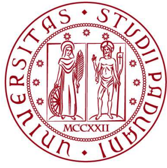


UNIVERSITÀ
DEGLI STUDI
DI PADOVA

DIPARTIMENTO
DI INGEGNERIA
DELL'INFORMA
ZIONE

DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE

CORSO DI LAUREA IN INGEGNERIA INFORMATICA

"EMBEDDED COMPUTER VISION PER RILEVAMENTO OGGETTI"

Relatore: Prof. / Dott Nicola Bellotto

Laureando/a: Davide Baggio

ANNO ACCADEMICO 2022 – 2023

Data di laurea 29/09/2023

"Siamo uomini di classe, signori si nasce, non si diventa."

— FC

Ringraziamenti

Ringrazio innanzitutto il Prof. Nicola Bellotto, relatore della mia tesi, per l'aiuto fornitomi durante la stesura del lavoro.

Ringrazio con affetto i miei genitori e mia sorella per essermi stati vicini in ogni momento durante gli anni di studio.

E infine ringrazio la mia ragazza e i miei amici per tutti i bellissimi anni passati insieme e le avventure vissute.

Padova, Settembre 2023 Davide Baggio

Sommario

La visione artificiale è un campo in continua crescita nell'ambito della tecnologia dell'informazione, con applicazioni in settori diversi come il monitoraggio industriale, la sorveglianza, la robotica e l'automazione. Questa tesi si propone di esplorare l'implementazione di sistemi di computer vision embedded per il rilevamento di oggetti, capitalizzando l'efficienza di dispositivi come Arduino e schede integrate. L'obiettivo è realizzare una soluzione che permetta l'identificazione in tempo reale di oggetti, superando le limitazioni di risorse computazionali e di memoria tipiche dei dispositivi embedded. L'utilizzo di tecnologie come Arduino offre vantaggi significativi in termini di costo, consumo energetico e dimensioni compatte, rendendo il sistema adatto a situazioni in cui portabilità ed efficienza sono cruciali. Questo studio si concentra sulla progettazione e l'implementazione di algoritmi leggeri di computer vision, capaci di operare su hardware con risorse limitate senza compromettere l'accuratezza e la robustezza del rilevamento. Un aspetto distintivo di questa tesi è l'approccio pratico adottato, con l'effettiva realizzazione di un sistema di riconoscimento oggetti su piattaforma embedded. I risultati ottenuti saranno valutati attraverso test e confronti con soluzioni tradizionali. In sintesi, questa tesi mira a dimostrare l'efficacia di un sistema di computer vision embedded per il rilevamento di oggetti, aprendo opportunità in vari settori. L'utilizzo di dispositivi compatti come Arduino democratizza questa tecnologia, rendendola accessibile anche in contesti con risorse limitate e aprendo nuove prospettive nello sviluppo della visione artificiale.

Indice

1 Introduzione **1**

1.1 Motivazioni 1

1.2 Obiettivi 2

2 Stato dell'arte **3**

2.1 Computer vision 3

2.1.1 Le immagini 3

2.1.2 Deep learning 4

2.1.3 MobileNets 6

2.2 I microcontrollori 7

2.2.1 Microcontrollori noti 8

2.3 Machine learning su microcontrollori 9

2.4 I software e le librerie più utilizzate 10

3 Ambiente di sviluppo per computer vision **13**

3.1 Hardware 13

3.2 Hardware con potenza di calcolo inferiore 14

3.3 Software e librerie 15

3.3.1 Edge Impulse 15

3.3.2 YOLOv8 16

3.3.3 Arduino IDE 16

4 Sviluppo e preparazione del modello di computer vision **19**

4.1 La raccolta dei dati 19

4.2 Il dataset 19

4.3 Training della rete neurale 20

4.3.1 Edge Impulse 20

4.3.2 YOLOv8 22

4.4 Gli ottimizzatori 23

4.5 Un processo fondamentale: Quantizzazione 25

4.6 Training: Risultati 25

4.7 Esportazione del modello 28

5 Esecuzione del codice su schede **29**

5.1 Configurazione dell'ambiente di sviluppo 29
5.2 Il modello YOLOv8 30
5.3 La cattura delle Immagini 31
5.4 Inferenza in tempo reale 31
5.5 Upload del Codice 32

6 Test e risultati **33**
6.1 EON e acceleratore hardware 34

7 Conclusioni **35**
7.1 Ostacoli più comuni 35
7.2 I risultati di YOLOv8 37
7.3 Migliorie e sviluppi futuri 38

Bibliografia **39**

Frammenti dei Codici **43**

Elenco delle figure

2.1 Architettura di una Deep Neural Network [17] 4
2.2 Tipiche funzioni di Attivazione [17] 5
2.3 Filtri di Convoluzione Standard 6
2.4 Filtri di Convoluzione in profondità 6
2.5 Filtri di Convoluzione in spazio 6
2.6 Struttura di un Microcontrollore [23] 8

4.1 Vettori delle "features" 21
4.2 Stochastic Gradient Descent 24
4.3 Batch Gradient Descent 24
4.4 mini Batch Gradient Descent 24
4.5 Parametri MobileNetV2 26
4.6 Specifiche MobileNetV2 26
4.7 Performance dell'addestramento Edge Impulse 27
4.8 Performance dell'addestramento YOLOv8 27
4.9 Specifiche YOLOv8 27

5.1 OV7675 29
5.2 *Arduino Nano 33 BLE* e *OV7675* Pinout [1] 30

6.1 Differenza dati e risultati 33
6.2 Risultati: utilizzo memoria 34

7.1 Risultati: utilizzo memoria 37

Elenco delle tabelle

2.1 Standard MobileNet vs MobileNet (filtri convoluzione separabili) [10] 7

3.1 Schede utilizzate 13

Elenco delle Equazioni

4.1 Precision 25
4.2 Recall 26
4.3 F1 Score 26

Termininologia

IA: Intelligenza Artificiale

IoT: Internet of Things

CPU: Central Processing Unit

GPU: Graphics Processing Unit

RAM: Random Access Memory

ROM: Read Only Memory

PSRAM: Pseudo Static RAM

FLASH: Solid state memory

FAT: File Allocation Table (File System)

SPI: (Serial Peripheral Interface)

I2C: Inter Integrated Circuit

TINYML: Tiny Machine Learning

CNN: Convolutional Neural Network

YOLO: You Only Look Once

LOGITS: misure dell'affidabilità

MIPS: Million Instructions Per Second

OPI PSRAM: Octa Peripheral Interface PSRAM

UART: Universal Asynchronous Receiver/-Transmitter

Capitolo 1

Introduzione

Negli ultimi decenni, l'area di ricerca della Computer Vision ha conosciuto un rapido sviluppo, portando a significativi avanzamenti nelle applicazioni reali. Tra queste applicazioni, i sistemi embedded si sono affermati come uno dei principali ambienti di sviluppo, grazie alla loro crescente pervasività in una vasta gamma di dispositivi e applicazioni, tra cui robotica, veicoli autonomi, telefoni cellulari, telecamere di sicurezza e molti altri [26].

La scelta di focalizzare questa tesi sulla Computer Vision per sistemi embedded è guidata da diverse motivazioni che saranno discusse in questo capitolo introduttivo. Queste motivazioni sono strettamente legate agli obiettivi di ricerca che ci siamo prefissati e alle dimostrazioni che intendiamo sviluppare nel corso di questa tesi.

1.1 Motivazioni

Le ragioni che spingono a dedicare questa tesi alla Computer Vision per sistemi embedded sono molteplici.

In un primo luogo i sistemi embedded sono diventati una parte integrante della nostra vita quotidiana. Troviamo questi sistemi in elettrodomestici, dispositivi medici, dispositivi IoT, droni e molte altre applicazioni. La capacità di dotare questi dispositivi di funzionalità di Computer Vision apre nuove opportunità per migliorare l'efficienza, l'automazione e l'intelligenza di tali sistemi. Creare dunque un modello di questo tipo permetterebbe di studiare un ambiente in continuo aggiornamento esplorando le implementazioni tipiche e valutando le possibili criticità.

In secondo luogo i sistemi embedded sono spesso alimentati da batterie o da fonti energetiche limitate, il che rende essenziale l'ottimizzazione delle risorse energetiche. La progettazione di algoritmi di Computer Vision che siano efficienti dal punto di vista energetico è una sfida importante, ma anche una necessità per garantire la praticità e la durata delle batterie dei dispositivi.

E infine una delle motivazioni più importanti è la possibilità di creare un sistema isolato in grado di fornire stabilità e sicurezza durante le operazioni di riconoscimento degli oggetti, al contrario di un ordinario personal computer che comunica spesso in modo indiscriminato con la rete e che è facilmente accessibile da altri dispositivi.

1.2 Obiettivi

Gli obiettivi che questa tesi si pone sono:

- Fornire una panoramica sullo stato dell'arte del Computer Vision evidenziando le sfide chiave, i risultati significativi e le aree di ricerca in evoluzione.
- Progettare e implementare algoritmi di Computer Vision ottimizzati per sistemi embedded, con un'attenzione particolare alla gestione efficiente delle risorse computazionali ed energetiche.
- Creare un sistema di Computer Vision sicuro e isolato in grado di svolgere inferenza in tempo reale e riconoscere con una discreta precisione gli oggetti per il quale è stato addestrato.

In sintesi, questa tesi si propone di esplorare e sviluppare soluzioni avanzate di Computer Vision per sistemi embedded, affrontando le sfide specifiche di questa disciplina e contribuendo così a migliorare la qualità e l'efficienza dei dispositivi embedded nella nostra vita quotidiana.

Nel capitolo successivo, procederemo con una revisione approfondita della letteratura esistente nel campo della Computer Vision per sistemi embedded, discutendo delle soluzioni hardware e software più affidabili.

In seguito si parlerà dei dispositivi e framework che verranno utilizzati nel progetto, con particolare attenzione alle specifiche e performance dichiarate dai produttori. Verrà poi addestrata una rete neurale con caratteristiche particolari adatta ai dispositivi embedded e con il compito di svolgere inferenza in tempo reale per il riconoscimento di oggetti. Sarà necessario poi eseguire il codice sulle schede facendo attenzione ai metodi di caricamento e relativi bootloader. Infine verranno dedicati due capitoli nei quali si eseguiranno i test e le valutazioni per poi passare alle considerazioni finali spiegando anche le criticità di un progetto di questo tipo.

Capitolo 2

Stato dell’arte

2.1 Computer vision

Computer Vision è un campo dell’intelligenza artificiale che mira a insegnare ai computer a "vedere" e comprendere il mondo visuale come lo fanno gli esseri umani. Un aspetto fondamentale della Computer Vision è la gestione delle immagini come input, poiché queste costituiscono una delle principali fonti d’informazioni visive.

2.1.1 Le immagini

Le immagini [17] sono rappresentate digitalmente come una matrice di pixel, dove ciascun pixel corrisponde a un singolo punto dell’immagine. Una delle scelte fondamentali da fare riguarda il tipo di immagine da utilizzare come input per l’analisi e l’elaborazione. Due tipi d’immagini comunemente utilizzati sono le immagini RGB (Red, Green, Blue) e le immagini grayscale (scala di grigi).

Le immagini RGB sono composte da tre canali di colore - rosso, verde e blu - per ciascun pixel. Ogni canale rappresenta l’intensità del colore corrispondente e la combinazione di questi tre canali produce una vasta gamma di colori visibili. Questo tipo di immagine è ampiamente utilizzato per la visione a colori e consente di catturare e analizzare dettagli cromatici.

D’altra parte, le immagini grayscale hanno un solo canale di colore per pixel e vengono rappresentate in scala di grigi. Ogni pixel ha un valore che varia da 0 a 255, dove 0 rappresenta il nero assoluto e 255 il bianco assoluto. Questo tipo di immagine viene utilizzato per semplificare l’analisi, in quanto riduce la complessità del calcolo e richiede meno risorse di archiviazione rispetto alle immagini RGB.

La scelta tra immagini RGB e grayscale dipende dalla natura del problema che si vuole risolvere e dalle informazioni necessarie per l’elaborazione. Se l’analisi si concentra principalmente sulle caratteristiche cromatiche, l’uso di immagini RGB è consigliato per catturare i dettagli del colore. Al contrario, se l’obiettivo è riconoscere forme, contorni o texture, le immagini grayscale possono essere sufficienti, consentendo una maggiore efficienza computazionale.

2.1.2 Deep learning

Il Deep Learning [17] è una branca dell'intelligenza artificiale (IA) che si basa sull'uso di reti neurali profonde per l'apprendimento automatico delle rappresentazioni dei dati. Questo approccio consente alle macchine di apprendere direttamente dai dati, senza la necessità di essere programmate esplicitamente per compiere specifiche attività. Le reti neurali profonde, chiamate così perché sono composte da molteplici strati di neuroni artificiali, sono in grado di elaborare dati complessi e di riconoscere modelli e caratteristiche nascoste nei dati. Grazie alla loro capacità di apprendere rappresentazioni gerarchiche dei dati, il Deep Learning ha rivoluzionato diversi campi, come la computer vision, il riconoscimento del linguaggio naturale, il riconoscimento vocale e molte altre applicazioni che richiedono analisi avanzate di grandi quantità di dati. Il successo del Deep Learning è stato alimentato dalla crescente disponibilità di dati di addestramento, dall'aumento della potenza di calcolo delle GPU e dalla continua ricerca e sviluppo di nuove architetture di reti neurali sempre più efficienti e performanti.

Le Convolutional Neural Network (CNN) [17] sono un tipo di architettura di reti neurali profonde (deep learning) ampiamente utilizzate nel campo della computer vision. Queste reti sono progettate appositamente per l'analisi di immagini e sono caratterizzate da alcune importanti caratteristiche che le rendono particolarmente efficaci nel riconoscimento di pattern visivi complessi.

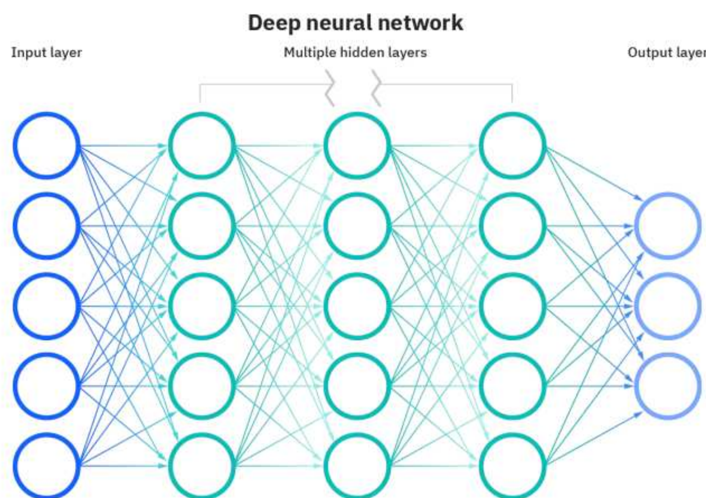


Figura 2.1: Architettura di una Deep Neural Network [17]

- Convoluzione: La convoluzione è il cuore delle CNN. Consiste nell'applicazione di uno o più filtri (chiamati kernel) all'immagine di input per estrarre caratteristiche rilevanti. Durante questa operazione, il filtro si sposta su tutta l'immagine eseguendo operazioni di moltiplicazione tra i valori dei pixel e i pesi del filtro e sommando i risultati. Questo processo crea una mappa delle caratteristiche (feature map) che evidenzia i pattern rilevanti presenti nell'immagine.

- Strato di Pooling: Dopo la fase di convoluzione, è comune utilizzare uno strato di pooling per ridurre la dimensione della mappa delle caratteristiche e ridurre l'overfitting. Lo strato di pooling riduce la dimensionalità dell'immagine, riducendo il numero di parametri e rendendo la rete più gestibile. Tipi comuni di pooling sono il max pooling e il mean pooling.

- Funzioni di Attivazione: Per introdurre non linearità nella rete, dopo ogni strato di convoluzione o pooling, viene applicata una funzione di attivazione. La più comune è la Rectified Linear Unit (ReLU), che imposta a zero tutti i valori negativi, introducendo una non linearità e migliorando la capacità della rete di apprendere caratteristiche complesse.

- Strato Fully Connected: Dopo diverse fasi di convoluzione e pooling, la rete può includere uno o più strati fully connected (o dense). Questi strati prendono le caratteristiche estratte e le collegano a una o più unità di output, tipicamente utilizzate per la classificazione o il rilevamento di oggetti.

- Condivisione dei Pesi: Una delle caratteristiche distintive delle CNN è la condivisione dei pesi. Questo significa che i filtri utilizzati durante la convoluzione sono applicati in modo identico su diverse parti dell'immagine. Questa tecnica riduce il numero di parametri della rete e permette di apprendere pattern invarianti rispetto alla posizione, rendendo la rete più efficiente e robusta.

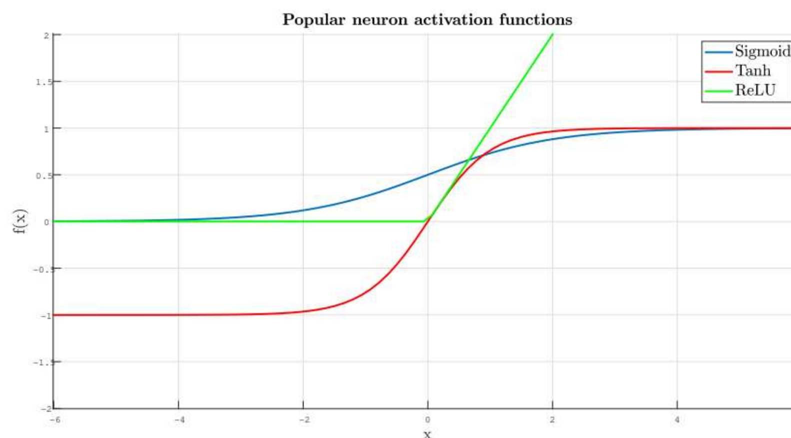


Figura 2.2: Tipiche funzioni di Attivazione [17]

Grazie a queste caratteristiche, le Convolutional Neural Network si sono dimostrate molto potenti nel riconoscimento di pattern visivi, come oggetti, volti, caratteri e molto altro. Sono ampiamente utilizzate in applicazioni di computer vision, come rilevamento oggetti, classificazione di immagini, elaborazione delle immagini mediche, guida autonoma e molto altro. La capacità delle CNN di apprendere automaticamente caratteristiche rilevanti dalle immagini e di gestire grandi quantità di dati le rende uno strumento fondamentale per l'analisi visiva e l'intelligenza artificiale.

2.1.3 MobileNets

I Modelli MobileNets [10] rappresentano una famiglia di architetture di reti neurali profonde ottimizzate per l'esecuzione su dispositivi mobili e a risorse limitate con lo scopo di implementare sistemi *tinyML* (machine learning su piccoli dispositivi). Sviluppati da Google, questi modelli sono progettati specificamente per affrontare le sfide di efficienza computazionale e di consumo energetico, consentendo il deploy di applicazioni di computer vision avanzate su smartphone, dispositivi IoT e altre piattaforme embedded.

La caratteristica distintiva dei MobileNets è l'utilizzo di due strategie principali: la convoluzione separabile e la convoluzione a stride variabile [10]. La convoluzione separabile divide l'operazione di convoluzione in due fasi separate: una convoluzione in profondità e una convoluzione in spazio. Questo permette di ridurre notevolmente il numero di operazioni di moltiplicazione e di somma, riducendo così il carico computazionale. La convoluzione a stride variabile, invece, consente di controllare la dimensione delle feature map durante la fase di convoluzione, permettendo un'ulteriore riduzione delle dimensioni e dei calcoli.

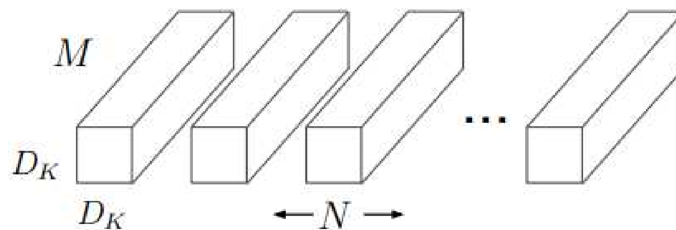


Figura 2.3: Filtri di Convoluzione Standard

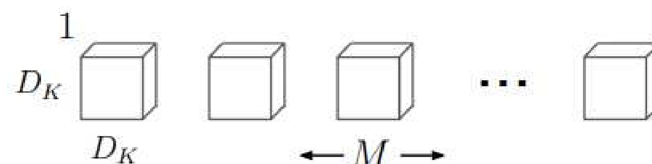


Figura 2.4: Filtri di Convoluzione in profondità

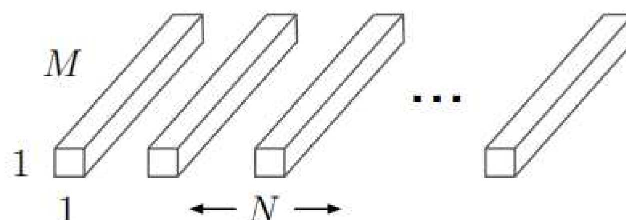


Figura 2.5: Filtri di Convoluzione in spazio

L'uso di queste strategie rende i MobileNets molto più leggeri rispetto ad altre architetture di CNN, pur mantenendo un'elevata precisione e un'ottima capacità di apprendimento delle

caratteristiche dell'immagine. Questi modelli sono particolarmente adatti per applicazioni in tempo reale, come il riconoscimento di oggetti, la classificazione di immagini e il tracciamento facciale, in cui è necessario ottenere risultati rapidi ed efficienti su dispositivi con risorse limitate.

Gli esempi dimostrano che l'utilizzo di filtri di convoluzione separabili anziché quelli standard consente di ridurre il numero di parametri di oltre l'85%.

Modello	Precisione ImageNet	Operazioni di Multiply-Accumulate (milioni)	Milioni di Parametri
Conv MobileNet	71.7%	4866	29.3
MobileNet	70.6%	569	4.2

Tabella 2.1: Standard MobileNet vs MobileNet (filtri convoluzione separabili) [10]

Grazie alla loro versatilità e alle prestazioni ottimali su piattaforme a basso consumo, i modelli MobileNets sono diventati una scelta popolare per lo sviluppo di applicazioni di computer vision su dispositivi mobili ed embedded. La continua ricerca e l'ottimizzazione delle architetture di MobileNets stanno rendendo sempre più accessibili e potenti le soluzioni di deep learning su una vasta gamma di dispositivi, aprendo nuove opportunità nell'ambito dell'intelligenza artificiale distribuita e on-device.

In MobileNets è presente un parametro α. Il parametro α è un iper parametro che controlla la larghezza della rete neurale. Esso controlla il numero di canali nelle diverse fasi del modello, consentendo di ridurre la complessità e la dimensione complessiva del modello.

Più specificamente, α è un valore compreso tra 0 e 1 (di solito tra 0,25 e 1) che moltiplica il numero di filtri nelle diverse fasi dell'architettura. Quando α è 1, la rete avrà le dimensioni e la complessità standard (MobileNet originale). Se α è minore di 1, si riducono proporzionalmente il numero di filtri e, di conseguenza, il numero complessivo di parametri della rete.

2.2 I microcontrollori

I microcontrollori [5, 15] rappresentano una classe fondamentale di dispositivi elettronici che svolgono un ruolo cruciale nell'automazione e nell'elettronica embedded. Sono presenti in numerose applicazioni quotidiane, spaziando dalle apparecchiature domestiche agli strumenti industriali, dai dispositivi medici alle automobili, e sono essenziali per il funzionamento di una vasta gamma di sistemi intelligenti e interattivi.

Questi dispositivi integrano su un unico chip una CPU, una quantità di memoria RAM e ROM, interfacce di input/output e, talvolta, periferiche specializzate. La loro caratteristica principale è l'autonomia e la capacità di eseguire compiti predefiniti in modo ripetitivo e affidabile, sfruttando poche risorse di calcolo.

L'introduzione dei microcontrollori ha rivoluzionato il modo in cui progettiamo e realizziamo sistemi elettronici. Prima della loro diffusione, molte funzioni richiedevano circuiti complessi e costosi; oggi, un microcontrollore può gestire tali funzioni in modo efficiente ed economico.

La miniaturizzazione dei processi produttivi ha portato a un continuo aumento della potenza di calcolo dei microcontrollori, permettendo di implementare algoritmi sofisticati e funzionalità

avanzate, come la comunicazione wireless, il riconoscimento di pattern e persino l’intelligenza artificiale su piattaforme embedded.

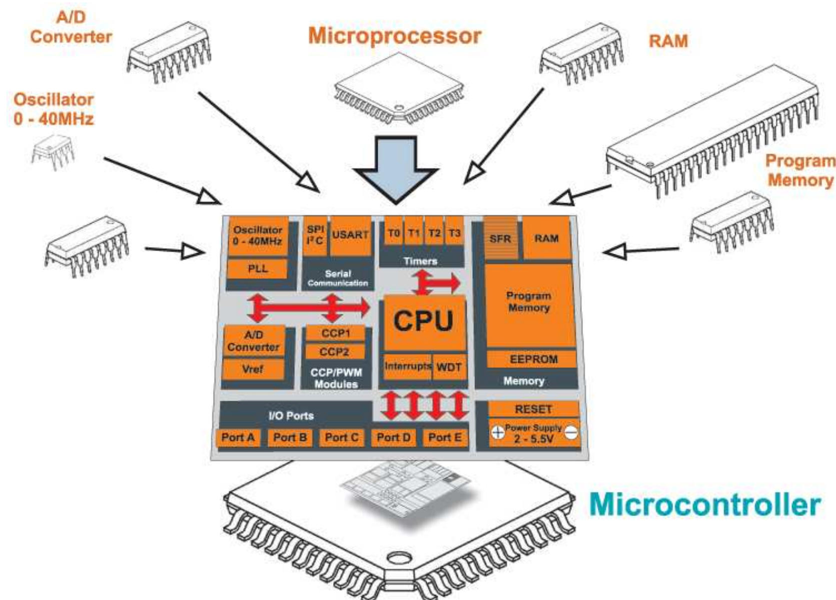


Figura 2.6: Struttura di un Microcontrollore [23]

In questa tesi, ci concentreremo sull’analisi delle caratteristiche, delle prestazioni e delle applicazioni dei microcontrollori, esaminando come essi si integrino nei dispositivi embedded e nella realizzazione di progetti pratici. Verranno esplorate le sfide e le opportunità nell’utilizzo di tali dispositivi, evidenziando le principali linee di ricerca e sviluppo che stanno plasmando il futuro dei microcontrollori e delle tecnologie a loro associate.

I microcontrollori rappresentano uno degli elementi centrali della rivoluzione tecnologica in corso, poiché sono alla base di molte innovazioni che hanno cambiato il nostro modo di vivere e lavorare.

2.2.1 Microcontrollori noti

Di seguito una breve lista dei microcontrollori più conosciuti e con il quale avviene lo sviluppo di sistemi intelligenti [15, 20].

- *Atmega* (di Microchip Technology, utilizzati in schede Arduino): La famiglia Atmega è il cuore di diverse schede Arduino e si distingue per la sua semplicità di programmazione e l’ampia comunità di supporto.

- *Raspberry Pi*: Anche se tecnicamente è un singolo computer a scheda ridotta (SBC - Single Board Computer), è comune includerlo in questa lista poiché offre funzionalità simili a un microcontrollore, oltre a un’ampia gamma di capacità di elaborazione e connettività.

- *STM32* (Famiglia STM32 di STMicroelectronics): Questa famiglia di microcontrollori basati su core ARM Cortex-M è ampiamente utilizzata in diverse applicazioni industriali e di automazione.

- *ESP8266 e ESP32* (di Espressif Systems): Questi microcontrollori sono noti per la loro connettività Wi-Fi e Bluetooth, rendendoli ideali per progetti IoT (Internet of Things).
- *PIC* (di Microchip Technology): I microcontrollori PIC sono stati utilizzati a lungo in applicazioni industriali, mediche e automotive, e sono apprezzati per la loro stabilità e affidabilità.
- *MSP430* (di Texas Instruments): Questi microcontrollori sono noti per il loro basso consumo energetico, rendendoli ideali per dispositivi a batteria o applicazioni a basso consumo.
- *NXP* LPC (Famiglia LPC di NXP Semiconductors): Una serie di microcontrollori ad alte prestazioni e basso consumo, ampiamente utilizzati in applicazioni industriali, automobilistiche e IoT.
- *AVR* (di Microchip Technology): La famiglia di microcontrollori AVR è stata popolare per molti anni e viene utilizzata in diverse applicazioni, inclusi progetti industriali e hobbistici.
- *Teensy* (di PJRC): Questi microcontrollori sono noti per la loro potenza e velocità, offrendo funzionalità avanzate in una dimensione compatta, rendendoli ideali per progetti con requisiti elevati di prestazioni.

2.3 Machine learning su microcontrollori

Il machine learning riveste un ruolo di fondamentale importanza nell'ambito degli sviluppi dei microcontrollori [5], poiché offre soluzioni capaci di superare le limitazioni hardware tipiche di tali dispositivi. Questa sezione illustrerà le ragioni principali per cui l'applicazione del machine learning è cruciale in questo contesto.

- Ottimizzazione delle risorse: Grazie alla capacità di implementare algoritmi di machine learning leggeri e altamente efficienti, i microcontrollori possono gestire complessi compiti di elaborazione dati pur disponendo di risorse computazionali e memoria limitate. Ciò permette di ottenere risultati significativi anche in ambienti con spazio e potenza ridotti.
- Adattabilità dinamica: I modelli di machine learning possono essere addestrati per apprendere dai dati di input in tempo reale, consentendo ai microcontrollori di adattarsi rapidamente ai cambiamenti nelle condizioni operative o ambientali. Ciò evita la necessità di programmazioni statiche e rigide, consentendo un'interazione più flessibile e responsiva.
- Riconoscimento di pattern: L'introduzione del machine learning permette di sviluppare algoritmi per il riconoscimento di pattern, come il riconoscimento facciale, vocale o la classificazione di oggetti. Queste funzionalità rendono i microcontrollori in grado di interpretare e interagire con l'ambiente circostante in modo intelligente e intuitivo.

- Ottimizzazione dell'energia: La progettazione di algoritmi di machine learning leggeri e ottimizzati per il consumo energetico consente ai microcontrollori di operare in modo più efficiente, riducendo il consumo energetico e ampliando l'autonomia delle batterie. Questa caratteristica è particolarmente importante in applicazioni di sensori a basso consumo energetico.

- Automazione intelligente: L'integrazione del machine learning nei microcontrollori abilita i dispositivi embedded ad apprendere dai dati raccolti, migliorando progressivamente le loro prestazioni e adattandosi a nuove situazioni. Questa capacità di adattamento rende possibile una maggiore automazione in diverse applicazioni.

- Risparmio di tempo e costi: L'utilizzo del machine learning semplifica il processo di sviluppo di soluzioni sofisticate su microcontrollori, riducendo il tempo e le risorse necessarie per la progettazione e l'implementazione manuale di algoritmi complessi.

- Intelligenza decentralizzata: Integrare il machine learning nei microcontrollori consente di realizzare sistemi intelligenti a livello locale, riducendo la dipendenza da comunicazioni costanti con server remoti per l'elaborazione dei dati. Ciò migliora la reattività del sistema e riduce il carico sulla rete.

Il machine learning rappresenta una componente chiave nello sviluppo di microcontrollori intelligenti, efficienti ed efficaci, consentendo l'implementazione di funzionalità avanzate e adattabili in ambienti con risorse hardware limitate. Questa integrazione offre vantaggi significativi e apre prospettive promettenti per l'evoluzione delle tecnologie embedded in svariate applicazioni.

2.4 I software e le librerie più utilizzate

Esistono diverse soluzioni open-source e framework [5] per l'implementazione di modelli di machine learning su dispositivi embedded, con particolare attenzione ai microcontrollori. Alcuni di questi strumenti sono stati specificamente progettati per ridurre l'overhead e ottimizzare l'esecuzione su hardware con risorse limitate.

- TensorFlow Lite (TFL) rappresenta una solida piattaforma utilizzata comunemente per l'addestramento e la distribuzione di reti neurali su dispositivi mobili. TensorFlow Lite Micro (TFLM) è una versione dedicata ai microcontrollori, tuttavia, attualmente, il suo supporto è limitato a specifiche piattaforme hardware e operazioni TensorFlow. È interessante notare che uTensor, basato su TFLM, offre un runtime ancora più leggero e si integra con strumenti di elaborazione grafica e raccolta dati.

- Embedded Learning Library (ELL), sviluppato da Microsoft, costituisce un'alternativa open-source per effettuare il machine learning su dispositivi embedded, inclusi Arduino e micro:bit. Fornisce un compilatore che consente la conversione di modelli esterni in codice binario.

- NanoEdge AI Studio è una soluzione desktop user-friendly pensata appositamente per dispositivi STM32. Questo software permette di creare librerie personalizzate per il progetto, con l'obiettivo di ottimizzare sia l'accuratezza dei modelli che l'utilizzo della memoria.

- Edge Impulse è un servizio cloud focalizzato sullo sviluppo di modelli di machine learning per dispositivi edge. Offre un ambiente di addestramento online e consente l'implementazione dei modelli su diverse piattaforme.

- Artificial Intelligence for Embedded Systems (AIfES) è una libreria versatile che supporta una vasta gamma di microprocessori. È in grado di consentire l'addestramento di reti neurali direttamente sui dispositivi, sebbene attualmente sia limitato alle reti feed-forward, con piani futuri di integrazione per reti convoluzionali.

- microTVM è un'estensione del framework Tensor Virtual Machines (TVM) focalizzata sulla distribuzione di reti neurali su microcontrollori. Tuttavia, è ancora in fase di sviluppo e il suo supporto per diversi dispositivi è limitato.

- Seedot è un linguaggio specifico progettato per la creazione di algoritmi di machine learning su microcontrollori, introducendo tecniche di riduzione dello spazio di ricerca e ottimizzazioni per le operazioni computazionalmente costose.

- hls4ml è un pacchetto di Python che agevola la conversione di modelli preesistenti in linguaggio High-Level Synthesis (HLS), consentendo la personalizzazione dei modelli secondo le specifiche esigenze. Tuttavia, è importante notare che al momento supporta solo una selezione limitata di tipi di reti.

- YOLOv8 [24] è un modulo Python dedicato all'addestramento di reti neurali per computer vision, consentendo l'esportazione dei modelli in diversi formati, tra cui *.tflite* (TensorFlow Lite).

- MicroFlow [6], un framework di compilazione di modelli neurali scritto in Rust, si distingue per le prestazioni superiori nell'inferenza in tempo reale, permettendo ai sistemi embedded, inclusi quelli con architettura a 8 bit, di utilizzare meno memoria FLASH e RAM mantenendo comunque una precisione comparabile a TensorFlow Lite.

Questi strumenti offrono diverse opzioni e approcci per lo sviluppo di soluzioni di machine learning su microcontrollori, consentendo di adattare le tecnologie alle specifiche esigenze del progetto.

Capitolo 3

Ambiente di sviluppo per computer vision

3.1 Hardware

In questo progetto, sono state utilizzate diverse componenti fondamentali al fine di creare un sistema embedded di computer vision. Il sistema si basa sulla scheda Xiao ESP32S3 [21], che dispone di un microprocessore Xtensa LX7 dual core con velocità fino a 240 MHz, 8 MB di PSRAM e 8 MB di memoria FLASH. Inoltre, questa scheda offre la possibilità di integrare una scheda SD con capacità fino a 32 GB con sistema di file FAT.

Un'altra opzione considerata per il progetto è l'utilizzo della scheda Arduino Nano 33 BLE [16], la quale rappresenta un ambiente meno potente rispetto alla prima opzione. Tuttavia, è comunque presa in considerazione al fine di ottenere una visione complessiva sulle prestazioni del sistema.

Scheda	Microprocessore	Frequenza	Architettura	PSRAM o SRAM	FLASH
XIAO_ESP32S3	Xtensa LX7 dual-core	240MHz	32 bit	8MB	8MB
Arduino Nano 33 BLE	ARM Cortex-M4F	64MHz	32 bit	256KB	1MB

Tabella 3.1: Schede utilizzate

- Scheda *Xiao ESP32S3*: La scheda Xiao ESP32S3 rappresenta la piattaforma di base di questo progetto. Dotata di un microprocessore Xtensa dual core operante fino a 240MHz, 8MB di PSRAM e 8MB di FLASH, offre una potenza di calcolo sufficiente per implementare algoritmi di machine learning ed elaborare dati complessi. Queste caratteristiche consentono di sfruttare le capacità avanzate della scheda per applicazioni di computer vision e machine vision.

  – *Telecamera OV2640*: La telecamera OV2640 è un modulo di imaging compatto e popolare, spesso utilizzato in progetti embedded e IoT. Questa telecamera offre una risoluzione fino a 2 megapixel ed è in grado di catturare immagini e video con dettagli sorprendenti. La sua versatilità la rende ideale per applicazioni di computer vision, sorveglianza, robotica e molto altro ancora.

- *Arduino Nano 33 BLE*: L'Arduino Nano 33 BLE è una piccola ma potente scheda di sviluppo con Bluetooth Low Energy. Basata sul nRF52840, supporta progetti IoT. Implementare una rete neurale per computer vision può consentire il rilevamento oggetti, riconoscimento gesti, classificazione, monitoraggio e filtraggio video in tempo reale, ma richiede ottimizzazioni per la limitata potenza di calcolo e memoria della scheda.
  - *Telecamera OV7675*: La telecamera OV7675 è un modulo di imaging compatto e utilizzato in varie applicazioni embedded. Con una risoluzione massima di 0,3 megapixel, offre la capacità di acquisire immagini e video con discreta qualità.

3.2 Hardware con potenza di calcolo inferiore

L'utilizzo di schede meno potenti come Arduino Uno avrebbe comportato notevoli sfide e limitazioni. La scelta di utilizzare schede più performanti come la scheda Xiao ESP32S3 o la scheda Arduino Nano 33 BLE è stata determinante per garantire un'elaborazione efficiente e precisa dei dati di computer vision. Ad esempio, considerando la scheda Arduino Uno, caratterizzata da un microcontrollore ATmega328P a 8 bit e una frequenza di clock di 16 MHz, ci saremmo trovati ad affrontare limiti significativi in termini di potenza di calcolo e capacità di memoria disponibile.

Il progetto richiede la gestione di complessi algoritmi di machine learning e l'elaborazione di immagini ad alta risoluzione. La presenza di un microprocessore più potente, come quello presente nelle schede scelte, ha permesso di affrontare tali operazioni in modo più efficiente, consentendo di ottenere prestazioni ottimali e tempi di risposta ridotti.

L'architettura a 8 bit di Arduino Uno [25] limita significativamente la capacità di elaborazione dei dati e la precisione nei calcoli. Inoltre, la dimensione ridotta della SRAM, solo 2KB, avrebbe comportato notevoli limitazioni nella memorizzazione dei dati e del codice, rendendo complicata l'implementazione di algoritmi di machine learning e l'elaborazione di immagini ad alta risoluzione.

Un altro aspetto critico sarebbe stata la compatibilità limitata con le librerie e i framework di machine learning, che spesso richiedono architetture a 32 bit per funzionare correttamente. Questo avrebbe comportato la necessità di implementare soluzioni di adattamento o creare versioni personalizzate delle librerie, aumentando la complessità dello sviluppo.

Tuttavia, è importante notare che l'evoluzione delle tecnologie e degli algoritmi di machine learning sta rapidamente portando a modelli sempre più ottimizzati e compatibili con dispositivi a bassa potenza. Si prevede che in futuro, grazie a tecniche di compressione, quantizzazione e ottimizzazione avanzate, i modelli di reti neurali potranno essere notevolmente ridotti in dimensione, rendendo possibile l'implementazione su microcontrollori a basso consumo energetico e a bassa capacità.

Inoltre, ci sono già progetti e framework, come TensorFlow Lite Micro e uTensor, che si concentrano sulla realizzazione di core runtime estremamente leggeri e su strategie di compressione per adattare modelli di machine learning ai dispositivi embedded, inclusi quelli a 8 bit. Questi

sforzi mirano a rendere il machine learning più accessibile anche per i microcontrollori meno potenti, aprendo la strada a nuove applicazioni di computer vision e intelligenza artificiale in ambienti a risorse limitate.

3.3 Software e librerie

Il software utilizzato invece si divide in base a due processi distinti ovvero il *training* di una rete neurale in grado di riconoscere degli oggetti e l'effettiva implementazione di un sistema embedded in grado di effettuare un *inference* in *real-time*.

Per il *training* di una rete neurale e la successiva esportazione del modello generato, è stato deciso di utilizzare Edge Impulse con la controparte più precisa ma decisamente più pesante in termini di memoria YOLOv8.

3.3.1 Edge Impulse

L'impiego di Edge Impulse [8] nella realizzazione della rete neurale per il nostro sistema embedded è stato guidato da molteplici considerazioni tecniche e pratiche. La piattaforma Edge Impulse ha dimostrato di offrire un ecosistema altamente specializzato e ottimizzato per lo sviluppo di modelli di intelligenza artificiale destinati a dispositivi edge con limitate capacità computazionali. Il suo approccio all-in-one ha significativamente semplificato il processo di sviluppo, consentendo la raccolta e l'etichettatura dei dati di addestramento all'interno di un'interfaccia intuitiva e ben strutturata. Questo si è rivelato particolarmente vantaggioso in termini di efficienza temporale, permettendoci di concentrare maggiormente l'attenzione sulla progettazione del modello stesso e sulla sua adattabilità alle specifiche esigenze del nostro sistema embedded.

Un aspetto cruciale che ha influenzato la scelta di Edge Impulse è la sua capacità di ottimizzare automaticamente i modelli neurali attraverso l'uso di tecniche di pruning e quantizzazione. Questa funzionalità è risultata essenziale nel garantire che la rete neurale potesse essere esportata con successo su una piattaforma embedded, preservando al contempo le prestazioni e l'accuratezza del modello. Inoltre, Edge Impulse ha fornito strumenti di validazione e test che ci hanno permesso di valutare l'efficacia del modello su dati precedentemente non visti, consentendoci di ottimizzarlo ulteriormente prima dell'implementazione finale.

Un ulteriore vantaggio dell'utilizzo di Edge Impulse è stata la sua integrazione semplificata con librerie e framework ampiamente utilizzati nell'ecosistema embedded, consentendo una transizione agevole dalla fase di sviluppo alla fase di integrazione. Questo ha garantito una maggiore flessibilità e facilità nell'adattare il modello alle specifiche esigenze del sistema embedded, senza dover affrontare complessità e sfide tecniche aggiuntive.

Complessivamente, la decisione di impiegare Edge Impulse come piattaforma chiave nella creazione della rete neurale per il nostro sistema embedded si è dimostrata fondamentale per raggiungere il giusto equilibrio tra prestazioni, efficienza e compatibilità. La sua combinazione di strumenti intuitivi, ottimizzazione automatica e integrazione agevole ha contribuito in modo

significativo al successo del nostro progetto, permettendoci di sviluppare una rete neurale performante e pronta per l'esportazione su un ambiente embedded.

3.3.2 YOLOv8

L'adozione di YOLOv8 [24] nel processo di addestramento della rete neurale ha rappresentato una scelta tattica per affrontare sfide specifiche di rilevamento e classificazione nell'ambito del nostro progetto di computer vision. YOLOv8, noto per la sua elevata accuratezza nel rilevamento oggetti in tempo reale, ha dimostrato un potenziale significativo nell'incremento delle performance dell'analisi visiva. Tuttavia, è importante sottolineare che questa scelta ha portato a un trade-off tra prestazioni e consumo di risorse. I modelli generati a partire da YOLOv8 risultano anche 100 volte più grandi in termini di memoria rispetto a quelli sviluppati attraverso Edge Impulse.

Questa considerazione si è rivelata cruciale nel contesto di una piattaforma embedded con risorse computazionali limitate. Il notevole aumento della dimensione dei modelli può comportare sfide nella gestione della memoria e nell'efficienza del calcolo, compromettendo la capacità del sistema embedded di eseguire l'analisi visiva con fluidità e reattività. Tuttavia, la decisione di considerare YOLOv8 è stata guidata dalla necessità di massimizzare l'accuratezza del rilevamento oggetti, soprattutto in scenari complessi in cui la precisione è prioritaria.

3.3.3 Arduino IDE

Nel corso della fase di sviluppo dei nostri progetti, la decisione di adottare Arduino IDE come principale ambiente di programmazione ha dimostrato di essere una scelta non solo pragmatica ma anche strategicamente vantaggiosa. Questo software, noto per la sua interfaccia utente intuitiva e la vasta community di supporto, si è rivelato fondamentale per semplificare il processo di caricamento dei programmi sia sulla scheda Xiao_ESP32S3 che sulla scheda Arduino Nano 33 BLE. La familiarità di Arduino IDE ha ridotto significativamente la curva di apprendimento, consentendoci di concentrarci maggiormente sulla progettazione e l'implementazione dei programmi stessi, anziché sul superamento di ostacoli tecnici relativi all'ambiente di sviluppo.

È opportuno sottolineare che, sebbene sia possibile ottenere direttamente il firmware precompilato per entrambe le schede, questa opzione introduce alcune considerazioni importanti. I firmware precompilati, sebbene semplifichino il processo di implementazione, limitano notevolmente la nostra capacità di apportare personalizzazioni e adattamenti specifici alle esigenze del nostro progetto. Questa restrizione potrebbe rivelarsi vincolante in futuro, specialmente se dovesse emergere la necessità di ottimizzare ulteriormente i programmi o affrontare situazioni di sviluppo particolarmente complesse.

D'altro canto, l'approccio basato su Arduino IDE ci offre un maggiore controllo e flessibilità sul processo di sviluppo. Questo ambiente di programmazione ci permette di adattare i programmi in modo dinamico e reattivo, affrontando le sfide tecniche che possono emergere durante il processo di sviluppo. Inoltre, la capacità di apportare modifiche personalizzate in ogni fase

del progetto, senza dover affrontare vincoli o limitazioni, è un vantaggio cruciale che potrebbe rivelarsi fondamentale per il successo a lungo termine del nostro lavoro.

Capitolo 4

Sviluppo e preparazione del modello di computer vision

4.1 La raccolta dei dati

In un progetto di computer vision, il processo di raccolta del dataset è un passo fondamentale per garantire un'adeguata base di dati per l'addestramento del modello. La raccolta coinvolge la cattura d'immagini e video rappresentativi del contesto d'interesse, seguita da un'accurata etichettatura delle regioni d'interesse all'interno di queste immagini, seguendo lo stile di etichettatura tipica di YOLO [24]. Questo processo richiede un'analisi attenta e dettagliata di ciascuna immagine, individuando e contrassegnando con precisione le aree di interesse in modo che il modello potesse apprendere efficacemente i pattern di riconoscimento.

Questa etichettatura in stile YOLO fornisce al modello le informazioni necessarie per comprendere la posizione e la classe degli oggetti nell'immagine. Ogni etichetta contiene due valori: le coordinate relative al centro della bounding box (indicando la posizione rispetto alle dimensioni dell'immagine) e le dimensioni della bounding box (espressa come altezza e larghezza relative). Inoltre, viene associato un indice numerico a ciascuna classe (ad esempio, 0 per "auto", 1 per "pedone", ecc.), che permette al modello di riconoscere e classificare diversi tipi di oggetti.

Tuttavia, è importante sottolineare che, al fine di risparmiare tempo e risorse, è stata fatta la scelta di utilizzare un dataset preesistente già disponibile. Questo dataset è stato scaricato da Roboflow, una fonte affidabile e ben strutturata per dati di computer vision. L'uso di un dataset pronto ha semplificato notevolmente la fase di raccolta e preparazione dei dati, consentendo di concentrarci maggiormente sull'addestramento e l'ottimizzazione del modello stesso.

4.2 Il dataset

La scelta di utilizzare il dataset "Person Image Dataset" [18] fornito da Roboflow, nonostante la possibilità di raccogliere le immagini direttamente dalle telecamere utilizzate nel progetto, è una decisione strategica mirata a ottimizzare la precisione dei modelli generati. Roboflow offre diversi vantaggi cruciali che giustificano questa scelta. Innanzitutto, fornisce un ampio e

diversificato repository di dati già annotati, consentendo di risparmiare tempo prezioso nella fase di acquisizione e annotazione dei dati, un processo notoriamente dispendioso. In secondo luogo, la disponibilità di dati provenienti da diverse fonti e contesti può arricchire il dataset di addestramento, esponendo il modello a una varietà di situazioni che potrebbero non essere state catturate dalle telecamere locali. Questa diversità nei dati può aumentare la robustezza del modello, rendendolo più affidabile e generalizzato.

Nonostante l'approccio abbia accelerato il processo, l'attenzione alla qualità e alla pertinenza del dataset rimaneva un fattore cruciale per garantire la validità delle prestazioni del modello nel contesto specifico del progetto di analisi visiva.

In questa tesi si prende in considerazione il rilevamento di oggetti generalizzato; è ovvio dunque che se l'applicazione di un modello tinyML deve essere eseguita in un ambiente specifico si consiglia l'utilizzo di un dataset mirato che sia costruito su immagini acquisite dalle telecamere in dotazione evitando alla rete neurale di generare features inutili.

4.3 Training della rete neurale

Per allenare una rete neurale al riconoscimento degli oggetti è possibile utilizzare dei modelli già preesistenti forniti da entrambe le piattaforme viste in precedenza (YOLO o Edge Impulse) oppure crearne una dove i parametri assumono inizialmente un valore casuale.

4.3.1 Edge Impulse

Il processo di addestramento di una rete neurale per il riconoscimento degli oggetti attraverso Edge Impulse rappresenta un'affascinante convergenza tra l'analisi dei dati visivi e l'innovazione tecnologica. Questo processo coinvolge diverse fasi cruciali che culminano nella creazione di un modello predittivo ottimizzato. Inizialmente, i dati visivi vengono raccolti e preparati, sottoposti a una serie di trasformazioni e pre-processing per assicurare che siano adatti all'addestramento.

Questo procedimento coinvolge diverse fasi intricatamente connesse al fine di estrarre informazioni rilevanti dai dati visivi iniziali. In primo luogo, i dati grezzi vengono acquisiti e caricati nella piattaforma Edge Impulse. Qui, attraverso l'interfaccia utente, gli sviluppatori possono applicare una serie di tecniche di pre-processing, come la normalizzazione e la riduzione del rumore, per garantire che i dati siano omogenei e pronti per l'elaborazione. Successivamente, vengono impiegate tecniche di estrazione delle features, come le trasformate di Fourier e l'estrazione di statistiche di base, al fine di catturare aspetti salienti dai dati visivi, come pattern, bordi e texture. Queste features estratte vengono poi organizzate in vettori multidimensionali, che costituiscono l'ingresso della rete neurale.

Figura 4.1: Vettori delle "features"

In questo caso particolare la classe *"0"* indica la classe di elementi *"persona"* e le caratteristiche rappresentate in questo grafico possono essere estratte nel processo precedente all'inferenza *real-time* con un utilizzo massimo di memoria RAM di 4KB in un periodo medio di 11ms.

L'addestramento della rete neurale inizia con la costruzione di un modello di riconoscimento oggetti vincolato. Vengono considerati diversi parametri tra cui il formato dell'input, i pesi per l'inizializzazione di MobileNetV2, il valore di alpha (usato per costruire MobileNetV2) e il numero di classi di oggetti da riconoscere. MobileNetV2 viene inizializzato utilizzando l'architettura pre addestrata e gli eventuali pesi specificati, mentre un nuovo strato convoluzionale è aggiunto per creare un *"head"* che sarà utilizzato per il riconoscimento degli oggetti. L'output del modello consiste in logit di dimensioni *(B, H/8, W/8, num_classes)*, dove B rappresenta la dimensione del batch, H e W sono l'altezza e la larghezza dell'immagine d'input, e *num_classes* è il numero totale di classi, compreso lo sfondo.

La funzione *"train"* è il cuore del processo di addestramento del modello di riconoscimento oggetti vincolato utilizzando il framework Edge Impulse. Questa funzione svolge una serie di passaggi chiave per costruire, addestrare e valutare il modello. Vediamo ora una spiegazione dettagliata dei principali passaggi all'interno di questa funzione:

- Costruzione del Modello: è basato sull'architettura pre addestrata MobileNetV2, che viene inizializzato utilizzando i parametri specificati come la forma dell'input, i pesi pre addestrati (se disponibili), il valore di alpha e il numero totale di classi, inclusa quella di sfondo.

- Preparazione dei Dati: i dataset di addestramento e validazione vengono trasformati

da etichette di bounding box a mappe di segmentazione, che sono adatte al modello di riconoscimento oggetti vincolato. Questo processo permette al modello di apprendere a riconoscere oggetti in base alle mappe di segmentazione dei dati d'input.

- Definizione della Loss Function: Viene definita una funzione di perdita pesata specifica per questa architettura di rete neurale. Questa loss function tiene conto del peso degli oggetti e del background, il che è particolarmente rilevante in un problema di riconoscimento oggetti vincolato.
- Compilazione del Modello: Il modello viene compilato utilizzando la loss function definita e l'ottimizzatore Adam con il tasso di apprendimento specificato.
- Addestramento del Modello: Il modello viene addestrato utilizzando il dataset di addestramento, con un numero di epoche specificato. Durante l'addestramento, le metriche di valutazione vengono monitorate attraverso una serie di callback, come il calcolo dell'F1-score e il progresso dell'addestramento.
- Salvataggio del Miglior Modello: Alla fine delle epoche, il modello migliore viene salvato in un percorso specificato utilizzando i pesi ottenuti dalla migliore valutazione dell'*F1 score* (eq. 4.3). Questo modello sarà poi utilizzato per le valutazioni successive.
- Trasformazione Finale: Per garantire che il modello sia pronto per le previsioni, un ultimo strato di Softmax viene aggiunto all'output del modello. Questo assicura che le uscite siano probabilistiche, consentendo al modello di fornire una distribuzione di probabilità su tutte le classi possibili.

4.3.2 YOLOv8

Il processo di addestramento attraverso YOLOv8 rappresenta invece un ottimo compromesso per quanto riguarda la facilità con il quale è possibile creare un modello a discapito però delle dimensioni di esso.

Si tratta di una sequenza di operazioni per addestrare un modello di rilevamento degli oggetti YOLOv8 utilizzando la libreria *"ultralytics"*. In primo luogo, un oggetto YOLO viene inizializzato utilizzando il file di pesi pre addestrati *'yolov8n.pt'*, che rappresenta il modello di base per l'addestramento successivo.

Durante la fase di addestramento, il modello viene istruito a riconoscere oggetti all'interno d'immagini utilizzando il dataset specificato nel file YAML *'./data.yaml'*. Il modello viene addestrato per un totale di 200 epoche, con un batch size di 64 immagini e dimensioni dell'immagine di 160x160 pixel. L'addestramento avviene sulla CPU specificato dal parametro *'device'*. L'opzione *'verbose'* abilitata consente di visualizzare informazioni dettagliate sull'addestramento, mentre i parametri *'workers'* e *'cache'* contribuiscono a ottimizzare il processo di caricamento dei dati durante l'addestramento.

Una volta completato l'addestramento, il modello viene esportato nel formato TensorFlow Lite attraverso il metodo *'export'*. La dimensione dell'immagine in pixel di 96x96 è specificata

tramite il parametro *'imgsz'*, e l'opzione *'int8'* indica che il modello subirà una quantizzazione int8 durante l'esportazione. Questo processo ha lo scopo di ridurre la dimensione del modello per l'esecuzione efficiente su dispositivi con risorse limitate.

Tuttavia, è importante notare che il modello YOLOv8, anche in versione compressa tramite TensorFlow Lite e con la quantizzazione int8, può essere ancora notevolmente più grande rispetto a modelli più semplici. Questo è dovuto alla natura complessa dell'architettura YOLOv8, che è progettata per rilevare oggetti in modo preciso e dettagliato. Pertanto, sebbene la quantizzazione int8 e l'ottimizzazione di TensorFlow Lite riducano le dimensioni del modello rispetto alla versione originale, YOLOv8 potrebbe comunque richiedere risorse computazionali significative rispetto a modelli più leggeri, ma offre un notevole potenziale in termini di accuratezza nel riconoscimento degli oggetti.

4.4 Gli ottimizzatori

Durante il processo di "*fit*" della *CNN*, il modello deve convergere verso il minimo della loss function presa in considerazione. Per raggiungere questo minimo (assoluto) evitando anche il possibile *overfitting* della rete, vengono utilizzati degli ottimizzatori [13, 14] che consentono di velocizzare il raggiungimento dell'obiettivo.

Tra gli ottimizzatori più conosciuti vi sono Adam, Stochastic Gradient Descent (SGD), Batch Gradient Descent (BGD) e Mini-Batch Gradient Descent (mini-BGD), ciascuno con caratteristiche uniche che influenzano l'efficacia del processo di apprendimento.

Adam, acronimo di Adaptive Moment Estimation, è un ottimizzatore che combina le caratteristiche di SGD con momenti adattivi. Utilizzando stime del primo e secondo momento dei gradienti, Adam adatta il tasso di apprendimento per ciascun parametro. Questo ottimizzatore risulta particolarmente efficace in scenari con funzioni di perdita stocastiche o non stazionarie, poiché adatta dinamicamente il tasso di apprendimento in base alla storia dei gradienti calcolati.

Lo Stochastic Gradient Descent (SGD) è uno dei metodi più fondamentali e ampiamente utilizzati per l'addestramento delle reti neurali, inclusa la CNN. In ogni iterazione, SGD aggiorna i pesi del modello spostandoli nella direzione opposta al gradiente istantaneo calcolato su un singolo esempio di addestramento. Sebbene SGD possa talvolta convergere in maniera più lenta rispetto ad altri ottimizzatori più complessi, è meno suscettibile di rimanere bloccato in minimi locali.

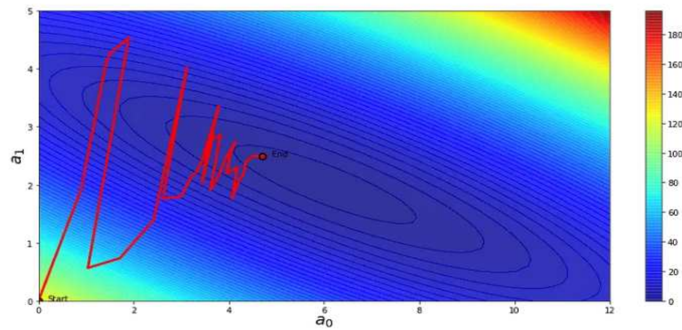


Figura 4.2: Stochastic Gradient Descent

Batch Gradient Descent (BGD), noto anche come gradient descent batch, calcola il gradiente della funzione di perdita rispetto all'intero set di addestramento in ogni iterazione. Sebbene questa approccio possa garantire una direzione di aggiornamento accurata, potrebbe risultare computazionalmente oneroso in scenari con grandi set di dati, poiché richiede il calcolo del gradiente su tutti i dati di addestramento.

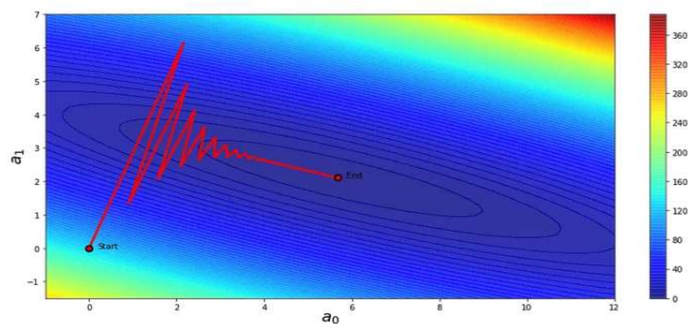


Figura 4.3: Batch Gradient Descent

Mini-Batch Gradient Descent (mini-BGD) rappresenta un compromesso tra SGD e BGD. Invece di calcolare il gradiente su tutto il set di addestramento, mini-BGD calcola il gradiente su piccoli sottoinsiemi chiamati *mini batch*. Questo approccio beneficia della parallelizzazione e dell'ottimizzazione del calcolo, consentendo un aggiornamento più frequente dei pesi rispetto a BGD e riducendo la varianza dell'aggiornamento rispetto a SGD.

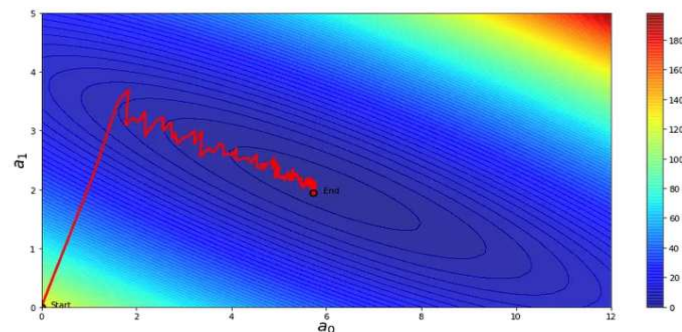


Figura 4.4: mini Batch Gradient Descent

4.5 Un processo fondamentale: Quantizzazione

Tra le tecniche di ottimizzazione che mirano a migliorare l'efficienza dell'inferenza in tempo reale la quantizzazione [7] assume un ruolo fondamentale.

La quantizzazione int8, o quantizzazione a 8 bit interi, consiste nel rappresentare i pesi e i valori di attivazione dei modelli di machine learning utilizzando solamente 8 bit anziché i 32 bit tipicamente utilizzati. Questo processo comporta una significativa riduzione delle dimensioni del modello, che risulta particolarmente critica per l'esecuzione su dispositivi con risorse limitate. La quantizzazione int8 riduce notevolmente il carico computazionale e la quantità di memoria richiesta per immagazzinare i parametri del modello, consentendo l'esecuzione su dispositivi con capacità di calcolo inferiori.

Nonostante la riduzione del numero di bit, la quantizzazione int8 è progettata per preservare l'accuratezza del modello. Questo è possibile grazie a strategie di approssimazione intelligenti che minimizzano l'errore di approssimazione durante la conversione dei valori a 32 bit in formati a 8 bit. Inoltre, durante la fase di addestramento, i modelli possono essere allenati tenendo conto delle sfide introdotte dalla quantizzazione int8, assicurando che l'accuratezza sia ottimizzata anche nelle operazioni d'inferenza a 8 bit.

Un altro vantaggio cruciale della quantizzazione int8 è il suo impatto positivo sulla durata della batteria e sulla velocità di esecuzione dei modelli. La riduzione della complessità computazionale e delle richieste di memoria si traduce in un minore consumo di energia e tempi di esecuzione più rapidi. Questo rende la quantizzazione int8 particolarmente adatta per applicazioni in tempo reale, dove la velocità di risposta è fondamentale.

Nonostante i numerosi vantaggi, la quantizzazione comporta alcune sfide e compromessi. Poiché si utilizzano meno bit per rappresentare i valori, potrebbe verificarsi un aumento dell'errore di approssimazione, specialmente nei modelli molto complessi. Inoltre, alcuni modelli possono richiedere una ricalibrazione o un tuning specifico dopo la quantizzazione per mantenere l'accuratezza desiderata.

4.6 Training: Risultati

Prima di entrare in merito dei risultati del training è doveroso fare alcune precisazioni sui valori utilizzati per fornire una stima della precisione della nostra rete neurale. Nell'ambito del deep learning o più in generale nell'inferenza statistica si utilizzano principalmente tre grandezze: *Precision*, *Recall* e *F1 score* [19].

- *Precision*: La precisione con cui si identificano i veri positivi rispetto a tutti i positivi trovati (veri o falsi).

$$Precision = \frac{TruePositive}{TruePositive + FalsePositive} \tag{4.1}$$

- *Recall*: La capacità di trovare veri positivi rispetto a tutti i positivi effettivi.

$$Recall = \frac{TruePositive}{TruePositive + FalseNegative} \tag{4.2}$$

- *F1 score*: La media armonica di *Precision* e *Recall* che restituisce l'accuratezza generale del sistema.

$$F1 = 2 \times \frac{Precision \times Recall}{Precision + Recall} \tag{4.3}$$

Il modello *MobileNetV2* addestrato attraverso Edge Impulse è costituito dai seguenti parametri:

```
================================================================================
Total params: 26,770
Trainable params: 24,610
Non-trainable params: 2,160
________________________________________________________________________________
```

Figura 4.5: Parametri MobileNetV2

Secondo le specifiche e i test definiti dai progettisti di MobileNetV2 [10] l'accuratezza dell'inferenza in tempo reale si distribuisce in questo modo:

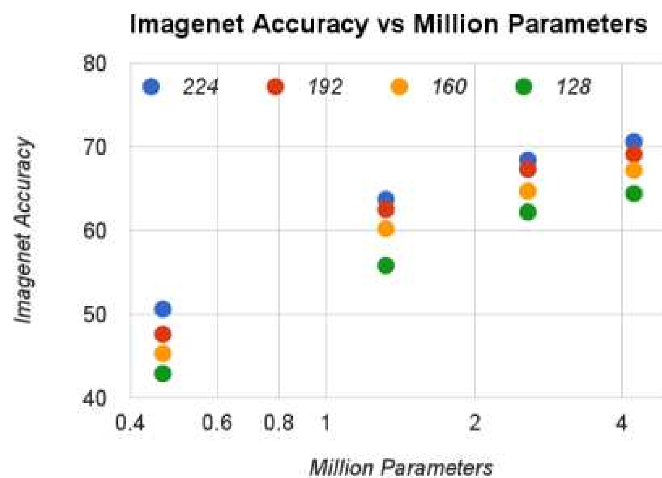


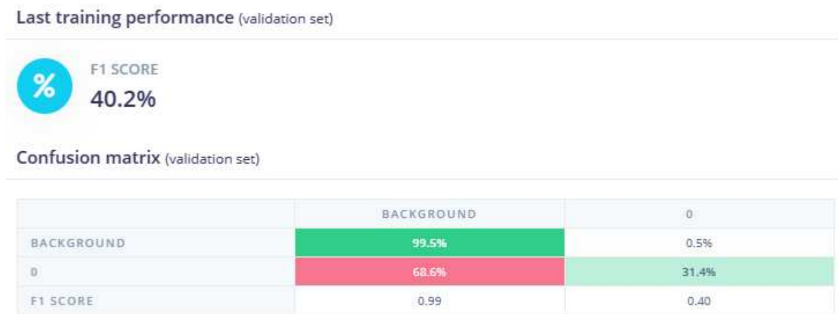
Figura 4.6: Specifiche MobileNetV2

Dove *"224"*, *"192"*, *"160"*, *"128"* indicano le dimensioni delle immagini ovvero i tensor di input della rete neurale.

Ci aspettiamo, di fronte a questi dati dunque che la nostra *CNN* performi molto peggio rispetto a un'accuratezza del 45%, valore nominale medio per un modello di 400K parametri circa, dato che il nostro modello è meno di un decimo della dimensione, specialmente se i tensor di input sono *70x70*.

Nonostante ciò la precisione rimane comunque attorno al 40% poiché l'accuratezza tende ad aumentare quando l'obiettivo è identificare un singolo oggetto anziché molti, infatti questo

semplifica il compito, riduce le ambiguità e richiede reti meno complesse con una varietà di classi limitata.



	BACKGROUND	0
BACKGROUND	99.5%	0.5%
0	68.6%	31.4%
F1 SCORE	0.99	0.40

Figura 4.7: Performance dell'addestramento Edge Impulse

In questi dati un fattore molto importante da considerare è il fatto che questo modello per il 68% delle volte, nonostante sia presente l'oggetto per il quale è stato addestrato, non riesce a identificarlo correttamente.

YOLOv8 invece ci fornisce una panoramica più dettagliata sul processo di training restituendo anche i grafici relativi:

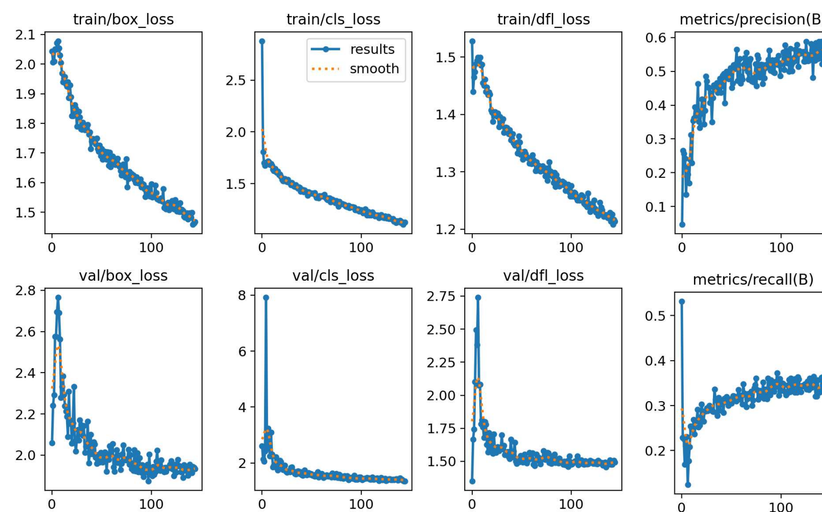


Figura 4.8: Performance dell'addestramento YOLOv8

Quello che si nota subito è che la Precisione e la Recall del modello sono più alte rispetto alla rete addestrata attraverso Edge Impulse, per un *F1* pari al 47% (eq. 4.3). L'aumento dell'accuratezza generale avviene a discapito della dimensione della rete: si tratta di un modello di circa 3 milioni di parametri post quantizzazione.

```
YOLOv8n summary: 225 layers, 3157200 parameters, 3157184 gradients
```

Figura 4.9: Specifiche YOLOv8

Ricordiamo inoltre che questa *CNN* non ha la struttura di un MobileNet e non dispone dunque di tutte le ottimizzazioni generate da questi ultimi.

4.7 Esportazione del modello

L'ultima fase relativa allo sviluppo di una rete neurale è l'effettiva esportazione di un modello. Edge impulse offre una funzione di *"deployment"* che converte automaticamente il modello in una libreria Arduino o firmware direttamente caricabili sui microcontrollori. Discuteremo a seguito del processo di caricamento su scheda.

Per quanto riguarda YOLOv8 una volta esportato il modello in *.tflite* (7.3), è necessario convertirlo in un formato comprensibile da Arduino ovvero in *C* o *C++*. Questo avviene attraverso il tool *xxd* che converte i file binari array di esadecimali in linguaggio *C++*.

Listing 4.1: Esempio Modello

```
unsigned char __best_int8_tflite[] = {
   0x20, 0x00, 0x00, 0x00, 0x54, 0x46, 0x4c, 0x33, 0x00, 0x00, 0x00,
      0x00,
   0x14, 0x00, 0x20, 0x00, 0x1c, 0x00, 0x18, 0x00, 0x14, 0x00, 0x10,
      0x00,
   0x0c, 0x00, 0x00, 0x00, 0x08, 0x00, 0x04, 0x00, 0x14, 0x00, 0x00,
      0x00,
   0x1c, 0x00, 0x00, 0x00, 0x80, 0x00, 0x00, 0x00, 0xf4, 0x00, 0x00,
      0x00,
   0xac, 0x53, 0x2e, 0x00, 0xbc, 0x53, 0x2e, 0x00, 0x74, 0x2f, 0x2f,
      0x00,
   0x03, 0x00, 0x00, 0x00, 0x01, 0x00, 0x00, 0x00, 0x04, 0x00, 0x00,
      0x00,
   ...
}
unsigned int __best_int8_tflite_len = 3157200;
```

Capitolo 5

Esecuzione del codice su schede

Come già presentato nel capitolo 3 la scelta della piattaforma hardware è la prima decisione critica nel processo d'implementazione. Essa deve essere basata sulle esigenze specifiche del progetto, tenendo conto delle limitazioni di risorse, come la potenza di calcolo della CPU, la disponibilità di memoria RAM e ROM. In questo contesto, Arduino ed ESP32 sono piattaforme comuni e versatili.

5.1 Configurazione dell'ambiente di sviluppo

Un passaggio cruciale è la configurazione dell'ambiente di sviluppo, che include la scelta degli strumenti di programmazione e debugging. Questa configurazione deve essere eseguita in modo accurato per garantire un ambiente di sviluppo efficiente.

Tutto ciò che segue è stato implementato attraverso Arduino IDE con lo scopo di gestire l'ambiente di sviluppo, i modelli e l'inferenza in tempo reale della nostra rete per Computer Vision.

Di fondamentale importanza è la gestione delle periferiche poiché ogni scheda e relativa telecamera possiedono collegamenti attraverso pin specifici. Per la telecamera *OV7675* esistono librerie progettate da Arduino come *"OV767X"* [4] che in automatico configurano alcuni pin e il progettista non deve fare altro che collegare fisicamente le porte corrette:

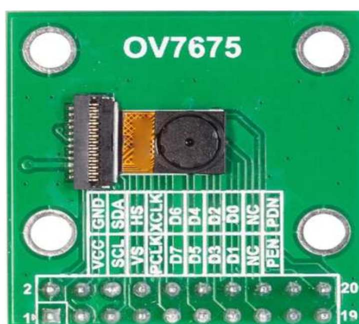


Figura 5.1: OV7675

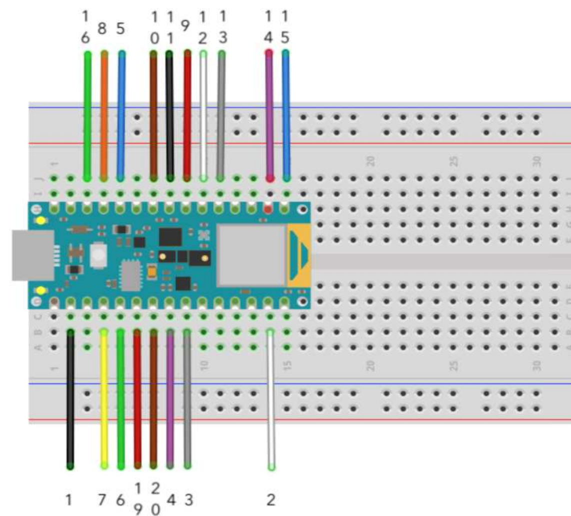


Figura 5.2: *Arduino Nano 33 BLE* e *OV7675* Pinout [1]

In altri casi come per la scheda integrata *XIAO_ESP32S3* è necessario fornire al codice il *"Pinout"* corretto poiché i collegamenti fisici sono già presenti:

Listing 5.1: *XIAOESP32S3* e *OV2460* Pinout

```
#define CAMERA_MODEL_XIAO_ESP32S3
#define PWDN_GPIO_NUM -1
#define RESET_GPIO_NUM -1
#define XCLK_GPIO_NUM 10
#define SIOD_GPIO_NUM 40
#define SIOC_GPIO_NUM 39

#define Y9_GPIO_NUM 48
#define Y8_GPIO_NUM 11
#define Y7_GPIO_NUM 12
#define Y6_GPIO_NUM 14
#define Y5_GPIO_NUM 16
#define Y4_GPIO_NUM 18
#define Y3_GPIO_NUM 17
#define Y2_GPIO_NUM 15
#define VSYNC_GPIO_NUM 38
#define HREF_GPIO_NUM 47
#define PCLK_GPIO_NUM 13
```

Un'ulteriore configurazione fondamentale è il supporto per le schede integrate *ESP32*[2] fornito dal *board manager* di *Espressif Systems*.

5.2 Il modello YOLOv8

Una volta ottenuto il modello da YOLOv8, teoricamente è possibile integrarlo nella libreria Arduino di Edge Impulse, sostituendo i file corrispondenti nelle cartelle "tflite-model" e "model-

parameters". Tuttavia, è fondamentale notare che questa operazione non è stata implementata nel contesto del progetto. La ragione principale risiede nelle dimensioni considerevoli del modello YOLOv8, che ammonta a circa 3 megabyte con la conseguente allocazione in RAM della sua *arena* che può essere anche 10 volte più grande. Questa dimensione supera notevolmente le capacità di memoria delle schede, rendendo praticamente impossibile l'esecuzione del modello su tale piattaforma.

Invece, all'interno del progetto, ci concentreremo sull'analisi delle prestazioni del modello YOLOv8. Questa valutazione si baserà sui risultati dei test di validazione svolti durante la fase di addestramento del modello. Esamineremo come il modello si comporta potenzialmente durante l'operazione su una piattaforma compatibile, valutando la sua efficienza nel rilevamento degli oggetti in tempo reale.

5.3 La cattura delle Immagini

La fase di cattura delle immagini prevede che la telecamera venga inizializzata con tutte le configurazioni necessarie ovvero i pin di collegamento, il formato e la dimensione delle immagini:

In seguito avviene la cattura non appena l'inferenza del ciclo precedente è terminata. E' dunque importante attendere il segnale dal thread piuttosto che eseguire uno *sleep(Time)* con tempo non sempre preciso per evitare che l'immagine possa essere caricata in memoria durante il processo di inferenza. L'acquisizione stessa delle immagini è un processo relativamente semplice; sarà necessario leggere i dati in ingresso dalla periferica collegata è salvarli in un buffer abbastanza capiente.

In alcuni casi è necessario ridimensionare l'immagine appena salvata poichè deve coincidere con i tensor di input della rete neurale e in altri casi è importante convertire anche il formato (es. da RGB565 a RGB888).

Tutte queste funzioni sono definite in modo intuitivo all'interno della libreria Arduino generata da Edge Impulse (vedi sezione Frammenti dei Codici). Quest'ultima contiene ovviamente il modello della *CNN* precedentemente addestrato, fondamentale per il passaggio successivo: l'inferenza in tempo reale.

5.4 Inferenza in tempo reale

L'inferenza avviene attraverso la chiamata alla funzione *run_classifier()* che dopo aver fatto passare l'immagine attraverso la rete neurale, salva i risultati nel puntatore *result* e ritorna una *"flag"* per indicare se la classificazione è avvenuta con successo o meno.

Per terminare non rimane altro che stampare i risultati a schermo accedendo al puntatore precedentemente modificato. In future implementazioni sarebbe certamente più comodo e intuitivo riuscire a stampare a schermo il video della cattura con relativi *Bounding Box* degli oggetti identificati.

5.5 Upload del Codice

L'upload dei codici su schede ESP32 o Arduino Nano attraverso strumenti come esptools o bossac è un processo cruciale nell'ambito dello sviluppo di dispositivi embedded. In breve, questo processo comporta la connessione fisica della scheda al computer host tramite un cavo USB, quindi l'utilizzo del tool appropriato per trasferire il codice sorgente compilato sulla scheda.

Per quanto riguarda le schede ESP32, esptools è uno strumento comunemente utilizzato per questo scopo. Dopo aver configurato correttamente il file di configurazione per la scheda e il firmware, esptools stabilisce una connessione seriale con la scheda, cancella la memoria flash se necessario e quindi scrive il nuovo firmware sulla scheda. Questo processo richiede una conoscenza dettagliata della configurazione e delle specifiche della scheda ESP32 [9].

Nel caso delle schede Arduino Nano o altre schede basate su microcontrollori ATMEL, bossac è uno strumento diffuso per l'upload del codice. Questo tool comunica con la scheda tramite la porta seriale (COM) e scrive il firmware compilato nella memoria flash della scheda. È importante selezionare la porta seriale giusta e impostare correttamente i parametri, come la velocità di trasmissione, per garantire un upload affidabile [3].

In entrambi i casi, l'upload del codice richiede una serie di passaggi che devono essere eseguiti con precisione per evitare errori. È fondamentale anche assicurarsi che la scheda sia correttamente alimentata e connessa al computer host. Una volta completato l'upload, il codice inizia a eseguire sulla scheda, consentendo il funzionamento dell'applicazione o del sistema embedded programmato.

Capitolo 6

Test e risultati

La fase di test della rete neurale riguarda per lo più una valutazione empirica che dimostra quanto efficientemente il modello restituisce un risultato attendibile durante il processo di inferenza.

Consideriamo il modello addestrato in precedenza, con *F1 score* nominale del 40% (cap. 4). Si può affermare, dati anche i risultati del training della rete stessa, che l'accuratezza complessiva della *CNN* non è del tutto soddisfacente. Lo dimostra il fatto che in molti casi il valore della *"Precision"* è particolarmente bassa e dunque vengono identificati oggetti anche quando non sono presenti.

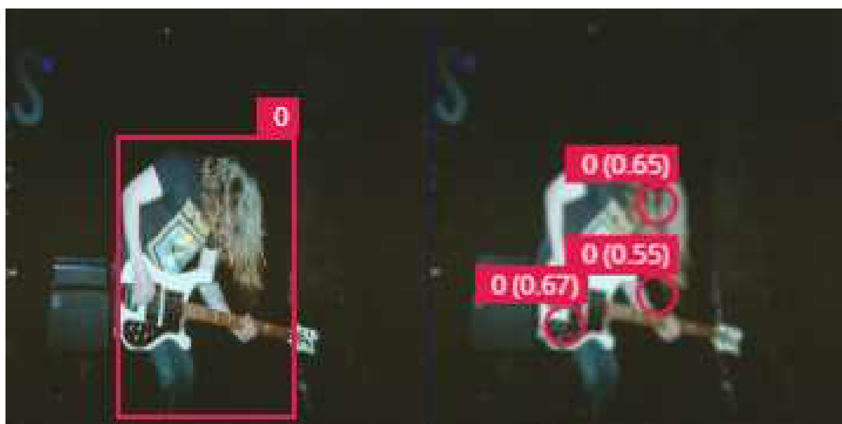


Figura 6.1: Differenza dati e risultati

Per i modelli come i MobileNets (cap. 2) è consigliabile che l'oggetto da identificare sia l'unico nel frame catturato. Questo compromesso è utile poichè, essendo il modello ridotto in dimensioni, non ha la possibilità di salvare un sufficiente numero di features in grado di differenziare in modo preciso oggetti simili tra loro.

Due elementi molto importanti da considerare sono anche la memoria utilizzata dal modello e il tempo necessario per effettuare l'inferenza. Ogni rete neurale richiede una certa quantità di memoria RAM per permettere il calcolo dell'output. Per essere efficienti tutta la rete viene caricata in memoria così da evitare troppi accessi alla memoria flash ovviamente molto più lenta. Questa area di memoria viene identificata come *"tensor arena"* e deve essere sufficientemente grande per conservare tutti i parametri della rete oltre ai valori intermedi di ogni nodo della rete stessa.

Figura 6.2: Risultati: utilizzo memoria

La conseguenza ovvia è che maggiore sarà la dimensione dei tensor di input, maggiore sarà la dimensione della *CNN* e maggiore anche la quantità di memoria utilizzata.

Per quanto riguarda il tempo di inferenza sarà necessario avvolgere il processo all'interno di un calcolo di Δt attraverso la funzione *millis()* di Arduino.

Il tempo medio stimato per l'inferenza si aggira intorno ai:

- 557ms per Arduino Nano 33 BLE.
- 1863ms per Xiao_ESP32S3.

I risultati relativi ai tempi di inferenza sono stati causa di ulteriori approfondimenti poichè, in base alle caratteristiche dell'hardware utilizzato, ci si sarebbe aspettato che la scheda Xiao_ESP32S3 avrebbe performato decisamente meglio. Lo dimostrano anche i dati di un paper dell'ICIST 2017 [12].

Nonostante ciò il Δt è 3 volte superiore rispetto alla controparte.

6.1 EON e acceleratore hardware

Durante la fase di *"deployment"* del modello addestrato, esiste l'opzione di esportare la *CNN* attraverso un compilatore chiamato *EON*.

EON [11] è un compilatore progettato per semplificare il processo di integrazione di modelli di machine learning su dispositivi embedded. Prende in input un file Tensorflow Lite Flatbuffer contenente i pesi del modello e produce in output file *.cpp* e *.h* che includono i pesi del modello scomposti e le funzioni per preparare ed eseguire l'inferenza del modello. Questo approccio offre un'alternativa più leggera rispetto all'approccio tradizionale di Tflite Micro, consentendo una migliore efficienza nell'esecuzione di inferenze di machine learning.

Purtroppo questo compilatore non permette alla scheda di attivare l'acceleratore hardware *ESP-NN*, limitando quindi le risorse del sistema e rallentando i MIPS. Per ovviare a questo inconveniente è necessario disabilitare il compilatore *EON* e usufruire dell'acceleratore *ESP-NN* utilizzato nel paper di Marcelo Rovai[22].

In seguito a questa implementazione le performance in termini di inferenza in tempo reale per la scheda Xiao_ESP32S3 sono migliorate notevolmente. Si passa da un Δt medio di 1863ms a un Δt medio di 274ms sviluppando in pieno tutte le capacità computazionali di questa scheda integrata.

Capitolo 7

Conclusioni

Per concludere, questo progetto di TinyML ha dimostrato la potenza e la versatilità delle reti neurali profonde quando applicate all'edge computing. L'uso di Edge Impulse ha semplificato significativamente il processo di addestramento e l'implementazione su microcontrollori come l'ESP32S3 e l'Arduino Nano 33 BLE. La capacità di eseguire l'object detection in tempo reale su queste schede rappresenta un notevole progresso nell'integrazione di algoritmi di intelligenza artificiale in dispositivi embedded di dimensioni ridotte.

Durante la realizzazione di questo progetto, si incontrano diversi passaggi cruciali. Prima di tutto è necessario acquisire familiarità con la raccolta e la preparazione dei dati, che costituiscono il fondamento dell'addestramento di reti neurali. In secondo luogo, l'importanza dell'ottimizzazione dei modelli: la limitata potenza di calcolo e memoria disponibile nei microcontrollori richiede un attento bilanciamento tra precisione e risorse. Inoltre, si sperimenta la necessità di affrontare sfide legate all'hardware, come l'integrazione di sensori e la gestione delle comunicazioni.

Questo progetto racchiude l'importanza della documentazione dettagliata e della risoluzione dei problemi, poiché spesso si sono verificati errori durante lo sviluppo che richiedevano una soluzione creativa. Nel complesso, questo percorso fornisce una preziosa esperienza nella progettazione di sistemi embedded con intelligenza artificiale, dimostrando le immense opportunità e sfide che questo campo offre. L'apprendimento e l'adattamento sono fondamentali per sfruttare appieno il potenziale dell'edge AI e contribuire all'evoluzione di soluzioni sempre più intelligenti ed efficienti per le applicazioni reali.

7.1 Ostacoli più comuni

Durante tutte le fasi del progetto sono stati riscontrati numerosi ostacoli: a partire dall'addestramento del modello per arrivare all'accelerazione hardware della scheda integrata.

In primo luogo uno dei problemi più complicati da risolvere è il *"trade-off"* tra dimensione della rete neurale e l'accuratezza in fase di inferenza. Il collo di bottiglia è stata la scheda Arduino Nano 33 BLE che ha una limitata memoria RAM. Il modello più preciso che viene prodotto in questo elaborato possiede soltanto un tensor di input pari a $\{70 \times 70 \times 1\}$ per un

F1 score che fatica a raggiungere il 40%. Si tratta di immagini in input molto sfocate dove i dettagli e le features sono difficilmente estraibili.

Un altro degli ostacoli ostici è la frammentazione della memoria nel momento di assegnamento dello spazio necessario per la *"tensor arena"*. Innanzitutto è fondamentale che la dimensione del modello sia sufficientemente piccola da poter essere caricata interamente in memoria tenendo conto anche della dimensione del buffer dell'immagine che viene catturata precedentemente all'inferenza. Una volta appurato ciò l'unico modo per risolvere questo inconveniente è definire l'allocazione statica dell'*arena* durante la fase di compilazione dello sketch Arduino attraverso l'argomento "-DEI_CLASSIFIER_ALLOCATION_STATIC".

In ultimo luogo per la scheda Xiao_ESP32S3 sono emerse due questioni importanti:

- Per la comunicazione tra scheda e telecamera OV2640 è necessario che venga abilitata la OPI PSRAM: una tecnologia di comunicazione ad alta velocità che permette al microcontrollore o al processore di accedere rapidamente ai dati memorizzati nella PSRAM offrendo offre una maggiore larghezza di banda rispetto alle tradizionali interfacce SPI o I2C.

- A volte *esptool* avvia uno *"stub loader"* ovvero un pezzo di codice aggiuntivo che sovrascrive il bootloader originale ESP32. Questo di solito viene utilizzato perchè contiene *UART routines* ottimizzate ma molto spesso entra in conflitto con il codice stesso da caricare. Sarà sufficiente aggiungere *–no-stub* come argomento nella fase di upload del codice.

7.2 I risultati di YOLOv8

Come già espresso nel capitolo relativo all'esecuzione del codice (cap. 5) il modello YOLOv8 non è stato utilizzato nelle schede prese in considerazione poichè genera una rete troppo "pesante" per la memoria a disposizione. Nonostante ciò l'addestramento YOLO della rete ci fornisce molte informazioni sulla validazione della *CNN* come output di un batch di dati che viene appunto validato.

Figura 7.1: Risultati: utilizzo memoria

E' noto da queste immagini che la rete è in grado di identificare discretamente gli oggetti per la quale è stata addestrata. Ipotizzando di avere una scheda Raspberry Pi nella quale potrebbe essere eseguita, l'inferenza sarebbe più precisa dei MobileNets precedentemente analizzati.

Considerando però i prezzi, i consumi e la versatilità delle schede Arduino Nano 33 BLE e Xiao_ESP32S3 le dimensioni della rete YOLOv8 non valgono il piccolo incremento di accuratezza.

Nonostante ciò YOLOv8 o i successivi aggiornamenti dei moduli YOLO possono rappresentare un punto di partenza per future implementazioni di reti neurali profonde in grado di essere eseguite in schede che dispongono di poche risorse, considerando innanzitutto una forte ottimizzazione di questi modelli.

7.3 Migliorie e sviluppi futuri

Nel contesto dell'object detection su sistemi embedded, l'ottimizzazione è fondamentale per garantire prestazioni efficienti e accurate.

Dalle considerazioni hardware, un possibile miglioramento potrebbe riguardare l'utilizzo di hardware specializzato, come acceleratori hardware per deep learning (ad esempio, GPU o acceleratori AI dedicati). Questo consentirebbe una maggiore velocità di elaborazione e la possibilità di gestire modelli più complessi. Inoltre, l'ottimizzazione dell'architettura del sistema, compresa la scelta di componenti a basso consumo energetico, può estendere la durata della batteria per dispositivi alimentati a batteria. L'espansione della memoria RAM per poter contenere modelli più grandi come YOLOv8 è un'altra soluzione da considerare anche se si andrebbe incontro a sistemi generalmente più costosi.

Dal lato software, è possibile implementare miglioramenti come la quantizzazione dei pesi e delle attivazioni del modello, per accelerare l'inferenza senza compromettere significativamente la qualità dei risultati. L'ottimizzazione del codice sorgente, l'uso di librerie di inferenza ottimizzate e la parallelizzazione delle operazioni possono anche aumentare l'efficienza computazionale. Inoltre, l'implementazione di tecniche di pruning per ridurre le dimensioni del modello senza perdere significativamente in accuratezza può essere un'opzione valida per sistemi embedded con risorse limitate.

Anche l'addestramento che richiede sempre meno parametri mantenendo una precisazione accettabile è oggetto di sviluppo futuro poichè è il fattore limitante principale nelle implementazioni di questo genere.

Infine, l'integrazione di algoritmi di riduzione del rumore e miglioramento delle immagini può migliorare la precisione dell'object detection in condizioni sfavorevoli, come scarsa illuminazione o rumore visivo, specialmente se si parla di immagini notevolmente ridimensionate. Complessivamente, una combinazione di miglioramenti hardware e software può contribuire a ottimizzare i sistemi embedded di object detection, consentendo loro di offrire prestazioni migliori e più efficienti in una varietà di applicazioni.

Bibliografia

[1] *Arduino Nano 33 BLE Sense*. URL: https://docs.edgeimpulse.com/docs/development-platforms/officially-supported-mcu-targets/arduino-nano-33-ble-sense.

[2] *arduino-esp32/package/package_esp32_index.template.json at master · espressif/arduino-esp32*. URL: https://github.com/espressif/arduino-esp32/blob/master/package/package_esp32_index.template.json.

[3] *arduino/arduino-flash-tools*. original-date: 2015-07-02T15:11:07Z. 15 Set. 2023. URL: https://github.com/arduino/arduino-flash-tools.

[4] *Arduino_OV767X Library for Arduino*. original-date: 2020-01-30T15:29:54Z. 11 Set. 2023. URL: https://github.com/arduino-libraries/Arduino_OV767X.

[5] Luigi Boldrin. «Stato dell'arte del machine learning su microcontrollori». Tesi di dott. Università degli Studi di Padova, 2023. URL: https://thesis.unipd.it/handle/20.500.12608/44047.

[6] Matteo Carnelos. «MicroFlow: A Rust TinyML Compiler for Neural Network Inference on Embedded Systems». Tesi di dott. Università degli Studi di Padova, 2023. URL: https://thesis.unipd.it/handle/20.500.12608/46961.

[7] Ram Cherukuri. *What Is int8 Quantization and Why Is It Popular for Deep Neural Networks?* URL: https://it.mathworks.com/company/newsletters/articles/what-is-int8-quantization-and-why-is-it-popular-for-deep-neural-networks.html.

[8] *Edge Impulse Documentation*. URL: https://docs.edgeimpulse.com/docs/.

[9] *Esptool.py Documentation - ESP32 - — esptool.py latest documentation*. URL: https://docs.espressif.com/projects/esptool/en/latest/esp32/.

[10] Andrew G. Howard et al. *MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications*. 16 Apr. 2017. arXiv: 1704.04861[cs]. URL: http://arxiv.org/abs/1704.04861.

[11] *Introducing EON: Neural Networks in Up to 55% Less RAM and 35% Less ROM*. 20 Set. 2020. URL: https://www.edgeimpulse.com/blog/introducing-eon.

[12] Jovan Ivković e Jelena Lužija Ivković. «Analysis of the Performance of the New Generation of 32-bit Microcontrollers for IoT and Big Data Application». In: (2017).

[13] Namrata Kapoor. *Optimizers*. Numpy Ninja. 19 Nov. 2020. URL: https://www.numpyninja.com/post/optimizers.

[14] Robert Kwiatkowski. *Batch, Mini-Batch and Stochastic Gradient Descent for Linear Regression*. Medium. 15 Giu. 2021. URL: https://towardsdatascience.com/batch-mini-batch-and-stochastic-gradient-descent-for-linear-regression-9fe4eefa637c.

[15] *Microcontrollore*. In: *Wikipedia*. Page Version ID: 132985589. 13 Apr. 2023. URL: https://it.wikipedia.org/w/index.php?title=Microcontrollore&oldid=132985589#Sistemi_di_sviluppo.

[16] *Nano 33 BLE | Arduino Documentation*. URL: https://docs.arduino.cc/hardware/nano-33-ble.

[17] Francesco Pasti. «Computer Vision Based Person Detection on Embedded Systems». Tesi di dott. Università degli Studi di Padova, 2022.

[18] *Person Image Dataset*. Roboflow. URL: https://universe.roboflow.com/sangarsh-s/person-jobpr/dataset/1.

[19] *Precision and recall*. In: *Wikipedia*. Page Version ID: 1175630050. 16 Set. 2023. URL: https://en.wikipedia.org/w/index.php?title=Precision_and_recall&oldid=1175630050.

[20] Partha Pratim Ray. «A review on TinyML: State-of-the-art and prospects». In: *Journal of King Saud University - Computer and Information Sciences* 34.4 (1 apr. 2022), pp. 1595–1623. ISSN: 1319-1578. DOI: 10.1016/j.jksuci.2021.11.019. URL: https://www.sciencedirect.com/science/article/pii/S1319157821003335.

[21] Redazione. *XIAO ESP32S3: il modulo ultra compatto di Seeed Studio*. Elettronica Open Source. URL: https://it.emcelettronica.com/xiao-esp32s3-il-modulo-ultra-compatto-di-seeed-studio.

[22] Marcelo Rovai. *TinyML Made Easy: Image Classification*. Hackster.io. URL: https://www.hackster.io/mjrobot/tinyml-made-easy-image-classification-cb42ae.

[23] *Struttura di un microcontrollore*. URL: http://www.elemania.altervista.org/pic/pic1.html.

[24] Ultralytics. *YOLOv8 Documentation*. URL: https://docs.ultralytics.com/.

[25] *UNO R3 | Arduino Documentation*. URL: https://docs.arduino.cc/hardware/uno-rev3.

[26] *What Is Computer Vision: Applications, Benefits and How to Learn It*. Simplilearn.com. 28 Apr. 2021. URL: https://www.simplilearn.com/computer-vision-article.

Frammenti dei Codici

Listing 7.1: Build Model

```
def build_model(input_shape: tuple, weights: str, alpha: float,
    num_classes: int) -> tf.keras.Model:
    # Crea modello mobile_net_V2 a partire da (HW, HW, C) input
       --> (HW/8, HW/8, C) output
    mobile_net_v2 = MobileNetV2(input_shape=input_shape,
       weights=weights, alpha=alpha, include_top=True)

    for layer in mobile_net_v2.layers:
       if type(layer) == BatchNormalization:
          layer.momentum = 0.9
    # "taglia" MobileNet dove arriva a 1/8th della risoluzione di
       input;
    cut_point = mobile_net_v2.get_layer('block_6_expand_relu')
    # Fissa un "head" addizionale
    model = Conv2D(filters=32, kernel_size=1, strides=1,
            activation='relu', name='head')(cut_point.output)
    logits = Conv2D(filters=num_classes, kernel_size=1, strides=1,
               activation=None, name='logits')(model)
    return Model(inputs=mobile_net_v2.input, outputs=logits)
```

Listing 7.2: Dati

```
train_segmentation_dataset =
    train_dataset.map(dataset.bbox_to_segmentation(
   output_width_height, num_classes_with_background)).batch(32,
      drop_remainder=False).prefetch(1)
validation_segmentation_dataset =
    validation_dataset.map(dataset.bbox_to_segmentation(
   output_width_height, num_classes_with_background)).batch(32,
      drop_remainder=False).prefetch(1)
validation_dataset_for_callback = validation_dataset.batch(32,
    drop_remainder=False).prefetch(1)
```

Listing 7.3: Addestramento

```
callbacks.append(
  tf.keras.callbacks.ModelCheckpoint(best_model_path,
  monitor='val_f1', save_best_only=True, mode='max',
     save_weights_only=True, verbose=0))
model.fit(train_segmentation_dataset,
    validation_data=validation_segmentation_dataset,
    epochs=num_epochs, callbacks=callbacks, verbose=0)
```

Listing 7.4: Softmax

```
softmax_layer = Softmax()(model.layers[-1].output)
model = Model(model.input, softmax_layer)
```

Listing 7.5: YOLOv8

```
from ultralytics import YOLO

model = YOLO('yolov8n.pt')
results = model.train(data='./data.yaml', epochs=200, batch=64,
    imgsz=160, device='cpu', verbose=True, workers=12, cache=True)

model.export(format='tflite', imgsz=96, int8=True)
```

Listing 7.6: Formato Immagini

```
#define EI_CAMERA_RAW_FRAME_BUFFER_COLS 160
#define EI_CAMERA_RAW_FRAME_BUFFER_ROWS 120
#define EI_CAMERA_FRAME_BYTE_SIZE 3
```

Listing 7.7: Cattura Immagini

```
snapshot_buf = (uint8_t*)malloc(EI_CAMERA_RAW_FRAME_BUFFER_COLS *
    EI_CAMERA_RAW_FRAME_BUFFER_ROWS * EI_CAMERA_FRAME_BYTE_SIZE);

  // check if allocation was successful
if(snapshot_buf == nullptr) {
  ei_printf("ERR: Failed to allocate snapshot buffer!\n");
  return;
}

ei::signal_t signal;
signal.total_length = EI_CLASSIFIER_INPUT_WIDTH *
    EI_CLASSIFIER_INPUT_HEIGHT;
signal.get_data = &ei_camera_get_data;
```

```
if (ei_camera_capture((size_t)EI_CLASSIFIER_INPUT_WIDTH,
    (size_t)EI_CLASSIFIER_INPUT_HEIGHT, snapshot_buf) == false) {
  ei_printf("Failed to capture image\r\n");
  free(snapshot_buf);
  return;
 }
```

Listing 7.8: Conversioni

```
int ei_camera_cutout_get_data(size_t offset, size_t length, float
    *out_ptr);
void resizeImage(int srcWidth, int srcHeight, uint8_t *srcImage, int
    dstWidth, int dstHeight, uint8_t *dstImage, int iBpp);
void cropImage(int srcWidth, int srcHeight, uint8_t *srcImage, int
    startX, int startY, int dstWidth, int dstHeight, uint8_t *dstImage,
    int iBpp);
```

Listing 7.9: Run Classifier

```
extern "C" EI_IMPULSE_ERROR run_classifier(signal_t *signal,
    ei_impulse_result_t *result, bool debug = false)
{
  const ei_impulse_t impulse = ei_default_impulse;
  return process_impulse(&impulse, signal, result, debug);
}
```

Listing 7.10: Tempo di Inferenza

```
long int startTime = millis();

// inferenza

long int endTime = millis();
long int elapsed = endTime - startTime;
```