

UNIVERSITÀ POLITECNICA DELLE MARCHE

FACOLTÀ DI INGEGNERIA

Corso di LAUREA IN INGEGNERIA ELETTRONICA E DELLE TECNOLOGIE DIGITALI

Implementazione di algoritmi di Intelligenza Artificiale non supervisionata per il rilevamento di guasti ad alta impedenza su piattaforma STM32 in scenari di Cold Ironing

Tesi di Laurea di:
Diego Lombi

Relatore: Chiar.mo
Prof. **Emanuele Principi**

Correlatori:
Stefano Squartini
Giulia Tanoni

Anno Accademico 2025/2026

Abstract

La presente tesi ha come obiettivo lo studio e l'analisi di un modello di Intelligenza Artificiale non supervisionata per il rilevamento di guasti ad alta impedenza (High Impedance Fault, HIF) in scenari di Cold Ironing e la successiva implementazione su una piattaforma embedded STM32.

In tali applicazioni, i collegamenti elettrici tra banchina e nave possono essere sottoposti a condizioni ambientali e sollecitazioni meccaniche che favoriscono il degrado dell'isolamento. Gli HIF risultano particolarmente critici poiché sono caratterizzati da correnti di guasto relativamente ridotte e da un comportamento non lineare, caratteristiche che possono renderne difficile l'individuazione mediante le tradizionali protezioni di sovracorrente.

Il metodo analizzato si basa su un autoencoder con architettura Temporal Convolutional Network (TCN), addestrato esclusivamente su condizioni di funzionamento sano. Il rilevamento dell'anomalia viene effettuato attraverso l'errore di ricostruzione dei segnali trifase di tensione e corrente, successivamente aggregato a livello di segmento e confrontato con una soglia di decisione.

Per rendere possibile il porting sulla piattaforma STM32, il modello originario è stato adattato ai vincoli di memoria mediante la riduzione del numero di canali nascosti e la successiva quantizzazione INT8. Tali interventi hanno consentito di ridurre significativamente l'occupazione di memoria fino a rendere il modello compatibile con le risorse disponibili sulla board.

Infine, sono state confrontate le prestazioni delle diverse configurazioni sia su PC sia sulla piattaforma embedded, analizzando l'effetto delle modifiche introdotte sulla capacità di rilevamento degli HIF e sui tempi di inferenza. I risultati evidenziano il compromesso tra riduzione delle risorse richieste, accuratezza del modello e costo computazionale dell'esecuzione su microcontrollore.

Indice

Elenco delle figure **iv**

Elenco delle tabelle **v**

1 Introduzione e stato dell'arte **2**
1.1 Contesto applicativo e Cold Ironing 2
1.2 High Impedance Fault 3
1.3 Stato dell'arte sul rilevamento degli HIF 4
1.4 Approcci di anomaly detection 5
1.5 Obiettivo del lavoro 6
1.6 Organizzazione della tesi 7

2 Modello e metodologia **8**
2.1 Architettura TCN autoencoder 8
2.2 Dataset e criterio di anomaly detection 11

3 Piattaforma e implementazione su STM32 **13**
3.1 Piattaforma hardware 13
3.2 Ambiente software e strumenti di sviluppo 14

4 Metodo di sviluppo e implementazione **16**
4.1 Conversione e validazione del modello 16
4.2 Prime verifiche sulla piattaforma 17
4.3 Generazione degli input per la board 18
4.4 Vincoli di memoria e ottimizzazione del modello 18
4.5 Implementazione del firmware 20
4.6 Setup hardware 21
4.7 Problematica relativa alla SD Card 23
4.8 Soluzione alternativa tramite comunicazione USB 24

5 Risultati e discussione **26**
5.1 Confronto tra i formati ONNX e TensorFlow Lite 26
5.2 Effetto della quantizzazione 27
5.3 Risultati ottenuti su STM32 31
5.4 Analisi dei tempi di inferenza sulla STM32 33

5.5 Sintesi dei risultati 35

6 Conclusioni **37**

Bibliografia **40**

Ringraziamenti **42**

Elenco delle figure

1.1 Schema semplificato di un sistema di cold ironing. 3
1.2 Confronto tra le correnti trifase in condizioni di funzionamento sano e in presenza di High Impedance Fault, nella finestra temporale compresa tra 0,45 s e 0,55 s. 4

2.1 Schema generale della procedura di pre-processing e anomaly detection utilizzata 9
2.2 Architettura dell'autoencoder TCN nella configurazione finale a 56 hidden channels quantizzata INT8, comprensiva della fase di post-processing per l'ottenimento della label di classe. 11

4.1 Effetto della modalità di crop sul frame score: confronto tra crop standard e crop *center*. 18
4.2 Schema a blocchi del firmware implementato sulla piattaforma STM32. 22

5.1 Confronto tra le confusion matrix della configurazione a 56 hidden channels prima e dopo la quantizzazione INT8. 28
5.2 Confronto delle metriche di errore dei modelli a 56 hidden channels rispetto al modello originale a 64 hidden channels in formato Float32. 29
5.3 Distribuzione degli anomaly score del modello quantizzato sul dataset di test, con indicazione della soglia di decisione τ. I segmenti sani e quelli caratterizzati da High Impedance Fault sono rappresentati separatamente. 30
5.4 Ripartizione delle operazioni eseguite dal modello quantizzato tra aritmetica INT8, floating point e conversioni tra i due formati. 31
5.5 Anomaly score ottenuti per i 10 segmenti elaborati sulla piattaforma STM32 e confronto con la soglia di decisione τ. 32

Elenco delle tabelle

4.1 Confronto tra la rete originale e la rete quantizzata. 19
4.2 Confronto dell'occupazione di memoria delle principali configurazioni analizzate tramite STM32CubeAI durante il porting del modello. . . 20

5.1 Confronto delle principali metriche di classificazione. I risultati sulla STM32 sono riferiti ai 10 segmenti effettivamente elaborati sulla board. 27
5.2 Confronto degli anomaly score e delle predizioni ottenuti sul PC e sulla piattaforma STM32 per i medesimi 10 segmenti di test. 33
5.3 Confronto dei tempi medi di inferenza per singolo frame nelle diverse configurazioni considerate. 34
5.4 Sintesi delle prestazioni di classificazione, della complessità computazionale, dei tempi di inferenza e dei requisiti di memoria delle principali configurazioni considerate. 36

Capitolo 1

Introduzione e stato dell'arte

1.1 Contesto applicativo e Cold Ironing

Nei sistemi portuali le condizioni costiere, come umidità e salsedine, portano a un maggiore degrado delle apparecchiature elettriche ed elettroniche, causando, in particolare, problemi nel corretto isolamento degli impianti di media tensione.
In questo caso si fa riferimento a uno scenario di *cold ironing*, o *shore-to-ship power*. Esso consiste nell'elettrificazione delle banchine portuali con lo scopo di fornire energia elettrica alle navi ormeggiate, permettendo loro di spegnere i motori ausiliari a combustibili fossili usati per i servizi di bordo. Il fine è, dunque, quello di ridurre le emissioni, eliminando lo smog in porto.
L'introduzione del cold ironing porta però con sé nuove criticità dal punto di vista della protezione e del monitoraggio degli impianti. I cavi di collegamento nave-banchina sono infatti sottoposti a cicli continui di connessione e scollegamento, a sollecitazioni meccaniche dovute al moto ondoso e agli ormeggi, e a un ambiente particolarmente aggressivo per i materiali isolanti, caratterizzato da umidità elevata, salsedine e forti escursioni termiche. Questi fattori accelerano l'invecchiamento dell'isolamento e aumentano la probabilità che si verifichino cedimenti localizzati, specialmente sui cavi di media tensione utilizzati per l'alimentazione a bordo.

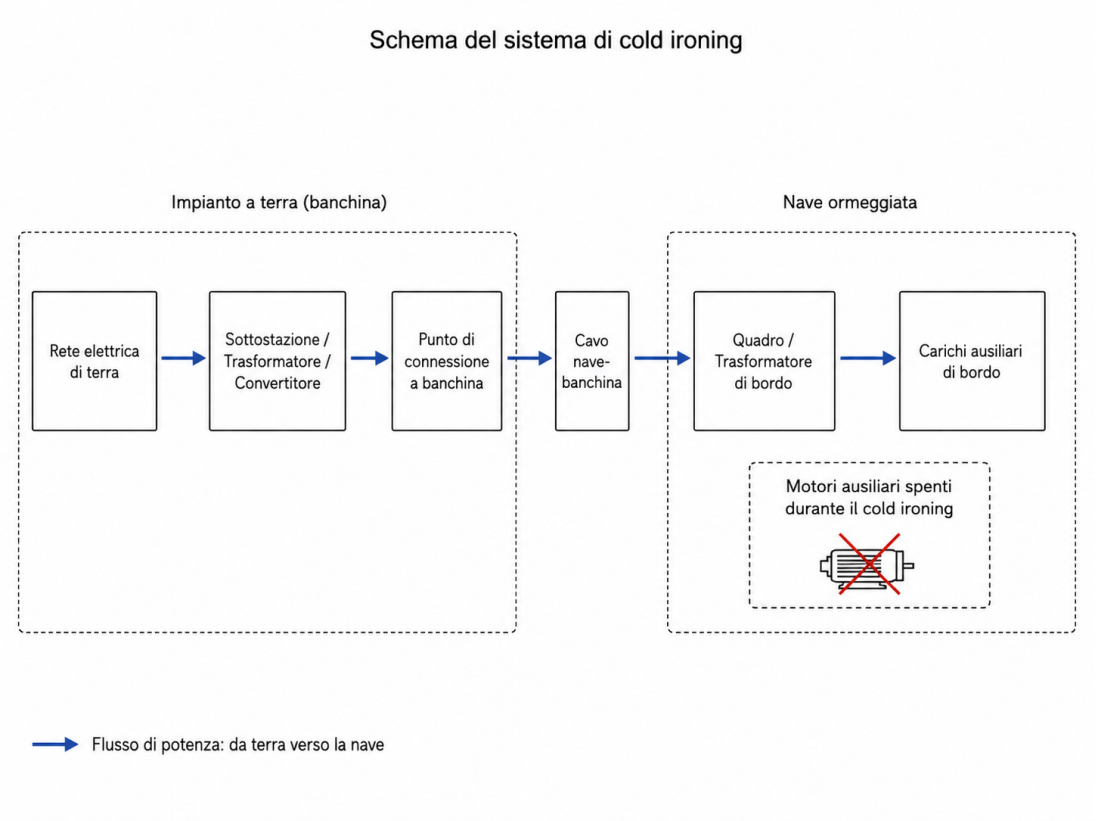


Figura 1.1: Schema semplificato di un sistema di cold ironing.

1.2 High Impedance Fault

Quando un conduttore energizzato entra in contatto con una superficie ad alta resistenza, come asfalto bagnato, calcestruzzo o terreno umido, si può generare un guasto ad alta impedenza (*High Impedance Fault*, HIF). A differenza dei guasti convenzionali, un HIF produce correnti di guasto ridotte e fortemente variabili, spesso di natura intermittente e non lineare, che possono mantenersi su livelli molto vicini a quelli del normale funzionamento di carico. Proprio per questo motivo, i sistemi di protezione tradizionali basati sulla soglia di sovracorrente non sono quasi mai in grado di distinguere un HIF da una normale variazione di carico, lasciando il guasto attivo e non rilevato per periodi prolungati.

Questa situazione è particolarmente critica in un contesto come quello portuale: un HIF non rilevato può degenerare in un arco elettrico sostenuto, con rischio di incendio,

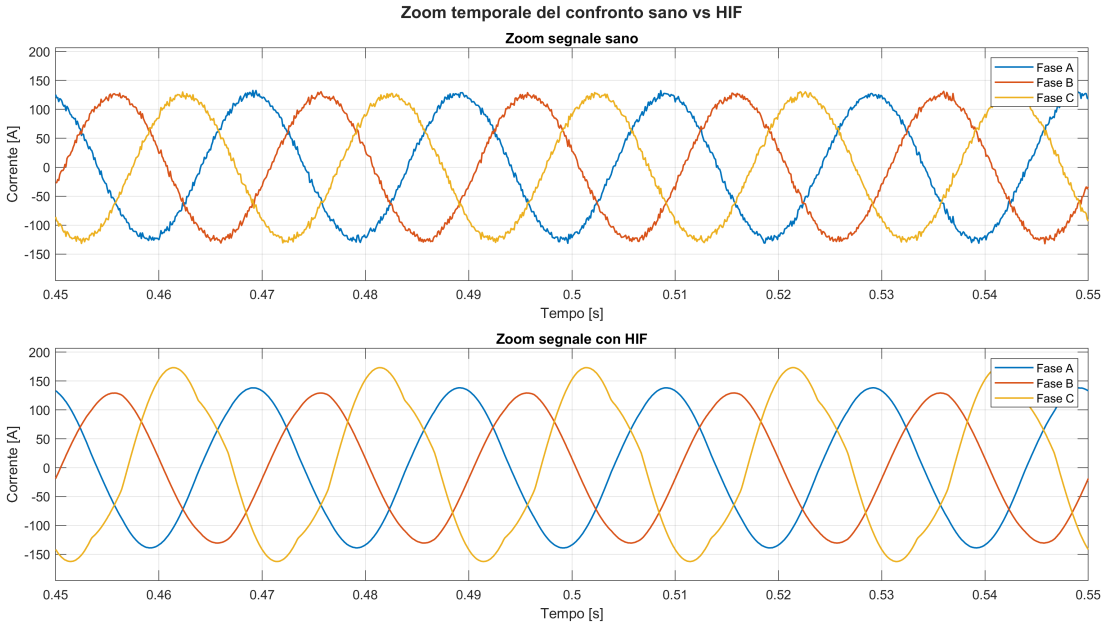


Figura 1.2: Confronto tra le correnti trifase in condizioni di funzionamento sano e in presenza di High Impedance Fault, nella finestra temporale compresa tra 0,45 s e 0,55 s.

danneggiamento dell'impianto e pericolo per il personale e per l'imbarcazione stessa. Lo studio e l'analisi di tali problematiche sono ulteriormente complicate dal fatto che i guasti ad alta impedenza reali sono eventi rari, pericolosi da riprodurre e difficili da acquisire sul campo; ciò rende estremamente limitata la disponibilità di dataset con guasti reali etichettati, ostacolando così lo sviluppo e la validazione di metodi di rilevamento supervisionati, che necessiterebbero di numerosi esempi di guasto per l'addestramento.

1.3 Stato dell'arte sul rilevamento degli HIF

Le protezioni tradizionali usate nelle reti elettriche si basano principalmente su relè di sovracorrente, che intervengono quando la corrente supera una certa soglia. Come descritto nella sezione precedente, questo tipo di protezione non funziona bene con gli HIF, perché la corrente di guasto resta troppo simile a quella normale. Aljohani e Habiballah [1] confermano questo problema nella loro rassegna sugli HIF, spiegando che la corrente di un HIF in media tensione è di solito inferiore a 75 A, un valore

troppo basso per far scattare le protezioni classiche. Gli autori dividono le tecniche di rilevamento proposte in letteratura in quattro gruppi: le protezioni tradizionali appena descritte, i metodi basati sull'analisi del segnale, i metodi matematici e i metodi basati sull'intelligenza artificiale.

Tra le tecniche di analisi del segnale, la più usata è la trasformata wavelet. Lai et al. [2] propongono un metodo che analizza tensione e corrente tramite trasformata wavelet, dopo aver modellato il guasto con lo stesso modello a due diodi. Il loro metodo riesce a distinguere un HIF da un funzionamento normale con un errore molto basso, circa il 2,5%.

Altri lavori usano invece tecniche di intelligenza artificiale supervisionata, cioè addestrate su esempi sia di funzionamento sano sia di guasto. Rameshrao et al. [3] usano una rete neurale di tipo LSTM per rilevare gli HIF nei microgrid, ottenendo buoni risultati anche quando la produzione di energia fotovoltaica varia molto o quando la rete cambia configurazione. Questi metodi però hanno un limite comune: per essere addestrati hanno bisogno di molti dati di guasto reali ed etichettati, cioè segnali di cui si sa già con certezza se rappresentano un guasto oppure no.

Ciò è proprio il problema principale nel contesto del cold ironing. Ding et al. [4] spiegano che l'ambiente portuale è molto difficile da monitorare e che, per questo motivo, mancano dataset di guasti reali con cui addestrare modelli supervisionati. Pertanto, in questo lavoro si è scelto un approccio diverso, che non richiede esempi di guasto per l'addestramento: gli approcci di anomaly detection non supervisionata descritti nella sezione seguente.

1.4 Approcci di anomaly detection

Le tecniche di anomaly detection si dividono in due categorie principali.

Nella prima, chiamata supervisionata, il modello viene addestrato usando sia dati sani sia dati di guasto etichettati. Nella seconda, chiamata non supervisionata, il modello viene addestrato usando solo dati sani: durante il funzionamento il modello segnala come anomalia tutto ciò che si discosta troppo da quanto imparato durante

l'addestramento. Come spiegato nella sezione precedente, per il problema degli HIF nel cold ironing questo secondo approccio è preferibile, perché i dati di guasto reali sono difficili da ottenere.

Una delle tecniche più usate per l'anomaly detection non supervisionata è l'autoencoder. Il funzionamento è semplice: la rete viene addestrata a ricostruire in uscita lo stesso segnale che riceve in ingresso, usando solo segnali sani. Se durante il funzionamento il segnale in ingresso rappresenta un guasto, la rete non riesce a ricostruirlo correttamente, e l'errore tra segnale originale e segnale ricostruito diventa più alto. Questo errore, chiamato errore di ricostruzione, viene quindi usato come indicatore di anomalia.

Thill et al. [5] applicano questo principio usando una rete convoluzionale temporale (TCN), la stessa tipologia di rete usata in questa tesi, al posto delle più comuni reti ricorrenti come le LSTM. Il loro autoencoder TCN ottiene risultati migliori rispetto a un autoencoder basato su LSTM. Il risultato conferma l'efficacia delle architetture TCN per l'anomaly detection di serie temporali.

Un confronto simile viene fatto anche nel lavoro di Xhani et al. [6], svolto sullo stesso problema degli HIF nel cold ironing. Nel loro studio vengono confrontati diversi metodi: un indicatore tradizionale basato sulla tensione (59N), due tecniche di machine learning non supervisionato più semplici (Isolation Forest e One-Class SVM) e due autoencoder, uno basato su LSTM e uno basato su TCN. I risultati mostrano che l'indicatore tradizionale ottiene le prestazioni peggiori, i metodi di machine learning semplici funzionano meglio e l'autoencoder TCN supera nettamente quello basato su LSTM.

Sulla base di questi risultati, in questo lavoro si è scelto di sviluppare un autoencoder basato su rete TCN, addestrato solo su dati di funzionamento sano.

1.5 Obiettivo del lavoro

L'attività di studio e sviluppo svolta riguarda l'implementazione di un metodo di *High Impedance Fault Detection* su piattaforma board STM32 di STMicroelectronics,

tramite un algoritmo di AI non supervisionata in scenari di *cold ironing*.
La soluzione proposta consiste nell'implementazione di un modello di AI non supervisionata per *anomaly detection* su una piattaforma STM32. Trattandosi di un approccio non supervisionato, il modello viene addestrato esclusivamente su dati di funzionamento sano, senza richiedere esempi di guasto.

1.6 Organizzazione della tesi

La tesi è organizzata in sei capitoli. Dopo il presente capitolo introduttivo, dedicato al contesto applicativo, al problema degli High Impedance Fault e allo stato dell'arte relativo alle tecniche di rilevamento, il Capitolo 2 descrive il modello utilizzato e la metodologia adottata, approfondendo l'architettura TCN autoencoder, il dataset e il criterio di anomaly detection.
Il Capitolo 3 è dedicato alla piattaforma hardware STM32 e all'ambiente software e agli strumenti di sviluppo utilizzati.
Il Capitolo 4 descrive il metodo di sviluppo e implementazione del modello sulla board, illustrando la conversione e validazione della rete, i vincoli di memoria incontrati e le operazioni di compressione e quantizzazione adottate per rispettarli, l'implementazione del firmware e il sistema di comunicazione con il PC.
Nel Capitolo 5 vengono presentati e discussi i risultati ottenuti, confrontando le diverse versioni del modello e analizzando sia le prestazioni di classificazione sia i tempi di inferenza sulla piattaforma embedded.
Infine, il Capitolo 6 raccoglie le conclusioni del lavoro e sintetizza i principali risultati e limiti emersi durante l'attività.

Capitolo 2

Modello e metodologia

Il processo di rilevamento dell'anomalia è articolato in una sequenza di operazioni che comprende il pre-processing dei segnali, la suddivisione in frame, la ricostruzione mediante il modello TCN autoencoder e il successivo calcolo dello score di anomalia. Una rappresentazione schematica dell'intera procedura è riportata in Figura 2.1.

2.1 Architettura TCN autoencoder

Per comprendere il funzionamento del modello fornito, ho approfondito le basi teoriche dell'intelligenza artificiale seguendo il libro *Dive into Deep Learning* [7], concentrandomi in particolare sulle reti neurali convoluzionali (CNN, *Convolutional Neural Network*), poiché condividono lo stesso principio di funzionamento delle reti convoluzionali temporali (TCN, *Temporal Convolutional Network*) su cui si basa il modello utilizzato in questo progetto.

Il modello fornitomi è infatti un **autoencoder basato su architettura TCN**, progettato per ricostruire le serie temporali multicanale date in ingresso. La rete è organizzata secondo una struttura encoder-bottleneck-decoder: una prima fase di *encoding* comprime l'informazione fino al raggiungimento di un collo di bottiglia (*bottleneck*), segue una fase di *decoding* che espande nuovamente la rappresentazione fino a ricostruire il segnale originale. La rete viene addestrata esclusivamente su dati sani: in questo modo, quando in fase di inferenza essa riceve in ingresso dati corrotti da

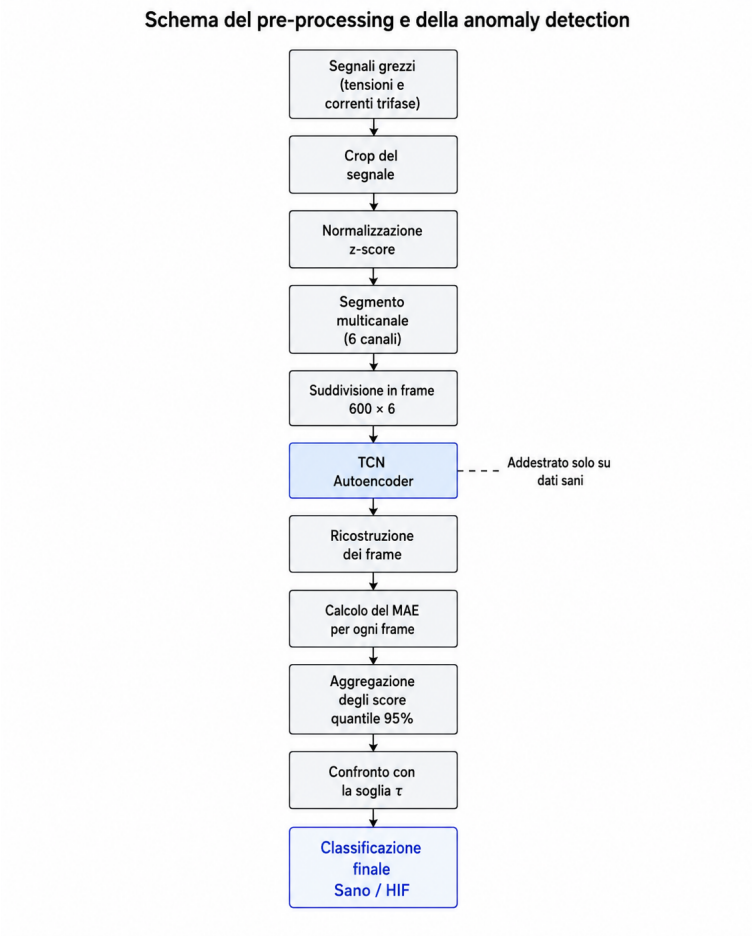


Figura 2.1: Schema generale della procedura di pre-processing e anomaly detection utilizzata

un guasto ad alta impedenza, non è in grado di ricostruirli correttamente e produce, di conseguenza, un errore di ricostruzione più elevato rispetto al caso di funzionamento normale. È proprio questo scarto tra segnale originale e segnale ricostruito a costituire l'indicatore di rilevamento dell'anomalia.

Dal punto di vista architetturale, la rete riceve in ingresso un frame $X^{(k)} \in \mathbb{R}^{N\times 6}$, dove le 6 componenti corrispondono alle tre tensioni e alle tre correnti di fase. Il frame viene innanzitutto elaborato da un blocco convoluzionale 1×1, che porta il numero di canali interni a 64 nella configurazione originale del modello. Come descritto nel Capitolo 3, tale valore è stato successivamente ridotto a 56 durante la fase di adattamento della rete ai vincoli della piattaforma embedded. Successivamente il segnale viene elaborato da una serie di **blocchi convoluzionali residui dilatati**, con fattori di dilatazione pari a 1, 2, 4 e 8, ciascuno seguito da normalizzazione di gruppo (*Group Normalization*) e da una funzione di attivazione non lineare; ad esso si aggiungono due strati di *downsampling*, realizzati tramite convoluzioni con passo (*stride*) 2, che riducono progressivamente la risoluzione temporale del segnale. La struttura del decoder è speculare rispetto a quella dell'encoder: le stesse operazioni vengono eseguite in ordine inverso, sostituendo le convoluzioni di downsampling con convoluzioni trasposte (*transposed convolution*), fino a riportare il segnale alla lunghezza e al numero di canali originali.

L'utilizzo di blocchi residui dilatati consente alla rete di ottenere un **campo ricettivo** (*receptive field*) molto più ampio a parità di profondità della rete stessa, senza per questo aumentare in modo eccessivo il numero di parametri. Questa caratteristica è particolarmente importante nel contesto dei guasti ad alta impedenza, il cui comportamento nel tempo è spesso intermittente e non stazionario: un campo ricettivo ampio permette infatti alla rete di catturare sia le distorsioni localizzate sulla forma d'onda, sia le dipendenze temporali a più lungo termine che caratterizzano l'evoluzione del guasto durante la durata del segmento acquisito.

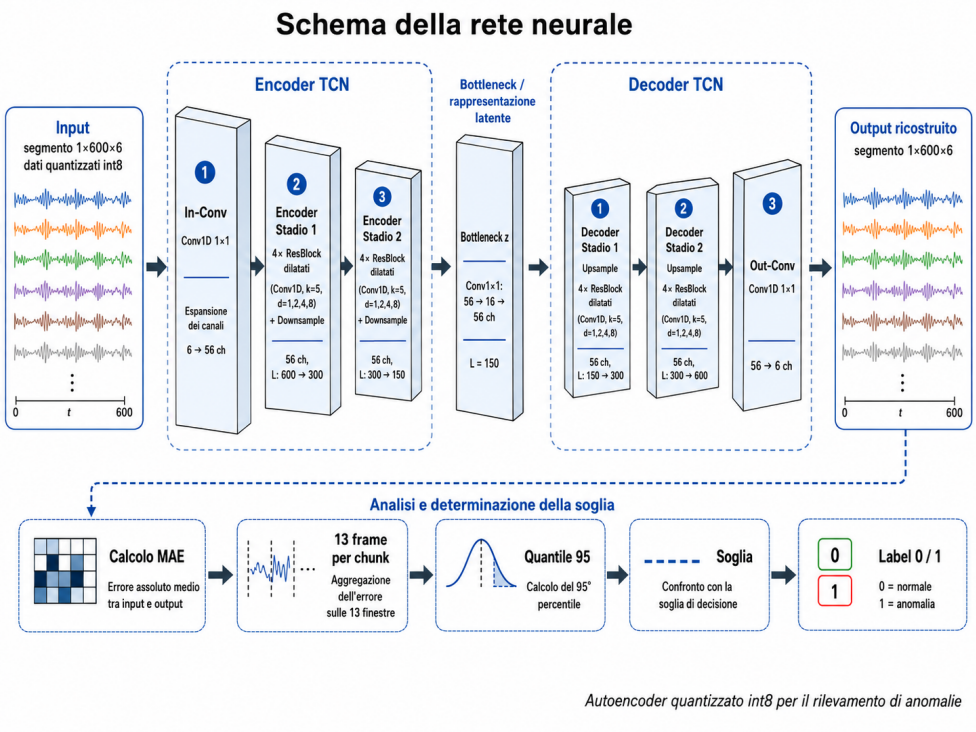


Figura 2.2: Architettura dell'autoencoder TCN nella configurazione finale a 56 hidden channels quantizzata INT8, comprensiva della fase di post-processing per l'ottenimento della label di classe.

2.2 Dataset e criterio di anomaly detection

Per quanto riguarda il dataset utilizzato per la validazione del modello, esso è costituito da 1500 simulazioni complessive: 750 caratterizzate da funzionamento sano e 750 con guasto ad alta impedenza. Ciascuna simulazione ha una durata di 1 secondo, con segnali campionati a una frequenza di 10 kHz; da ciascuna simulazione viene poi estratta un'istanza multicanale di durata pari a 0,8 s, che costituisce il **segmento di forma d'onda** (*waveform segment*) [6].

Considerando la frequenza di campionamento di 10 kHz, il segmento di 0,8 s contiene complessivamente 8000 campioni. Nell'implementazione adottata per il porting vengono utilizzati 7800 campioni, corrispondenti a 13 frame consecutivi di 600 campioni ciascuno; i 200 campioni rimanenti non partecipano quindi alla successiva elaborazione.

Ogni segmento viene a sua volta suddiviso in frame di lunghezza fissa, ciascuno co-

stituito da 600 campioni. Il rilevamento del guasto può quindi avvenire a due livelli differenti: a livello di singolo frame oppure a livello dell'intero segmento. Si definisce così e_k l'errore di ricostruzione calcolato sul singolo frame k-esimo, che viene indicato come *frame-level score*.

Gli errori calcolati sui singoli frame vengono poi aggregati per ottenere un unico punteggio di anomalia riferito all'intero segmento, denominato *segment-level score* s, che viene confrontato con una soglia di decisione τ stabilita in fase di training.

$$s = Q_q\left(\{e_k\}_{k=1}^{T}\right) \tag{2.1}$$

$$e_k = \frac{1}{6L}\sum_{n=1}^{L}\sum_{c=1}^{6}\left|X_{n,c}^{(k)} - \hat{X}_{n,c}^{(k)}\right| \tag{2.2}$$

dove $Q_q(\cdot)$ indica il q-esimo quantile calcolato sull'insieme di valori a cui è applicato (in questo caso l'insieme degli errori e_k dei T frame del segmento), mentre $X_{n,c}^{(k)}$ e $\hat{X}_{n,c}^{(k)}$ rappresentano rispettivamente il valore del segnale originale e il corrispondente valore ricostruito dalla rete, relativi al campione temporale n e al canale c del frame k.

Nel caso considerato, ciascun frame è costituito da $L = 600$ campioni e ogni segmento utilizzato nell'implementazione viene suddiviso in $T = 13$ frame. Come funzione di aggregazione degli errori di ricostruzione è stato adottato il quantile al 95%, pertanto $q = 0.95$.

Capitolo 3

Piattaforma e implementazione su STM32

3.1 Piattaforma hardware

La board su cui è stato portato il modello è la STEVAL-STWINKT1B di STMicroelectronics, un kit di sviluppo pensato per applicazioni IoT industriali, basato sul microcontrollore **STM32L4R9ZIJx**, in package UFBGA144, con core ARM Cortex-M4 dotato di unità floating point (FPU). La board dispone di 2 MB di memoria Flash e circa 640 kB di RAM, valori che si sono rivelati determinanti nella fase di ottimizzazione della rete, poiché il modello nella sua versione originale non entrava in Flash.

Nel progetto ho configurato il clock di sistema partendo da un oscillatore esterno ad alta frequenza (HSE) da 16 MHz, moltiplicato tramite il PLL interno per ottenere una frequenza di funzionamento del core (SYSCLK) di 120 MHz. È stato inoltre mantenuto attivo l'oscillatore interno HSI48, necessario per generare il clock a 48 MHz richiesto dalla periferica USB, come descritto di seguito.

Per programmare e debuggare la scheda ho utilizzato il debugger/programmatore **ST-Link**, collegato alla board tramite interfaccia SWD (Serial Wire Debug). Lo ST-Link permette sia di scaricare il firmware compilato sulla Flash del microcontrollore tramite STM32CubeProgrammer, sia di effettuare debug passo-passo direttamente da

STM32CubeIDE, funzionalità che ho sfruttato soprattutto nelle fasi iniziali di test del firmware e durante l'individuazione del problema di alimentazione della SD, descritto nella sezione 4.7.

3.2 Ambiente software e strumenti di sviluppo

Per la realizzazione del lavoro sono stati necessari alcuni tool e software, descritti di seguito. Per quanto riguarda il modello, mi sono stati forniti: il modello sviluppato in Python, già addestrato precedentemente, tutti gli iperparametri ottenuti dall'addestramento, i pesi, un dataset di test e lo script utilizzato per l'addestramento della rete.
Come IDE ho utilizzato Visual Studio Code, con cui ho scritto e testato tutti gli script Python del progetto: lo script di conversione del modello da PyTorch a ONNX, lo script per l'inferenza sul PC, lo script per la generazione del dataset quantizzato a 8 bit da inviare alla board (`input.py`) e lo script `host_usb_bridge.py` per la comunicazione seriale con la scheda.
Il codice sorgente sviluppato in Python, comprensivo degli script di conversione, inferenza e comunicazione con la board, insieme ai risultati dell'inferenza ottenuti su PC, è disponibile nella seguente repository GitHub [8].
Un secondo strumento utilizzato è stato il tool online Edge Impulse, sfruttato per importare il modello ONNX ed esportarlo in formato TensorFlow Lite, così da poter confrontare il comportamento della rete anche in quest'ultimo formato durante i test di inferenza sul PC, oltre che in ONNX.
Per quanto riguarda la board, ho utilizzato cinque software della ST:

- **STM32CubeMX**: tool che serve per configurare l'hardware della scheda. L'ho utilizzato per generare il progetto STM32CubeIDE della board STEVAL-STWINKT1B, configurando le periferiche necessarie (GPIO per i LED, timer TIM2 per la misura dei tempi di inferenza, periferica USB Device in modalità CDC) e per gestire il file di progetto `.ioc`, da cui viene rigenerato il codice di inizializzazione ogni volta che si modifica la configurazione hardware.

- **STM32CubeIDE**: IDE ufficiale di STMicroelectronics con cui ho sviluppato il firmware, cioè tutta la logica applicativa scritta in `main.c` e nei file di gestione della USB (lettura dei chunk dal PC, esecuzione dell'inferenza frame per frame, calcolo del MAE e del quantile 95, invio dei risultati).

- **STM32CubeAI Studio**: serve per ottimizzare il codice sviluppato e simulare il porting sulla scheda, risultando utile per capire la memoria occupata. In particolare l'ho usato per analizzare l'occupazione di FLASH (pesi) e RAM (buffer di attivazione) prima ancora di scriverlo sulla board, e per generare automaticamente il codice C ottimizzato della rete (i file `network.c`, `network_data.c` e i relativi header), integrato poi nel progetto su STM32CubeIDE.

- **STM32CubeProgrammer**: utilizzato per scrivere la Flash della board, cioè per caricare fisicamente il firmware compilato (file `.elf`) sulla memoria del microcontrollore.

- **Developer Cloud**: utilizzato per ottimizzare il modello. Nello specifico, l'ho usato per quantizzare a 8 bit il modello a partire dai pesi in floating point, fornendo come input di calibrazione la matrice estratta dal dataset di validazione, così da ridurre l'occupazione di Flash rispettando i vincoli di memoria della board.

Infine, per programmare il chip STEVAL-STWINKT1B, ho utilizzato lo ST-Link, il debugger/programmatore integrato nel kit che permette di caricare il firmware sulla scheda e di effettuare debug via SWD direttamente da STM32CubeIDE.
Il progetto firmware completo per STM32CubeIDE, pronto per essere eseguito sulla piattaforma STEVAL-STWINKT1B, è disponibile come release nella stessa repository [9].

Capitolo 4

Metodo di sviluppo e implementazione

4.1 Conversione e validazione del modello

Una volta compreso il funzionamento teorico del modello, il primo passo pratico è stato lo studio approfondito dello script utilizzato per l'addestramento della rete.
Ho quindi realizzato un mio script per la conversione del modello dal formato nativo PyTorch al formato standard **ONNX** (*Open Neural Network Exchange*), passaggio necessario per poter successivamente importare la rete negli strumenti di STMicroelectronics. In seguito ho creato uno script in grado di eseguire delle inferenze direttamente sul PC, così da poter validare il comportamento del modello prima ancora di portarlo sulla board.
A tale scopo ho analizzato tutte le funzioni del codice originale che potevano tornarmi utili e ho scritto lo script riprendendo la stessa logica di pre-elaborazione dei dati già utilizzata nello script di addestramento.
Ho inoltre utilizzato la piattaforma cloud **Edge Impulse** per esportare la rete anche in formato **TensorFlow Lite**, pur non essendo questo lo scopo principale della piattaforma, così da poter confrontare il comportamento del modello convertito in due formati differenti. Ho quindi eseguito l'inferenza sul PC utilizzando sia il modello in formato ONNX sia quello in formato TFLite, con l'obiettivo di verificarne la coerenza

dei risultati prima del porting sulla board.
Tra gli output prodotti dalle inferenze sono stati salvati il frame-level score per ciascun frame elaborato, i tempi necessari all'esecuzione delle singole inferenze e le predizioni ottenute sul dataset di test. Tali risultati vengono analizzati e confrontati nel Capitolo 5.
Durante questa fase di validazione ho osservato che, modificando la modalità di ritaglio (*crop*) del segnale in ingresso, ad esempio impostandola su *center* anziché sulla modalità originariamente prevista (*last*), il modello smette di funzionare correttamente, restituendo punteggi di errore elevati e classificando quindi tutti i segnali come affetti da guasto. Questo comportamento probabilmente è dovuto al fatto che, nei primi campioni del segnale, i valori assunti sono significativamente più elevati rispetto al resto della forma d'onda; poiché la rete è stata addestrata utilizzando sempre la stessa porzione finale del segnale, non ha mai osservato durante il training questi valori iniziali più alti, e di conseguenza non è in grado di ricostruirli correttamente.
Ho inoltre osservato che, riducendo la lunghezza del frame da $L = 800$ a $L = 600$ campioni, le prestazioni del modello peggiorano leggermente: in particolare, un segnale sano viene classificato erroneamente come affetto da guasto (falso positivo) con una frequenza maggiore rispetto al caso in cui si utilizzi $L = 800$. Ciò probabilmente è dovuto al fatto che avendo meno campioni, il risultato è più incerto.

4.2 Prime verifiche sulla piattaforma

Una volta validato il comportamento del modello sul PC, il passo successivo è stato prendere confidenza con la piattaforma hardware e con gli strumenti di sviluppo descritti nel Capitolo 3, scrivendo alcuni firmware dimostrativi: un programma di *blinking LED* per verificare la corretta configurazione della board e un secondo firmware per testare la comunicazione tramite SDMMC.

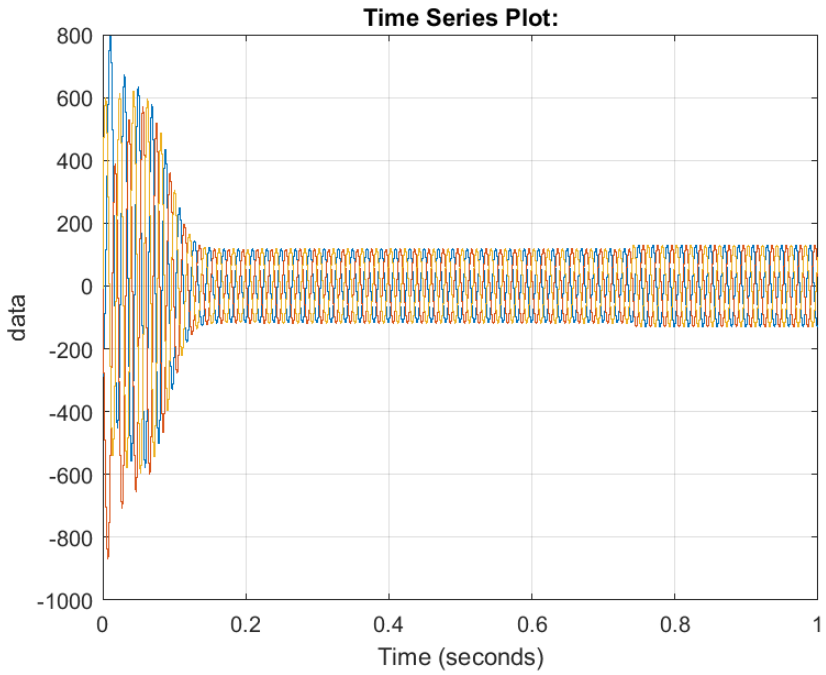


Figura 4.1: Effetto della modalità di crop sul frame score: confronto tra crop standard e crop *center*.

4.3 Generazione degli input per la board

Superata questa fase di validazione software, ho realizzato uno script Python per generare il file di input in formato binario, necessario per poter alimentare il modello direttamente sulla board.

4.4 Vincoli di memoria e ottimizzazione del modello

Analizzando il modello con STM32CubeAI, ho riscontrato che esso occupa circa 2,9 MB per i soli pesi in memoria Flash e circa 450 kB per i buffer di attivazione in RAM, a fronte di una disponibilità sulla board di soli 2 MB di Flash e circa 640 kB di RAM. Il modello, nella sua configurazione originale, risultava quindi troppo grande per essere caricato sulla scheda, rendendo necessarie la compressione e ottimizzazione. Per affrontare questo problema, il primo intervento è stata la compressione tramite la piattaforma Cube AI, testando le tre modalità disponibili: *low*, *mid* e *high*. Poiché questa operazione da sola non era sufficiente a rientrare nei vincoli di memoria, sono

passato alla **quantizzazione a 8 bit** tramite lo strumento STM32 Edge AI Developer Cloud. A seguito della quantizzazione, l'occupazione di Flash si è ridotta a circa 2,17 MB, valore comunque ancora superiore ai 2 MB disponibili sulla board.

Parametro	**Rete originale**	**Rete quantizzata a 56 canali**
Memoria FLASH occupata	$\sim$ 2.9 MB	$\sim$ 1.6 MB
Memoria RAM occupata	$\sim$ 450 kB	$\sim$ 170 kB
Riduzione memoria FLASH	–	$\sim$ 44.8%
Riduzione memoria RAM	–	$\sim$ 62.2%
Rappresentazione dei pesi	Float32	Int8
Numero di MACC	$\sim 336 \times 10^6$	$\sim 264 \times 10^6$
Operazioni eseguite in INT8	0%	29.5%
Operazioni eseguite in floating point	100%	68.2%
Conversioni tra INT8 e floating point	0%	2.3%

Tabella 4.1: Confronto tra la rete originale e la rete quantizzata.

La tabella mostra il confronto tra la rete originale e la configurazione finale ottenuta combinando la riduzione dei canali nascosti da 64 a 56 con la quantizzazione a 8 bit, descritta nel dettaglio più avanti in questa sezione. Si osserva inoltre che il numero di MACC resta invariato tra le due configurazioni, poiché la quantizzazione modifica la rappresentazione numerica dei dati senza alterare il numero di operazioni aritmetiche eseguite dalla rete. Analizzando più nel dettaglio le operazioni effettivamente eseguite dal modello quantizzato, solo il 29,5% avviene in aritmetica INT8, mentre il 68,2% rimane in floating point e il restante 2,3% è costituito da conversioni tra i due formati, dovute principalmente al mantenimento in floating point delle operazioni di normalizzazione.

A questo punto ho valutato, come prima possibile soluzione, l'acquisto di una board dotata di maggiore capacità di Flash o, quantomeno, di una memoria esterna per l'espansione della Flash stessa. Si è tuttavia successivamente optato per un intervento diretto sulla struttura della rete.

Oltre alla quantizzazione, si è reso quindi necessario intervenire anche sulla dimensione

della rete stessa: il primo tentativo è stato ridurre il numero di canali in ingresso, passando dai 6 canali originali a soli 3 o 4 canali. Questa strada si è tuttavia rivelata poco efficace, poiché il primo strato della rete proietta comunque l'ingresso sul numero di canali interni configurato per gli strati nascosti (*hidden channels*) e lavora sempre con quel numero di canali, poi solo alla fine la rete viene riportata al numero di canali originale.

Procedendo in questo modo, il numero di pesi effettivamente risparmiato riducendo i soli canali in ingresso risultava non sufficiente. Ho quindi agito direttamente sul numero di canali nascosti, riducendolo da 64 a 56. Con questa modifica la rete non quantizzata occupava circa 2,2 MB; una volta applicata nuovamente la quantizzazione, l'occupazione di Flash si è ridotta a circa 1,6 MB e quella di RAM a circa 170 kB, valori compatibili con le risorse disponibili sulla board.

I risultati dell'analisi effettuata tramite STM32CubeAI sulle principali configurazioni considerate sono riassunti nella Tabella 4.2. Il confronto evidenzia come, né la sola quantizzazione della rete originale né la sola riduzione del numero di canali nascosti siano sufficienti a rispettare il limite di memoria FLASH della piattaforma. La configurazione implementabile viene ottenuta combinando la riduzione da 64 a 56 canali nascosti con la quantizzazione del modello.

Hidden channels	**Formato**	**FLASH [KB]**	**RAM [KB]**	**Implementabile**
64	Float32	2855	451	No
64	Quantizzato	2161	196	No
56	Float32	2190	395	No
56	Quantizzato	1656	171	Sì

Tabella 4.2: Confronto dell'occupazione di memoria delle principali configurazioni analizzate tramite STM32CubeAI durante il porting del modello.

4.5 Implementazione del firmware

L'ultimo passaggio è stato lo sviluppo del codice necessario ad eseguire l'inferenza direttamente sulla board. Ho scelto di eseguire l'intera fase di pre-elaborazione dei

dati (*pre-processing*) lato PC, in Python: il file di input inviato alla board contiene quindi valori già normalizzati e quantizzati. Questa scelta è dovuta al fatto che il solo firmware occupa più di 1900 kB di Flash, e includendo anche la parte di pre-processing si sarebbe superato il limite di 2 MB disponibile sul microcontrollore.
Il firmware realizzato elabora sequenzialmente le 1500 simulazioni del dataset di validazione. Come descritto in precedenza, ciascuna simulazione è composta da 7800 campioni; poiché ogni frame elaborato dalla rete è costituito da 600 campioni, ogni simulazione viene suddivisa esattamente in 13 frame.
La logica di elaborazione è organizzata in due cicli `for` annidati. Il ciclo più esterno scorre l'indice di simulazione da 1 a 1500 e, per ciascuna iterazione, riceve dal PC un buffer contenente l'intero segmento di input. Essendo i dati quantizzati a 8 bit, la dimensione di tale buffer è pari a:

$$7800 \cdot 6 \text{ B} = 46{,}8 \text{ kB} \tag{4.1}$$

Il ciclo più interno, da 1 a 13, estrae dal buffer un frame alla volta (600 campioni per 6 canali, pari a 3600 B) e ne esegue l'inferenza tramite la rete. Per ciascun frame, l'output della rete — anch'esso una matrice 600×6 — viene dequantizzato e confrontato con l'input dequantizzato per calcolare l'errore assoluto medio (MAE).
Al termine dei 13 frame di una simulazione, i rispettivi valori di MAE vengono aggregati calcolando il quantile 95 e confrontati con la soglia τ: se il punteggio la supera, la simulazione viene classificata come guasto (label 1), altrimenti come condizione sana (label 0).
Score, label predetta e label vera vengono infine raccolti in un file CSV, che consente di ricostruire la *confusion matrix* e di calcolare gli indici di valutazione (precision, recall, F1-score) presentati nei risultati.

4.6 Setup hardware

Inizialmente avevo configurato anche le periferiche SDMMC e il middleware FATFS, poiché l'idea iniziale prevedeva di leggere il dataset dalla microSD e di salvare su di

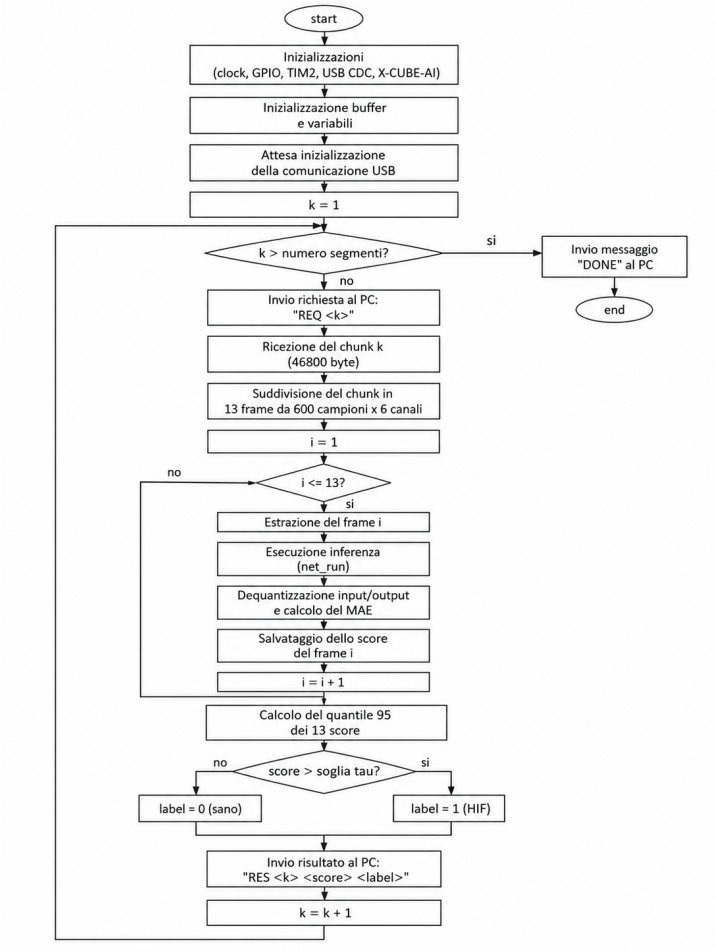


Figura 4.2: Schema a blocchi del firmware implementato sulla piattaforma STM32.

essa anche i risultati prodotti dalla board.
A seguito del problema di alimentazione riscontrato sulla SD (sezione 4.7), ho aggiornato il firmware basandolo interamente sulla comunicazione USB, che serve contemporaneamente ad alimentare la scheda e a scambiare i dati con il PC.
Di conseguenza, nel progetto sono rimaste configurate solo le periferiche effettivamente necessarie: i GPIO, tra cui il pin collegato al LED che segnala l'esecuzione delle inferenze, il timer TIM2, impiegato per misurare i tempi di esecuzione, e la periferica USB Device, configurata in modalità CDC, cioè come porta seriale virtuale. Nel clock è inoltre mantenuto attivo l'oscillatore interno HSI48, necessario al corretto funzionamento della USB: grazie a questa configurazione, quando la board viene collegata al PC, Windows la riconosce come una porta COM, permettendo allo script Python di comunicare con il firmware.

4.7 Problematica relativa alla SD Card

Durante i primi test del firmware è emerso un problema di comunicazione con la scheda SD. Inizialmente ho ipotizzato si trattasse di un errore nella configurazione hardware, ma il problema si è ripresentato anche eseguendo sulla board il firmware realizzato da Luca Marcantonio [10], il che ha permesso di escludere un errore di configurazione legato al mio progetto.
Ho poi provato a sostituire la scheda SD, senza però ottenere risultati diversi. Quindi ho ipotizzato che la causa potesse risiedere nel clock oppure nell'alimentazione; nonostante abbia ridotto la frequenza di clock fino a 400 kHz, il problema è rimasto invariato.
Misurando infine la tensione ai capi dei condensatori C3 e C4, collegati direttamente al pin di alimentazione della SD, ho riscontrato un valore di soli 1,7 V su entrambi, a fronte dei 3,3 V attesi. Ciò evidenzia la presenza di un guasto hardware sulla linea di alimentazione della scheda SD.

4.8 Soluzione alternativa tramite comunicazione USB

Non potendo utilizzare la SD, ho deciso di lasciare i file direttamente sul PC, sfruttando la porta micro-USB della board per lo scambio dei dati. In questo modo la board riceve dal PC solo i byte necessari a eseguire l'inferenza corrente.
Il funzionamento è basato su un semplice scambio a richiesta: la board chiede un chunk, il PC lo legge dal file binario e glielo invia, la board lo elabora e ne restituisce il risultato. Il protocollo, implementato via USB CDC, è il seguente:

```
MCU --> PC : REQ <k>\r\n
PC  --> MCU: 46800 byte del chunk k
MCU --> PC : RES <k> <score> <label>\r\n
MCU --> PC : DONE\r\n
```

Con il comando `REQ <k>` la board segnala di essere pronta a ricevere il segmento `k`; lo script Python legge quindi dal file `dataset_int8.bin` il blocco corrispondente e lo invia alla scheda.
Ricevuto il chunk, il firmware lo suddivide nei 13 frame da 600 campioni e ne esegue l'inferenza secondo la logica descritta nella sezione 4.5. Al termine la board invia al PC la riga `RES <k> <score> <label>` e, una volta processati tutti i segmenti, con l'invio del messaggio `DONE` si segnala la fine dell'elaborazione, permettendo allo script Python di chiudere i file.
Lato PC, lo script `host_usb_bridge.py` gestisce l'intero scambio: apre la porta COM della board, risponde alle richieste `REQ` leggendo il chunk corretto dal file binario e, alla ricezione di ogni riga `RES`, confronta la label predetta con quella vera contenuta in `labels_uint8.bin`, salvando il tutto in un file CSV; i messaggi di log inviati dalla board vengono invece raccolti in un file di report separato. Ho inoltre introdotto un breve ritardo nel firmware subito dopo l'inizializzazione della USB, per far sì che la prima richiesta venga inviata dopo che lo script Python abbia effettivamente aperto la porta COM.
Dai test è emerso che il tempo di esecuzione complessivo è piuttosto elevato a causa della complessità computazionale della rete, rendendo poco pratica l'elaborazione del-

l'intero dataset sulla STM32; un'analisi dettagliata dei tempi di inferenza è riportata nella sezione 5.4.

Capitolo 5

Risultati e discussione

I risultati ottenuti permettono di valutare separatamente tre aspetti principali del lavoro: la correttezza della conversione del modello in formati differenti, l'effetto della quantizzazione sulle prestazioni di classificazione e, infine, la sostenibilità computazionale dell'esecuzione sulla piattaforma STM32.

5.1 Confronto tra i formati ONNX e TensorFlow Lite

La prima fase di confronto ha riguardato la configurazione a 56 hidden channels non quantizzata, eseguita sul PC sia in formato ONNX sia in formato TensorFlow Lite. Sul dataset completo di 1500 simulazioni i due formati hanno prodotto la stessa confusion matrix, costituita da 735 veri negativi, 15 falsi positivi, 47 falsi negativi e 703 veri positivi.

Ne risultano un'accuracy pari al 95,87%, una precision del 97,91%, una sensitivity, o recall, del 93,73%, una specificity del 98,00% e un F1-score del 95,78%.

Il fatto che le versioni ONNX e TensorFlow Lite forniscano sostanzialmente gli stessi score e le stesse predizioni mostra che la conversione di formato non ha alterato in modo significativo il comportamento della configurazione considerata. Le differenze

osservate nei tempi di esecuzione sono pertanto attribuibili principalmente al runtime utilizzato per effettuare l'inferenza.
Sul PC, infatti, il tempo medio misurato è stato di circa 8,644 ms per frame per la configurazione ONNX a 56 hidden channels Float32 e di circa 14,762 ms per la corrispondente configurazione TensorFlow Lite. In questo specifico caso ONNX Runtime risulta quindi più efficiente del runtime TensorFlow Lite sulla CPU utilizzata.

5.2 Effetto della quantizzazione

La quantizzazione a 8 bit è risultata necessaria principalmente per soddisfare i vincoli di memoria della board. La rete originale, infatti, non poteva essere caricata nella Flash disponibile, mentre la riduzione del numero di canali nascosti da 64 a 56, seguita dalla quantizzazione, ha permesso di ridurre l'occupazione fino a circa 1,6 MB di Flash e circa 170 kB di RAM.
La quantizzazione ha però modificato il comportamento numerico della rete. Sul dataset completo di 1500 simulazioni, il modello ONNX quantizzato ha prodotto 750 veri negativi, nessun falso positivo, 244 falsi negativi e 506 veri positivi. Le principali metriche ottenute sono riassunte nella Tabella 5.1.

Metrica	**ONNX 56 F32**	**ONNX 56 INT8**	**TFLite 56 F32**	**STM32 56 INT8**
Accuracy	95,87%	83,73%	95,87%	80,00%
Precision	97,91%	100,00%	97,91%	80,00%
Sensitivity / Recall	93,73%	67,47%	93,73%	80,00%
Specificity	98,00%	100,00%	98,00%	80,00%
F1-score	95,78%	80,57%	95,78%	80,00%

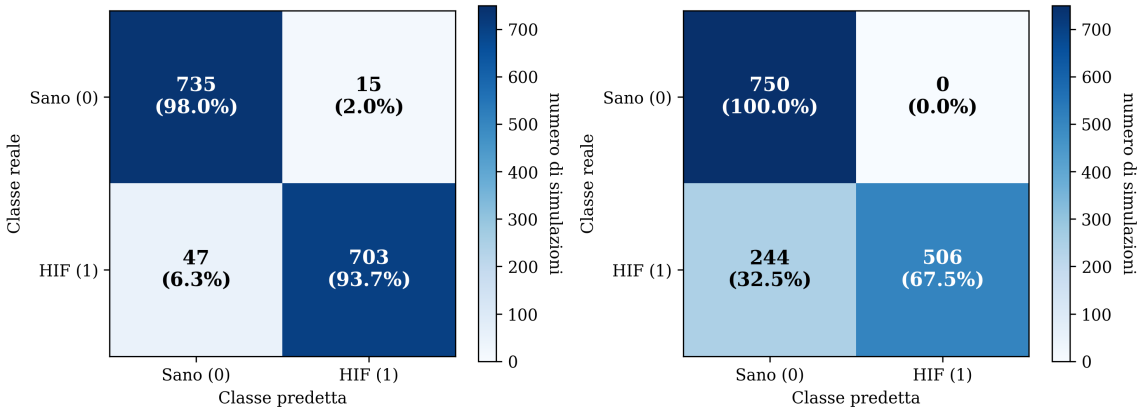
Tabella 5.1: Confronto delle principali metriche di classificazione. I risultati sulla STM32 sono riferiti ai 10 segmenti effettivamente elaborati sulla board.

È importante osservare che l'aumento della precision e della specificity al 100% nel modello quantizzato non corrisponde a un miglioramento complessivo della rete. Tali valori derivano infatti dall'assenza di falsi positivi: il modello quantizzato non classifica nessun campione sano come guasto. Allo stesso tempo, però, il numero di falsi

negativi aumenta da 47 a 244. Di conseguenza, la sensitivity diminuisce dal 93,73% al 67,47%, indicando che una parte significativa dei guasti reali non viene più rilevata. Il modello quantizzato risulta quindi più conservativo nella classificazione del guasto: tende ad assegnare più frequentemente la classe sana, riducendo i falsi allarmi ma aumentando il numero di HIF non riconosciuti. In un'applicazione di rilevamento dei guasti questo aspetto è particolarmente rilevante, poiché un falso negativo rappresenta un guasto reale che il sistema non è riuscito a individuare.

Per visualizzare direttamente l'effetto della quantizzazione sulle prestazioni di classificazione, il confronto è stato effettuato mantenendo invariata l'architettura a 56 hidden channels. In questo modo le differenze osservate possono essere attribuite alla sola quantizzazione del modello.

La configurazione Float32 produce 735 veri negativi, 15 falsi positivi, 47 falsi negativi e 703 veri positivi, mentre la versione quantizzata INT8 produce 750 veri negativi, nessun falso positivo, 244 falsi negativi e 506 veri positivi. Le relative confusion matrix sono riportate in Figura 5.1.



(a) 56 hidden channels, Float32.

(b) 56 hidden channels, INT8.

Figura 5.1: Confronto tra le confusion matrix della configurazione a 56 hidden channels prima e dopo la quantizzazione INT8.

Per valutare quantitativamente l'effetto delle modifiche apportate all'architettura originale, è stato effettuato un confronto tra gli errori di ricostruzione prodotti dalle di-

verse configurazioni del modello. Come riferimento è stata utilizzata la rete originale con 64 hidden channels e pesi in formato Float32.

Per ciascun frame è stata calcolata la differenza tra l'errore di ricostruzione prodotto dal modello considerato e quello ottenuto dalla rete originale:

$$\Delta_k = \left| e_k^{\text{modello}} - e_k^{64F} \right|, \tag{5.1}$$

dove e_k^{64F} rappresenta l'errore di ricostruzione del k-esimo frame ottenuto con il modello originale a 64 canali nascosti in formato Float32.

A partire da tali differenze sono stati calcolati il valore medio dell'errore assoluto (MAE), l'errore quadratico medio (MSE), l'errore massimo e la deviazione standard. I risultati sono riportati in Figura 5.2.

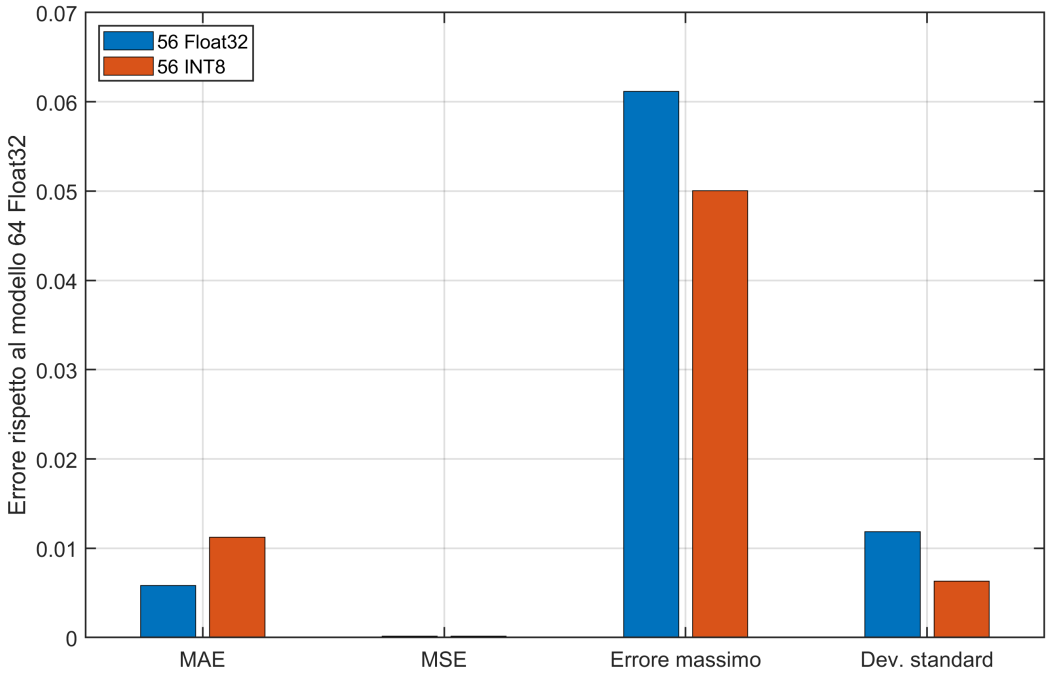


Figura 5.2: Confronto delle metriche di errore dei modelli a 56 hidden channels rispetto al modello originale a 64 hidden channels in formato Float32.

La distribuzione degli anomaly score ottenuti sul dataset di test è mostrata in Figura 5.3. La soglia di decisione τ consente di separare gran parte dei campioni sani da quelli caratterizzati da HIF; sono tuttavia presenti alcuni campioni HIF con

score inferiore alla soglia, corrispondenti ai falsi negativi osservati nelle metriche di classificazione.

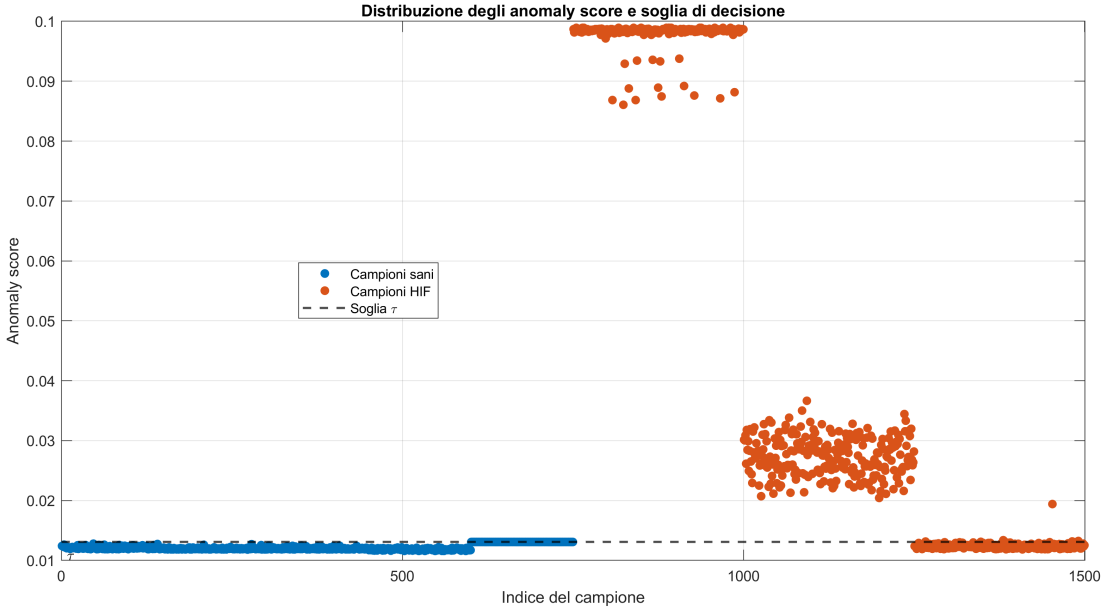


Figura 5.3: Distribuzione degli anomaly score del modello quantizzato sul dataset di test, con indicazione della soglia di decisione τ. I segmenti sani e quelli caratterizzati da High Impedance Fault sono rappresentati separatamente.

L'analisi del codice generato da STM32CubeAI ha inoltre mostrato che il modello ottenuto non è completamente eseguito in aritmetica INT8. Circa il 29,5% delle operazioni risulta eseguito in INT8, mentre circa il 68,2% rimane in floating point; la parte restante, pari a circa il 2,3%, è costituita da operazioni di conversione tra INT8 e floating point. In particolare, le operazioni di normalizzazione vengono mantenute in floating point e richiedono quindi conversioni intermedie dei tensori. Questo limita il vantaggio prestazionale che ci si potrebbe attendere dalla quantizzazione.

Tale comportamento è visibile anche nei tempi misurati sul PC: il modello ONNX quantizzato a 56 hidden channels richiede mediamente circa 9,435 ms per frame, contro gli 8,644 ms della corrispondente configurazione Float32, con un incremento di circa il 9,15%. La quantizzazione ha quindi prodotto un vantaggio fondamentale in termini di memoria, rendendo possibile il porting sulla board, ma non ha comportato un miglioramento del tempo di inferenza sul PC.

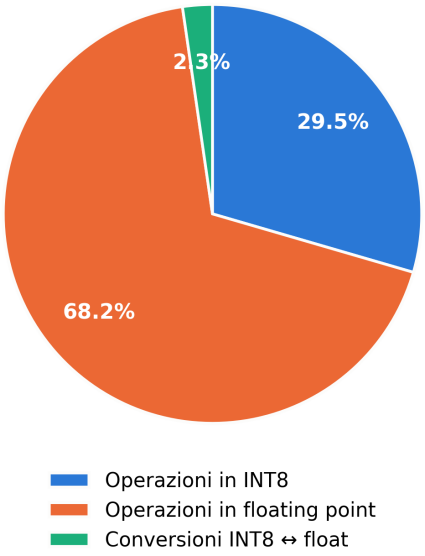


Figura 5.4: Ripartizione delle operazioni eseguite dal modello quantizzato tra aritmetica INT8, floating point e conversioni tra i due formati.

5.3 Risultati ottenuti su STM32

A causa dell'elevato tempo richiesto per l'esecuzione del modello sulla STM32, non è stato possibile completare l'inferenza sull'intero dataset di 1500 simulazioni in tempi compatibili con una sperimentazione pratica, come mostrato dalla stima riportata nella sezione 5.4.

È stato quindi eseguito un test su un sottoinsieme di 10 file, costituito da 5 campioni sani e 5 campioni con HIF.

Sui suddetti 10 file sono stati ottenuti 4 veri negativi, 1 falso positivo, 1 falso negativo e 4 veri positivi, corrispondenti a un'accuracy, precision, sensitivity, specificity e F1-score tutte pari all'80%. Questi risultati mostrano che l'intera catena di elaborazione implementata sulla board è funzionante e che il modello è effettivamente in grado di distinguere campioni sani e campioni affetti da guasto anche dopo il porting.

È tuttavia necessario sottolineare che le metriche ottenute sulla board non possono essere confrontate direttamente, dal punto di vista statistico, con quelle ricavate dalle 1500 simulazioni elaborate sul PC. Un campione di soli 10 elementi è infatti troppo ridotto perché una singola classificazione errata non abbia un peso molto elevato sul

risultato finale: in questo caso, ciascun errore modifica l'accuracy di 10 punti percentuali. Il valore dell'80% deve quindi essere considerato come una verifica funzionale del porting e non come una stima definitiva delle prestazioni del modello sulla board. Per analizzare più nel dettaglio il comportamento del modello sulla piattaforma embedded, in Figura 5.5 sono riportati gli anomaly score ottenuti per i 10 segmenti elaborati sulla STM32, insieme alla soglia di decisione τ.

Si osserva come i due campioni classificati erroneamente presentino score prossimi alla soglia, mentre gli HIF correttamente individuati raggiungono valori sensibilmente superiori.

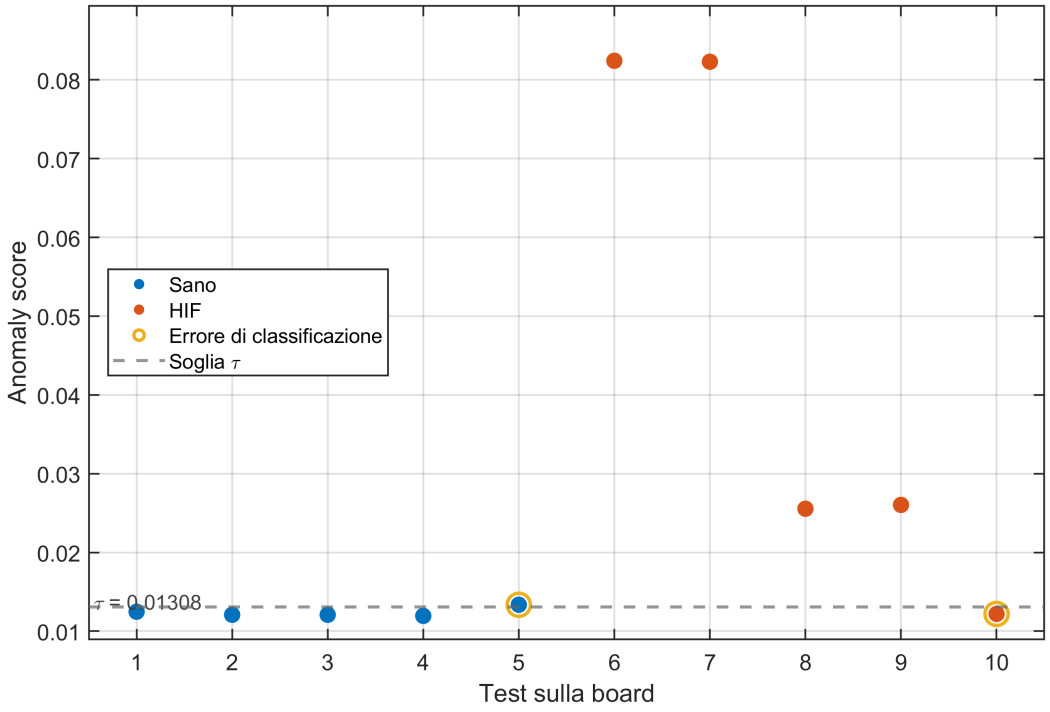


Figura 5.5: Anomaly score ottenuti per i 10 segmenti elaborati sulla piattaforma STM32 e confronto con la soglia di decisione τ.

Il campione sano classificato erroneamente come guasto presenta uno score pari a 0,013358, mentre il campione HIF non rilevato presenta uno score pari a 0,012194. Al contrario, diversi HIF correttamente riconosciuti producono score sensibilmente più elevati, compresi tra 0,025 e 0,082 circa. Questo comportamento è coerente con la presenza di alcuni casi più difficili da separare e con la perdita di sensibilità già osservata sul modello quantizzato durante i test sul PC.

Per verificare più nel dettaglio la corrispondenza tra l'esecuzione software e quella embedded, i medesimi 10 segmenti elaborati sulla STM32 sono stati processati anche su PC utilizzando il modello ONNX a 56 canali nascosti quantizzato. I risultati sono confrontati nella Tabella 5.2.

Test	**Campione**	**Classe reale**	**Score PC**	**Score STM32**	**$\lvert\Delta s\rvert$**	**Pred. PC**	**Pred. STM32**
1	`ls1_817`	Sano	0.012153	0.012472	0.000319	Sano	Sano
2	`ls2_799`	Sano	0.011994	0.012086	0.000092	Sano	Sano
3	`ls3_765`	Sano	0.011924	0.012085	0.000161	Sano	Sano
4	`ls4_766`	Sano	0.011732	0.011945	0.000213	Sano	Sano
5	`normal_714`	Sano	0.013081	0.013358	0.000277	Sano	HIF
6	`rnconcrete_722`	HIF	0.098609	0.082412	0.016197	HIF	HIF
7	`rnconcrete_914`	HIF	0.098609	0.082293	0.016316	HIF	HIF
8	`wetconcrete_724`	HIF	0.025841	0.025549	0.000292	HIF	HIF
9	`wetconcrete_751`	HIF	0.025962	0.026035	0.000073	HIF	HIF
10	`wetsand_244`	HIF	0.011999	0.012194	0.000195	Sano	Sano

Tabella 5.2: Confronto degli anomaly score e delle predizioni ottenuti sul PC e sulla piattaforma STM32 per i medesimi 10 segmenti di test.

Le predizioni ottenute nelle due piattaforme coincidono in 9 casi su 10. L'unica differenza si verifica per il campione `normal_714`, il cui score risulta estremamente prossimo alla soglia di decisione: sul PC il valore è pari a 0.013081, mentre sulla STM32 aumenta a 0.013358, determinando il passaggio dalla classe sana alla classe HIF.

Il falso negativo relativo al campione `wetsand_244`, invece, è presente in entrambe le esecuzioni.

5.4 Analisi dei tempi di inferenza sulla STM32

Il principale limite emerso durante il porting riguarda il tempo di esecuzione. Dalle 130 inferenze misurate, corrispondenti ai 13 frame di ciascuno dei 10 file analizzati, è stato ottenuto un tempo medio pari a circa 22,973 s per singolo frame. La variabilità

tra le diverse misure è molto ridotta, a conferma del fatto che il tempo è determinato principalmente dal costo computazionale della rete e non dalla comunicazione USB.

Configurazione	**Piattaforma**	**Tempo medio per frame**
ONNX 56 hidden channels Float32	PC	8,644 ms
ONNX 56 hidden channels INT8	PC	9,435 ms
TensorFlow Lite 56 hidden channels Float32	PC	14,762 ms
56 hidden channels INT8	STM32	22,973 s

Tabella 5.3: Confronto dei tempi medi di inferenza per singolo frame nelle diverse configurazioni considerate.

Poiché ogni segmento richiede 13 inferenze, il tempo necessario per elaborare un file completo è pari a circa:

$$22,973 \cdot 13 \simeq 298,6 \text{ s} \simeq 4,98 \text{ min.} \tag{5.2}$$

Estendendo tale valore all'intero dataset di 1500 simulazioni si ottiene una durata teorica complessiva pari a circa:

$$4,98 \cdot 1500 \simeq 7470 \text{ min} \simeq 124,5 \text{ h} \simeq 5,18 \text{ giorni.} \tag{5.3}$$

Il confronto complessivo dei tempi medi è riportato nella Tabella 5.3.
Per fornire un ulteriore termine di paragone, è utile calcolare il rapporto tra il tempo di elaborazione di un singolo frame e la sua durata reale. Il segnale è campionato a 10 kHz e ciascun segmento, della durata di 0,78 s, è suddiviso in 13 frame; la durata di un singolo frame risulta quindi pari a:

$$0,78/13 \simeq 0,06 \text{ s} = 60 \text{ ms.} \tag{5.4}$$

Rapportando il tempo medio di elaborazione di una singola inferenza sulla STM32 (22,973 s) alle 13 inferenze necessarie per l'elaborazione di un intero segmento, si ottiene:

$$22,973/13 \simeq 1,77 \text{ s.} \tag{5.5}$$

Questo valore rappresenta la quota media di tempo di elaborazione imputabile a ciascuna delle 13 inferenze rispetto al tempo complessivo richiesto per processare

un intero segmento, ed evidenzia come il carico computazionale della rete risulti comunque eccessivo per un'elaborazione in tempo reale del segnale sulla piattaforma STM32.

Il valore misurato sulla board è spiegabile considerando la notevole complessità della rete, pari a circa 264 milioni di operazioni per ciascuna inferenza, eseguite su un microcontrollore Cortex-M4 a 120 MHz. A questo si aggiunge il fatto che il modello non è completamente INT8 ma contiene numerose elaborazioni in floating point e conversioni tra i due formati numerici. La misura temporale viene effettuata direttamente attorno alla chiamata di esecuzione della rete; pertanto, il periodo di circa 23 s non include il tempo necessario alla trasmissione dei dati tramite USB, ma rappresenta principalmente il tempo occorrente alla rete per completare l'inferenza.

Il porting può quindi essere considerato riuscito dal punto di vista funzionale e dei vincoli di memoria, ma l'architettura attuale non risulta adatta a un'elaborazione rapida o in tempo reale sulla piattaforma STM32L4 utilizzata. Per un'eventuale prosecuzione del lavoro sarebbe opportuno intervenire sull'architettura del modello, riducendone ulteriormente la complessità e privilegiando operatori compatibili con un'esecuzione completamente INT8.

5.5 Sintesi dei risultati

I principali risultati ottenuti nelle diverse fasi di sviluppo sono riassunti nella Tabella 5.4. Il confronto permette di evidenziare il compromesso introdotto dalle modifiche necessarie per rendere il modello compatibile con le risorse della piattaforma embedded, sia in termini di capacità di classificazione sia di occupazione di memoria e tempo di esecuzione.

La riduzione del numero di canali nascosti da 64 a 56 comporta una diminuzione della complessità computazionale da circa 336,05 a 259,37 milioni di MACC per inferenza, corrispondente a una riduzione di circa il 22,8%. La successiva quantizzazione non riduce ulteriormente il numero di MACC, anzi lo fa leggermente aumentare, da circa 259,37 a circa 264,59 milioni (+5,22 milioni), a causa delle operazioni aggiuntive

Configurazione	Pesi	MACC	Test	Accuracy	Precision	Recall	F1-score	Tempo/frame	FLASH req.	RAM req.
64 ch., Float32 – PC	Float32	336.05 M	1500	98.40%	96.90%	100.00%	98.43%	25.571 ms	2855 KB	451 KB
56 ch., Float32 – PC	Float32	259.37 M	1500	95.87%	97.91%	93.73%	95.78%	8.644 ms	2190 KB	395 KB
56 ch., quantizzato – PC	INT8	264.59 M	1500	83.73%	100.00%	67.47%	80.57%	9.435 ms	1656 KB	171 KB
56 ch., quantizzato – STM32	INT8	264.59 M	10	80.00%	80.00%	80.00%	80.00%	22.973 s	1656 KB	171 KB

Tabella 5.4: Sintesi delle prestazioni di classificazione, della complessità computazionale, dei tempi di inferenza e dei requisiti di memoria delle principali configurazioni considerate.

di conversione tra rappresentazione INT8 e floating point introdotte dal processo di quantizzazione. Quest'ultima consente comunque una significativa diminuzione dell'occupazione di memoria e l'impiego di rappresentazioni numeriche a precisione ridotta.

I risultati ottenuti si confermano coerenti con quanto discusso nel Capitolo 1. Il modello a 56 canali, con un'accuracy del 95,87% e una sensitivity del 93,73% sul dataset di 1500 simulazioni, si avvicina alle prestazioni riportate per approcci supervisionati come quello di Rameshrao et al. [3], pur non avendo mai visto esempi di guasto durante l'addestramento. Anche il confronto con il lavoro di Xhani et al. [6], che valuta lo stesso tipo di autoencoder TCN sullo stesso problema, conferma che l'architettura scelta è tra le più efficaci fra quelle non supervisionate testate, superando sia l'indicatore tradizionale 59N sia l'autoencoder basato su LSTM.

La perdita di prestazioni osservata dopo la quantizzazione, con la sensitivity che scende dal 93,73% al 67,47%, mostra che il compromesso tra risorse disponibili e accuratezza del modello, discusso in generale nella sezione 1.3 a proposito delle diverse tecniche di rilevamento, si presenta anche quando si opera sullo stesso modello riducendone la complessità per adattarlo a una piattaforma embedded.

Capitolo 6

Conclusioni

L'attività di tirocinio ha permesso di realizzare e validare l'intera catena necessaria al porting di un modello di anomaly detection per il rilevamento di High Impedance Fault su una piattaforma embedded STM32.

Il lavoro è partito dall'analisi del modello TCN autoencoder sviluppato in PyTorch e dalla comprensione della logica di rilevamento basata sull'errore di ricostruzione. Successivamente il modello è stato convertito in formato ONNX e validato sul PC, confrontandone inoltre il comportamento con la versione TensorFlow Lite. I risultati hanno mostrato che la conversione ONNX–TFLite conserva sostanzialmente inalterate le capacità di classificazione del modello originale, che sul dataset di 1500 simulazioni raggiunge un'accuracy del 95,87% e una sensitivity del 93,73%.

La principale difficoltà del porting è stata rappresentata dai vincoli di memoria della STEVAL-STWINKT1B. La riduzione dei canali nascosti e la successiva quantizzazione a 8 bit hanno permesso di portare l'occupazione del modello entro i limiti della flash e della RAM disponibili, rendendo possibile l'esecuzione effettiva della rete sulla board. Tale risultato rappresenta uno degli obiettivi principali raggiunti durante il lavoro.

La quantizzazione ha tuttavia evidenziato un compromesso tra occupazione di memoria e prestazioni. Pur consentendo il deployment sulla STM32, il modello quantizzato presenta una riduzione dell'accuracy all'83,73% sul dataset completo e, soprattutto, una diminuzione della sensitivity al 67,47%. Inoltre, la rete generata non viene

eseguita completamente in INT8 e conserva una quota significativa di elaborazioni in floating point, limitando il beneficio della quantizzazione dal punto di vista della velocità.

Il test finale sulla board, effettuato su 10 file a causa dell'elevato tempo di elaborazione, ha confermato il corretto funzionamento dell'intero sistema: trasferimento dei dati tramite USB CDC, suddivisione dei segmenti in frame, inferenza, ricostruzione, calcolo del MAE, aggregazione tramite quantile e classificazione finale. Sul sottoinsieme analizzato è stata ottenuta un'accuracy dell'80%; tale valore deve tuttavia essere interpretato con cautela a causa del numero molto ridotto di campioni.

Il principale limite rimasto è quindi di natura computazionale. Un tempo medio di circa 22,97 s per frame porta a quasi 5 minuti per segmento e a una stima di oltre 5 giorni per l'elaborazione dell'intero dataset. La soluzione sviluppata dimostra dunque la fattibilità tecnica del porting del modello su microcontrollore, ma evidenzia anche la necessità di una successiva ottimizzazione dell'architettura qualora si voglia arrivare a un'applicazione realmente utilizzabile sul campo.

In conclusione, il tirocinio ha consentito di completare con successo tutte le principali fasi di una procedura di deployment di una rete neurale su sistema embedded: conversione, validazione software, analisi dei vincoli di memoria, ottimizzazione, quantizzazione, generazione del codice C, integrazione firmware e verifica sperimentale sulla board. I risultati ottenuti forniscono inoltre indicazioni precise sui limiti dell'implementazione attuale e individuano chiaramente gli aspetti su cui intervenire per rendere il sistema più rapido ed efficiente in sviluppi futuri.

Bibliografia

[1] A. Aljohani and I. Habiballah, "High-impedance fault diagnosis: A review," *Energies*, vol. 13, no. 23, p. 6447, 2020.

[2] T. M. Lai, L. A. Snider, and E. Lo, "Wavelet transform based relay algorithm for the detection of stochastic high impedance faults," *Electric Power Systems Research*, vol. 76, no. 8, pp. 626–633, 2006.

[3] A. G. Rameshrao, E. Koley, and S. Ghosh, "A lstm-based approach for detection of high impedance faults in hybrid microgrid with immunity against weather intermittency and n-1 contingency," *Renewable Energy*, vol. 198, pp. 75–90, 2022.

[4] K. Ding, C. Yao, Y. Li, Q. Hao, Y. Lv, and Z. Huang, "A review on fault diagnosis technology of key components in cold ironing system," *Sustainability*, vol. 14, no. 10, p. 6197, 2022.

[5] M. Thill, W. Konen, H. Wang, and T. Bäck, "Temporal convolutional autoencoder for unsupervised anomaly detection in time series," *Applied Soft Computing*, vol. 112, p. 107751, 2021.

[6] E. Xhani, M. A. Khan, G. Tanoni, G. Comodi, and E. Principi, "Unsupervised high-impedance fault detection in a cold ironing microgrid," in *SEST*, 2026.

[7] A. Zhang, Z. C. Lipton, M. Li, and A. J. Smola, *Dive into Deep Learning*. Cambridge University Press, 2023, disponibile anche online su https://d2l.ai.

[8] D. Lombi, “repository,” https://github.com/diegolombii/repository, 2026, repository contenente gli script Python sviluppati e i risultati dell’inferenza su PC.

[9] ——, “repository, release v1.0 – firmware stm32,” https://github.com/diegolombii/repository/releases/tag/v1.0, 2026, release contenente il progetto firmware STM32CubeIDE per la piattaforma STEVAL-STWINKT1B.

[10] L. Marcantonio, “Firmware di test per la piattaforma steval-stwinkt1b,” https://github.com/Luca-m3/repository, 2024.

Ringraziamenti

Desidero innanzitutto ringraziare il Prof. Emanuele Principi, il Prof. Stefano Squartini e la Dott.ssa Giulia Tanoni per avermi dato la possibilità di svolgere questo lavoro di tesi, per la disponibilità dimostrata e per il supporto fornito durante la sua realizzazione.

Ringrazio la mia famiglia e le persone a me care per la vicinanza e il sostegno che mi hanno dimostrato nel corso di questi anni.

Ringrazio inoltre i miei amici per tutti i momenti di svago, e non solo, condivisi durante questi anni.

Un ringraziamento speciale va ai miei colleghi, che oggi posso chiamare amici, senza i quali sarebbe stato impensabile affrontare questi anni e con i quali ho condiviso momenti memorabili.

Ringrazio infine tutte le persone che, in modi diversi, hanno fatto parte di questi anni e hanno contribuito a rendere questo percorso unico e irripetibile.