acquisire il mutex
2  if (xSemaphoreTake(camera->camera_mutex, pdMS_TO_TICKS
       (5000)) != pdTRUE)
3      {
4          ESP_LOGE(TAG, "Timeout acquisizione mutex
               fotocamera");
5          return ESP_ERR_TIMEOUT;
6      }
7  // Acquisisci il frame della camera
8  camera_fb_t *fb = esp_camera_fb_get();
9  if (!fb)
10     {
11         ESP_LOGE(TAG, "Errore acquisizione frame
               fotocamera");
12         xSemaphoreGive(camera->camera_mutex);
13         return ESP_FAIL;
14     }
15 xSemaphoreGive(camera->camera_mutex);
```

Listing 3.1: Acquisizione frame della camera con mutex

È stata inoltre implementata una *message queue*, una coda di comunicazione asincrona che consente di disaccoppiare l'acquisizione del *frame buffer* della camera, dall'elaborazione dell'inferenza. Può contenere un numero arbitrario di messaggi in coda, con puntatori alle foto e il tipo di inferenza richiesta. Ciò garantisce la responsività del sistema mentre le inferenze procedono in parallelo sulla task dedicata.

```
ai_task_message_t message = {
        .inference_type = inference_type,
        .image_buffer = frame_copy,
        .image_size = photo_size,
        .timestamp = (uint32_t)(esp_timer_get_time() /
           1000000)
    };
// Invio del messaggio (Producer)
if (xQueueSend(ai_task_queue, &message, pdMS_TO_TICKS
   (5000)) != pdTRUE) {
        ESP_LOGE(TAG, "Timeout invio messaggio");
        free(frame_copy);
        return ESP_ERR_TIMEOUT;
    }
// Ricezione del messaggio (Consumer)
if (xQueueReceive(ai_task_queue, &message, portMAX_DELAY)
   == pdTRUE) {

            if (message.inference_type == 1) {
                yolo_inference_result_t result;
                if (inference_process_image_yolo(message.
                   image_buffer, message.image_size, &
                   result)) {
                    ESP_LOGI(TAG, "Inferenza completata
                       con successo");
                } else {
                    ESP_LOGE(TAG, "Errore durante l'
                       inferenza");
                }
        }
    }
```

Listing 3.2: Invio frame per inferenza asincrona

3.3.3 Interfacce di test

In questa sezione si descrivono le due interfacce sviluppate per l'interazione e il *testing* del sistema: la CLI (*Command Line Interface*) e la GUI (*Graphical User Interface*) *web-based*, per permettere l'invio di comandi alla scheda e la visualizzazione di eventuali log di debug.

La CLI è stata la prima ad essere sviluppata, e, diversamente dalla GUI, consente un controllo granulare sul sistema e, soprattutto, non presenta l'overhead associato alla connessione Wi-Fi e alle richieste HTTP.

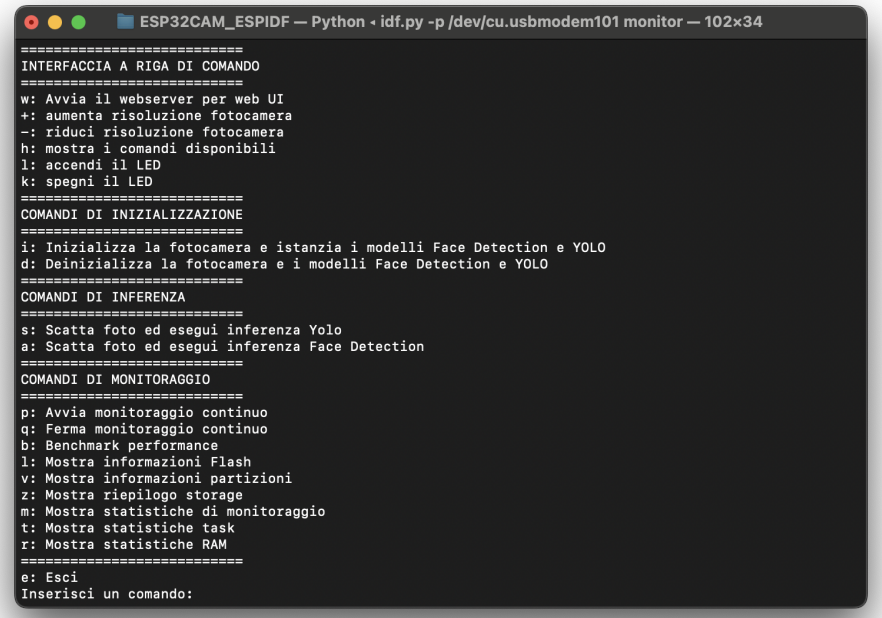


Figura 3.3: *Command Line Interface*

Tale interfaccia, basata su connessione seriale USB CDC, permette di interagire direttamente con numerose funzioni implementate nei componenti software, tra cui l'inizializzazione della camera e dei modelli in flash, lo scatto di una foto, l'esecuzione di inferenza, il cambio della risoluzione di scatto, l'avvio del web server, e una serie di comandi per effettuare il monitoraggio delle risorse. Questa soluzione è stata ottima

per poter inviare comandi alla scheda e poterci inizialmente interagire, particolarmente utile per le fasi iniziali di *debug* del firmware. Tuttavia, il principale limite di questa interfaccia, consiste nel non poter valutare visivamente i risultati qualitativi: non è possibile visualizzare l'immagine scattata, nè le *bounding boxes* risultanti dalle inferenze, delle quali è possibile stampare solo le coordinate numeriche.

Per superare questa limitazione, è stata sviluppata anche una GUI *web-based*, per permettere in modo rapido e in tempo reale di visualizzare le immagini acquisite e i risultati delle inferenze.

Figura 3.4: *Graphical User Interface*

La comunicazione tra la GUI e il firmware avviene tramite richieste HTTP di tipo GET e POST, che invocano specifiche funzioni implementate all'interno dei componenti. Questa soluzione, pur introducendo un leggero *overhead* dovuto alla connessione di rete, offre un notevole vantaggio in termini di accessibilità e analisi visiva dei risultati.

Capitolo 4

Esperimenti, Analisi e Risultati

In questo capitolo, vengono illustrati gli esperimenti e le implementazioni eseguite, con l'obiettivo di portare un modello *state-of-the-art* di *object detection* in tempo reale, sul microcontrollore a risorse limitate ESP32-S3, proponendo possibili ottimizzazioni per trovare un bilanciamento nelle prestazioni, sia in tempo di inferenza che in accuratezza.

4.1 Metriche quantitative di valutazione

Le metriche adottate per valutare i modelli implementati sono quelle utilizzate nella letteratura dell'*object detection*. La base geometrica su cui si fondano queste metriche è detta *Intersection over Union* (IoU). Essa descrive quanto due *bounding boxes* (quella predetta dal modello, e il *ground truth*) si sovrappongano tra loro, e si può calcolare mediante la seguente formula:

$$\text{IoU} = \frac{\text{Area}(B_{pred} \cap B_{gt})}{\text{Area}(B_{pred} \cup B_{gt})} \tag{4.1}$$

dove:

- B_{pred} rappresenta *la bounding box* predetta,
- B_{gt} rappresenta la *bounding box* del *ground truth*.

Il valore della IoU varia tra 0 e 1: più è alto, maggiore è la sovrapposizione tra le due *bounding boxes*.

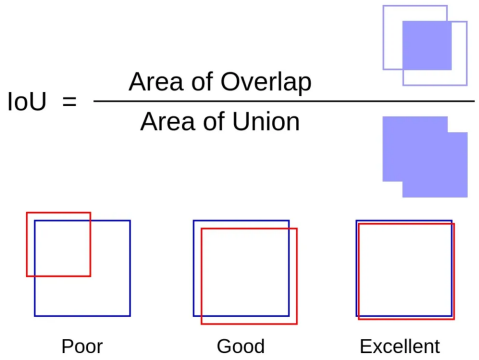


Figura 4.1: Intersection Over Union

In base ad una *IoU Threshold* IoU_{th} arbitraria (solitamente pari a 0.5), una predizione viene considerata correttamente localizzata se IoU $\geq$ IoU_{th}. Successivamente, combinando questo criterio con la classe predetta, le predizioni vengono classificate come:

- **True Positive (TP):** *la bounding box* è correttamente localizzata e la classe predetta corrisponde a quella reale
- **False Positive (FP):** la *bounding box* non corrisponde ad alcun oggetto reale, oppure la classe predetta è errata, anche se la localizzazione è corretta
- **False Negative (FN):** un oggetto reale non è stato rilevato da alcuna predizione valida

Grazie a quest'ultime, si possono calcolare:

$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}} \quad (4.2) \qquad \text{Recall} = \frac{\text{TP}}{\text{TP} + \text{FN}} \quad (4.3)$$

Intuitivamente, più alta è la **Precision**, e più è probabile che quando il modello fa una predizione, essa sia giusta. Più è alta la **Recall**, e più il modello è bravo a non mancare gli oggetti presenti nell'immagine. Nella pratica, esiste spesso un compromesso tra *precision* e *recall*: aumentando una, si tende a ridurre l'altra.

Fissata una *IoU Threshold* IoU_{th}, l'accuratezza di un modello di *object detection* su di una specifica classe è definita come **Average Precision** (**AP**), la quale consiste nell'area sotto la curva *precision-recall*:

$$\text{AP@50} = \int_0^1 P(R)\, dR \quad \text{con } \text{IoU}_{\text{th}} = 0.5 \tag{4.4}$$

Se il modello è multi-classe, la sua accuratezza media su tutte le classi è definita come **Mean Average Precision** (**mAP**). Sia fissata una *IoU Threshold* IoU_{th}, la mAP rappresenta la media delle AP calcolate su tutte le C classi di oggetti.

$$\text{mAP@50} = \frac{1}{C} \sum_{k=1}^{C} (\text{AP}_k \text{ a } \text{IoU}_{\text{th}} = 0.5) \tag{4.5}$$

Dunque, se un modello è monoclasse, mAP@50 coincide con AP@50.

La **mAP@[.50:.95]** è la metrica più rigorosa, introdotta dal COCO *evaluation protocol*, essa consiste nel calcolare l'mAP su dieci diverse soglie di IoU (da 0.50 a 0.95 con passi di 0.05), per poi fare la media di questi 10 valori. Formalmente, si può esprimere come:

$$\text{mAP}[@.50 : .95] = \frac{1}{|\mathcal{T}|} \sum_{t \in \mathcal{T}} \left(\frac{1}{C} \sum_{k=1}^{C} (\text{AP}_k \text{ a } \text{IoU}_{\text{th}} = t) \right) \tag{4.6}$$

dove $\mathcal{T} = \{0.50, 0.55, \ldots, 0.95\}$ è l'insieme dei 10 valori discreti di IoU e C è il numero di classi.

4.2 YOLOv11 e dataset COCO

Il modello scelto per il *deploy* è YOLOv11[9], una delle più recenti iterazioni della famiglia You Only Look Once.

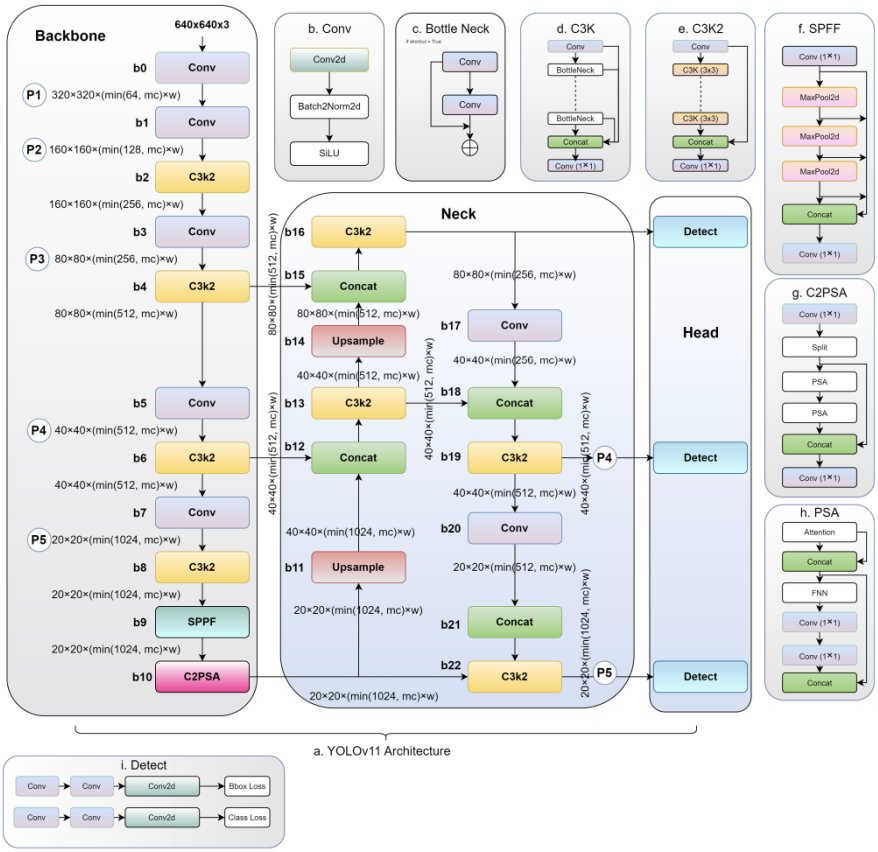


Figura 4.2: Architettura YOLOv11

YOLOv11 estende le potenzialità delle versioni precedenti, come YOLOv8[10], migliorando le *performance* in termini di velocità e accuratezza.

L'architettura è composta da tre macro-blocchi principali:

- **Backbone:** responsabile dell'estrazione delle *feature* dalle immagini in input, impiega blocchi convoluzionali residui per generare *feature maps* a diverse risoluzioni.

- **Neck:** combina le informazioni provenienti dai vari livelli della *backbone* attraverso operazioni di *upsampling* e concatenazione, fornendo rappresentazioni multi-scala utili per la rilevazione di oggetti di dimensioni differenti.
- **Head:** genera le predizioni finali, producendo per ciascuna *bounding box* le coordinate, la confidenza e la classe corrispondente.

Tra le innovazioni architetturali, il modulo C3k2 sostituisce il vecchio C2f, un'implementazione del *Cross Stage Partial Bottleneck* più efficiente, migliorando la propagazione dei gradienti. Viene inoltre introdotto un nuovo modulo C2PSA (*Cross Stage Partial with Spatial Attention*), il quale implementa un meccanismo di attenzione spaziale che ne migliora l'accuratezza.

La funzione di attivazione utilizzata nei *layer* convoluzionali è la SiLU(*Sigmoid Linear Unit*), preferita alla ReLU per la sua maggiore stabilità numerica e capacità di convergenza:

$$\text{SiLU}(x) = x \cdot \sigma(x) = \frac{x}{1 + e^{-x}} \tag{4.7}$$

La loss function di tale modello è definita come la somma pesata di tre componenti principali:

$$L = w_{box} \cdot L_{box} + w_{cls} \cdot L_{cls} + w_{dfl} \cdot L_{dfl} \tag{4.8}$$

dove:

- L_{box} misura l'accuratezza delle *bounding boxes*,
- L_{cls} valuta la correttezza della classificazione e include la stima della confidenza,
- L_{dfl} affina la regressione delle *bounding boxes*, trattandola come un problema di distribuzione,
- w_{box}, w_{cls} e w_{dfl} sono pesi di bilanciamento dei rispettivi termini.

Ultralytics distribuisce diverse varianti del modello [11], che si differenziano per numero di parametri, velocità di inferenza e accuratezza, permettendo di bilanciare prestazioni e requisiti computazionali in base al contesto applicativo.

Tabella 4.1: Versioni del modello YOLOv11

Modello	**Image size**	**Size(MB)**	**Parametri**	**GFLOPs**
YOLOv11n	640	5.6	2.6M	6.5
YOLOv11s	640	19.3	9.4M	21.5
YOLOv11m	640	40.6	20.1M	68
YOLOv11l	640	51.4	25.3M	86.9
YOLOv11x	640	114.3	56.9M	194.9

Tutte le varianti sono pre-addestrate sul dataset COCO (*Common Objects in Context*) [12], un *benchmark* ampiamente utilizzato per la valutazione di modelli di *object detection*. Il dataset include 80 classi, con circa 118K immagini di training, 5K di validazione e 20K di test, costituendo un punto di partenza ideale per attività di *fine-tuning*.

Figura 4.3: Esempi di immagini del COCO *Dataset*.

4.3 YOLOv11n su ESP32-S3

In un contesto a risorse computazionali limitate, come quello dell'ESP32-S3, la scelta del modello ricade su YOLOv11n, la versione più compatta della famiglia YOLOv11, capace di mantenere un buon equilibrio tra accuratezza e complessità computazionale.

4.3.1 Deployment

Tale modello viene distribuito in formato FLOAT32 e sebbene il binario del modello sia sufficientemente compatto da poter risiedere nella flash del microcontrollore, la *tensor arena*, ossia l'area di memoria utilizzata per l'allocazione dei tensori durante l'inferenza, eccede gli 8 MB di PSRAM disponibili sulla scheda. Diventa quindi necessaria una *pipeline* che coinvolga la quantizzazione.

Il binario del modello in formato Pytorch (.pt) viene innanzitutto esportato in un formato ONNX (*Open Neural Network Exchange*).

Il formato ONNX rappresenta uno standard aperto per la serializzazione di modelli di *deep learning*, pensato per l'interoperabilità tra *framework* e dispositivi diversi. A differenza del formato PyTorch, ottimizzato per il *training*, ONNX definisce un grafo statico ottimizzato per l'inferenza, riducendo *overhead* e migliorando l'efficienza computazionale. Alcuni dei miglioramenti rilevanti che apporta sono:

- **Operation fusion e costant folding**: alcune serie di operazioni semplici e sequenziali, vengono convertite in un'unica operazione ottimizzata (per esempio, un *layer* convoluzionale e un *batch normalization* vengono fusi in un unico *layer*), e i calcoli costanti vengono precalcolati,
- **Rimozione nodi inutili**: alcune operazioni non necessarie all'inferenza vengono eliminate,

- **Preallocazione**: alcune strutture dati ausiliarie all'inferenza vengono preallocate in modo efficiente.

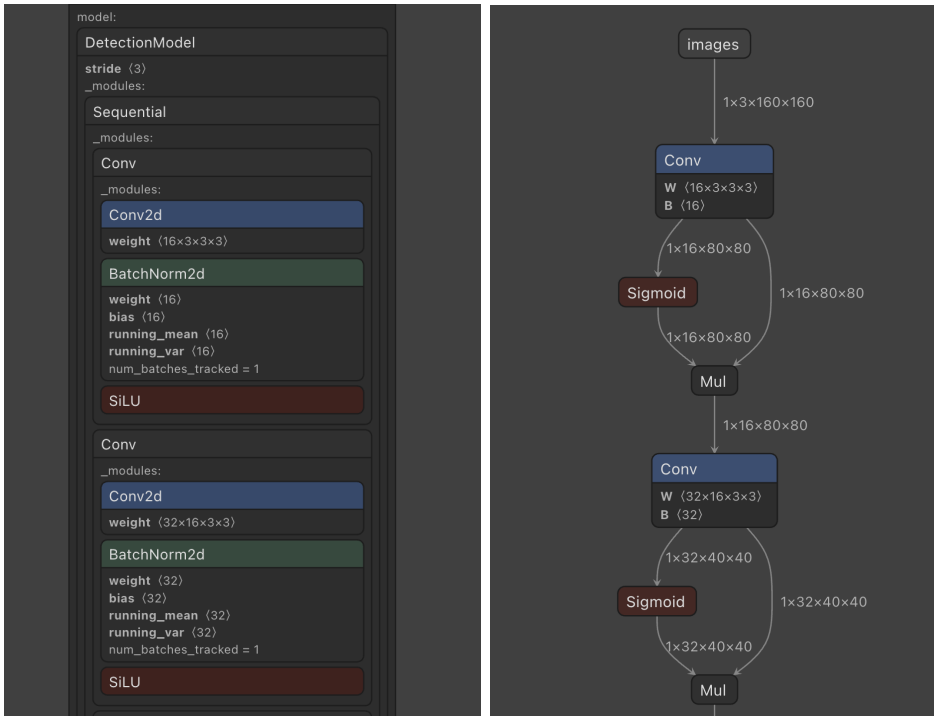


Figura 4.4: Differenza layer convoluzionali di YOLOv11n in formato Pytorch (sinistra) e in ONNX (destra) su Netron

Il file binario in formato ONNX ottenuto è più grande del Pytorch, in particolare 10.5 MB, proprio perchè tutte le strutture dati necessarie sono già allocate e incluse al suo interno.

Inoltre, per ridurre la dimensione del modello e il suo carico computazionale, viene eseguita *Post-Training Quantization* (PTQ) statica, per convertirlo da FLOAT32 a INT8, determinando i parametri di quantizzazione prima del *deploy*. Per fare ciò, è necessario un dataset di calibrazione che permetta di determinare i fattori di scala e gli *zero-point* dei vari layer del modello, per minimizzare la perdita dovuta al cambio di scala dei valori.

E' importante che il dataset di calibrazione che viene utilizzato, sia simile al dominio su cui il modello andrà a fare inferenza. Per questo motivo,

sono state scattate, utilizzando la camera integrata nel microcontrollore, circa 120 immagini destinate alla calibrazione.

Figura 4.5: Esempi di immagini del *Calibration Dataset*

La calibrazione crea un file da 2.9MB di dimensione in un formato .espdl compatibile con il motore di inferenza del microcontrollore.

Un aspetto rilevante del processo di inferenza che viene subito notato, riguarda il formato delle immagini acquisite dalla camera integrata, che opera unicamente in formato JPEG compresso. Tali immagini devono essere decodificate in RGB888 (8 bit per canale) per risultare compatibili con il modello, e questa conversione costituisce la parte predominante del tempo di preprocessing, variando in base alla risoluzione di acquisizione.

Tabella 4.2: Tempi di *decode* da JPEG a RGB888

Risoluzione	**Pixel**	**Tempo (ms)**
128×128	$16,384$	4
240×240	$57,600$	12
320×240	$76,800$	16
320×320	$102,400$	21
480×320	$153,600$	30
640×480	$307,200$	59
800×600	$480,000$	89
1280×720	$921,600$	168
1280×1024	$1,310,720$	239
1600×1200	$1,920,000$	346
1920×1080	$2,073,600$	370

4.3.2 Problemi

Dopo il *deploy* su ESP32-S3, il modello YOLOv11n evidenzia alcune criticità prestazionali e qualitative. Una prima analisi sperimentale evidenzia tempi di inferenza elevati, fortemente dipendenti dall'*image size* [13].

L'*image size* è un parametro che definisce la dimensione alla quale ciascuna immagine viene ridimensionata prima di essere fornita in input al modello. Poiché YOLO opera su immagini quadrate, fissata un'*image size* x, l'input al primo strato convoluzionale è un tensore di forma (B, C, H, W), dove $H = W = x$, C rappresenta il numero di canali e B la dimensione del batch.

Questo parametro risulta rilevante per il tempo di inferenza del modello, in quanto stabilisce la dimensione delle feature maps intermedie. Poiché ONNX genera un grafo statico, l'image size deve essere fissata e nota in fase di esportazione, dato che le forme di questi tensori intermedi vengono determinate staticamente.

Tabella 4.3: Tempi di inferenza di YOLOv11n INT8 (Pre-trained COCO) su ESP32-S3 al variare dell'image size

Image Size	**Pixel**	**Tempo (ms)**
64×64	$4,096$	480 ± 2
96×96	$9,216$	1126 ± 3
128×128	$16,384$	2068 ± 4
160×160	$25,600$	3307 ± 3
320×320	$102,400$	14043 ± 11
640×640	$409,600$	59240 ± 34

Tali tempi di inferenza rappresentano un limite significativo, rendendo il modello poco adatto e inefficace in un contesto di applicazione reale: per un'*image size* di 640, l'inferenza supera i 59 secondi.

L'*image size* con cui il modello viene esportato in ONNX è rilevante non solo per il tempo di inferenza, ma anche per la sua accuratezza.

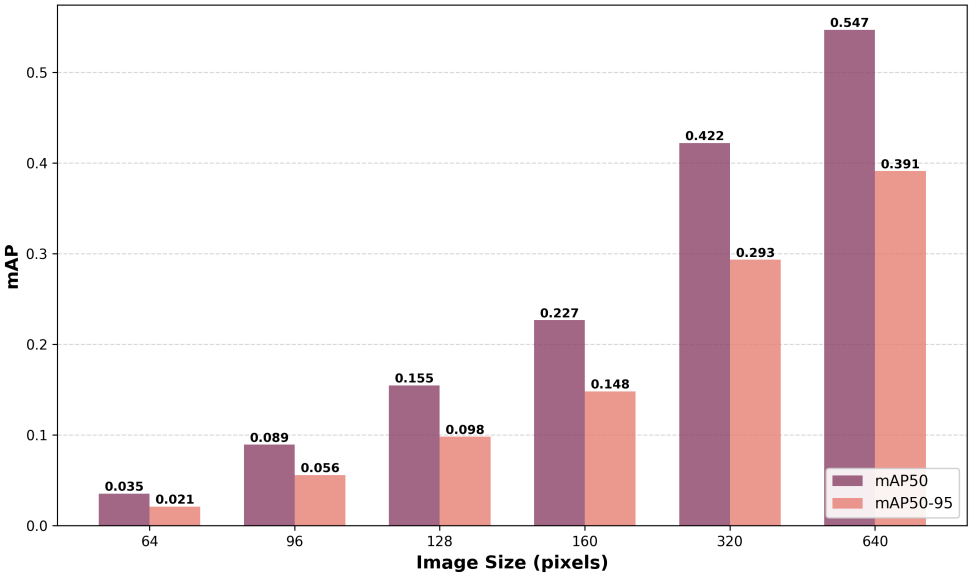


Figura 4.6: Tendenza mAP di YOLOv11 in funzione dell'*image size*, su COCO val2017

Oltre alle limitazioni computazionali, il modello mostra un'accuratezza fortemente degradata in fase di *deploy*. Tale fenomeno è attribuibile in parte agli effetti della quantizzazione INT8, che introducono errori di approssimazione nei pesi e nelle attivazioni, ma soprattutto al *domain gap* tra il *dataset* COCO, composto da immagini ad alta qualità, e le immagini JPEG compresse e rumorose acquisite dalla camera del microcontrollore.

(a) True Positive (b) False Negative

Figura 4.7: YOLOv11n riesce a rilevare persone solo se molto vicine alla camera

In questo contesto, si definisce *domain gap* la discrepanza tra due *dataset*, che, pur contenendo oggetti appartenenti alle stesse classi, presentano differenze significative nelle caratteristiche visive e contestuali delle immagini. Tale differenza può derivare da numerosi fattori: nel contesto di questa tesi, è principalmente dovuta al dispositivo di acquisizione, il quale introduce degradazioni visive quali rumore e compressione, riducendo la qualità complessiva dell'immagine. Tuttavia, un *domain gap* può essere causato anche da variazioni di illuminazione, angolazione di ripresa, occlusioni, o, nel caso di scenari *outdoor*, perfino dalle condizioni meteorologiche[14].

In modo analogo, si può definire *resolution gap* la differenza sostanziale di *image size* tra le immagini utilizzate durante l'addestramento e quelle fornite in input al modello in fase di inferenza. Un modello addestrato a riconoscere un oggetto a una determinata scala può incontrare difficoltà nel rilevarlo correttamente quando il numero di pixel in input è significativamente più basso, a causa della perdita di dettagli discriminanti. Questo aspetto è particolarmente rilevante in questo caso, in quanto, se si vuole mantenere un ridotto tempo di inferenza, sarà necessario fornire al modello immagini da elaborare con *image size* molto più piccola di quella su cui è stato pre-addestrato.

Negli esperimenti successivi, verranno quindi applicati *fine-tuning* di *resolution adaptation* e di *domain adaptation* per mitigare queste discrepanze e migliorare le prestazioni del modello nelle condizioni di applicazione reali.

4.4 Proposta di ottimizzazione: YOLOv11micro

In questa sezione si propone una possibile ottimizzazione di YOLOv11n, sia per diminuirne il carico computazionale del modello, sia per migliorarne l'accuratezza *on deploy*, in particolare per la camera dell'ESP32-S3 utilizzato.

4.4.1 Model Scaling

Il problema principale legato al tempo di inferenza di YOLOv11n, è dovuto al suo numero di parametri e di GFLOPs (*Giga Floating-point Operations*) i quali, seppur minori rispetto alle altre sue varianti, risultano ancora sfidanti per un microcontrollore.

Analizzando e confrontando le architetture di YOLOv11n e YOLOv11s, emerge che la principale differenza tra le due varianti non risiede nella tipologia o nel numero di *layer*, bensì nella larghezza (*width*), ovvero nel numero di canali dei *layer* convoluzionali. In particolare, YOLOv11n utilizza circa la metà dei canali rispetto alla versione *small*. Seguendo questa logica di *model scaling*, è stata creata una nuova variante denominata YOLOv11micro, il cui nome richiama sia il *target* di *deploy* su microcontrollori, sia le dimensioni ridotte rispetto alla versione nano [15]. Nello specifico, la *width* è stata dimezzata, ottenendo così un modello con un numero di canali pari alla metà rispetto a YOLOv11n, con una conseguente riduzione significativa della dimensione complessiva.

Inoltre, il modello è stato specializzato su una singola classe, al fine di concentrare la capacità di apprendimento su un solo tipo di oggetto e ridurre ulteriormente la complessità della *detection head*. La classe scelta è *person*, in quanto ampiamente rappresentata in dataset pubblici come COCO e facilmente verificabile in fase di test sul campo.

Il modello YOLOv11micro monoclasse ottenuto possiede meno di un terzo dei parametri rispetto YOLOv11n, e presenta un tempo di inferenza

circa quattro volte inferiore.

Tabella 4.4: Confronto di dimensione dei modelli

Modello	**Size(MB)**	**Parametri**	**GFLOPs**	**Classi**
YOLOv11n	5,6	2,624,080	6,5	80
YOLOv11micro	2	864,600	2,7	80
YOLOv11micro	1,9	816,219	2,4	1

Tabella 4.5: Tempi di inferenza di YOLOv11micro INT8 (con pesi random) su ESP32-S3 al variare dell'*image size*

Image Size	**Pixel**	**Tempo (ms)**
64×64	$4,096$	123 ± 2
96×96	$9,216$	274 ± 3
128×128	$16,384$	500 ± 4
160×160	$25,600$	813 ± 3
320×320	$102,400$	3545 ± 10
640×640	$409,600$	15686 ± 24

```
                 from  n    params  module                                       arguments
  0                -1  1       232  ultralytics.nn.modules.conv.Conv             [3, 8, 3, 2]
  1                -1  1      1184  ultralytics.nn.modules.conv.Conv             [8, 16, 3, 2]
  2                -1  1      1888  ultralytics.nn.modules.block.C3k2            [16, 32, 1, True, 0.25]
  3                -1  1      9280  ultralytics.nn.modules.conv.Conv             [32, 32, 3, 2]
  4                -1  1      7232  ultralytics.nn.modules.block.C3k2            [32, 64, 1, True, 0.25]
  5                -1  1     36992  ultralytics.nn.modules.conv.Conv             [64, 64, 3, 2]
  6                -1  1     22016  ultralytics.nn.modules.block.C3k2            [64, 64, 1, True]
  7                -1  1     73984  ultralytics.nn.modules.conv.Conv             [64, 128, 3, 2]
  8                -1  1     87040  ultralytics.nn.modules.block.C3k2            [128, 128, 1, True]
  9                -1  1     41344  ultralytics.nn.modules.block.SPPF            [128, 128, 5]
 10                -1  1     63424  ultralytics.nn.modules.block.C2PSA           [128, 128, 1]
 11                -1  1         0  torch.nn.modules.upsampling.Upsample         [None, 2, 'nearest']
 12            [-1, 6]  1         0  ultralytics.nn.modules.conv.Concat           [1]
 13                -1  1     30208  ultralytics.nn.modules.block.C3k2            [192, 64, 1, True]
 14                -1  1         0  torch.nn.modules.upsampling.Upsample         [None, 2, 'nearest']
 15            [-1, 4]  1         0  ultralytics.nn.modules.conv.Concat           [1]
 16                -1  1      8704  ultralytics.nn.modules.block.C3k2            [128, 32, 1, True]
 17                -1  1      9280  ultralytics.nn.modules.conv.Conv             [32, 32, 3, 2]
 18           [-1, 13]  1         0  ultralytics.nn.modules.conv.Concat           [1]
 19                -1  1     24064  ultralytics.nn.modules.block.C3k2            [96, 64, 1, True]
 20                -1  1     36992  ultralytics.nn.modules.conv.Conv             [64, 64, 3, 2]
 21           [-1, 10]  1         0  ultralytics.nn.modules.conv.Concat           [1]
 22                -1  1     95232  ultralytics.nn.modules.block.C3k2            [192, 128, 1, True]
 23       [16, 19, 22]  1    267123  ultralytics.nn.modules.head.Detect           [1, [32, 64, 128]]
YOLO11micro summary: 231 layers, 816,219 parameters, 816,203 gradients, 2.4 GFLOPs
```

Figura 4.8: *Snapshot* dell'architettura di YOLOv11micro

4.4.2 Argomenti di Training

Tutti gli esperimenti descritti nelle sezioni successive condividono una configurazione di base coerente, qui di seguito riportata, salvo dove diversamente indicato.

L'addestramento è stato eseguito utilizzando SGD (*Stochastic Gradient Descent*) come ottimizzatore. Tale scelta è motivata dalla sua comprovata stabilità e dalla superiore capacità di generalizzazione a confronto con altri ottimizzatori adattivi, nonostante richieda un tempo di convergenza maggiore. Per l'ottimizzatore, sono stati mantenuti i valori di default per il *Momentum* (0.937) e per il *Weight Decay* (0.0005).

Per la gestione del tasso di apprendimento (LR), è stato utilizzato il *Learning Rate Scheduler* di default di Ultralytics. Il *Final Learning Rate Fraction* (lrf) è stato mantenuto costante a 0.01, mentre il tasso iniziale (lr0) è stato variato unicamente in funzione dello specifico *training*.

Il modello è sempre stato addestrato in modalità monoclasse sulla classe zero, limitando l'apprendimento delle *detection* esclusivamente alla classe person. Inoltre, sono stati utilizzati 8 *workers*, per parallelizzare ed efficientare il training sulla Nvidia Tesla T4 utilizzata.

Per la *Data Augmentation*, sono state mantenute le trasformazioni standard proposte da Ultralytics (variazioni HSV, traslazione, *scaling*, *flipping*, ecc.). È stata data particolare enfasi alla *mosaic augmentation*, una semplice tecnica che consiste nel disporre più immagini del *training set* in una unica, a forma di mosaico. Cruciale per mitigare l'*overfitting* e potenziare la generalizzazione, di questa tecnica ne è stata modificata l'intensità e la durata in base alla tipologia di *training* in esame.

Tale configurazione fissa è stata mantenuta come base in tutti gli esperimenti successivi, al fine di garantire la coerenza metodologica.

4.4.3 Warm-up

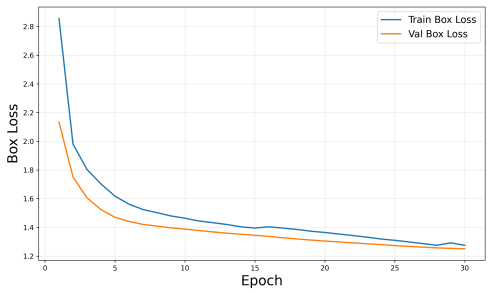
Prima di procedere con l'adattamento alle risoluzioni target, è stato eseguito un training preliminare di *warm-up* a 640×640 per 30 epoche su YOLOv11micro, utilizzando la classe person del dataset COCO.

Questa fase non mira a portare il modello a convergenza completa su tale risoluzione, poiché ciò comporterebbe l'apprendimento di dettagli che verrebbero inevitabilmente persi nella successiva fase di *resolution adaptation*. L'obiettivo è piuttosto quello di consentire al modello di apprendere feature semantiche di alto livello, utili a stabilizzare e velocizzare le fasi di *training* successive.

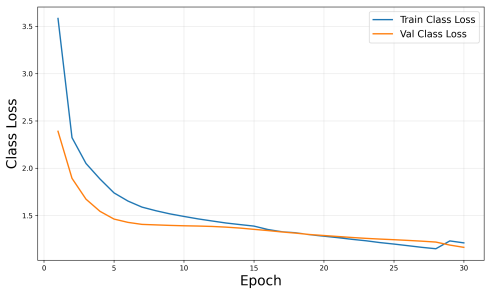
Inoltre, tale esperimento fornisce un punto di riferimento per valutare la rapidità di apprendimento del modello YOLOv11micro e per confrontarne l'accuratezza con quella del modello YOLOv11n

Tabella 4.6: Argomenti di *training, warm-up*

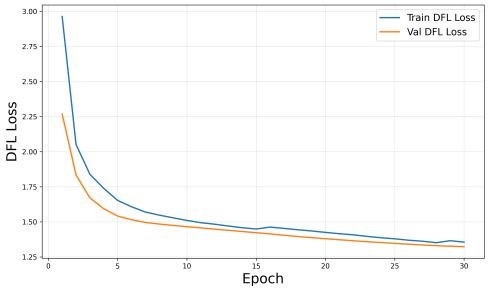
Iperparametro	**Valore**
freeze	0
batch	16
epochs	30
imgsz	640
close mosaic	15
mosaic	0.5
amp	True
lr0	0.01



(a) Box Loss

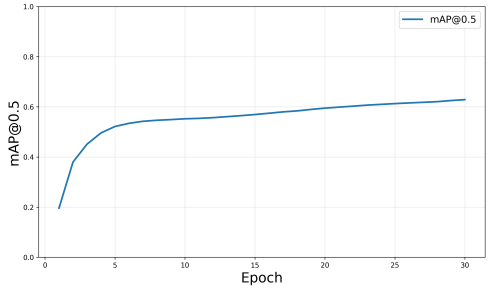


(b) Class Loss

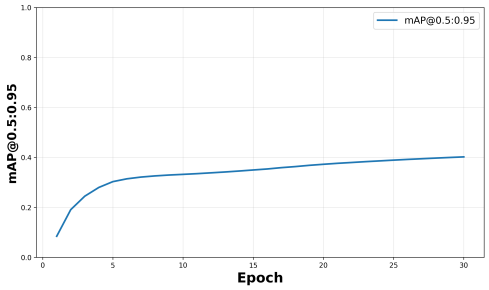


(c) DFL Loss

Figura 4.9: Progressione delle componenti della Loss durante il training



(a) mAP@50



(b) mAP@[50:95]

Figura 4.10: Avanzamento dell'mAP durante il training

Essendo monoclasse, l' mAP del modello coincide con l'AP relativa alla classe person. Grazie alle sue dimensioni ridotte, YOLOv11micro mostra una veloce capacità di apprendimento, raggiungendo una AP@50:95 pari a circa 0.4 sulla classe person dopo sole 30 epoche.

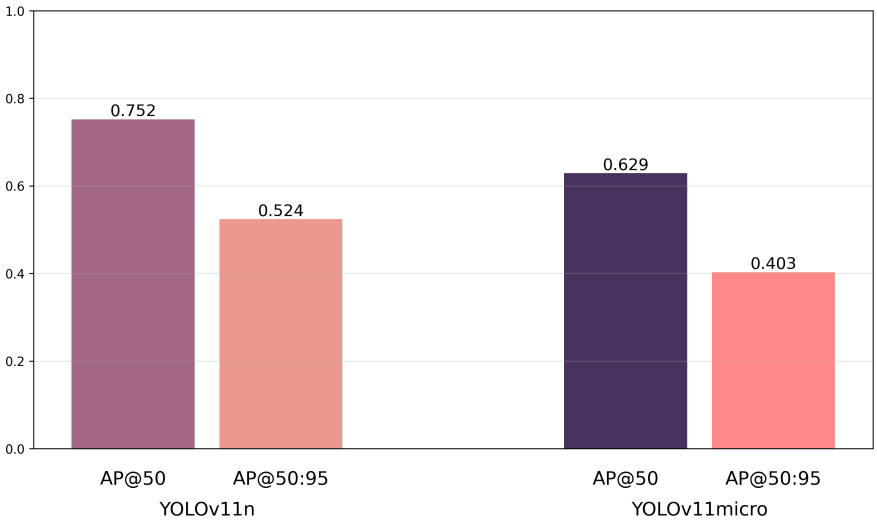
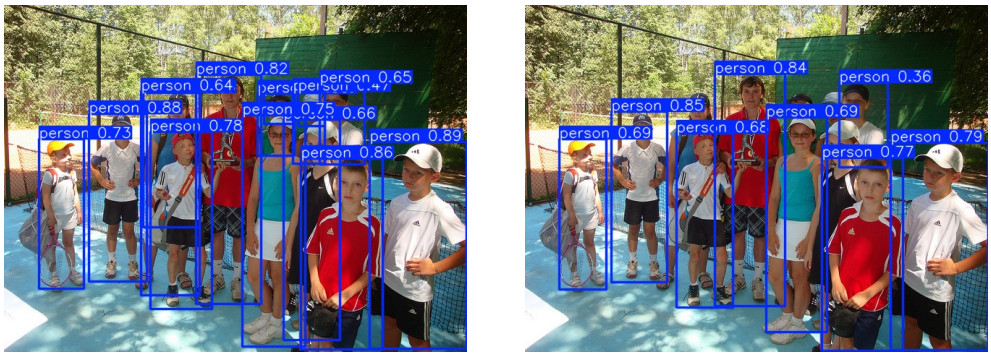


Figura 4.11: Confronto *Average Precision* classe person tra YOLOv11n e YOLOv11micro, su val2017 COCO

Come previsto, il modello non supera le prestazioni del corrispettivo YOLOv11n alla stessa risoluzione, ma questo risultato è coerente con l'obiettivo di questa fase, che è principalmente quello di fornire un punto di partenza solido per l'adattamento successivo a risoluzioni inferiori.



(a) YOLOv11n (b) YOLOv11micro

Figura 4.12: Esempio di confronto inferenza, YOLOv11micro non rileva quattro persone parzialmente occluse, immagine da COCO val2017

4.4.4 Resolution Adaptation

L'obiettivo di questa fase è migliorare l'accuratezza di YOLOv11micro sulle *image size target* di *deploy*, riducendo al minimo la perdita di prestazioni derivante dalla diminuzione della risoluzione di input [16].

Come discusso in precedenza, esportare il modello con una *image size* inferiore riduce significativamente il tempo di inferenza, ma può penalizzare l'accuratezza a causa della perdita di dettagli dell'immagine. Una volta fissato l'obiettivo di mantenere il tempo di inferenza al di sotto di un secondo, si è scelto di addestrare YOLOv11micro su *image size* di 64, 96, 128 e 160 (Tabella 4.5), al fine di analizzare l'impatto della risoluzione sulla precisione del modello.

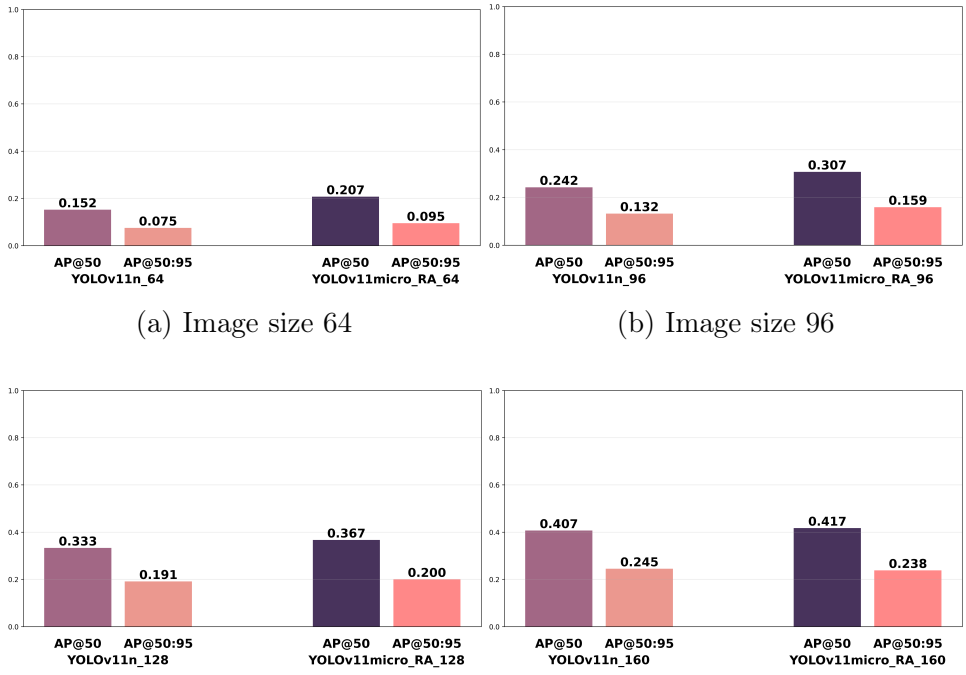
A partire dal modello ottenuto nella fase di *warm-up*, è stato eseguito un nuovo *training* di 200 epoche sul dataset COCO per ciascuna risoluzione x. Per ogni esperimento, è stata selezionata l'epoca che ha ottenuto la miglior AP *person*, sul set di validazione COCO val2017. La riduzione dell'*image size* ha permesso di aumentare il *batch size* da 16 a 64, grazie al minor consumo di memoria della GPU. La *mosaic augmentation* è stata mantenuta solo nelle prime epoche, poiché con immagini di dimensioni così ridotte, c'è il rischio di ottenere mosaici di immagini troppo piccole.

Tabella 4.7: Argomenti di *training, resolution adaptation*

Iperparametro	**Valore**
freeze	0
batch	64
epochs	200
imgsz	x
close mosaic	180
mosaic	0.7
amp	False
lr0	0.01

Per convenzione, si definisce con YOLOv11n_y il modello YOLOv11n esportato con *image size y*, e con YOLOv11micro_RA_y il modello

YOLOv11micro ottenuto tramite la fase di *Resolution Adaptation* sulla stessa *image size*.



(a) Image size 64 (b) Image size 96

(c) Image size 128 (d) Image size 160

Figura 4.13: Confronto dell'*Average Precision* sulla classe *person* tra YOLOv11n_y e YOLOv11micro_RA_y su COCO val2017.

Come atteso, la riduzione della risoluzione comporta un calo significativo dell'accuratezza rispetto al *training* di *warm-up* su 640×640.

Tuttavia, è interessante notare come YOLOv11micro, pur avendo meno di un terzo dei parametri di YOLOv11n, riesca, una volta riaddestrato specificamente su ciascuna *image size*, ad ottenere un'*Average Precision* superiore o comparabile a quella del corrispondente YOLOv11n_y esportato direttamente senza riaddestramento. Questo risultato evidenzia la capacità del modello ridotto di adattarsi in maniera sufficiente a risoluzioni più piccole.

(a) YOLOv11n_96 (b) YOLOv11micro_RA_96

Figura 4.14: Esempio di confronto inferenza, YOLOv11micro dopo resolution adaptation su imgsz x piccola, rileva persone meglio di YOLOv11n esportato con medesima imgsz, immagine da COCO val2017

4.4.5 Domain Adaptation

I modelli YOLOv11micro ottenuti, riescono a raggiungere una maggiore accuratezza su *image size* piccole, ma l'accuratezza *on deploy* continua ad essere degradata. Questo, è dovuto al *domain gap* che continua ad esistere tra le immagini contenute in COCO e le immagini che vengono acquisite dalla camera del microcontrollore. Per risolvere questa problematica, si è deciso di creare un dataset *esp-person* su cui effettuare un *fine-tuning* di *domain adaptation*.

In particolare, il *dataset* è stato creato scattando numerose foto mediante la camera dell'ESP32-S3 includendo soggetti diversi (amici e familiari) da angolazioni, pose e distanze differenti. Il dataset è stato poi etichettato a mano utilizzando la piattaforma Roboflow, mediante lo standard di YOLO su *task* detection per le etichette.

Il formato YOLO utilizza un file di testo (.txt) separato per ogni immagine, contenente un'annotazione per riga. Ogni riga del file di etichetta definisce un'istanza di oggetto e segue rigorosamente il formato con valori separati da spazi:

$$< class\_id > \ < center_x > \ < center_y > \ < width > \ < height > \quad (4.9)$$

Dove class_id rappresenta la classe dell'istanza (sempre 0 in questo caso, lavorando sulla classe person), e gli altri quattro valori rappresentano la bounding box, normalizzati tra 0 e 1 dividendo le coordinate originali in pixel per la dimensione totale dell'immagine.

Nello specifico, il *dataset* è composto da 1300 immagini, dando priorità alla qualità e differenziazione delle stesse, piuttosto che alla quantità. Di queste, 1000 sono state dedicate al *train set* per massimizzare il numero di immagini viste dal modello, 150 per il *train set*, e altrettante 150 per il *test set*. Inizialmente, il *dataset* era composto da 1200 immagini contenenti sempre almeno un'istanza di person, ma per diminuire i falsi positivi dei modelli, si è deciso di aggiungere anche altre 100 immagini di *background*, distribuite tra i tre set.

Figura 4.15: Esempi di immagini contenute nel dataset esp-person

Per allenare i modelli *resolution adapted* su questo dataset, si è deciso infine di utilizzare *progressive unfreezing*, in quanto ha mostrato di rag-

giungere una buona accuratezza sul *dataset target*, mantenendo ridotta la perdita di *Average Precision* sul dataset di COCO[17]. Si è quindi inizialmente allenata solo la *detection head*, prendendo sempre il modello risultante dall'ultima epoca, per poi effettuare un *fine-tuning* completo sull'intera rete a basso *learning rate*, dalla quale viene preso il modello con *Average Precision* migliore sul *validation set*.

Tabella 4.8: Argomenti di *training, domain adapatation*

Head

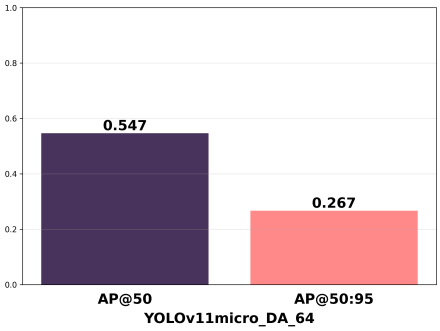
Iperparametro	**Valore**
freeze	23
batch	64
epochs	50
imgsz	x
close mosaic	30
mosaic	0.5
amp	False
lr0	0.005

Full

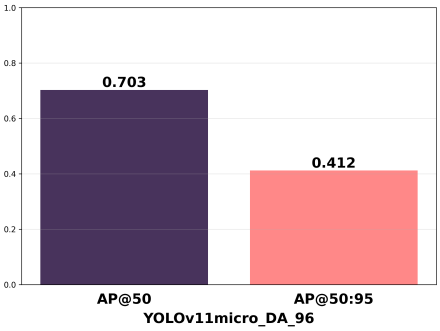
Iperparametro	**Valore**
freeze	0
batch	64
epochs	100
imgsz	x
close mosaic	60
mosaic	0.5
amp	False
lr0	0.001

Inoltre, dopo un'analisi degli errori di quantizzazione *layerwise*, si è deciso empiricamente di portare in *mixed-precision* INT16, i *layer* convoluzionali iniziali della rete che contribuivano maggiormente all'errore *graphwise*. In particolare, i *layer* /model.0/conv/Conv, /model.1/conv/Conv, /model.2/cv1/conv/Conv, /model.2/cv2/conv/Conv e /model.4/cv1/conv/Conv. Si è inoltre implementata l'*equalization* nel processo di quantizzazione, in quanto essa riesce a migliorare la qualità della quantizzazione senza aggiungere *overhead* computazionale rilevante[18].

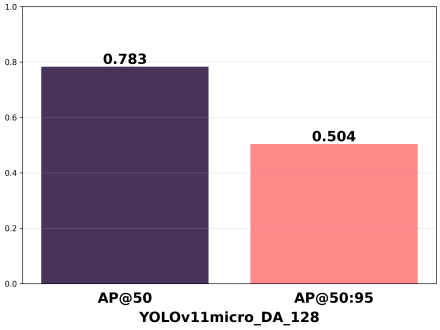
Per convenzione, definiamo in seguito YOLOv11micro_DA_y il modello ottenuto in questa fase di *domain adaptation* su *image size* y, partendo da YOLOv11micro_RA_y.



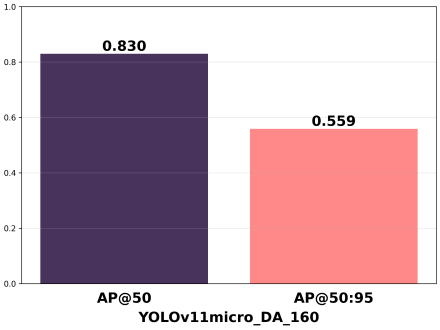
(a) Image size 64



(b) Image size 96



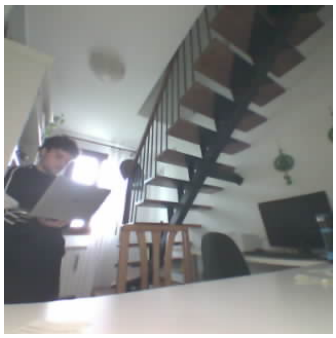
(c) Image size 128



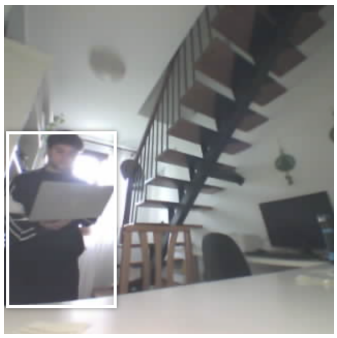
(d) Image size 160

Figura 4.16: Confronto dell'*Average Precision* sulla classe *person* di YOLOv11micro dopo *domain adaptation*, su *esp-person test set*.

In sintesi, i risultati ottenuti evidenziano che, sebbene la riduzione dell'*image size* influisca negativamente sull'accuratezza, l'adattamento al dominio *target* ha permesso al modello di comprendere meglio le immagini della camera ed effettuare rilevazioni stabili direttamente sul microcontrollore.



(a) YOLOv11micro_RA_96



(b) YOLOv11micro_DA_96

Figura 4.17: Esempio di confronto inferenza *on deploy*

Il *dataset* esp-person è stato inoltre utilizzato come set di calibrazione, per migliorare la stabilità della quantizzazione. I nuovi parametri di quantizzazione, in particolare i *layer* INT16, permettono una diminuzione dell'errore di quantizzazione, al costo di qualche millisecondo aggiunto al tempo di inferenza.

Tabella 4.9: Tempi di inferenza di YOLOv11micro INT8 con nuovi argomenti di quantizzazione, su ESP32-S3 al variare dell'*image size*

Image Size	**Pixel**	**Tempo (ms)**
64×64	$4,096$	128 ± 2
96×96	$9,216$	289 ± 4
128×128	$16,384$	546 ± 3
160×160	$25,600$	873 ± 10

Figura 4.18: *Person detection* su ESP32-S3 con i quattro modelli ottenuti. Dall'alto a sinistra procedendo in senso antiorario: YOLOv11micro_DA_64/96/128/160

Conclusioni

In questo progetto si è riusciti a portare, bilanciando l'accuratezza con l'efficienza computazionale, un modello You Only Look Once per l'*Object Detection* direttamente su un microcontrollore. Tra i quattro modelli sviluppati, la scelta dell'uno rispetto all'altro dipende dall'applicazione specifica e dai parametri di priorità. Il modello con *image size* pari a 64×64 risulta particolarmente utile nei casi in cui sia necessario eseguire più inferenze al secondo e rilevare istanze a breve distanza. Al contrario, se l'attenzione è rivolta alla massima accuratezza e alla capacità di individuare soggetti a distanze di alcuni metri, i modelli con *image size* 128×128 o 160×160 rappresentano le soluzioni più adatte.

Il modulo complessivo, costituito dal microcontrollore, dalla camera e dal modello sviluppato, può trovare numerose applicazioni pratiche: sia come sistema autonomo, sia come parte integrante di un sistema più ampio, in cui i risultati dell'elaborazione (come la presenza di istanze e le *bounding boxes* rilevate) vengono trasmessi ad altri dispositivi, con unità di controllo proprie, responsabili di prendere decisioni.

Per quanto riguarda gli sviluppi futuri, un primo passo rilevante sarebbe l'ampliamento del *dataset* di *domain adaptation*, che costituisce un pilastro fondamentale per migliorare l'accuratezza complessiva del modello. Ulteriori miglioramenti potrebbero riguardare la struttura architetturale della rete, riducendone ulteriormente la complessità o sostituendone interi moduli, come la *backbone*. Si potrebbe inoltre variare la classe oggetto di rilevazione, modificando quindi lo scopo applicativo, oppure estendere

il modello a una configurazione multiclasse. Infine, sarebbe interessante valutare l'estensione di questo approccio ad altri modelli di *computer vision*, per analizzarne vantaggi e limiti in confronto a quello implementato in questo contesto.

Elenco delle figure

Elenco delle tabelle

Bibliografia

[1] J. Schmidhuber, "Deep learning in neural networks: An overview," *Neural Networks*, vol. 61, p. 85–117, Jan. 2015. [Online]. Available: http://dx.doi.org/10.1016/j.neunet.2014.09.003

[2] K. O'Shea and R. Nash, "An introduction to convolutional neural networks," 2015. [Online]. Available: https://arxiv.org/abs/1511.08458

[3] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, "You only look once: Unified, real-time object detection," 2016. [Online]. Available: https://arxiv.org/abs/1506.02640

[4] S. S. Saha, S. S. Sandha, and M. Srivastava, "Machine learning for microcontroller-class hardware: A review," *IEEE Sensors Journal*, vol. 22, no. 22, p. 21362–21390, Nov. 2022. [Online]. Available: http://dx.doi.org/10.1109/JSEN.2022.3210773

[5] D. Liu, Y. Zhu, Z. Liu, Y. Liu, C. Han, J. Tian, R. Li, and W. Yi, "A survey of model compression techniques: past, present, and future," *Frontiers in Robotics and AI*, vol. Volume 12 - 2025, 2025. [Online]. Available: https://www.frontiersin.org/journals/robotics-and-ai/articles/10.3389/frobt.2025.1518965

[6] Espressif Systems. (2025) Esp32-s3 documentation. [Online]. Available: https://www.espressif.com/sites/default/files/documentation/esp32-s3_technical_reference_manual_en.pdf

[7] ——. (2024) Esp-idf programming guide, esp32 series. [Online]. Available: https://docs.espressif.com/projects/esp-idf/en/stable/esp32/index.html

[8] Real Time Engineers Ltd. (2024) Freertos documentation overview. [Online]. Available: https://www.freertos.org/Documentation/00-Overview

[9] R. Khanam and M. Hussain, "Yolov11: An overview of the key architectural enhancements," 2024. [Online]. Available: https://arxiv.org/abs/2410.17725

[10] N. Jegham, C. Y. Koh, M. Abdelatti, and A. Hendawi, "Yolo evolution: A comprehensive benchmark and architectural review of yolov12, yolo11, and their previous versions," 2025. [Online]. Available: https://arxiv.org/abs/2411.00201

[11] Ultralytics. (2024) Ultralytics yolov11 documentation. [Online]. Available: https://docs.ultralytics.com/models/yolo11/

[12] T.-Y. Lin, M. Maire, S. Belongie, L. Bourdev, R. Girshick, J. Hays, P. Perona, D. Ramanan, C. L. Zitnick, and P. Dollár, "Microsoft coco: Common objects in context," 2015. [Online]. Available: https://arxiv.org/abs/1405.0312

[13] J. Moosmann, H. Müller, N. Zimmerman, G. Rutishauser, L. Benini, and M. Magno, "Flexible and fully quantized lightweight tinyissimoyolo for ultra-low-power edge systems," *IEEE Access*, vol. 12, p. 75093–75107, 2024. [Online]. Available: http://dx.doi.org/10.1109/ACCESS.2024.3404878

[14] M. Jeon, J. Seo, and J. Min, "Da-raw: Domain adaptive object detection for real-world adverse weather conditions," 2024. [Online]. Available: https://arxiv.org/abs/2309.08152

[15] C.-Y. Wang, A. Bochkovskiy, and H.-Y. M. Liao, "Scaled-yolov4: Scaling cross stage partial network," 2021. [Online]. Available: https://arxiv.org/abs/2011.08036

[16] Y. Hao, H. Pei, Y. Lyu, Z. Yuan, J.-R. Rizzo, Y. Wang, and Y. Fang, "Understanding the impact of image quality and distance of objects to object detection performance," 2022. [Online]. Available: https://arxiv.org/abs/2209.08237

[17] V. Gandhi and S. Gandhi, "Fine-tuning without forgetting: Adaptation of yolov8 preserves coco performance," 2025. [Online]. Available: https://arxiv.org/abs/2505.01016

[18] M. Nagel, M. van Baalen, T. Blankevoort, and M. Welling, "Data-free quantization through weight equalization and bias correction," 2019. [Online]. Available: https://arxiv.org/abs/1906.04721