Alma Mater Studiorum · Università di Bologna

Dipartimento di Informatica — Scienza e Ingegneria
Corso di Laurea Triennale in Informatica

Valutazione delle prestazioni del Model Splitting su dispositivi ESP32 in Reti IoT Multi-Hop

Relatore:
Chiar.mo Dott.
Trotta Angelo

Presentata da:
Cisbani Nicola

Correlatore:
Chiar.mo Dott.
Esposito Alfonso

Sessione LUGLIO 2026
Anno Accademico 2025/2026

Abstract

I recenti sviluppi nel *model splitting* e nella *split inference* mostrano come l'esecuzione distribuita di reti neurali su più dispositivi possa estendere le capacità dell'intelligenza artificiale verso nodi edge fortemente vincolati in termini di memoria e potenza di calcolo, evidenziando al contempo nuove criticità legate alla scelta dei punti di taglio, al costo di comunicazione tra nodi e alla gestione delle risorse hardware [12, 3]. In questo contesto, la tesi progetta, implementa e valuta sperimentalmente una pipeline di inferenza distribuita su una catena di microcontrollori ESP32-S3, in cui ciascun nodo esegue una porzione del modello MobileNetV2 quantizzato a `int8` e inoltra il tensore intermedio al nodo successivo tramite TCP. Il firmware, sviluppato in ESP-IDF con supporto Arduino e basato su TensorFlow Lite for Microcontrollers, adotta un'architettura single-threaded sequenziale e una Tensor Arena ibrida tra PSRAM esterna e SRAM interna, mentre un coordinatore Python esterno orchestra il caricamento dei segmenti di modello, l'invio dell'input e la raccolta delle metriche di latenza ed energia. Le configurazioni a 2, 3 e 4 nodi sono confrontate con una baseline a singolo nodo e tra due diverse strategie di posizionamento dei punti di split, isolando il contributo del tempo di calcolo locale da quello di trasmissione di rete e modellando quest'ultimo come funzione affine della dimensione del tensore. I risultati mostrano che la suddivisione consente di ripartire il carico computazionale e l'occupazione di memoria su più dispositivi constrained, rendendo possibile l'esecuzione di modelli che risulterebbero difficilmente gestibili. Al tempo stesso però, il costo computazionale aggiuntivo introdotto dallo split resta contenuto, mentre il forwarding costituisce la componente più critica e variabile della pipeline. La scelta del punto di split quindi deve privilegiare la riduzione dei tensori intermedi scambiati, soprattutto all'aumentare dei nodi coinvolti.

Indice

Elenco delle figure

Elenco delle tabelle

Capitolo 1

Introduzione

La crescente diffusione di dispositivi connessi in ambito industriale, domestico e ambientale rende l'elaborazione locale dei dati un requisito sempre più centrale nell'Internet of Things (IoT). Il paradigma dell'*Edge AI* risponde a questa esigenza spostando il carico computazionale verso i nodi periferici della rete, riducendo la dipendenza dal cloud, la latenza di risposta e la quantità di dati trasmessi a server remoti [4]. In questo contesto, i microcontrollori a basso consumo costituiscono una piattaforma di riferimento per eseguire algoritmi di apprendimento automatico direttamente alla sorgente dei dati, secondo l'approccio noto come *TinyML*.

Modelli neurali di uso comune, come MobileNet, ResNet o VGG, richiedono però risorse di memoria e calcolo spesso superiori a quelle disponibili su un singolo microcontrollore. Il *model splitting* affronta questo limite suddividendo il modello in partizioni sequenziali, assegnate a nodi distinti: ciascun dispositivo elabora la propria porzione e trasmette al successivo il tensore di attivazione intermedio. In questo modo, carico computazionale e occupazione di memoria vengono distribuiti tra più dispositivi cooperanti, rendendo potenzialmente eseguibili modelli altrimenti inaccessibili su hardware embedded [3, 12].

Questa tesi analizza sperimentalmente l'efficacia del *model splitting* su una piattaforma fortemente vincolata, composta da microcontrollori ESP32-S3 connessi in rete locale Wi-Fi. Oltre all'esecuzione distribuita del modello, il lavoro valuta i tre principali contributi alla latenza complessiva: inferenza locale, forwarding dei tensori intermedi e overhead dovuto alla suddivisione del grafo. Viene inoltre analizzato il profilo energetico del sistema, per verificarne la compatibilità con scenari IoT reali, spesso alimentati a batteria o da sorgenti a bassa potenza.

Il documento è organizzato come segue. Il Capitolo 2 introduce il background teorico sul *model splitting*, sul TinyML e sulla piattaforma ESP32-S3. Il Capitolo 3 descrive l'architettura hardware e software del sistema implementato, includendo pipeline distribuita, layout delle partizioni in flash, firmware, moduli di inferenza e comunicazione, raccolta delle metriche e coordinatore Python esterno. Il Capitolo 4 presenta i risultati sperimentali, confrontando due strategie di split del modello MobileNetV2 su configurazioni a 2, 3 e 4 nodi e analizzando inferenza locale, forwarding, latenza end-to-end stimata e consumo energetico. Le conclusioni sintetizzano i risultati principali e delineano possibili sviluppi futuri.

Capitolo 2

Background

Il background teorico di questo lavoro ruota attorno a tre idee strettamente collegate: la suddivisione del modello neurale tra più dispositivi, l'esecuzione dell'inferenza vicino alla sorgente dei dati e l'uso di microcontrollori ESP32 come piattaforma di calcolo. Questi tre elementi definiscono il contesto in cui è stato progettato il sistema sperimentale e chiariscono perché una pipeline distribuita possa essere più adatta di un'esecuzione monolitica su un singolo nodo [3, 4, 6].

2.1 Model Splitting

Nel lavoro di riferimento sullo *split learning*, una rete neurale viene vista come una sequenza di strati che può essere suddivisa in più sottoreti, ognuna assegnata a un agente diverso. Ogni agente conserva soltanto i parametri della propria sezione e comunica con gli agenti adiacenti scambiando le attivazioni intermedie durante il passaggio in avanti e, nel caso dell'addestramento distribuito, i gradienti durante il passaggio all'indietro [11].

In questa prospettiva, il *model splitting* non indica semplicemente la copia dello stesso modello su più dispositivi, ma la decomposizione strutturale del modello lungo la sua profondità. L'esecuzione diventa quindi una pipeline: il primo nodo riceve l'input, calcola gli strati che gli sono stati assegnati e trasmette il tensore prodotto al nodo successivo; il procedimento continua fino all'ultimo segmento, che restituisce l'output finale. La correttezza del risultato dipende dal fatto che l'ordine dei segmenti e il formato dei tensori scambiati rimangano coerenti con la rete originale.

L'interesse di questa tecnica nasce dal fatto che molti modelli neurali sono troppo onerosi per essere eseguiti integralmente su un singolo dispositivo embedded. Distribuendo gli strati tra più nodi si riducono la memoria richiesta da ciascun dispositivo, la dimensione del modello caricato localmente e l'arena necessaria all'inferenza. Il vantaggio non è però gratuito: a ogni punto di taglio corrisponde un trasferimento di dati intermedi, per cui il beneficio computazionale deve essere confrontato con il costo di comunicazione introdotto dalla rete [11, 3].

Nel progetto questa definizione viene applicata al caso specifico della *split inference*: non viene distribuito l'addestramento, ma solo l'esecuzione in avanti del modello già addestrato. La rete neurale viene partizionata in segmenti caricati su una catena lineare di ESP32-S3; ogni nodo esegue la propria porzione con TensorFlow Lite for Microcontrollers

e inoltra tramite un socket TCP raw permanente il tensore di uscita al nodo successivo. Questa scelta permette di isolare sperimentalmente il compromesso centrale del model splitting, cioè la riduzione del carico locale rispetto all'aumento del traffico tra nodi.

2.2 TinyML

Con *TinyML* si intende l'esecuzione di modelli di apprendimento automatico su microcontrollori e dispositivi embedded a bassissimo consumo energetico e su hardware fortemente vincolato in termini di memoria, capacità di calcolo e disponibilità energetica. L'interesse pratico di questo approccio nasce dalla possibilità di portare l'inferenza direttamente sul nodo che acquisisce il dato grezzo, evitando di trasmetterlo a un server esterno per l'elaborazione. In una rete IoT, ciò si traduce in una riduzione della latenza di risposta, in un minore consumo di banda e in una maggiore resilienza operativa, poiché il dispositivo può continuare a funzionare anche in assenza di connettività con infrastrutture remote.

Sul piano architetturale, le principali sfide del TinyML riguardano tre aspetti strettamente connessi. Il primo è la **compressione dei modelli**: le reti neurali di uso comune devono essere ridotte in termini di numero di parametri e precisione numerica per rientrare nei vincoli di memoria e flash dei microcontrollori. Tecniche come la quantizzazione a 8 bit (`int8`), il pruning e la distillazione della conoscenza consentono di mantenere una precisione accettabile riducendo in modo significativo l'occupazione di memoria e il costo computazionale [18, 19]. Il secondo aspetto riguarda il **runtime di inferenza**: framework come TensorFlow Lite for Microcontrollers (TFLM) sono progettati per operare anche in assenza di un sistema operativo completo e senza fare affidamento su allocazione dinamica della memoria, offrendo un'impronta ridotta e un insieme selezionato di operatori matematici ottimizzati per le istruzioni disponibili sull'hardware target [18, 4]. Il terzo aspetto concerne la **gestione della memoria**: nei microcontrollori la distinzione tra memoria volatile (SRAM), memoria non volatile (flash) e, dove disponibile, memoria esterna come la PSRAM impone scelte progettuali precise su dove allocare i pesi del modello, i tensori di attivazione e i buffer temporanei di calcolo.

2.3 Microcontrollori ESP32

Gli ESP32 sono microcontrollori progettati da Espressif per applicazioni connesse e a basso consumo. La famiglia ESP32-S3, in particolare, integra connettività Wi-Fi e Bluetooth Low Energy, un'architettura dual-core Xtensa LX7 e il supporto a memoria esterna PSRAM. Queste caratteristiche rendono tali dispositivi interessanti per scenari di edge computing, nei quali piccoli nodi distribuiti devono acquisire dati, elaborarli localmente e comunicare con altri dispositivi della rete [6, 8].

L'applicazione del *model splitting* a microcontrollori di questo tipo permette di superare, almeno in parte, i limiti imposti dalle risorse locali. Un modello neurale che sarebbe troppo grande per essere eseguito interamente su un singolo nodo può essere suddiviso in più segmenti, riducendo la quantità di memoria, flash e area temporanea richiesta da ciascun dispositivo. In una rete IoT, questa suddivisione consente inoltre di sfruttare

la presenza di più nodi già distribuiti nell'ambiente, trasformando la rete stessa in una piccola infrastruttura cooperativa per l'inferenza [4, 3].

Il vantaggio principale non riguarda solo la possibilità di eseguire modelli più complessi, ma anche una migliore distribuzione del carico computazionale. Ogni microcontrollore elabora soltanto una parte della rete neurale, con potenziali benefici in termini di memoria occupata, consumo istantaneo e reattività del singolo nodo. Questo approccio è particolarmente rilevante quando si vuole mantenere l'elaborazione vicina alla sorgente dei dati, evitando di inviare continuamente informazioni grezze verso un server esterno.

Allo stesso tempo, il model splitting introduce difficoltà specifiche. I tensori intermedi devono essere trasmessi tra i nodi, e la loro dimensione può rendere la comunicazione più costosa dell'elaborazione risparmiata. Lavori recenti su reti IoT multi-hop mostrano inoltre come il partizionamento del modello debba essere progettato congiuntamente rispetto al percorso di inferenza, alla precisione dei sotto-modelli e ai vincoli di latenza ed energia dei dispositivi extreme edge [1]. A questi aspetti si aggiungono la necessità di sincronizzare correttamente i nodi, gestire eventuali errori di comunicazione e mantenere coerenti formato, quantizzazione e ordine dei dati scambiati [11, 12].

L'uso di microcontrollori ESP32 in scenari di split inference rappresenta un compromesso tra capacità distribuita e vincoli embedded. Da un lato, la cooperazione tra nodi può rendere praticabile l'esecuzione di modelli non adatti a un singolo dispositivo; dall'altro, il costo della comunicazione e la scarsità di memoria impongono una progettazione attenta del partizionamento, della pipeline e del protocollo di scambio dei dati.

Capitolo 3

Architettura Hardware e Software

Questo capitolo descrive nel dettaglio l'architettura hardware e software del sistema sviluppato per l'esecuzione della *split inference* distribuita. Il sistema si basa su una pipeline di nodi microcontrollori ESP32-S3 che cooperano per eseguire segmenti sequenziali di un modello di deep learning. L'infrastruttura software è progettata in modo modulare e implementa un ciclo di esecuzione sequenziale deterministico, supportato da un coordinatore esterno scritto in Python che gestisce l'orchestrazione dei test e la raccolta centralizzata delle metriche prestazionali.

3.1 Contesto sperimentale e architettura distribuita

Il sistema implementa una pipeline lineare di *model splitting* su una rete locale wireless (LAN) composta da N nodi microcontrollori ESP32-S3. L'architettura generale segue un paradigma di **split computing**: ogni nodo esegue localmente una porzione sequenziale del modello neurale originario e trasmette le attivazioni intermedie al nodo successivo.

Una caratteristica fondamentale di questo design è che tutti i nodi eseguono lo stesso firmware a livello binario. Il ruolo specifico di ciascun dispositivo, cioè nodo iniziale, nodo intermedio o nodo finale della pipeline, viene configurato dinamicamente a runtime tramite parametri ricevuti via HTTP al momento del caricamento del segmento di modello. In particolare, il parametro `lasthop` indica se il nodo rappresenta l'ultimo stadio della catena e deve quindi estrarre la classe predetta tramite *argmax*, mentre `next_ip` specifica l'indirizzo IP del nodo successivo a cui inoltrare le attivazioni intermedie.

La trasmissione del modello e dei metadati di configurazione avviene tramite richieste HTTP sincrone, mentre l'invio dei tensori intermedi di attivazione tra i nodi della catena avviene tramite un socket TCP raw permanente (con connessione persistente). In entrambi i casi, l'intero traffico rimane confinato all'interno della rete locale. Questa scelta consente di escludere latenze esterne dovute a instradamenti geografici e di analizzare in modo più controllato e ripetibile il bilancio energetico e temporale tra elaborazione computazionale locale e *overhead* di rete introdotto dal trasferimento dei tensori.

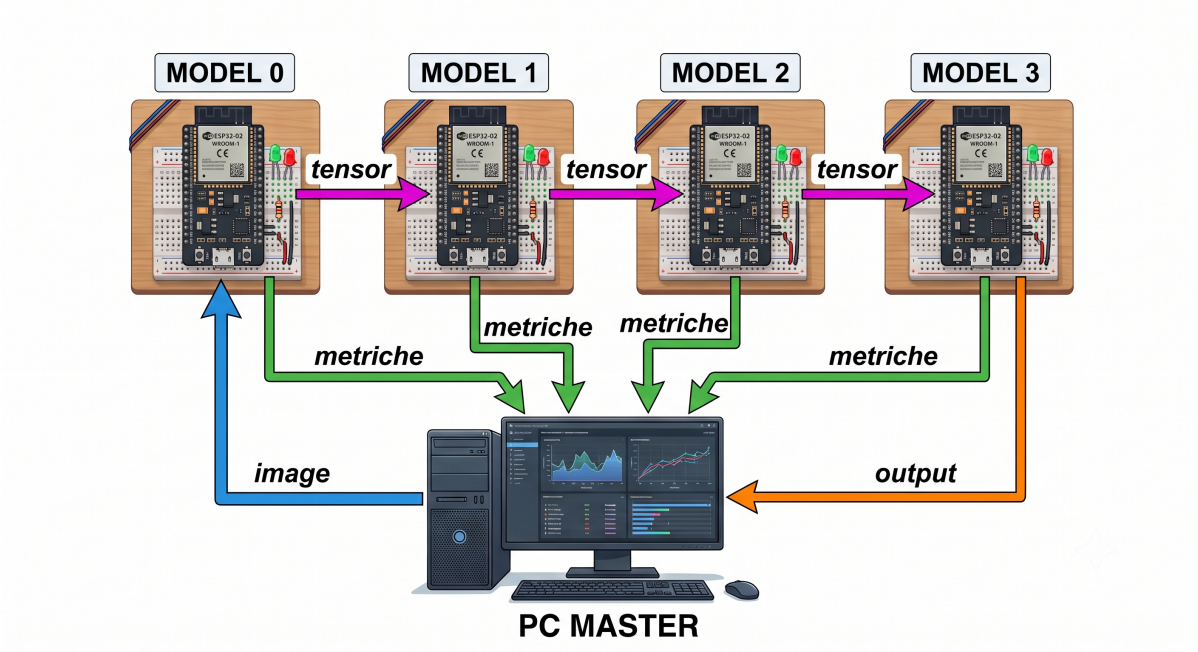


Figura 3.1: Architettura generale del sistema di Split Inference

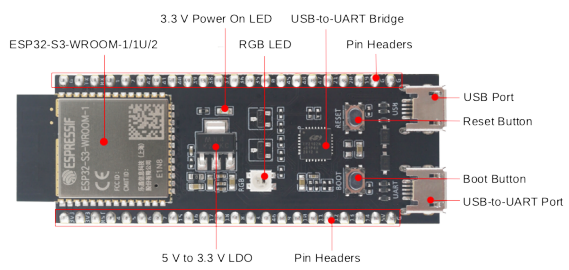
3.2 Architettura hardware

La piattaforma hardware selezionata per la sperimentazione è basata sul modulo ESP32-S3-WROOM-1. Si tratta di un sistema su chip (SoC) integrato ad alte prestazioni, particolarmente indicato per compiti di Edge AI grazie alla presenza di istruzioni vettoriali dedicate (estensioni d'istruzione XTensa per reti neurali) che accelerano i calcoli matematici alla base dei modelli di deep learning.

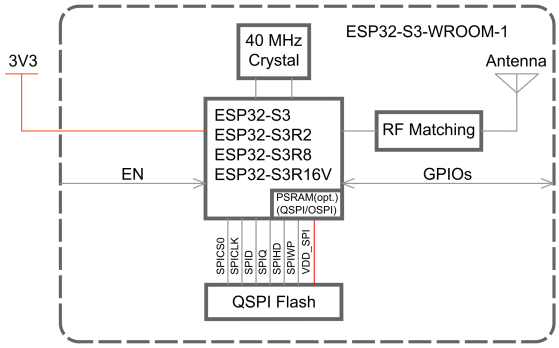
Le caratteristiche salienti della scheda includono:

- **microcontrollore**: Xtensa dual-core LX7 a 32 bit con frequenza di clock fino a 240 MHz.
- **Connettività**: Modulo Wi-Fi integrato a 2.4 GHz (802.11 b/g/n) e Bluetooth 5 (LE).
- **Sottosistema di Memoria**: SRAM interna accoppiata ad una memoria flash SPI e, elemento fondamentale per questo progetto, ad una memoria PSRAM esterna (Pseudo-Static RAM) ad alta capacità, collegata in modalità SPI ottale per massimizzare la larghezza di banda.

Nel progetto, i diversi tipi di memoria sono usati in modo strategico per aggirare i limiti tipici dei microcontrollori. La flash interna contiene il firmware compilato e una partizione riservata al modello neurale. La SRAM interna viene invece usata per lo stack e per le variabili di sistema che richiedono accessi rapidi. La PSRAM esterna, più capiente ma meno veloce, ospita l'arena di TensorFlow Lite Micro, cioè l'area di lavoro in cui vengono salvati i tensori di attivazione e il tensore di input. Quest'ultimo viene scritto direttamente in memoria durante la ricezione dal socket TCP permanente (o tramite richiesta HTTP nel caso del primo nodo della catena), evitando l'uso di buffer intermedi. Infine, i buffer temporanei usati dalle convoluzioni hanno una zona riservata



(a) Scheda di sviluppo ESP32-S3-DevKitC-1 [7]



(b) Schema a blocchi ESP32-S3-WROOM-1

Figura 3.2: Componenti hardware adottati nei test per valutare l'inferenza

nella SRAM interna. In questo modo, le operazioni più pesanti possono accedere alla memoria più veloce e si riduce il traffico verso la PSRAM.

Il microcontrollore ESP32-S3 integra un'architettura dual-core, che consentirebbe di separare la gestione del traffico di rete dall'esecuzione dell'inferenza. Questa separazione, tuttavia, non viene implementata, poiché non risulta necessaria ai fini del confronto tra le singole inferenze. Si preferisce quindi mantenere il firmware il più semplice possibile, riducendo l'overhead dovuto al context switch e concentrando l'elaborazione su un singolo thread deterministico.

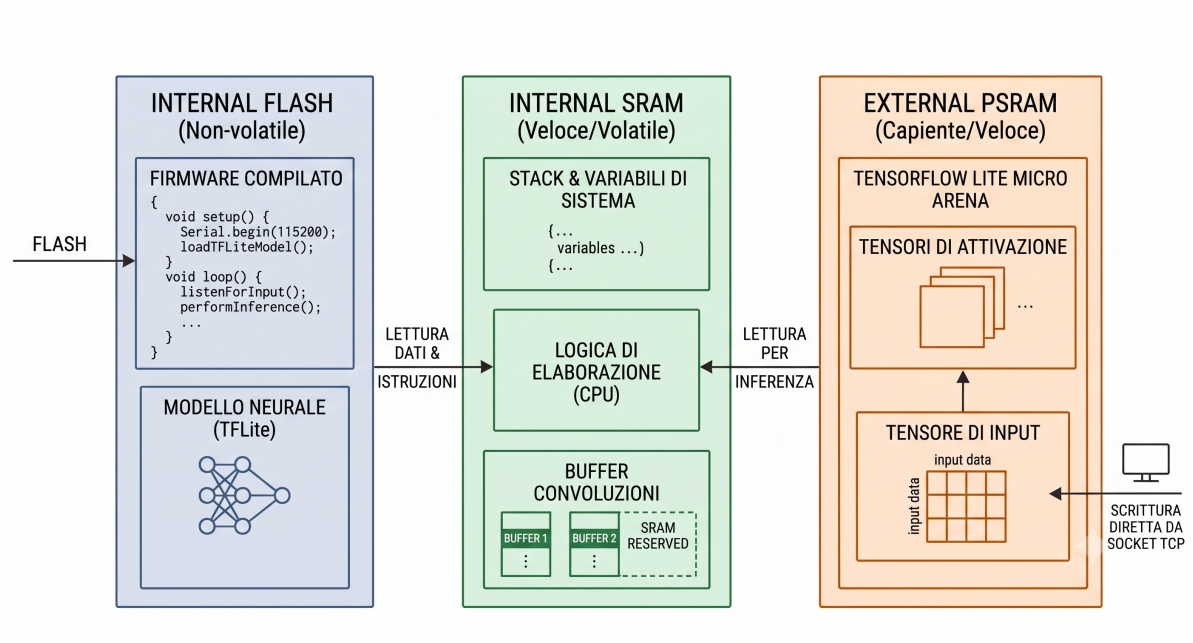


Figura 3.3: Architettura generale della meoria del Microcontrollore

3.2.1 Memoria e partizioni: Layout e Gestione del Modello

La tabella delle partizioni del dispositivo è stata adattata a una flash fisica complessiva da 16 MB. Per separare i dati del modello dal resto del sistema e supportare il caricamento dinamico di sotto-modelli di dimensioni elevate, è stata definita una partizione dati dedicata, denominata `model`, con una capacità di quasi 14 MB (`0xDE0000` byte).

Questa scelta consente di isolare i pesi della rete neurale dall'area riservata al firmware, garantendo al tempo stesso uno spazio contiguo e prevedibile per la memorizzazione del modello.

Name	Type	SubType	Offset	Size	Flags
nvs	data	nvs	0x9000	0x5000	
factory	app	factory	0x10000	0x200000	
model	data	0x40	0x210000	0xDE0000	
coredump	data	coredump	0xFF0000	0x10000	

Tabella 3.1: Tabella delle partizioni personalizzata per il dispositivo ESP32-S3

Per quanto concerne la gestione dei pesi a runtime, il firmware fa un uso estensivo delle API di sistema di basso livello dell'ESP-IDF per implementare una mappatura in memoria virtuale (**mmap**) dei pesi direttamente da flash, senza dover caricare il modello in RAM o in PSRAM.

Listing 3.1: Mappatura hardware della partizione tramite `esp_partition_mmap` in `network_server.cpp`. L'estratto è semplificato a scopo illustrativo; il codice completo gestisce il valore di ritorno con `markUploadError`.

```
1  // network_server.cpp - handleModelUpload(), fase UPLOAD_FILE_END
2  const void *ptr = nullptr;
3  esp_err_t err = esp_partition_mmap(model_partition, 0, model_size,
4                                     ESP_PARTITION_MMAP_DATA, &ptr,
5                                     &model_mmap_handle);
6  if (err != ESP_OK) {
7      markUploadError(500, "mmap failed");
8      return;
9  }
10 mapped_model_data = (const uint8_t *)ptr;
```

Come illustrato nel Listing 3.1, i pesi del sotto-modello vengono collegati allo spazio di indirizzamento virtuale della CPU XTensa LX7 tramite la chiamata a `esp_partition_mmap`. Questa scelta progettuale risulta di cruciale importanza per tre motivi:

- **Conservazione della PSRAM**: I pesi del modello neurale non occupano memoria volatile (SRAM o PSRAM). Infatti, risiedono interamente nella flash fisica e vengono letti dalla Memory Management Unit (MMU) hardware *on-demand* utilizzando un meccanismo di caching automatico del chip.

- **Supporto a modelli estesi**: Consente l'esecuzione di modelli con pesi le cui dimensioni superano la capacità totale di PSRAM fisicamente disponibile a bordo della scheda.

- **Sostituzione a caldo**: Il modello può essere aggiornato da remoto via rete senza richiedere un reboot hardware. L'handler HTTP si limita a invalidare il puntatore tramite `esp_partition_munmap()`, cancellare (*erase*) la partizione fisica, scrivere i nuovi dati e richiamare `esp_partition_mmap()`.

3.3 Architettura software

La struttura software implementata adotta un'architettura **single-threaded sequenziale**. Questa scelta è guidata dalla necessità di garantire il massimo livello di determinismo temporale durante le misurazioni: l'esecuzione di compiti di calcolo intensivo (come l'inferenza di reti neurali) su un unico thread previene i fenomeni di prelazione (*preemption*) e riduce al minimo l'overhead dovuto al cambio di contesto (*context switch*) introdotto dallo scheduler di FreeRTOS.

Il firmware è sviluppato all'interno dell'**Espressif IoT Development Framework (ESP-IDF)** [6], l'ambiente nativo messo a disposizione da Espressif per la programmazione dei dispositivi ESP32. L'uso di ESP-IDF permette di controllare in modo dettagliato le ottimizzazioni del compilatore, la configurazione del kernel FreeRTOS e le politiche di allocazione della memoria, aspetti particolarmente rilevanti in un sistema vincolato come quello analizzato. Per semplificare la gestione di componenti applicativi quali lo stack Wi-Fi e il server HTTP, il progetto integra inoltre la libreria `arduino-esp32` [15]. Ne risulta un approccio ibrido che combina la semplicità delle API Arduino con il controllo di basso livello e le possibilità di ottimizzazione offerte dall'ambiente nativo Espressif.

L'intero ciclo vitale del firmware è strutturato in modo da elaborare una sola richiesta alla volta. La logica software è suddivisa in quattro moduli funzionali interconnessi:

1. **Configurazione (config)**: Contiene i parametri globali per la connessione di rete e la definizione della struttura di instradamento (`NodeConfig`).

2. **Motore di Inferenza (inference)**: Gestisce l'inizializzazione del runtime di TensorFlow Lite per Microcontrollori e l'allocazione delle risorse di calcolo.

3. **Server di Rete (network_server)**: Gestisce l'interfaccia HTTP, il socket TCP e le funzioni di ricezione dati.

4. **Statistiche e Forwarding (metrics)**: Si occupa dell'instradamento sincrono dei risultati e dell'invio delle metriche al coordinatore esterno.

3.3.1 Configurazione e Iniezione Dinamica

Il modulo di configurazione definisce la struttura dati `NodeConfig`, la quale contiene l'indirizzo IP (`next_ip`) e un indicatore booleano (`last_hop`) che specifica se il nodo corrente è l'ultimo destinatario della pipeline.

Listing 3.2: Struttura NodeConfig

```
struct NodeConfig {
  IPAddress next_ip;
  bool last_hop; // true -> ultimo nodo
};
```

Una particolarità dell'architettura è che la configurazione del percorso dei dati viene iniettata dinamicamente in ciascun nodo tramite parametri di query HTTP all'atto dell'upload del modello (`next_ip`, `lasthop`, `master_ip`). Il design risultante è a **catena unidirezionale**: ogni nodo conosce soltanto l'elemento immediatamente successivo (`next_ip`) e il coordinatore centrale (`master_ip`) per l'invio delle metriche, senza alcuna conoscenza della topologia globale della rete. La configurazione iniziale del percorso dei dati e del modello avviene sulla porta HTTP `PORT` (8085), mentre la trasmissione dei tensori intermedi avviene su una porta TCP dedicata, impostata tramite la costante `TCP_PORT` (8086), utilizzando una connessione TCP permanente per eliminare l'overhead di setup della connessione ripetuto ad ogni inferenza. Nello stesso modulo vengono anche definite le costanti `TENSOR_ARENA_SIZE` e `ESP_NN_SCRATCH_SIZE`.

3.3.2 Motore di Inferenza TFLM e Tensor Arena Ibrida

Il modulo di inferenza incapsula la libreria TensorFlow Lite for Microcontrollers (TFLM) e le ESP Partitions API. All'avvio del sistema, il firmware individua la partizione *model* per ricavare il puntatore all'indirizzo di inizio della memoria (`model_partition`).

Per massimizzare l'efficienza delle risorse, il sistema adotta una **Tensor Arena Ibrida** distribuita tra PSRAM esterna e SRAM interna:

- **PSRAM esterna (6 MB)**: Utilizzata per contenere i tensori di attivazione e le matrici di lavoro dei diversi strati del modello. Sebbene la PSRAM OPI a 80 MHz sia fondamentale per l'elevata capacità richiesta, essa presenta latenze di accesso casuale non trascurabili.

- **SRAM interna (DRAM, 192 KB)**: Utilizzata per allocare un buffer di lavoro rapido (*scratch buffer*) per i calcoli temporanei delle operazioni di convoluzione (Conv2D e DepthwiseConv2D). Le routine matematiche di calcolo ottimizzate messe a disposizione dalla libreria `ESP-NN` di Espressif lavorano sul buffer veloce in SRAM interna, scrivendo in PSRAM soltanto il risultato finale, riducendo l'impatto della latenza della PSRAM esterna.

Listing 3.3: Allocazione della Tensor Arena in PSRAM esterna, in `setup_inference_partitions()` (inference.cpp).

```
// inference.cpp - setup_inference_partitions()
tensor_arena = (uint8_t *)ps_malloc(TENSOR_ARENA_SIZE);
```

Lo scratch buffer viene invece dichiarato come variabile globale a livello di file in `inference.cpp` e registrato nelle API ESP-NN all'interno di `init_interpreter()`, immediatamente dopo la chiamata ad `AllocateTensors()`, affinché il buffer sia valido per tutta la durata dell'esecuzione del modello.

Listing 3.4: Dichiarazione dello scratch buffer in SRAM interna e registrazione nelle API ESP-NN, in `init_interpreter()` (inference.cpp).

```
// inference.cpp - variabile globale di file
static uint8_t DRAM_ATTR esp_nn_scratch[ESP_NN_SCRATCH_SIZE];

// inference.cpp - init_interpreter(), dopo AllocateTensors()
esp_nn_set_conv_scratch_buf(esp_nn_scratch);
esp_nn_set_depthwise_conv_scratch_buf(esp_nn_scratch);
```

Per contenere l'impronta di memoria flash del binario (*binary footprint*), il firmware non registra l'intera libreria di operatori TFLM. Viene invece istanziato un risolutore personalizzato di tipo `tflite::MicroMutableOpResolver`, il quale mappa in modo selettivo solo ed esclusivamente gli undici operatori matematici necessari all'esecuzione delle architetture target studiate.

3.3.3 Server di Rete: Endpoint HTTP e Socket TCP

Il modulo `network_server` costituisce lo strato di comunicazione del microcontrollore. Esso gestisce in parallelo due distinti canali di comunicazione su porte differenti:

- **Server Web HTTP (porta 8085)**: istanzia la classe `WebServer` ed è dedicato alle operazioni di gestione, controllo, ispezione e caricamento del modello neurale.
- **Server TCP Raw (porta 8086)**: istanzia la classe `WiFiServer` ed è configurato per accogliere una connessione TCP permanente per lo scambio sequenziale e a bassissima latenza dei tensori intermedi tra i nodi della catena.

L'architettura software espone tre endpoint HTTP principali per la configurazione e il monitoraggio, elencati nella Tabella 3.2.

Endpoint / Porta	**Metodo / Tipo**	**Funzione principale**
`/upload_model`	HTTP POST (Multipart)	Caricamento modello e parametri di routing.
`/upload_input`	HTTP POST (Raw Binary)	Upload dell'input iniziale.
`/get_model_info`	HTTP GET	Ispezione dei metadati del modello attivo.
Porta `8086`	Socket TCP Raw	Trasmissione del tensore tra nodi adiacenti.

Tabella 3.2: Specifiche del server di rete implementato nel nodo IoT.

La gestione dei flussi di upload avviene tramite un meccanismo cooperativo a callback, integrato nel ciclo di polling del server. Il payload viene ricevuto in chunk sequenziali, limitati dalla dimensione della finestra TCP pari a 1460 byte; ogni chunk attiva la callback associata, che opera come una macchina a stati finiti guidata dalle transizioni della struttura `HTTPUpload` (`START`, `WRITE`, `END`) [13].

Nel caso dell'endpoint `/upload_model`, il processo è suddiviso nelle tre fasi atomiche illustrate nei Listing 3.5, 3.6 e 3.7. Nella fase iniziale (`UPLOAD_FILE_START`), il server estrae i parametri di routing necessari a configurare la struttura `node_config`. Successivamente, l'eventuale mappatura virtuale già presente viene invalidata tramite

`esp_partition_munmap()` e viene cancellata la porzione di memoria flash destinata a ospitare il nuovo file.

Listing 3.5: Allocazione e cancellazione della flash. La dimensione del modello viene letta dall'header HTTP personalizzato `size`, che il coordinatore Python deve includere nella richiesta multipart.

```
// La dimensione e' letta dall'header custom "size", non da Content-Length
size_t model_size = (size_t)server.header("size").toInt();

if (model_mmap_handle) {
    esp_partition_munmap(model_mmap_handle);
    mapped_model_data = nullptr;
    model_mmap_handle = 0;
}

// Allineamento della dimensione al blocco di cancellazione (sector size)
size_t erase_sz = ((model_size + model_partition->erase_size - 1) /
                   model_partition->erase_size) * model_partition->erase_size;

esp_err_t err = esp_partition_erase_range(model_partition, 0, erase_sz);
```

Durante la fase di scrittura (`UPLOAD_FILE_WRITE`), i chunk di dati vengono memorizzati direttamente in flash all'offset corrispondente al numero totale di byte già ricevuti.

Listing 3.6: Scrittura sequenziale del modello in flash.

```
esp_err_t err = esp_partition_write(model_partition, up.totalSize, up.buf, up.currentSize);
```

Al termine del trasferimento (`UPLOAD_FILE_END`), la partizione viene mappata nuovamente nello spazio di indirizzamento logico della CPU tramite `esp_partition_mmap()`. In questo modo, il modello diventa accessibile direttamente in memoria e può essere utilizzato per l'inizializzazione e l'hot-swap dell'interprete di inferenza.

Listing 3.7: Mappatura virtuale (mmap) e inizializzazione dell'interprete.

```
const void *ptr = nullptr;
esp_err_t err = esp_partition_mmap(model_partition, 0, model_size,
                                   ESP_PARTITION_MMAP_DATA, &ptr,
                                   &model_mmap_handle);
if (err != ESP_OK) {
    markUploadError(500, "mmap failed");
    return;
}
mapped_model_data = (const uint8_t *)ptr;
model_ready = init_interpreter();
if (!model_ready) {
    markUploadError(500, "Interpreter init failed");
}
```

Per l'endpoint di ricezione dati `/upload_input`, i singoli chunk HTTP vengono scritti direttamente nell'area di memoria del tensore di input, come mostrato nel Listing 3.8.

Listing 3.8: Scrittura diretta dei chunk nel tensore di input con protezione da overflow.

```
// Calcolo dell'offset corrente basato sui byte cumulativi ricevuti
size_t offset = raw.totalSize - raw.currentSize;

// Validazione dei limiti del tensore preallocato
if (offset + raw.currentSize <= input_tensor->bytes) {
    // Scrittura zero-copy nella memoria del tensore
    memcpy((int8_t *)input_tensor->data.int8 + offset, raw.buf, raw.currentSize);
} else {
    markUploadError(400, "Input upload overflow");
}
```

Poiché il tensore è già preallocato all'interno della `tensor_arena` dopo la chiamata iniziale a `AllocateTensors()`, il caricamento del payload durante lo streaming HTTP evita l'impiego di buffer intermedi, adottando un approccio *zero-copy*. La variabile semaforica `inference_pending` viene impostata a `true` solo alla ricezione del pacchetto di chiusura (`RAW_END`); in questo modo, l'esecuzione dell'inferenza tramite `Invoke()` avviene esclusivamente quando il dataset ricevuto è completo e coerente con le dimensioni attese dal modello.

3.3.4 Inoltro e Ricezione dei Tensori Intermedi tramite Socket TCP

Nello scenario di *model splitting*, una volta completata l'inferenza sulla porzione locale della rete neurale, il modulo `metrics` si occupa del trasferimento dei vettori di attivazione al nodo successivo. Per eliminare l'overhead tipico del protocollo HTTP, la trasmissione tra nodi adiacenti avviene tramite un socket TCP raw permanente gestito da un'istanza globale di `WiFiClient`.

Il protocollo di comunicazione prevede un meccanismo di controllo e validazione strutturato in tre fasi:

1. **Verifica del link:** Se il socket è disconnesso o l'IP del target cambia, la connessione esistente viene interrotta e ripristinata.

2. **Frammentazione e invio:** Viene trasmesso un header fisso di 4 byte indicante la dimensione del payload ($tensor\_size$), seguito dal flusso binario del tensore ripartito in chunk da 4096 byte per ottimizzare il buffer di rete.

3. **Validazione Zero-Copy e ACK:** Il server ricevente verifica che la dimensione dichiarata coincida esattamente con lo spazio allocato in PSRAM per l'array di input dell'interprete; in caso di esito positivo, i dati vengono scritti direttamente in memoria senza passaggi intermedi (*zero-copy*). La transazione si conclude con il ritorno di un byte di ACK (pari a 1).

La logica di inoltro e ricezione è implementata rispettivamente nelle funzioni `forward_data()` (Listing 3.9) e `handleTensorUpload()` (Listing 3.10).

Listing 3.9: Inoltro del tensore tramite socket TCP persistente in `forward_data()`.

```
// Gestione della connessione persistente e invio dei dati
if (!tcpClient.connected() || last_receiver_ip != next_ip) {
    tcpClient.stop();
    tcpClient.connect(next_ip.c_str(), TCP_PORT);
}
// Invio dell'header (dimensione) e del payload del tensore
tcpClient.write((uint8_t *)&size, sizeof(size));
tcpClient.write((uint8_t *)out->data.raw, size);
```

Listing 3.10: Ricezione non bloccante e scrittura zero-copy in `handleTensorUpload()`.

```
// Lettura header, validazione dimensione e scrittura zero-copy in PSRAM
activeClient.read((uint8_t *)&size_net, sizeof(size_net));
if (size_net == input_tensor->bytes) {
    activeClient.read((uint8_t *)input_tensor->data.int8 + total_read, to_read);
    activeClient.write(&ack, 1); // Invio del riscontro ACK
}
```

Contestualmente all'invio dei dati binari, il modulo `metrics` trasmette asincronamente tramite HTTP POST verso il coordinatore centrale un payload JSON contenente la telemetria prestazionale del nodo (Δt di inferenza locale, latenza di trasmissione `forward_ms` ed esito). Nel caso in cui il nodo operi come *last hop* della catena di elaborazione, il record JSON viene integrato con l'output finale del modello (`class_idx` e `class_score`), centralizzando così il monitoraggio e la raccolta dei dati predetti sul master.

3.4 Flusso di esecuzione del firmware

Il comportamento operativo del sistema segue la struttura bifase, basata sulle funzioni globali `setup()` e `loop()`. Dopo l'inizializzazione delle risorse hardware, della rete e dell'interprete TensorFlow Lite Micro, l'esecuzione procede all'interno del ciclo principale. La ricezione dei dati (sia tramite richieste HTTP che tramite socket TCP persistente) e l'esecuzione dell'inferenza non avvengono in parallelo, ma sono serializzate all'interno dello stesso flusso di controllo: durante la chiamata a `Invoke()` i server di rete non elaborano nuove richieste, mentre durante la ricezione del payload di rete il motore di inferenza rimane inattivo. Il coordinamento tra ricezione e inferenza è realizzato tramite il flag volatile `inference_pending`, che segnala al ciclo principale la disponibilità di un nuovo input da processare.

3.4.1 Polling e Pipeline Esecutiva

L'architettura del firmware adotta un approccio a polling sequenziale all'interno del ciclo principale `loop()`. Tale scelta rende il flusso di esecuzione prevedibile e riduce la necessità di primitive di sincronizzazione complesse, come mutex o semafori del sistema operativo. In questo modo si limita il rischio di conflitti di concorrenza nell'accesso alla memoria condivisa, in particolare alla `tensor_arena`, risorsa critica per l'esecuzione.

La pipeline esecutiva del nodo IoT si articola in quattro fasi principali:

1. **Gestione dello stack di rete.** A ogni iterazione del ciclo, la routine `handle_server_clients()` elabora le richieste HTTP pendenti sul server web (porta 8085) e controlla i dati in transito sul socket TCP raw permanente (porta 8086) per la ricezione dei tensori intermedi.

2. **Invocazione e isolamento dell'inferenza.** Quando lo strato di rete imposta il flag globale `inference_pending` a `true`, il firmware sospende temporaneamente l'elaborazione di nuove sessioni di rete e avvia l'esecuzione locale del modello tramite il metodo `Invoke()` dell'interprete TensorFlow Lite Micro. L'operazione viene isolata dal resto della pipeline e profilata misurandone la durata.

3. **Instradamento condizionale del tensore intermedio.** In base alla configurazione topologica del nodo, il flusso di esecuzione segue due comportamenti distinti:

   - se il dispositivo opera come nodo intermedio, l'output tensoriale parziale viene trasmesso al nodo successivo tramite il socket TCP permanente;
   - se il dispositivo corrisponde all'ultimo nodo della catena, il firmware applica un'operazione di *argmax* al vettore finale, estraendo l'indice della classe predetta e il relativo punteggio di confidenza.

4. **Invio della telemetria.** Al termine dell'elaborazione, la funzione `send_metrics()` invia al nodo master un payload JSON contenente le principali metriche di esecuzione.

Inoltre, nella routine di configurazione iniziale (`setup()`), viene chiamata esplicitamente la funzione `WiFi.setSleep(false)`. Questo disabilita la modalità di risparmio energetico (*WiFi power saving*) integrata nell'ESP32, la quale introduceva altrimenti latenze e ritardi significativi dovuti al passaggio in stato di riposo del chip RF durante i periodi di inattività, stabilizzando in tal modo i tempi di trasmissione di rete.

Questa organizzazione privilegia la semplicità del controllo e la riproducibilità delle misure sperimentali. L'assenza di esecuzione concorrente consente infatti di attribuire con maggiore precisione i tempi rilevati alle singole fasi della pipeline, separando il costo computazionale locale dal costo di comunicazione introdotto dal *model splitting*.

Capitolo 4

Valutazione delle prestazioni

Questo capitolo valuta sperimentalmente l'efficacia del *model splitting* su dispositivi ESP32-S3 collegati in una rete cooperativa. L'obiettivo non è soltanto misurare il tempo necessario a eseguire una singola inferenza, ma distinguere con chiarezza il contributo della computazione locale da quello della comunicazione tra nodi. In questo modo è possibile capire quando la suddivisione del modello è vantaggiosa e quando, invece, il trasferimento dei tensori intermedi diventa il principale limite prestazionale.

L'analisi è organizzata in quattro parti: prima vengono definite le metriche utilizzate, poi viene richiamata la configurazione del testbed, quindi vengono discussi i risultati di latenza e forwarding, e infine viene analizzato il consumo energetico del sistema.

4.1 Metriche prestazionali

Per separare i diversi contributi della pipeline distribuita sono state considerate le seguenti metriche:

- **Tempo di inferenza locale** (T_{inf}): indica il tempo, espresso in millisecondi, impiegato da ciascun ESP32-S3 per eseguire la propria partizione del modello tramite il metodo `Invoke()` dell'interprete TensorFlow Lite Micro. Questa misura descrive il costo computazionale puro del sotto-modello assegnato al nodo.

- **Tempo di forwarding** (T_{fwd}): misura il tempo necessario a inviare il tensore intermedio dal nodo corrente al nodo successivo. Nel sistema implementato comprende la preparazione del buffer, l'accesso alla PSRAM, l'apertura e gestione del socket TCP raw permanente e la trasmissione effettiva del payload.

- **Latenza end-to-end** (T_{e2e}): rappresenta il tempo complessivo stimato di una inferenza distribuita. Poiché non esiste un orologio unico sincronizzato condiviso tra i nodi, questa metrica non viene misurata direttamente da un unico timer globale, ma viene stimata sommando le metriche locali di calcolo e rete riportate autonomamente da ciascun dispositivo. In prima approssimazione, la latenza end-to-end è modellata come:

$$T_{e2e} \approx \sum_{i=1}^{N} T_{\text{inf},i} + \sum_{i=1}^{N-1} T_{\text{fwd},i} \tag{4.1}$$

dove N è il numero di nodi della catena. Questo approccio introduce una sottostima sistematica dovuta a latenze minori di dispatching o ritardi di rete inter-nodo esterni alle finestre temporali catturate dai singoli dispositivi.

- **Energia consumata (E)**: indica l'energia assorbita dal dispositivo nelle principali fasi operative. In particolare, sono stati distinti il consumo associato alla ricezione dell'input, alla computazione locale e all'inoltro del tensore verso il nodo successivo.

4.2 Configurazione del testbed e procedura di misura

Il testbed è composto da una catena di N schede ESP32-S3-DevKitC-1 identiche, configurate alla frequenza di clock massima di 240 MHz. Le schede sono mantenute in una disposizione fisica fissa e sono connesse alla stessa rete LAN Wi-Fi a 2.4 GHz della workstation master. La rete non è stata isolata dal traffico esterno: durante le prove erano presenti altri dispositivi connessi, così da riprodurre condizioni operative realistiche per uno scenario edge o TinyML domestico/industriale.

La workstation master esegue lo script Python di orchestrazione. Prima di ogni prova, lo script verifica lo stato dei nodi tramite l'endpoint `/get_model_info`, carica eventuali segmenti mancanti con `/upload_model`, configura il nodo successivo nella catena e avvia la pipeline inviando l'input quantizzato in formato `int8` all'endpoint `/upload_input` del primo dispositivo. Durante l'esecuzione, ciascun nodo elabora la propria partizione e invia al master le metriche raccolte tramite HTTP POST.

Ogni configurazione di split è stata valutata su 60 immagini. Questa scelta consente di calcolare medie e dispersioni statistiche e, soprattutto, di osservare la variabilità introdotta dalla comunicazione wireless, che risulta molto più instabile della computazione locale.

4.3 Risultati sperimentali e analisi comparativa

Si sono confrontate due diverse tecniche di suddivisione applicate al modello **MobileNetV2**, quantizzato interamente a `int8` e con input di dimensione $224 \times 224 \times 3$:

1. **prima tecnica (v1)**: utilizza punti di split collocati nelle prime fasi della rete, producendo partizioni iniziali piccole;

2. **seconda tecnica (v2)**: utilizza punti di split più profondi nel grafo, generando una prima partizione più grande;

Le configurazioni analizzate comprendono 2, 3 e 4 nodi, corrispondenti rispettivamente a 2, 3 e 4 partizioni del modello. I tempi medi di inferenza locale con i relativi scarti quadratici medi, e i tempi mediani di forwarding con i relativi intervalli interquartili $[Q_1, Q_3]$, sono riportati rispettivamente nelle Tabelle 4.1 e 4.2.

Configurazione	Partizione	Dimensione File	Tempo Calcolo (T_{inf})
1 Nodo (baseline)	Modello Intero	2,53 MB	$(4149,03 \pm 1,81)$ ms
1 Split (2 Nodi)	Part 0	136 KB	$(1933,93 \pm 0,58)$ ms
	Part 1	2,58 MB	$(2283,27 \pm 0,80)$ ms
	Totale v1	–	**4217,20 ms**
	Part 0	1,90 MB	$(3740,42 \pm 0,83)$ ms
	Part 1	757 KB	$(566,37 \pm 0,49)$ ms
	Totale v2	–	**4306,79 ms**
2 Split (3 Nodi)	Part 0	128 KB	$(1906,80 \pm 0,66)$ ms
	Part 1	730 KB	$(1254,83 \pm 0,62)$ ms
	Part 2	1,86 MB	$(1301,50 \pm 0,57)$ ms
	Totale v1	–	**4463,13 ms**
	Part 0	835 KB	$(3173,25 \pm 0,65)$ ms
	Part 1	1,07 MB	$(841,87 \pm 0,57)$ ms
	Part 2	757 KB	$(566,75 \pm 0,44)$ ms
	Totale v2	–	**4581,87 ms**
3 Split (4 Nodi)	Part 0	16 KB	$(1005,23 \pm 0,50)$ ms
	Part 1	114 KB	$(901,98 \pm 0,50)$ ms
	Part 2	730 KB	$(1254,78 \pm 0,61)$ ms
	Part 3	1,86 MB	$(1301,70 \pm 0,74)$ ms
	Totale v1	–	**4463,69 ms**
	Part 0	15,7 KB	$(1005,33 \pm 0,48)$ ms
	Part 1	823,6 KB	$(2162,53 \pm 0,70)$ ms
	Part 2	1,07 MB	$(841,73 \pm 0,52)$ ms
	Part 3	757 KB	$(566,68 \pm 0,47)$ ms
	Totale v2	–	**4576,27 ms**

Tabella 4.1: Tempi medi di inferenza locale (T_{inf}) con relativi scarti quadratici medi.

Configurazione	Tensore Output (Shape / Dim.)	Tempo Forwarding (T_{fwd})
1 Split (2 Nodi)	$14 \times 14 \times 192$ ($\approx 37,6$ KB)	$121,50$ ms $[104,50 - 246,00]$ ms
	Totale rete v1	**121,50 ms**
	$7 \times 7 \times 960$ ($\approx 47,0$ KB)	$117,50$ ms $[100,50 - 142,00]$ ms
	Totale rete v2	**117,50 ms**
2 Split (3 Nodi)	$28 \times 28 \times 192$ ($\approx 150,5$ KB)	$286,00$ ms $[235,00 - 314,00]$ ms
	$14 \times 14 \times 576$ ($\approx 112,9$ KB)	$209,50$ ms $[187,00 - 235,00]$ ms
	Totale rete v1	**495,50 ms**
	$14 \times 14 \times 576$ ($\approx 112,9$ KB)	$209,50$ ms $[187,00 - 235,00]$ ms
	$7 \times 7 \times 960$ ($\approx 47,0$ KB)	$117,50$ ms $[100,50 - 142,00]$ ms
	Totale rete v2	**327,00 ms**
3 Split (4 Nodi)	$56 \times 56 \times 96$ ($\approx 301,1$ KB)	$565,50$ ms $[535,00 - 622,00]$ ms
	$28 \times 28 \times 192$ ($\approx 150,5$ KB)	$286,00$ ms $[235,00 - 314,00]$ ms
	$14 \times 14 \times 576$ ($\approx 112,9$ KB)	$209,50$ ms $[187,00 - 235,00]$ ms
	Totale rete v1	**1061,00 ms**
	$56 \times 56 \times 96$ ($\approx 301,1$ KB)	$565,50$ ms $[535,00 - 622,00]$ ms
	$14 \times 14 \times 576$ ($\approx 112,9$ KB)	$209,50$ ms $[187,00 - 235,00]$ ms
	$7 \times 7 \times 960$ ($\approx 47,0$ KB)	$117,50$ ms $[100,50 - 142,00]$ ms
	Totale rete v2	**892,50 ms**

Tabella 4.2: Tempi mediani di trasmissione (T_{fwd}) e intervallo interquartile $[Q_1, Q_3]$

4.3.1 Comportamento del tempo di inferenza locale

La Tabella 4.1 evidenzia che la somma dei tempi di calcolo locale rimane relativamente stabile al variare del numero di partizioni. Il modello non suddiviso, eseguito su un singolo nodo, richiede in media $4149,03$ ms. Le configurazioni distribuite si collocano invece tra $4217,20$ ms e $4576,27$ ms, con un incremento massimo di circa 433 ms, pari a poco più del 10% rispetto alla baseline.

Questo risultato indica che la suddivisione del modello non comporta un'aumento significativo del lavoro computazionale. L'overhead osservato è riconducibile soprattutto alla trasformazione del grafo quantizzato in più sotto-grafi indipendenti. Quando un punto di split interrompe un blocco già ottimizzato, alcune fusioni tra operatori possono essere perse: operazioni che nel modello originale venivano eseguite da un unico kernel fuso possono essere espanse in più operazioni distinte, introducendo letture e scritture aggiuntive nella Tensor Arena [17, 18].

Un secondo contributo deriva dalla quantizzazione. Nei modelli interi a 8 bit, un valore reale r è rappresentato mediante parametri di scala e zero-point secondo una relazione affine del tipo $r = scale\,(q - zero\_point)$ [19, 20]. Se lo split produce un tensore con parametri di quantizzazione non direttamente compatibili con quelli attesi dal sotto-modello successivo, TensorFlow Lite deve inserire operazioni di riallineamento numerico, come moltiplicazioni, shift, arrotondamenti e saturazioni [21, 19]. Anche questo costo rimane contenuto, ma contribuisce all'overhead misurato.

Nel complesso, il tempo di inferenza locale risulta prevedibile e poco variabile: gli scarti quadratici medi delle singole partizioni restano nell'ordine di pochi millisecondi. Nel sistema analizzato, quindi, la componente computazionale è stabile; la principale fonte di variabilità va ricercata nella comunicazione tra nodi.

4.3.2 Comportamento del tempo di forwarding

Il tempo di forwarding T_{fwd} è la componente più sensibile della pipeline, come confermato dalla Tabella 4.2. A differenza della computazione locale, esso dipende da fattori esterni al modello: dimensione del tensore trasmesso, numero di hop, congestione del canale Wi-Fi, ritardi di accesso al mezzo, overhead TCP e possibili ritrasmissioni.

Gli split collocati nei layer iniziali producono feature map ancora ampie e, di conseguenza, payload elevati. Gli split più profondi riducono invece la risoluzione spaziale del tensore e abbassano il costo di rete. Questo spiega il miglioramento della variante v2: nello scenario a 3 nodi, il payload complessivo passa da $263,4$ KB nella v1 a $159,9$ KB nella v2, mentre il forwarding totale, espresso come somma dei tempi mediani, scende da $495,50$ ms a $327,00$ ms, con una riduzione di circa il 34%.

L'effetto è visibile anche nella latenza end-to-end. Nello scenario a 4 nodi, la v1 raggiunge una latenza media complessiva di $5654,51$ ms, mentre la v2 scende a $5515,46$ ms. La differenza è moderata perché la v2 presenta un tempo di computazione locale leggermente più alto; tuttavia, il dato conferma il principio progettuale principale: quando il costo computazionale resta quasi costante, la scelta dei punti di split deve minimizzare soprattutto la quantità di dati trasmessi.

4.3.3 Latenza end-to-end complessiva e confronto con la baseline

Per quantificare il costo pratico introdotto dallo split computing rispetto a un approccio monolitico, è utile confrontare la latenza end-to-end (T_{e2e}) delle configurazioni distribuite con la baseline eseguita su un singolo nodo (4149,03 ms). Questo confronto misura direttamente l'incremento del tempo di attesa percepito dall'applicazione master per ottenere il risultato finale dell'inferenza.

- **Scenario a 2 nodi (1 split)**: il tempo totale medio sale a 4392,90 ms per v1 (+5,88% rispetto alla baseline) e a 4450,59 ms per v2 (+7,27%). Il ritardo aggiuntivo introdotto dall'unico hop di rete è pari a 244 ms per v1 e 302 ms per v2. L'incremento resta contenuto e rende questa configurazione adatta quando l'obiettivo principale è distribuire un modello non caricabile interamente nella Flash o nella Tensor Arena di un singolo dispositivo.

- **Scenario a 3 nodi (2 split)**: con due hop di trasmissione intermedi, la latenza sale a 4944,98 ms per v1 (+19,18%) e a 4919,42 ms per v2 (+18,57%). Il delta rispetto alla baseline è di 796 ms per v1 e 770 ms per v2. In questo scenario v2 presenta un tempo di inferenza locale aggregato superiore (+118 ms rispetto a v1), ma un forwarding complessivo, calcolato come somma delle mediane, inferiore di 168,50 ms. Il vantaggio netto è quindi di soli 26 ms: l'*overhead* di rete non è ancora abbastanza dominante da rendere decisiva la scelta del punto di split.

- **Scenario a 4 nodi (3 split)**: con tre hop intermedi, la latenza end-to-end media si attesta a 5654,51 ms per v1 (+36,29%) e a 5515,46 ms per v2 (+32,93%). Il delta rispetto alla baseline è rispettivamente di 1505 ms e 1366 ms. In questa configurazione il risparmio di forwarding di v2, pari a 892,50 ms contro 1061,00 ms di v1, compensa la maggiore computazione locale (+113 ms) e produce un vantaggio netto di 139 ms. È quindi lo scenario in cui la scelta dei punti di split assume il peso progettuale maggiore.

In conclusione, la penalità temporale introdotta dalla pipeline distribuita cresce con il numero di hop: da circa 270 ms con un singolo split fino a 1,37–1,50 secondi con tre split. Per modelli con questo profilo computazionale, il vantaggio della v2 sulla v1 resta limitato finché il forwarding non diventa dominante rispetto alla computazione; con quattro nodi, invece, la riduzione del traffico intermedio produce un beneficio misurabile.

4.4 Analisi del forwarding e trade-off tra protocolli

I risultati sperimentali mostrano che il forwarding dei tensori intermedi è uno dei fattori più critici per l'inferenza distribuita. Nel sistema realizzato è stato utilizzato un socket TCP raw mantenuto aperto tra i nodi, così da evitare il costo di instaurazione della connessione a ogni trasferimento. La trasmissione wireless introduce comunque una variabilità non trascurabile, legata alla congestione del canale Wi-Fi, ai tempi di accesso al mezzo (CSMA/CA), all'overhead dello stack TCP/IP lwIP e alle possibili ritrasmissioni.

L'analisi statistica dei dati mostra che i tempi di forwarding presentano distribuzioni asimmetriche, caratterizzate da code positive pronunciate, dovute a picchi occasionali di

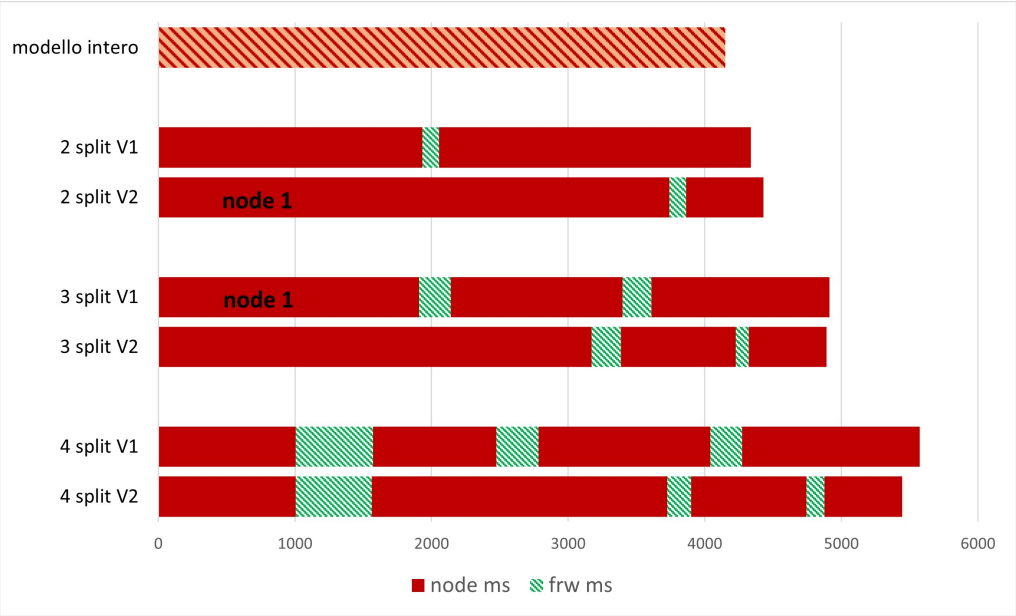


Figura 4.1: Latenza end-to-end complessiva rispetto alla baseline a singolo nodo.

latenza causati da interferenze, ritrasmissioni o contesa del canale. Per questa ragione la media aritmetica tende a sovrastimare il comportamento tipico del sistema: nelle analisi successive si privilegiano quindi la **mediana** e l'intervallo interquartile $[Q_1, Q_3]$, indicatori più robusti per descrivere la tendenza centrale delle prestazioni di rete.

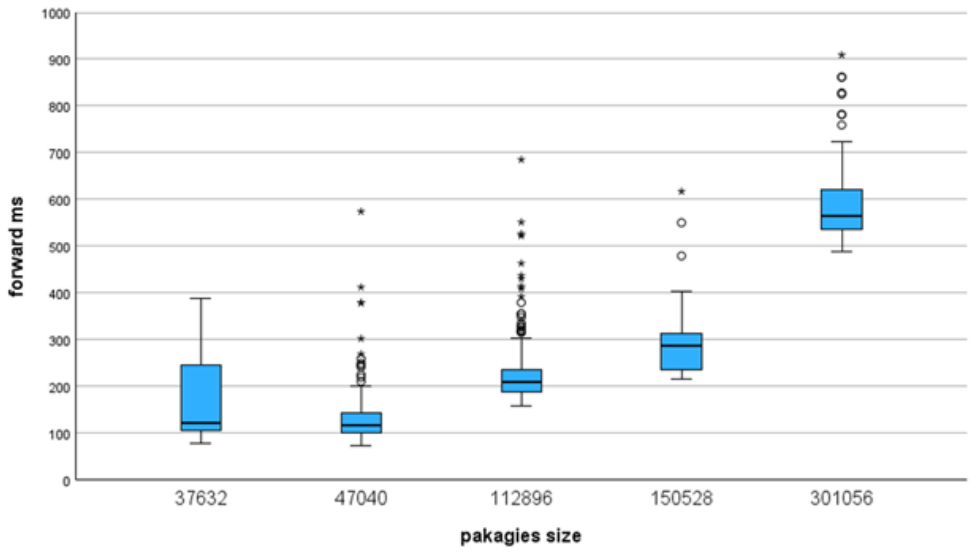


Figura 4.2: Box plot dei tempi di forwarding in funzione della dimensione del payload.

La Figura 4.2 mostra il box plot dei tempi di forwarding in funzione della dimensione dei pacchetti trasmessi. Le code positive delle distribuzioni sono particolarmente evidenti per i payload più grandi, come quello da $301,1$ KB ($56 \times 56 \times 96$), confermando l'importanza di usare le mediane per ridurre l'influenza delle fluttuazioni estreme sulle stime prestazionali.

Per modellare analiticamente le prestazioni del canale di comunicazione è stata adottata una formulazione semplificata del ritardo di trasmissione, coerente con il modello generale di rete proposto da Kurose e Ross [23]:

$$T_{\text{fwd}} = \frac{L}{R} + T_0 \tag{4.2}$$

dove L indica la dimensione del payload in bit, R il throughput e T_0 un termine di overhead fisso. Quest'ultimo rappresenta il costo minimo della trasmissione, assimilabile al tempo richiesto per inviare un pacchetto vuoto: comprende l'inizializzazione dell'invio, la gestione dei buffer in PSRAM e il contributo residuo introdotto dallo stack di comunicazione. Nel setup utilizzato, T_0 è stato stimato sperimentalmente a circa 50 ms.

La Figura 4.3 confronta i dati sperimentali con i limiti teorici e pratici del throughput:

- **Mediane misurate (punti verdi)**: indicano le mediane dei tempi di forwarding rilevate per ciascuna delle cinque dimensioni di pacchetto analizzate (37, 6 KB, 47, 0 KB, 112, 9 KB, 150, 5 KB e 301, 1 KB).
- **Limite teorico massimo (linea blu)**: calcolato con la formula $\frac{L}{R_{\max}} + T_0$, usando il throughput teorico massimo che la scheda ESP32-S3 supporta (7, 29 Mbps) [22].
- **Andamento empirico medio (linea arancione)**: calcolato usando il throughput reale medio di 5, 40 Mbps, corrispondente alla velocità di trasferimento effettivamente misurata nel testbed in presenza di traffico di rete e stack lwIP operante sul microcontrollore.

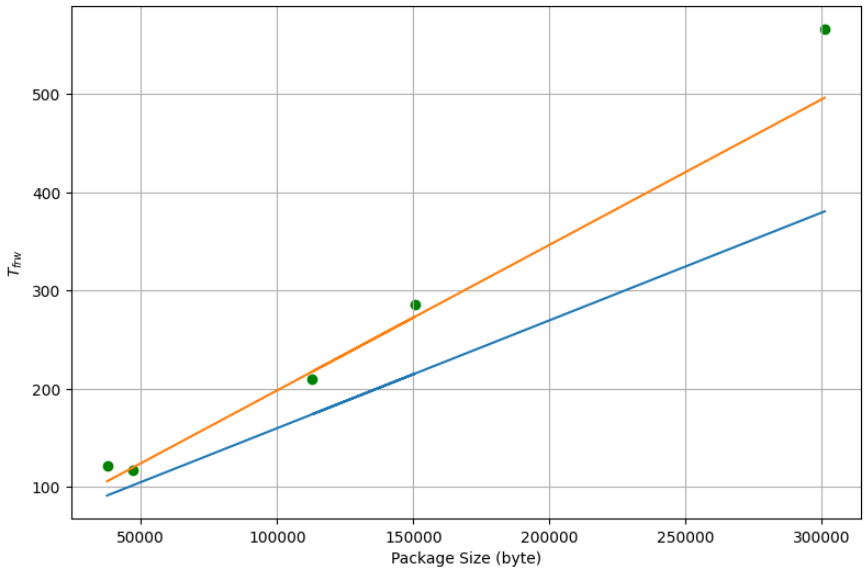


Figura 4.3: Confronto tra i tempi di forwarding mediani misurati (punti verdi), il limite teorico a 7, 29 Mbps (linea blu) e la curva empirica a 5, 40 Mbps (linea arancione), modellate secondo l'Equazione 4.2.

La buona aderenza dei punti verdi alla curva empirica (linea arancione) conferma che il modello adottato descrive in modo adeguato l'andamento osservato: il throughput effettivo del sistema, pari a $5,40$ Mbps, risulta coerente con le limitazioni hardware e di protocollo del dispositivo in un ambiente wireless realistico.

Questi risultati possono essere meglio contestualizzati confrontandoli con lo studio sperimentale di Jenhani et al. [2], che ha valutato sullo stesso hardware ESP32-S3 quattro protocolli di trasporto (TCP, UDP, ESP-NOW, BLE) per il forwarding di attivazioni intermedie di MobileNetV2. I risultati sono visibili nella Tabella 4.3. Gli autori distinguono esplicitamente la *latenza di trasmissione pura* del payload dal *tempo di setup* della connessione e dal *round-trip time (RTT)* complessivo, una scomposizione utile per interpretare anche i risultati di questa tesi:

- **UDP** ottiene la latenza di trasmissione più bassa in assoluto (pochi millisecondi anche per split profondi con tensori di pochi KB), grazie alla natura connectionless e all'MTU Wi-Fi relativamente ampio (1460–1472 byte). Tuttavia non garantisce affidabilità della consegna, aspetto che nel sistema qui sviluppato è invece richiesto per la correttezza dell'inferenza distribuita.

- **ESP-NOW**, pur avendo un payload massimo per pacchetto molto ridotto (250 byte), ottiene nel lavoro di Jenhani et al. l'RTT complessivo più basso tra tutti i protocolli testati, non per la velocità di trasmissione in sé ma perché il suo overhead di setup è quasi nullo (dell'ordine di poche decine di millisecondi, contro le centinaia o migliaia di millisecondi di TCP e UDP su Wi-Fi). Per tensori di grandi dimensioni, però, la frammentazione in pacchetti da 250 byte fa rapidamente crescere il numero di trasmissioni necessarie e, con esso, la latenza totale.

- **TCP**, sebbene garantisca affidabilità della consegna, mostra nello studio di riferimento la latenza di trasmissione più alta tra i protocolli IP, un effetto che gli autori attribuiscono principalmente all'overhead del three-way handshake, alla gestione degli ACK e, soprattutto per i payload più grandi, ai limiti del buffer TCP interno dell'ESP32, che può causare stalli e ritrasmissioni.

- **BLE** risulta il protocollo meno adatto al trasferimento di tensori di dimensioni non trascurabili, a causa dell'MTU contenuto (fino a 512 byte) che comporta una frammentazione molto più aggressiva rispetto agli altri protocolli.

Rispetto a questo quadro, la scelta architetturale adottata in questa tesi, ovvero un socket TCP raw permanente anziché connessioni HTTP/TCP riaperte a ogni trasferimento, risulta coerente con l'indicazione di Jenhani et al. secondo cui il costo di setup della connessione, più che la trasmissione del payload in sé, rappresenta spesso la componente dominante della latenza di rete su questi dispositivi. Eliminando l'handshake ripetuto a ogni inferenza, l'approccio qui implementato si avvicina concettualmente al comportamento di un protocollo a basso overhead di setup come ESP-NOW, pur mantenendo l'affidabilità tipica di TCP, a scapito però di una latenza di trasmissione pura comunque superiore a quella ottenibile con UDP.

Dal punto di vista progettuale emergono quindi tre indicazioni:

- **TCP/IP raw permanente**: rappresenta un compromesso ragionevole quando si privilegiano affidabilità della consegna e semplicità di integrazione, a patto di eliminare l'overhead di setup ripetuto, come fatto in questo lavoro.

Layers (Output shape)		block_2_expand		block_15_project		block_16_project_BN	
		(None, 56, 56, 48)		(None, 7, 7, 56)		(None, 7, 7, 112)	
Model size	(D1, D2)	752,6 kB, 11,8 MB		2,2 MB, 9,7 MB		2,7 MB, 9,2 MB	
Protocol	**Bytes per chunk**	**Latency**	**N. packets**	**Latency**	**N. packets**	**Latency**	**N. packets**
UDP	1472	145,1 ms	103	2,26 ms	2	5,2 ms	4
UDP	1460	83,9 ms	104	1,4 ms	2	3,2 ms	4
UDP	1200	98,3 ms	126	2,2 ms	3	3,7 ms	5
TCP	1472	558,7 ms	103	8,6 ms	2	19,2 ms	4
TCP	1460	563,3 ms	104	8,5 ms	2	19,3 ms	4
TCP	1200	393,9 ms	126	8,8 ms	3	15,719 ms	5
ESP-NOW	250	1897,0 ms	603	34,6 ms	11	69,2 ms	22
BLE	512	7305,94 ms	603	148,9 ms	11	272,9 ms	11

Tabella 4.3: Confronto latenze di trasmissione per diversi protocolli e punti di split, tratto da Jenhani et al. [2].

- **UDP con ACK applicativo**: potrebbe ridurre ulteriormente la latenza di trasmissione pura rispetto a TCP, mantenendo un controllo minimo sull'affidabilità tramite conferme a livello applicativo, particolarmente utile per tensori di dimensioni medio-grandi.
- **ESP-NOW**: offre un overhead di setup quasi nullo, ma il limite di payload per pacchetto ne consiglia l'uso soprattutto per split profondi, tensori molto piccoli o rappresentazioni compresse, condizioni in cui il numero di pacchetti necessari resta contenuto.

4.5 Analisi dei consumi energetici

La latenza non è l'unico parametro rilevante in un sistema TinyML. Nei contesti edge e IoT i nodi sono spesso alimentati a batteria o da sorgenti energetiche limitate, per cui la valutazione delle prestazioni deve includere anche il consumo energetico delle diverse fasi della pipeline. A questo scopo sono state misurate tensione e corrente assorbite dalla scheda ESP32-S3-DevKitC-1 durante l'esecuzione dell'esperimento.

Le misure sono state acquisite mediante un multimetro USB **FNIRSI FNB58**[1], interposto tra la sorgente di alimentazione e la porta USB della scheda, configurato con una frequenza di campionamento di 20 campioni al secondo. La tensione nominale osservata sulla linea USB è risultata pari a circa 5 V.

L'analisi energetica si concentra sulle quattro fasi attive principali della sequenza sperimentale:

1. **Upload del modello**: ricezione e scrittura in memoria Flash del sotto-modello da eseguire sul nodo.
2. **Upload dell'immagine**: ricezione del tensore di input da elaborare.
3. **Inferenza**: esecuzione locale della partizione tramite TFLM ed ESP-NN.

[1]Specifiche e documentazione del produttore: `https://www.fnirsi.com/products/fnb58`; manuale d'uso: `https://github.com/Rhomboid/fnb58-archive/blob/main/FNB58-manual-v2.2.pdf`

4. **Forwarding**: trasmissione del tensore di output verso il nodo.

Tra l'upload del modello e l'upload dell'immagine è presente inoltre un intervallo di attesa (*gap*), dovuto alla sincronizzazione e al caricamento dei modelli sugli altri nodi della catena. Questo intervallo non viene incluso nel totale delle fasi attive, ma risulta comunque utile per distinguere il consumo operativo da quello di mantenimento in stato di *idle*.

Per ciascuna fase, l'energia E è stata calcolata come prodotto tra la potenza media assorbita e la durata della fase. I valori riportati nella Tabella 4.4 sono ricavati dalla media di cinque misurazioni indipendenti, in modo da ridurre l'effetto di variazioni puntuali dovute alla rete o al comportamento istantaneo del microcontrollore.

Fase	Durata	Corrente Media	Potenza Media	Energia (J)
Upload Model	1663 ms	123,03 mA	0,6213 W	1,0333 J
Upload Image	800 ms	118,52 mA	0,5985 W	0,4788 J
Inferenza	1934 ms	132,66 mA	0,6694 W	1,2947 J
Forwarding	107 ms	114,35 mA	0,5774 W	0,0618 J
Totale	**4504 ms**	**126,16 mA**	**0,6369 W**	**2,8686 J**

Tabella 4.4: Consumi energetici medi per le fasi attive del modello MobileNetV2 v1 (modello 0) su ESP32-S3, ricavati dalla media di cinque misurazioni. I totali si riferiscono alla somma delle sole fasi attive, escludendo i gap di attesa del modello e della rete.

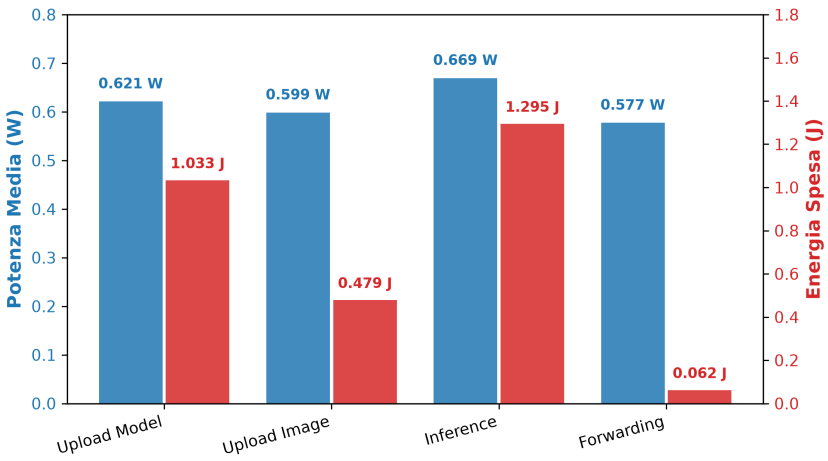


Figura 4.4: Confronto grafico tra la potenza elettrica media assorbita (in blu) e l'energia consumata (in rosso) nelle fasi operative.

La fase con la potenza media più elevata è l'inferenza locale, pari a $0,6497$ W, con una corrente media di $129,84$ mA, un risultato coerente con l'uso intensivo della CPU durante i calcoli convoluzionali e con l'impiego delle primitive ottimizzate di ESP-NN. Anche l'upload del modello mostra un assorbimento significativo ($0,6339$ W), riconducibile alla ricezione dei dati via Wi-Fi e alla contestuale scrittura in memoria Flash.

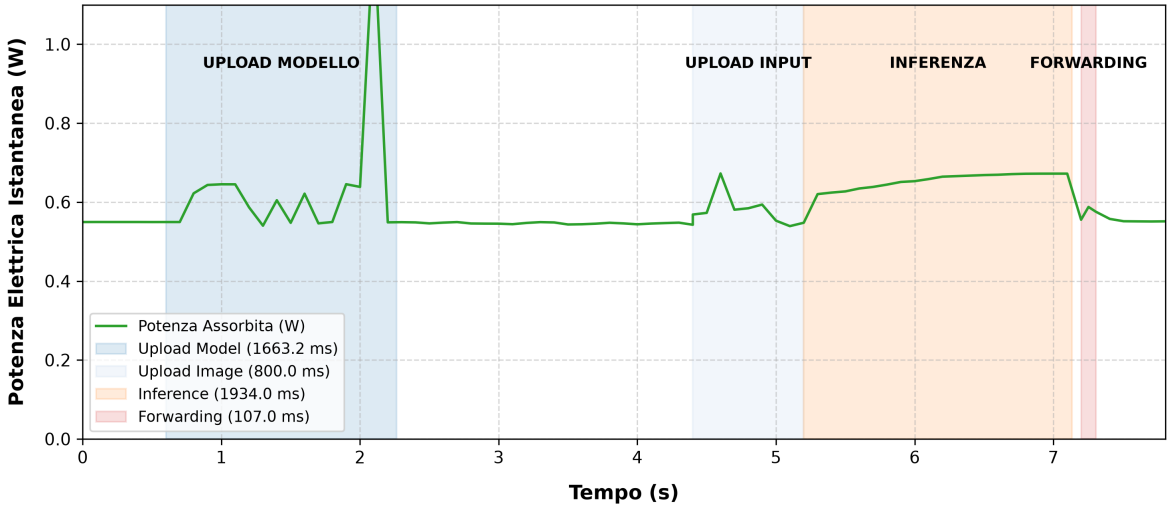


Figura 4.5: Profilo temporale della potenza elettrica istantanea assorbita dal microcontrollore ESP32-S3 durante le fasi della pipeline, rappresentativo di una singola misurazione.

L'upload dell'immagine e il forwarding presentano invece potenze medie più contenute, pari rispettivamente a $0,5792$ W e $0,5878$ W: valori verosimilmente legati a un'attività più leggera dell'interfaccia di rete, priva in questi casi dell'overhead aggiuntivo introdotto dalla scrittura in Flash o dal calcolo intensivo.

Il profilo temporale riportato in Figura 4.5 permette di distinguere chiaramente i diversi livelli di assorbimento del microcontrollore. Nelle fasi di inattività, ossia prima del caricamento, durante il gap di attesa tra 2,26 s e 4,4 s e dopo il forwarding, il consumo si mantiene stabilmente sul livello di base (*idle*), pari a circa $0,55$ W, corrispondente a una corrente di $\approx 109,5$ mA. Durante l'upload del modello, invece, l'attività di ricezione e scrittura in Flash produce un picco caratteristico che supera $1,2$ W al termine del trasferimento. Una volta completato il salvataggio, la potenza assorbita ritorna rapidamente alla linea di base. Complessivamente, l'energia richiesta per completare le quattro fasi attive considerate è pari a circa 2,87 J, equivalenti a 0,80 mWh. La quota maggiore è dovuta all'inferenza locale e all'upload del modello, mentre il forwarding incide in misura marginale a causa della breve durata della trasmissione. Il risultato evidenzia l'efficienza energetica complessiva dell'esecuzione su ESP32-S3 e conferma che l'ottimizzazione della comunicazione contribuisce non solo alla riduzione della latenza, ma anche al contenimento del consumo energetico complessivo della pipeline.

Conclusioni

Il lavoro di tesi ha analizzato il *model splitting* come tecnica per distribuire l'inferenza di reti neurali su più microcontrollori ESP32-S3. L'obiettivo era verificare se la cooperazione tra nodi potesse superare i limiti di memoria di un singolo dispositivo, mantenendo al tempo stesso latenze e consumi compatibili con scenari TinyML reali.

I risultati sperimentali mostrano che la risposta dipende dal bilanciamento tra due fattori: da un lato il carico computazionale locale,che rimane stabile e prevedibile, dall'altro il costo di comunicazione, che cresce con la dimensione dei tensori intermedi e con il numero di hop della rete. Da questa analisi emergono tre conclusioni principali.

1. **Lo splitting abilita l'esecuzione distribuita.** La somma dei tempi di inferenza locale resta vicina alla baseline a nodo singolo: rispetto ai $4149,03$ ms del modello intero, le configurazioni distribuite misurate si collocano tra $4217,20$ ms e $4581,87$ ms. L'overhead computazionale massimo è quindi contenuto, nell'ordine del 10%. Questo risultato indica che la suddivisione non altera in modo sostanziale il costo di calcolo complessivo: ogni nodo esegue una porzione più piccola della rete e la somma delle elaborazioni locali rimane confrontabile con l'esecuzione monolitica.

   Il contributo principale dello splitting è quindi la possibilità di rendere concretamente eseguibili modelli complessi su hardware vincolato. Distribuendo i pesi e le operazioni su più schede, il sistema può eseguire in modo cooperativo modelli o partizioni che supererebbero i limiti di memoria di un singolo nodo, trasformando un vincolo locale in un problema di coordinamento tra dispositivi. Questo aspetto è particolarmente importante in scenari IoT multi-hop, nei quali i nodi sono già organizzati in rete e possono collaborare senza richiedere necessariamente l'invio dei dati grezzi verso un server remoto. In questo modo, una parte dell'elaborazione rimane vicina alla sorgente del dato, con potenziali benefici anche in termini di scalabilità, autonomia dell'infrastruttura, sicurezza e riduzione della dipendenza dalla connettività cloud. In questa prospettiva, reti più grandi diventano candidati naturali per l'esecuzione distribuita: un modello che nel suo insieme richiederebbe troppa memoria per i pesi può essere suddiviso in 5 partizioni, rendendo possibile il caricamento su dispositivi edge con memoria limitata. La suddivisione, tuttavia, non deve essere casuale, lo split point deve essere scelto considerando congiuntamente la dimensione dei segmenti, il tempo di inferenza locale e la quantità di dati da trasferire al nodo successivo. I risultati ottenuti confermano quindi che il *model splitting* è una strategia efficace per estendere la classe di modelli eseguibili su microcontrollori, ma richiede una progettazione consapevole dell'intera pipeline distribuita.

2. **Il forwarding è il principale collo di bottiglia della pipeline.** I dati sperimentali mostrano che la comunicazione tra nodi è molto più variabile della computazione locale e dipende in modo diretto dalla dimensione dei tensori intermedi trasferiti. Nel caso a 3 nodi, v2 riduce il payload complessivo da $263,4$ KB a $159,9$ KB, portando il forwarding complessivo (somma dei tempi mediani) da $495,50$ ms a $327,00$ ms. Nello scenario a 4 nodi la stessa strategia riduce il traffico intermedio e il tempo di trasmissione totale da $1061,00$ ms a $892,50$ ms. In entrambi i casi il guadagno cresce con il numero di hop, confermando che l'impatto della dimensione del payload sul tempo di forwarding si amplifica man mano che aumentano i nodi attraversati dalla pipeline.

   Questo risultato indica che la scelta dei punti di split deve privilegiare tensori intermedi compatti, e non soltanto partizioni bilanciate in termini di dimensione dei file. In questa direzione, anche una progettazione del modello orientata a input di risoluzione inferiore, ad esempio $96 \times 96 \times 3$ invece di $224 \times 224 \times 3$, rappresenterebbe una leva complementare: riducendo le dimensioni dell'immagine in ingresso si riducono proporzionalmente anche le attivazioni intermedie generate da ciascun layer, con un beneficio atteso notevole sia sul tempo di inferenza sia, soprattutto, sul tempo di forwarding.

3. **L'implementazione deve considerare anche la Tensor Arena.** La memoria occupata dai pesi non è l'unico vincolo da valutare: durante l'inferenza TensorFlow Lite for Microcontrollers richiede anche una porzione di RAM temporanea per memorizzare attivazioni, buffer intermedi e strutture ausiliarie, indicata come **Tensor Arena**. Questo aspetto è particolarmente critico nei microcontrollori, perché tale memoria deve essere disponibile in modo contiguo e dipende non solo dalla dimensione del modello, ma anche dalla forma dei tensori prodotti dai layer e dagli operatori utilizzati. Di conseguenza, una partizione può risultare caricabile in flash o in memoria non volatile, ma non necessariamente eseguibile se la Tensor Arena richiesta supera la RAM effettivamente disponibile sul nodo. Un esempio significativo è dato da una possibile suddivisione di **VGG-16**: lo splitting può rendere gestibile la dimensione dei pesi delle singole partizioni, ma l'esecuzione con input $224 \times 224 \times 3$ richiede una Tensor Arena di circa $13\,034\,304$ **byte** ($\approx 12,4$ MB), valore non compatibile con la RAM interna dell'ESP32-S3. Riducendo l'input a $96 \times 96 \times 3$, il requisito stimato scende a circa $4\,718\,592$ **byte** ($\approx 4,5$ MB): il miglioramento è evidente. La progettazione dello split deve quindi includere anche il profilo di memoria dinamica di ciascun segmento, evitando partizioni che generano picchi di attivazioni troppo grandi anche quando i pesi risultano distribuiti in modo bilanciato.

4. **Il costo energetico dipende sia dal calcolo sia dalla comunicazione.** Dal punto di vista energetico, una iterazione completa composta dalle fasi attive di upload del modello, inferenza e forwarding consuma circa 2,87 J (0,80 mWh), come riportato in Tabella 4.4. Questo valore evidenzia che il consumo non è determinato soltanto dal calcolo locale. Anche la trasmissione dei tensori intermedi contribuisce al bilancio complessivo, soprattutto quando il payload è grande o attraversa più hop. In una pipeline distribuita, infatti, l'energia totale non dipende solo dalla potenza istantanea assorbita da ciascuna fase, ma anche dalla sua durata: una fase con potenza media più bassa può comunque incidere in modo rilevante se rimane

attiva per un intervallo di tempo più lungo. La fase di forwarding presenta una potenza media inferiore rispetto all'inferenza, ma può durare abbastanza a lungo da incidere in modo non trascurabile sull'energia totale. Questo comportamento conferma il legame tra ottimizzazione della comunicazione e sostenibilità energetica del sistema: ridurre la dimensione dei tensori intermedi, scegliere split point che producono payload più compatti e limitare il numero di hop non migliora soltanto la latenza end-to-end, ma riduce anche il tempo in cui i moduli radio rimangono attivi. Di conseguenza, l'efficienza energetica deve essere considerata un criterio di progetto al pari della memoria e della latenza, soprattutto in scenari alimentati a batteria o soggetti a duty cycle stringenti. In questi contesti, anche piccoli miglioramenti sul forwarding possono tradursi in un aumento dell'autonomia operativa dei nodi e in una maggiore sostenibilità della pipeline distribuita nel lungo periodo.

In sintesi, il *model splitting* si conferma una soluzione promettente per portare modelli neurali più complessi su dispositivi TinyML, purché la suddivisione venga progettata in modo congiunto rispetto a tre vincoli: memoria disponibile, dimensione dei tensori intermedi e qualità del collegamento wireless. L'adozione di protocolli a minore overhead, come UDP con conferma applicativa o ESP-NOW per payload molto piccoli, rappresenta una direzione di ottimizzazione prioritaria. Un ulteriore sviluppo naturale del lavoro consiste nella definizione di un algoritmo di partizionamento automatico, capace di scegliere lo split point sulla base del profilo del modello e delle condizioni effettive della rete.

Bibliografia

[1] Trotta, A., Esposito, A., Sciullo, L., Bononi, L. and Di Felice, M., 2026. Private Inference at the Extreme Edge: Joint Mixed Precision Quantization and Model Splitting in Multi-Hop IoT Networks. Disponibile a: https://cris.unibo.it/handle/11585/1051471.

[2] Jenhani, Z., Bensalem, M., Dizdarević, J. and Jukan, A., 2025. An Experimental Study of Split-Learning TinyML on Ultra-Low-Power Edge/IoT Nodes. arXiv preprint arXiv:2507.16594. Disponibile a: https://arxiv.org/abs/2507.16594.

[3] Kang, Y., Hauswald, J., Gao, C., Rovinski, A., Mudge, T., Mars, J. and Tang, L., 2017. Neurosurgeon: Collaborative Intelligence Between the Cloud and Mobile Edge. In: Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '17). DOI: https://doi.org/10.1145/3037697.3037698.

[4] Google AI Edge, 2024. LiteRT for Microcontrollers. [Online] Disponibile a: https://ai.google.dev/edge/litert/microcontrollers.

[5] FreeRTOS, 2026. The FreeRTOS Kernel. [Online] Disponibile a: https://www.freertos.org/Documentation/02-Kernel/01-About-the-FreeRTOS-kernel/01-FreeRTOS-kernel.

[6] Espressif Systems, 2026. ESP-IDF Programming Guide for ESP32-S3. [Online] Disponibile a: https://docs.espressif.com/projects/esp-idf/en/stable/esp32s3/index.html.

[7] Espressif Systems, 2023. ESP32-S3-DevKitC-1 v1.1 User Guide. [Online] Disponibile a: https://docs.espressif.com/projects/esp-dev-kits/en/latest/esp32s3/esp32-s3-devkitc-1/user_guide_v1.1.html#hardware-reference.

[8] Espressif Systems, 2024. ESP32-S3-WROOM-1 Datasheet. [Online] Disponibile a: https://documentation.espressif.com/esp32-s3-wroom-1_wroom-1u_datasheet_en.pdf.

[9] Espressif Systems, 2024. ESP-IDF Partition Tables Guide. [Online] Disponibile a: https://docs.espressif.com/projects/esp-idf/en/stable/esp32/api-guides/partition-tables.html.

[10] Espressif Systems, 2024. ESP-IDF Partition API Reference. [Online] Disponibile a: https://docs.espressif.com/projects/esp-idf/en/stable/esp32/api-reference/storage/partition.html.

[11] Distributed learning of deep neural network over multiple agents [Online] Disponibile a: https://arxiv.org/pdf/1810.06060

[12] Hara, T. and Sasabe, M., 2026. Optimization of Model Splitting, Placement, and Chaining for Multi-hop Split Learning and Inference. arXiv preprint arXiv:2604.25197. Disponibile a: https://arxiv.org/pdf/2604.25197.

[13] Avantmaker, 2024. ESP32 WebServer Library — File Upload Handling. In: Avantmaker References & Documentation, URL: https://avantmaker.com/references/esp32-arduino-core-index/esp32-webserver-library/esp32-webserver-library-upload/.

[14] Avantmaker, 2024. ESP32 WebServer Library — Client Request Handling. In: Avantmaker References & Documentation, URL: https://avantmaker.com/references/esp32-arduino-core-index/esp32-webserver-library/esp32-webserver-library-handleclient/.

[15] Espressif Systems, 2026. Arduino Core for ESP32. [Online] Disponibile a: https://github.com/espressif/arduino-esp32.

[16] Espressif Systems, 2026. ESP-NN Library. [Online] Disponibile a: https://github.com/espressif/esp-nn.

[17] Google AI Edge, 2024. TensorFlow Lite Operator Fusion. [Online] Disponibile a: https://ai.google.dev/edge/litert/models/optimize_ops.

[18] David, R., Duke, J., Jain, A., Janapa Reddi, V. et al., 2021. TensorFlow Lite Micro: Embedded Machine Learning on TinyML Systems. In: Proceedings of Machine Learning and Systems (MLSys 2021). arXiv preprint arXiv:2010.08655.

[19] Google AI Edge, 2024. TensorFlow Lite 8-bit Quantization Specification. [Online] Disponibile a: https://ai.google.dev/edge/litert/models/quantization_spec.

[20] Google AI Edge, 2024. Quantization Parameters and Scale Representation in TensorFlow Lite. [Online] Disponibile a: https://ai.google.dev/edge/litert/models/quantization.

[21] Google, 2024. Gemmlowp: a small self-contained matrix multiplication library - Output Pipeline. [Online] Disponibile a: https://github.com/google/gemmlowp/blob/master/doc/output.md.

[22] TutoDuino, 2023. ESP32 WiFi Performance. [Online] Disponibile a: https://tutoduino.fr/en/esp32-wifi-performance/.

[23] Kurose, J.F. and Ross, K.W., 2021. Computer Networking: A Top-Down Approach. 8th ed. Pearson.

Ringraziamenti