Università Politecnica delle Marche

Facoltà di Ingegneria

Corso di Laurea Triennale in Ingegneria Elettronica

Studio e sviluppo delle tecniche di scalabilità su dispositivo STM32 per algoritmi di monitoraggio non intrusivo del carico basati su deep learning

Study and Development of Scalability Techniques on STM32 Devices for Non-Intrusive Load Monitoring Algorithms Based on Deep Learning

Candidato:
Luca Marcantonio

Relatore:
Prof. Emanuele Principi

Correlatori:
Prof. Stefano Squartini

Dott.ssa Giulia Tanoni

Anno Accademico 2023/2024

Università Politecnica delle Marche

Facoltà di Ingegneria

Corso di Laurea Triennale in Ingegneria Elettronica

Studio e sviluppo delle tecniche di scalabilità su dispositivo STM32 per algoritmi di monitoraggio non intrusivo del carico basati su deep learning

Study and Development of Scalability Techniques on STM32 Devices for Non-Intrusive Load Monitoring Algorithms Based on Deep Learning

Candidato:
Luca Marcantonio

Relatore:
Prof. Emanuele Principi

Correlatori:
Prof. Stefano Squartini

Dott.ssa Giulia Tanoni

Anno Accademico 2023/2024

Università Politecnica delle Marche
Facoltà di Ingegneria
Corso di Laurea Triennale in Ingegneria Elettronica
Via Brecce Bianche – 60131 Ancona (AN), Italy

Ringraziamenti

Per prima cosa desidero ringraziare il professor Emauele Principi e la dottoressa Giulia Tanoni per la loro pazienza e disponibilità, per i loro consigli dati durante lo svolgimento del progetto, nonchè al professor Stefano Squartini per avermi dato questa opportunità ed avermi fatto appassionare con le sue lezioni.

Un ringraziamento speciale va ai miei familiari, che mi hanno dato l'opportunità e la possibilità di intraprendere questo percorso. A loro, ai nonni, agli zii ed Ilaria, che con il loro amore, la loro pazienza ed il loro sostegno mi hanno accompagnato, dimostrandosi un punto di riferimento fondamentale.

Desidero inoltre ringraziare tutti i miei amici, i colleghi e le persone che mi hanno supportato e che mi sono state vicine in questo percorso.

Ancona, Novembre 2024

Luca Marcantonio

Sommario

Il crescente fabbisogno energetico richiede strumenti efficaci per il monitoraggio e l'ottimizzazione dei consumi. Nella seguente tesi si esplora il porting di algoritmi di **Non-Intrusive Load Monitoring (NILM)** basati su reti neurali profonde su dispositivo embedded, focalizzandosi su un microcontrollore STM32.

L'implementazione sfrutta strumenti software specifici, come **STMcubeMX** e **X-cube-AI** per ottimizzare le prestazioni della rete neurale sul dispositivo. Sono state testate varie strategie di ottimizzazione, tra cui compressione e quantizzazione, confrontando le prestazioni ottenute su piattaforma embedded con quelle ottenute testando il modello su PC sfruttando **Python** e le sue librerie di machine learning.

I risultati dimostrano che, nonostante le limitazioni hardware imposte dal microcontrollore, le tecniche proposte consentono di ottenere un equilibrio tra precisione ed efficienza computazionale. L'analisi conclude con valutazioni per la possibile implementazione di modelli multi-appliance, aprendo la strada a sistemi di monitoraggio energetico in ambito domestico su dispositivi a basso consumo.

Indice

1 Introduzione **1**

2 Non Intrusive Load Monitoring **3**
- 2.1 Desrizione del problema 3
- 2.2 Metodo utilizzato 4
- 2.3 Dataset 6

3 Piattaforma Hardware e Software **7**
- 3.1 Hardware 7
- 3.2 Software 7

4 Implementazione su Microcontrollore **9**
- 4.1 Rete CNN in esame 9
- 4.2 Analisi degli input 11
- 4.3 STMcubeMX 12
  - 4.3.1 Reti testate 12
- 4.4 STMcubeIDE 14
  - 4.4.1 Descrizione del codice implementato 15
  - 4.4.2 Lettura e scrittura su scheda SD 15
  - 4.4.3 Creazione delle finestre ed inferenza 18

5 Risultati Ottenuti **19**
- 5.1 Risultati dell'inferenza su PC 21
- 5.2 Risultati dell'inferenza su piattaforma embedded 22
- 5.3 Confronto tra esecuzioni su PC e microcontrollore 23
  - 5.3.1 Differenze nei tempi di inferenza 23
  - 5.3.2 Impatto dell'ottimizzazione e compressione 24
- 5.4 Quantizzazione della rete tramite Developer cloud 26
  - 5.4.1 Impatto della quantizzazione 26
- 5.5 Sintesi dei risultati ottenuti 30

6 Conclusione **33**

Elenco delle figure

1.1 NILM 1
1.2 ILM 2

2.1 Esempio grafico dell'approccio 3
2.2 Rappresentazione grafica rete neurale 5
2.3 Esempio di rete CNN 5
2.4 Neurone con pesi e funzione di attivazione 6

3.1 Tabella caratteristiche del dispositivo e dispositivo utilizzato 7
3.2 Programmatore STLINK-V3MINIE 8

4.1 Analisi tramite Netron 10
4.2 Rappresentazione vettore di input 11
4.3 Report di analisi delle reti prese in esame 13
4.4 Descrizione grafica del codice 14
4.5 Diagramma di flusso main 16
4.6 Diagramma per funzioni di lettura e scrittura su SD 17
4.7 Diagramma di flusso inferenza 18

5.1 Risultati inferenza da PC rete originale 21
5.2 Risultati inferenza da PC rete ottimizzata 22
5.3 Errore assoluto tra il modello originale e ottimizzato in Python . . . 22
5.4 Risultati inferenza sul Microcontrollore 23
5.5 Differenze nei tempi di inferenza tra i modelli testati 24
5.6 Errori modelli processati in STM rispetto al PC 25
5.7 Analisi della rete quantizzata 27
5.8 Andamento output della rete quantizzata 28
5.9 Errore assoluto rete quantizzata 28
5.10 Tabella riassuntiva conclusiva dei modelli di maggiore interesse . . . 30

Capitolo 1

Introduzione

Negli ultimi decenni lo sviluppo tecnologico ha portato inevitabilmente ad una rapida crescita della richiesta di energia elettrica alla rete, a causa soprattutto dell'introduzione (nelle industrie come nelle abitazioni) di dispositivi elettronici [1]. Per affrontare questa sfida, col fine di gestire efficacemente il flusso energetico, uno degli strumenti più importanti per il monitoraggio è sicuramente il Non Intrusive Load Monitoring (NILM). [1]

Il NILM consiste nell'estrarre informazioni sui consumi energetici dei singoli carichi collegati alla rete partendo dal consumo elettrico complessivo, a differenza dell'ILM (Intrusive Load Monitoring) dove i singoli consumi sono misurati in maniera intrusiva con l'utilizzo di sensori posti in ogni dispositivo (una rappresentazione delle differenze si può trovare in Figura 1.1 e 1.2)

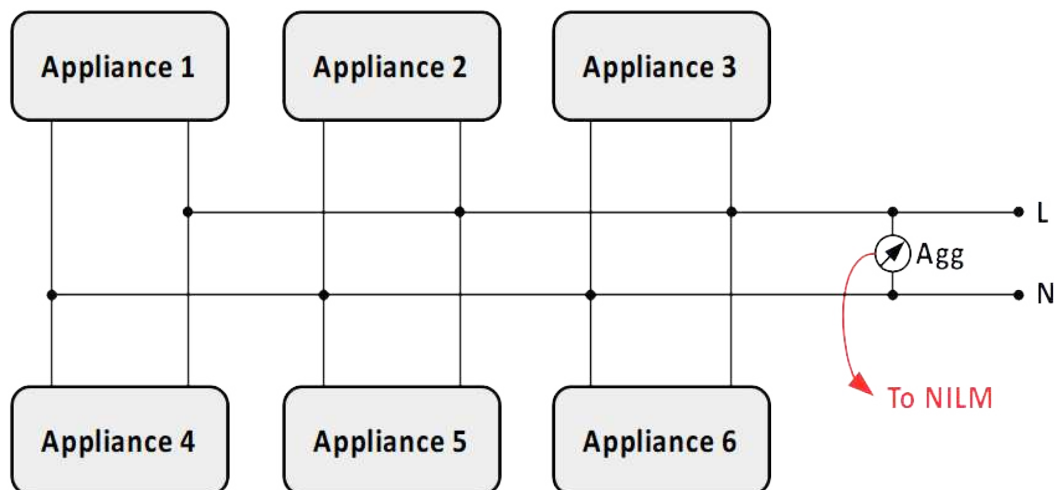

Figura 1.1: NILM

Negli ultimi decenni si sono sviluppati molti approcci per effettuare il NILM, tra cui quello che è sicuramente diventato lo stato dell'arte è l'utilizzo delle reti neurali profonde [2]. In questa tesi si descrive l'approccio effettuato attraverso l'utilizzo di una rete CNN per la disgregazione della potenza attiva assorbita da un solo

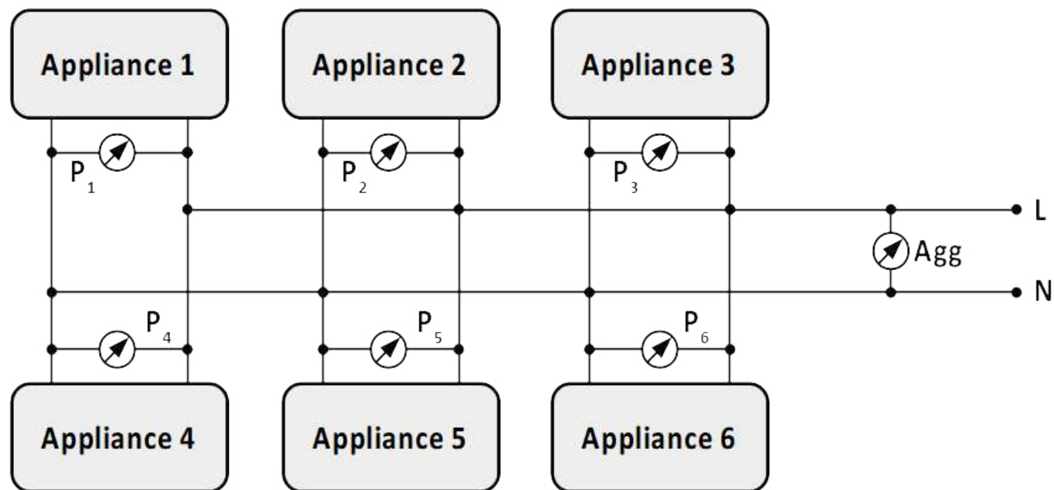

Figura 1.2: ILM

dispositivo (washing machine) partendo dalla potenza attiva richiesta dall'intera rete presa in esame (dataset UK-DALE, come descritto in 2.3).

L'obiettivo della tesi è il porting efficiente della rete neurale descritta su una piattaforma embedded (microcontrollore descritto in 3), fornita da STmcroelectronics. Lavorando quindi su un chip embedded si presta particolare attenzione a quantità di memoria occupata e tempi di inferenza andando a fare delle valutazioni qualitative e quantitative delle prestazioni (capitolo 5) ottenute sia utilizzando la rete CNN originale che effettuando ad essa delle operazioni di quantizzazione e compressione, in modo da ridurre la memoria occupata ed i tempi di inferenza.

Capitolo 2

Non Intrusive Load Monitoring

2.1 Desrizione del problema

Il sistema elettrico globale, negli ultimi decenni, sta vedendo una rapida crescita nella richiesta di potenza (in media tale richiesta è cresciuta nell'ultimo decennio del 3,4% all'anno [1]). Come descritto da un recente report dell'Eurostat, le abitazioni civili sono responsabili del consumo energetico annuale per un 27,6% [3], giocano quindi un ruolo fondamentale nell'ottica della riduzione dei consumi. La riduzione dei consumi è possibile anche grazie alle tecniche di monitoraggio dei carichi e di rilevamento di guasti, una delle più importanti è il NILM (Non Intrusive Load Monitoring).

Questa tecnica parte dall'osservazione del consumo energetico aggregato di una casa o edificio, per poi, attraverso degli algoritmi di disgregazione riuscire ad ottenere i consumi dei singoli dispositivi connessi alla rete (come visibile in Figura 2.1).

A differenza dell'ILM (Intrusive Load Monitoring), il quale richiede dei sensori posti in ogni appliance, il monitoraggio non intrusivo permette di monitorare l'intero sistema elettrico come unico punto di misurazione, riducendo i costi e la complessità di istallazione.

Per effettuare la disgregazione dei carichi si può ricorrere a varie tecniche che possono essere divise in tre principali categorie (Machine Learning, Pattern Matching e

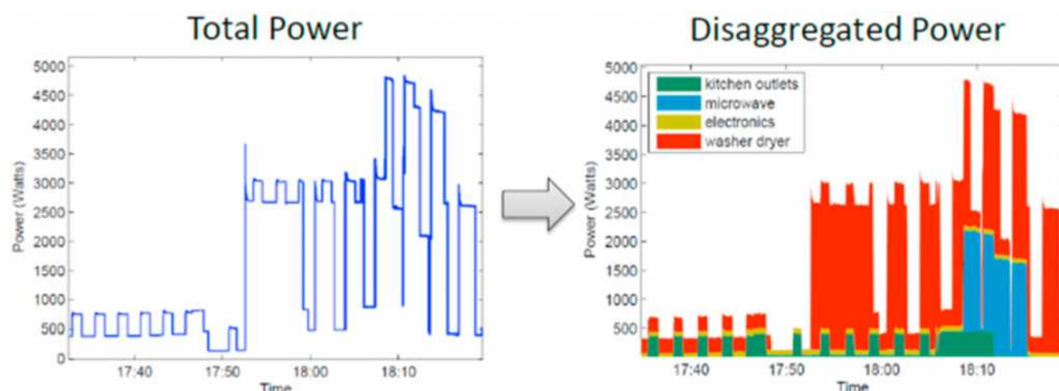

Figura 2.1: Esempio grafico dell'approccio

Single-channel Source Separation), in questo documento viene utilizzato un approccio di tipo Machine Learning (ML), che è ormai diventato lo stato dell'arte. [2]

Il principio generale dell'approccio, nel caso di N-1 dispositivi connessi ad una rete è descritto dalla formula:

$$p_{tot}(t) = f(p_1, ..., p_N, e) = \sum_{n=1}^{N-1} p_n(t) + e(t) = \sum_{n=1}^{N} p_n(t) \tag{2.1}$$

dove $\mathrm{p_{tot}(t)}$ è la potenza totale consumata, che può essere vista come somma delle potenze consumate da ogni appliance sommato alla quota di rumore inevitabilmente presente. L'obiettivo del NILM è quello di ottenere i singoli consumi $\mathrm{p_n(t)}$, cioè:

$$\overline{P} = \{p_1, p_2, ..., p_{N-1}, e\} = f^{-1}\{p_{tot}\} \tag{2.2}$$

Dove $\overline{P}$ è un vettore contenente la potenza disgregata di ogni carico. Una risoluzione analitica di questa espressione risulta impossibile da ottenere, anche se ogni apparecchio ha una propria "firma energetica", cioè un profilo unico in termini di consumo energetico. Per superare questa difficoltà, viene impiegato il Machine Learning, dove una rete neurale viene addestrata per riconoscere queste firme utilizzando dataset che contengono sia i dati di consumo aggregato che quelli disaggregati, consentendole di fare previsioni accurate.

2.2 Metodo utilizzato

Il machine learning è una branca dell'intelligenza artificiale che consente ai sistemi di apprendere automaticamente da una serie di dati senza essere programmati per compiti specifici. L'idea fondamentale è che le macchine possono fare previsioni e prendere decisioni partendo dall'acquisizione di una grande quantità di dati come il cervello umano.

Con l'ampliamento delle capacità computazionali e l'accesso a dati sempre più massivi si è riusciti a sviluppare tecniche sempre più potenti come il deep learning, che si basa sull'uso di reti neurali profonde. Le reti neurali profonde consistono in molteplici livelli (strati o layers) interconnessi tra loro, dove ogni livello riceve input dal precedente, effettua delle trasformazioni su questi input e produce un output per il livello successivo, una descrizione grafica dell'approccio è mostrata in Figura 2.2.

Per lo scopo del NILM possono essere tilizzate diverse tipologie di reti, tra cui le CNN (Convolutional Neural Networks [4]), inizialmente sviluppate per il riconoscimento di immagini sono state poi adattate per l'analisi di sequenze temporali di dati, permettendo così il riconoscimento delle firme energetiche dei vari appliance. Gli strati caratteristici per le reti di tipo CNN sono (Figura 2.3):

- **Convoluzionale**: in questo strato è presente un filtro che viene applicato all'input (atraverso operazioni di convoluzione) per ottenere dati sulle variazioni e cambiamenti nel consumo che possono essere ricondotti ai singoli dispositivi.

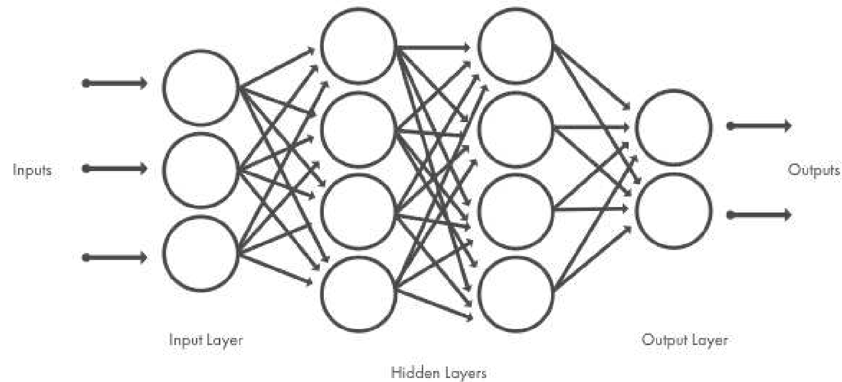

Figura 2.2: Rappresentazione grafica rete neurale

- **Pooling o Dense**: riduce la dimensionalità dei dati forniti dallo strato precedente, permettendo alla rete di essere più robusta a rumori e piccole variazioni.

- **Fully Connected**: permette di elaborare i dati ottenuti e produrre la previsione finale sul consumo di un dato apparecchio.

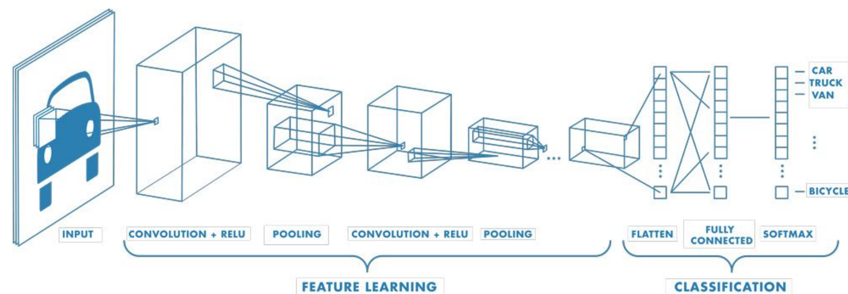

Figura 2.3: Esempio di rete CNN

In ogni livello di una rete neurale sono presenti dei neuroni artificiali (ispirati ai neuroni biologici del cervello), i quali elaborano istruzioni effettuando delle somme degli input opportunamente pesati, come si può vedere in Figura 2.4 i pesi associati all'input determinano l'influenza di ciascun valore sull'output del neurone. Durante l'addestramento, la rete neurale è quindi in grado di modificare i suoi pesi per ridurre l'errore rispetto ad un risultato atteso, affinando la capacità di riconoscere una determinata "firma energetica". Di conseguenza, una rete addestrata riesce a rispondere in modo efficace anche all'inserimento di nuovi dati di cui non si conoscono i risultati attesi in base a quanto appreso nella fase di addestramento.

Un altro elemento fondamentale nella generazione dell'output di un neurone è la funzione di attivazione, che introduce una non linearità nel sistema rendendolo in grado di riconoscere determinati pattern complessi.

Nell'insieme, pesi e funzioni di attivazione determinano lo spazio di memoria occupato dalla rete stessa, fondamentale se la si vuole implementare su un chip con memoria limitata (come vedremo in capitolo 4).

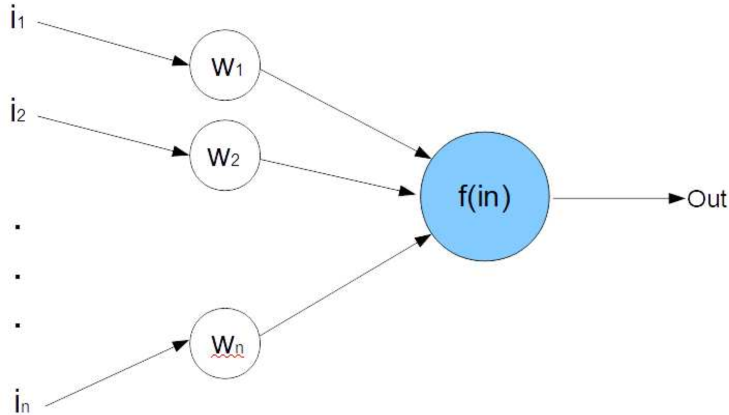

Figura 2.4: Neurone con pesi e funzione di attivazione

2.3 Dataset

I dataset sono delle raccolte strutturate ed ampie di dati, provenienti da una singola fonte e destinati ad un determinato progetto. Nel caso del NILM sono fondamentali durante l'addestramento e la verifica delle prestazioni di una rete neurale progettata per la disgregazione del carico, essi contengono infatti campionamenti di periodi molto lunghi di tempo dei consumi sia aggregati (che serviranno in input alla rete) che non (utili per l'addestramento). Alcuni dei dataset più utilizzati in questo ambito sono:

- **UK-DALE** [5]: dataset contenente dati raccolti da cinque abitazioni del Regno unito con un campionamento della potenza aggregata ogni 1 secondo (portata ad una lettura ogni 6 secondi attraverso un downsampling) e misurazioni della potenza disgregata di ogni apparecchio (come frigo, lavatrice, lavastoviglie e bollitore).

- **REFIT** [6]: dataset proveniente da 21 abitazioni del Regno Unito con campionamento della potenza sia aggregata che disaggregata si un campione ogni 8 secondi.

Capitolo 3

Piattaforma Hardware e Software

Per il progetto si utilizzano a pieno i prodotti messi a disposizione da STmicroelectronics (sia hardware che software) sfruttando un tool creato appositamente per la gestione e l'implementazione di reti neurali su piattaforme embedded.

3.1 Hardware

Per la tesi viene utilizzato il chip STEVAL-STWINKT1B [7], progettato per semplificare applicazioni industriali avanzate, attraverso sensori appositi e l'elevato grado di connettività che il dispositivo offre (Figura 3.1). Le caratteristiche del dispositivo di nostro interesse per gli scopi della tesi sono visibili in tabella 3.1.

Per poter programmare il micrcontrollore si utilizza un STLINK (visibile in Figura 3.2). L'STLINK-V3MINIE è un programmatore e debugger fornito da STM per facilitare la comunicazione tra il PC ed il chip permettendo il caricamento del firmware, la scrittura della memoria e la visualizzazione in tempo reale delle variabili durante l'esecuzione del programma (funzione di debug) [8].

3.2 Software

Per questo progetto sono stati utilizzati due dei software messi a disposizione da STmicroelectronics per facilitare la programmazione delle proprie MCUs e MPUs, nonchè software classici per la gestione degli input ed output:

Microcontrollore	STEVAL-STWINKT1B
Frequenza di clock	120 MHz
Memoria FLASH	2048 Kbyte
Memoria RAM	192 Kbyte
Connettività	Bluetooth, micro SD, USB
GPIO	16 Canali

Figura 3.1: Tabella caratteristiche del dispositivo e dispositivo utilizzato

Figura 3.2: Programmatore STLINK-V3MINIE

1. **STM32CubeIDE** [9]: ambiente di sviluppo integrato (IDE) che fornisce un ambiente per la scrittura di codice in linguaggio C/C++. L'IDE offre inoltre supporto per il debugging e le simulazioni, riducendo la necessità di passare tra più strumenti.

2. **STM32CubeMX** [10]: strumento grafico per la gestione delle periferiche dei microcontrollori STM, in grado di generare automaticamente codice in linguaggio C basandosi sulle scelte fatte graficamente dallo sviluppatore. Tale piattaforma viene opportunamente integrata dal tool X-cube-AI. [11]

   Il tool X-cube-AI è un'espansione messa a disposizione pubblicamente da STM, che estende le capacità di STM32CubeMX con la conversione automatica di algoritmi di intelligenza artificiale pre-addestrati in codice specifico per le board STM, le sue funzionalità principali sono:

   - Conversione automatizzata: il tool permette di importare reti addestrate in TensorFlow o Keras e di convertirle direttamente in linguaggio C
   - Supporto per modelli di reti di tipo CNN (utilizzata nel progetto)
   - Ottimizzazione delle prestazioni: il tool permette di sfruttare al massimo le risorse limitate delle board permettendo l'ottimizzazione della rete in termini di memoria occupata o tempo di inferenza

3. **Python** [12]: Linguaggio di programmazione di alto livello molto utilizzato nel campo dell'intelligenza artificiale e del machine learning per via della ricchezza di librerie specifiche (come descritto in 4.1). Per lo scopo del progetto, Python, ha permesso sia di convertire l'input in un formato compatibile con il microcontrollore utilizzato (descritto in 4.2), sia di calcolare i risultati delle inferenze della rete neurale al di fuori della piattaforma embedded per poi confrontarle con quelli ottenuti in STM.

4. **MATLAB** [13]: Piattaforma molto utilizzata per il calcolo numerico e matriciale, permette di creare interfacce utente e visualizzare in maniera grafica dati e funzioni. Per lo scopo del progetto viene utilizzato per la creazione dei grafici necessari per il confronto dei diversi modelli neurali.

Capitolo 4

Implementazione su Microcontrollore

4.1 Rete CNN in esame

In questo capitolo si procede con il porting della rete CNN sull'MCU; tale rete è stata generata ed addestrata tramite la libreria TensorFlow [14]. TensorFlow è una libreria per il machine learning che permette di velocizzare significativamente i calcoli fornendo una serie di funzioni e strumenti necessari per l'apprendimento. Questa libreria è una delle più utilizzate, spesso assieme a Keras. Keras è una API (Application Programming Interface) di Python ad alto livello per l'implementazione di reti neurali in poche righe di codice sfruttando una varietà di livelli (layers) pre-costruiti.

La rete neurale utilizzata è di tipo CNN e sfrutta l'architettura Sequence to point (seq2point) [15], riceve quindi in ingresso una finestra ampia di 99 campioni di consumo aggregato di potenza attiva e ne produce in output uno che rappresenta il consumo medio di potenza attiva della sola lavatrice all'interno dell'intera finestra. La rete implementata ha 7 layers (strati) di cui 2 convoluzionali e 32 neuroni con pesi, input ed output in formato float a 32 bit (4 Byte), una sua prima analisi tramite Netron (applicazione in grado di visualizzare i modelli di reti neurali [16]) è fornita in Figura 4.1.

Per via delle limitate capacità computazionali e di memoria del microcontrollore, ed in previsione di un possibile sviluppo futuro con l'integrazione di reti multi-appliance (in grado di disgregare più carichi, quindi reti più complesse, discusse nel capitolo 6), oltre alla rete originale sono state testate anche versioni ottimizzate o compresse della stessa, nelle quali i pesi vengono convertiti o compressi in formati più leggeri (paragrafo 4.3.1).

Lo sviluppo di queste reti inizia sempre da quella originale, partendo da una sua conversione in formato TensorFlow Lite. TesnsorFlow lite è una versione di TensorFlow ottimizzata per il porting su dispositivi embedded, consente una migliore integrazione con il tool x-cube-AI. come illustrato in Figura 4.1, nel caso di rete ottimizzata (utilizzando un'ottimizzazione di default tramite TensorFlow), i pesi di alcuni strati subiscono una conversione in formati più leggeri, come interi ad 8 bit, per ridurre il calcolo computazionale.

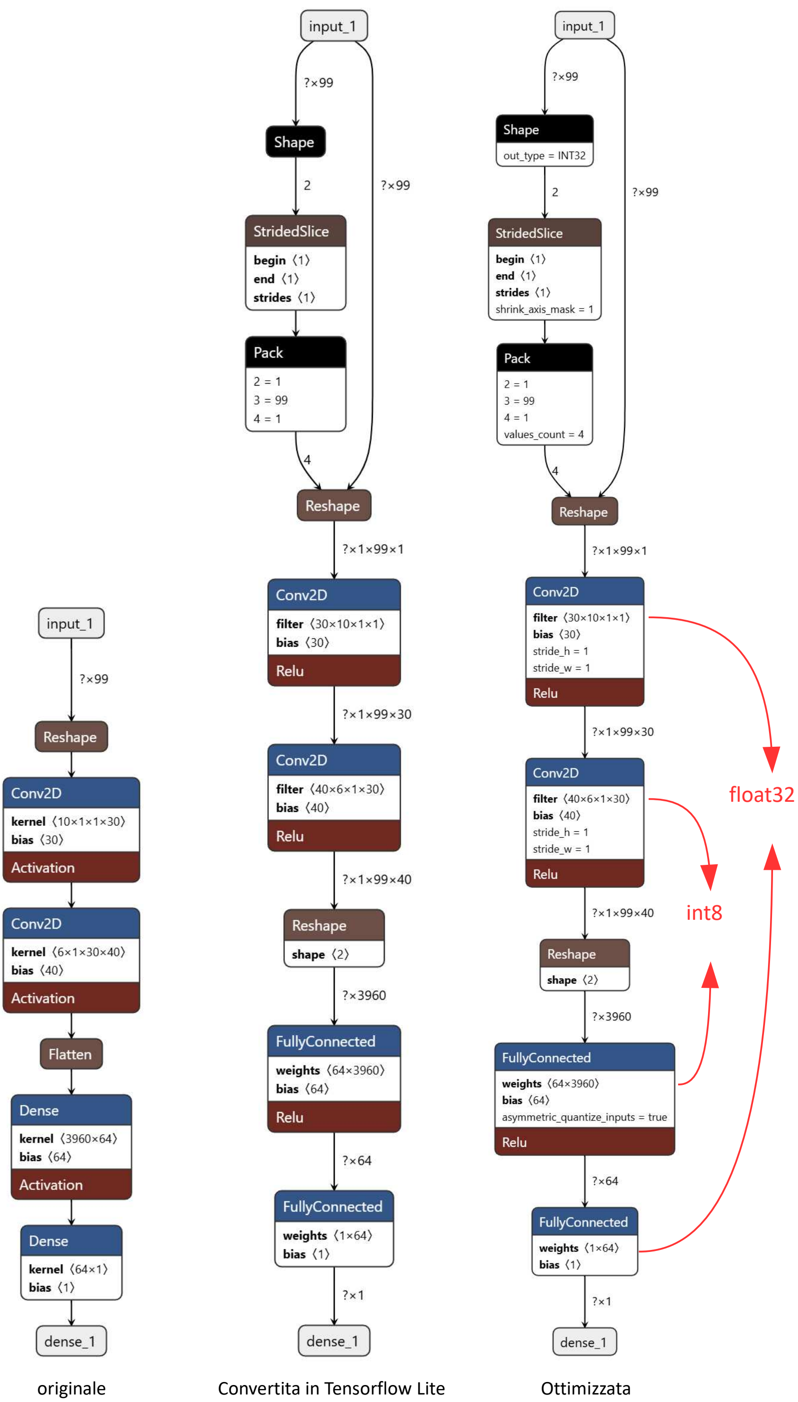

Figura 4.1: Analisi tramite Netron

4.2 Analisi degli input

L'insieme di dati presi come input vengono da una finestra di 7098 campioni presa all'interno del dataset UK-DALE (2.3). Tramite Python si converte tale finestra in un formato leggibile dall'MCU e dello stesso tipo richiesto dalla rete neurale (float32), i valori di consumo di potenza attiva vengono salvati poi in un file in formato binario, possibile grazie al codice implementato visibile in 4.1. Il file generato contenente la finestra di input viene salvato su scheda SD per essere letto dal chip (una rappresentazione del vettore preso in input è fornita in Figura 4.2).

Codice 4.1: Codice python per convertire i dati numpy in binari

```
#importo le librerie necessarie
import numpy as np

print("carico il file degli input da utilizzare")
data = np.load('input-DA-UTILIZZARE.npy')

#conversione degli input in float32
data_32 = data.astype(np.float32)

print("salvo il risultato in binario nel file input.bin")
data_32.tofile('input.bin')

input("premere invio per terminare")
```

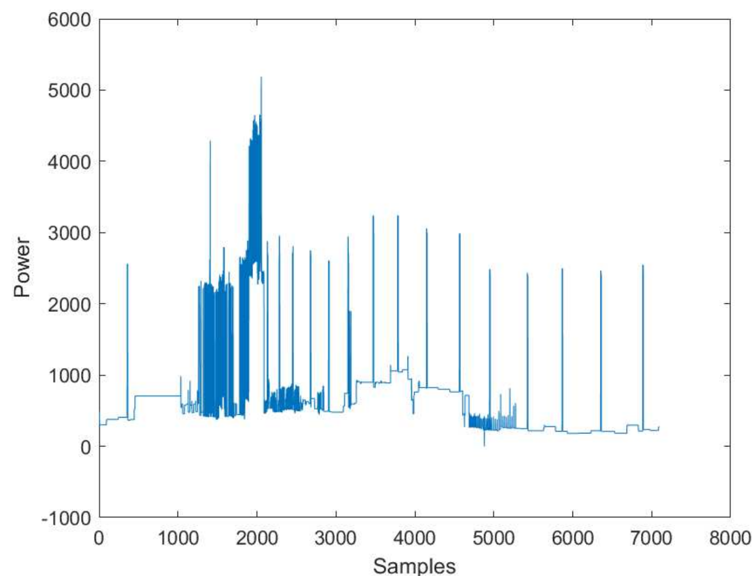

Figura 4.2: Rappresentazione vettore di input

Per processare tali dati, il vettore ottenuto viene, nel codice in STMcubeIDE suddiviso in finestre di 99 campioni per essere processate dalla rete (che ha lo stesso numero di input). Queste finestre vengono fatte scorrere nel vettore completo con

passo pari ad 1 campione, permettendo di ottenere una serie di sequenze sovrapposte che coprano interamente il vettore di input (paragrafo 4.4). La struttura a finestre facilita l'analisi sequenziale di dati e fornisce alla rete un flusso continuo di campioni per l'inferenza.

4.3 STMcubeMX

In questa fase si genera un nuovo progetto in STMcubeMX, dove si configurano opportunamente le porte ed i tool necessari:

- **Clock**: si imposta la frequenza di clock a 120MHz, sfruttando così al massimo le potenzialità dell'MCU.
- **FATFS**: si configura la libreria FatFS per il supporto in fase di lettura e scrittura di file nella scheda SD.
- **Timer**: si imposta il timer1 presente nel chip per poter valutare il tempo di ogni inferenza, volendo contare periodi di 1 us (frequenza pari a 1MHz) viene impostato un prescaler di:

$$PSC = \frac{f_{clock}}{f_{timer}} = \frac{120 \times 10^6}{10^6} = 120 \tag{4.1}$$

- **X-cube-AI**: si attiva il tool necessario per implementare nel chip in linguaggio C le reti neurali testate.

Il tool X-cube-AI offre inoltre degli strumenti molto importanti per lo scopo della tesi, tra cui:

- Analisi: si riesce in cubeMX, prima di importare la rete nel chip, ad ottenere un report che descrive perfettamente il modello che si vuole implementare e quanta memoria esso andrà ad occupare.
- Compressione: il tool permette di effettuare delle compressioni dei pesi della rete prima di importarla su piattaforma embedded, ottenendo così dei modelli compressi.

4.3.1 Reti testate

Tutti i modelli testati e confrontati partono dalla rete originale descritta in 4.1, la quale viene affiancata da versioni con pesi quantizzati o compressi, le reti testate includono:

1. **Originale**: modello CNN completo
2. **Convertita**: modello CNN convertito nel formato ".tflite" (TensorFlow Lite).

3. **Ottimizzata**: modello i cui pesi vengono ottimizzati, convertendo alcuni layers in interi tramite Python per ridurre l'occupazione di memoria ed i tempi di inferenza (Figura 4.1)

4. **Compressa**: versione del modello con compressione dei pesi, effettuata direttamente attraverso il tool fornito da STM, il quale permette diversi gradi di compressione, si definiscono così 3 nuovi modelli che differiscono nel fattore di compressione voluto: **fattore LOW**, **fattore MEDIUM**, **fatore HIGH**

Di seguito viene presentata una tabella che riassume i risultati del report di analisi generato da STMcubeMX (Figura 4.3).

	originale	convertita	ottimizzata	Fattore 'low'	Fattore 'medium'	Fattore 'high'
Dimensione file originale	3105 $Kbyte$	1024 $Kbyte$	261 $Kbyte$	1024 $Kbyte$	1024 $Kbyte$	1024 $Kbyte$
Memoria FLASH scritta	1055 $Kbyte$	1055 $Kbyte$	1055 $Kbyte$	296 $Kbyte$	168,73 $Kbyte$	168,73 $Kbyte$
Memoria RAM scritta	30,6 $Kbyte$	30,6 $Kbyte$	30,6 $Kbyte$	30,6 $Kbyte$	30,6 $Kbyte$	30,6 $Kbyte$
Guadagno di memoria	/	/	/	72,7%	84,9%	84,9%
Tipologia di input	Float 32	Float 32	Float 32	Float 32	Float 32	Float 32
Tipologia di output	Float 32	Float 32	Float 32	Float 32	Float 32	Float 32
Pesi	Float 32	Float 32	Float 32	Float 8	Float 4	Float 4
Numero di MACC	1'003'133	1'003'133	1'003'133	1'003'133	1'003'133	1'003'133
% di operazioni f_32 to f_32	100%	100%	100%	74,7%	74,7%	74,7%
% di operazioni f_32 to f_8	/	/	/	25,3%	/	/
% di operazioni f_32 to f_4	/	/	/	/	25,3%	25,3%

Figura 4.3: Report di analisi delle reti prese in esame

Da questa tabella si può osservare come i primi due modelli (originale e convertito) scrivano nel chip la stessa quantità di dati, ciò avviene perché il tool X-cube-AI converte internamente il file Keras in un file TensorFlow Lite più adatto per il porting nel chip.

Nel caso del modello ottimizzato la quantità di memoria scritta, come la tipologia dei pesi rimane la stessa delle versioni precedenti. Questo è probabilmente dovuto dal fatto che il tool X-cube-AI, riporta tutti i pesi in virgola mobile (float a 32 bit), non accettando un modello quantizzato solo parzialmente. Una soluzione alternativa, ottenuta forzando tutti i pesi ad interi tramite il developer cloud [18] fornito da STM è descritta nel capitolo 5.4.

Infine, per quanto riguarda le reti compresse con STM, si nota un miglioramento sempre maggiore al crescere del fattore di compressione. Tuttavia, come sarà illustrato nel capitolo 5, questo comporterà una perdita molto importante in termini di

precisione, poiché la compressione opera convertendo i pesi del modello da float a 32 bit a float ad 8 o 4 bit.

4.4 STMcubeIDE

L'STMcubeIDE è una piattaforma di sviluppo che permette la scrittura di codice in linguaggio C, partendo dalla configurazione iniziale fornita dal tool STMcubeMX. In questo progetto, STMcubeIDE viene usato per sviluppare un'applicazione che esegue diverse operazioni: legge l'input dalla scheda SD, crea le finestre di 99 campioni da fornire in ingresso alla rete neurale, esegue l'inferenza e salva i risultati nell'SD assieme ad un file di report che contiene i tempi di esecuzione di ogni inferenza (Figura 4.4)

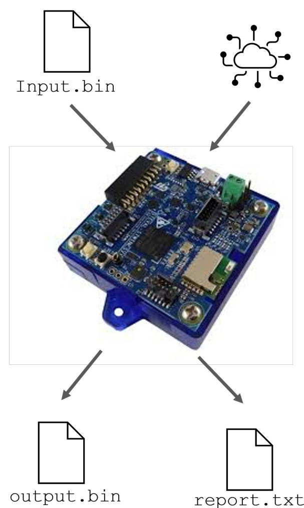

Figura 4.4: Descrizione grafica del codice

Per prima cosa nel main si effettuano delle definizioni iniziali che andranno a settare dei passaggi all'interno del codice (come descritto in 4.2):

- Nomi dei file da leggere in input e di quelli da scrivere in output.
- Tipo di dato da inserire in input alla rete, nel caso delle reti analizzate si ha il tipo float32.
- Standardizzazione dell'input: si settano le costanti per effettuare la standardizzazione.

Codice 4.2: Codice nell'IDE per definizioni e settaggi iniziali

```
1 #define F_OUT "output.bin" //file in cui vengono salvati gli output
2 #define F_REPORT "output.txt" //file di report per i tempi di infereza
```

```
3 #define F_INPUT "input.bin" //file dal quale si legge la
    finiestra in input
4 #define QUANTIZZAZIONE 0 //0 se no (input in float32) ed 1 se
     si (input in int8)
5 #if QUANTIZZAZIONE
6 typedef ai_i8 IN_TYPE;
7 #else
8 typedef ai_float IN_TYPE;
9 #endif
10 #define STANDARDIZZAZIONE 1
11 #define MEDIA        522.0
12 #define VARIANZA     814.0
13 #define OUT_EXP      3999.0
```

L'operazione di standardizzazione consente di ottenere output più affidabili, consiste nel sottrarre ad ogni valore la media e dividere per la deviazione standard calcolate su tutto l'input del training set (ovvero tutto il dataset UK-DALE) per poi moltiplicare l'output per il massimo calcolato per l'output desiderato. Nel caso del dataset UK-DALE l'operazione consiste in:

$$in_s[i] = \frac{(input[i] - 522)}{814} \tag{4.2}$$

$$out_s[i] = output[i] \cdot 3999 \tag{4.3}$$

4.4.1 Descrizione del codice implementato

Il programma principale si avvia, dopo le inizializzazioni necessarie, ricavando la lunghezza del file in input, passaggio essenziale per evitare problemi di memoria. Successivamente si decide di suddividere la lettura/scrittura dei dati in blocchi (chunk) da 10'000 valori, pari a 40'000 byte, per evitare un potenziale overflow della memoria heap del programma (usata per le allocazioni dinamiche).

Per ogni chunk il programma esegue un operazione di lettura dei valori dalla scheda. Questi valori vengono sottoposti ad un processo di standardizzazione. Una volta elaborati, i dati sono pronti per essere inseriti in input alla rete. Il programma genera quindi 10'000 finestre per ogni chunk, per ogni finestra si produce un output. Gli output generati vengono scritti nel file di output dedicato assieme ai tempi di inferenza. Un diagramma di flusso del progetto è visibile in Figura 4.5.

4.4.2 Lettura e scrittura su scheda SD

Le funzioni dedicate alla scrittura e lettura da scheda SD comprendono dei passaggi in comune, quali:

1. **Apertura del file corrispondente**: sia che la funzione legga o scriva valori nel file, per prima cosa si procede con l'apertura dello stesso, sfruttando le

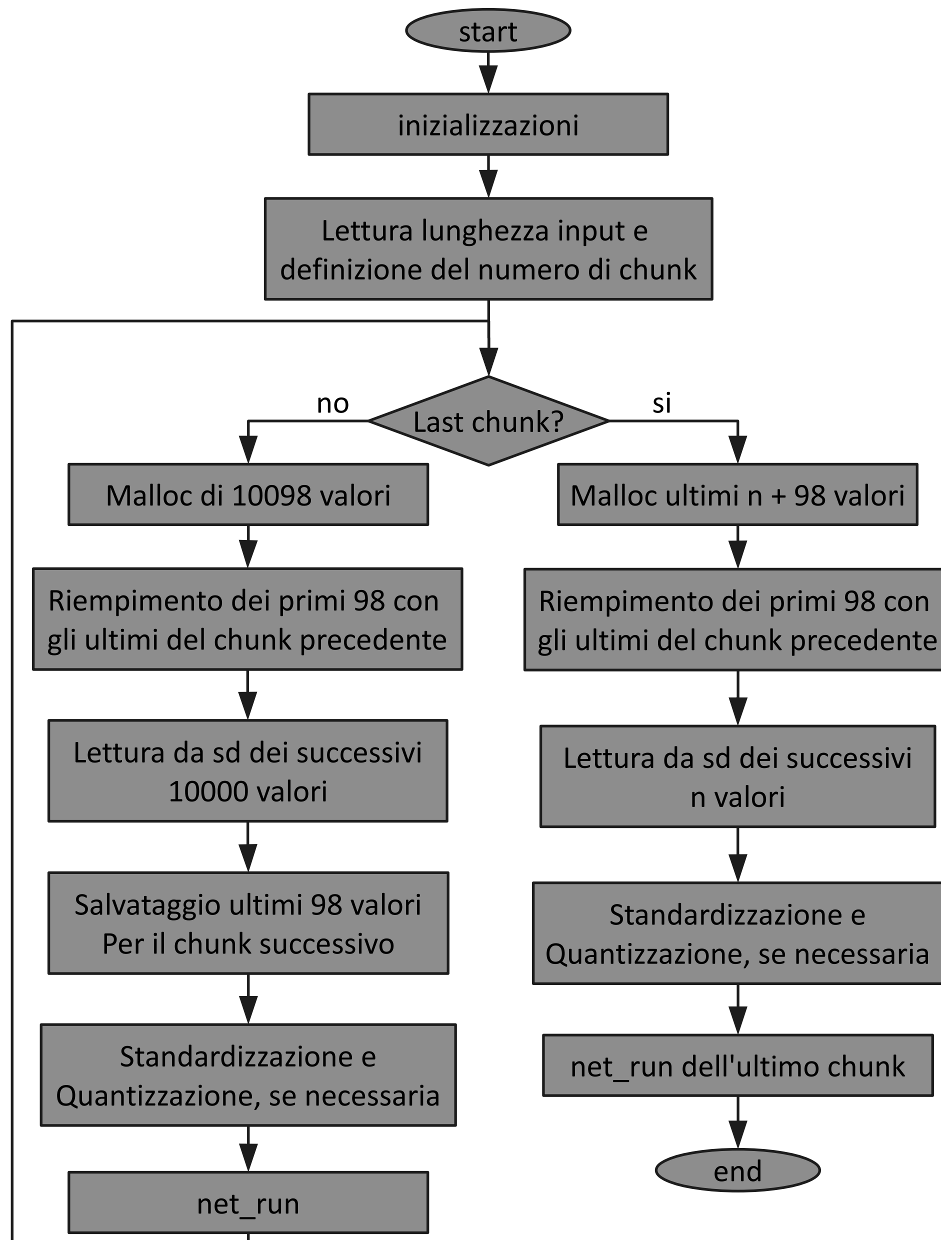

Figura 4.5: Diagramma di flusso main

intestazioni definite in 4.2 e la modalità di apertura del file, ovvero *"FA_READ"* nel caso di lettura ed *"FA_WRITE |FA_OPEN_APPEND"* (apre il file in scrittura posizionando il puntatore alla fine del file, in modo da non sovrascrivere i risultati precedenti) nel caso di scrittura.

2. **Offset**: nel caso di lettura si esegue manualmente un offset dei valori contenuti per portare il puntatore al chunk attualmente da esaminare.

3. **Lettura o Scrittura**: si esegue la vera e propria operazione di lettura o scrittura, tendendo in considerazione che un valore float occupa 32 bit (4 byte) all'interno del file, quindi per ogni valore da leggere o scrivere si vanno a considerare 4 byte di memoria.

4. **chiusura del file**: il file viene chiuso dopo aver effettuato l'operazione necessaria, in modo da salvare nell'SD il punto nel quale si è arrivati per evitare che errori improvvisi determinino la perdita dell'intero processo effettuato precedentemente.

Un diagramma che descrive le operazioni è visibile in Figura 4.6.

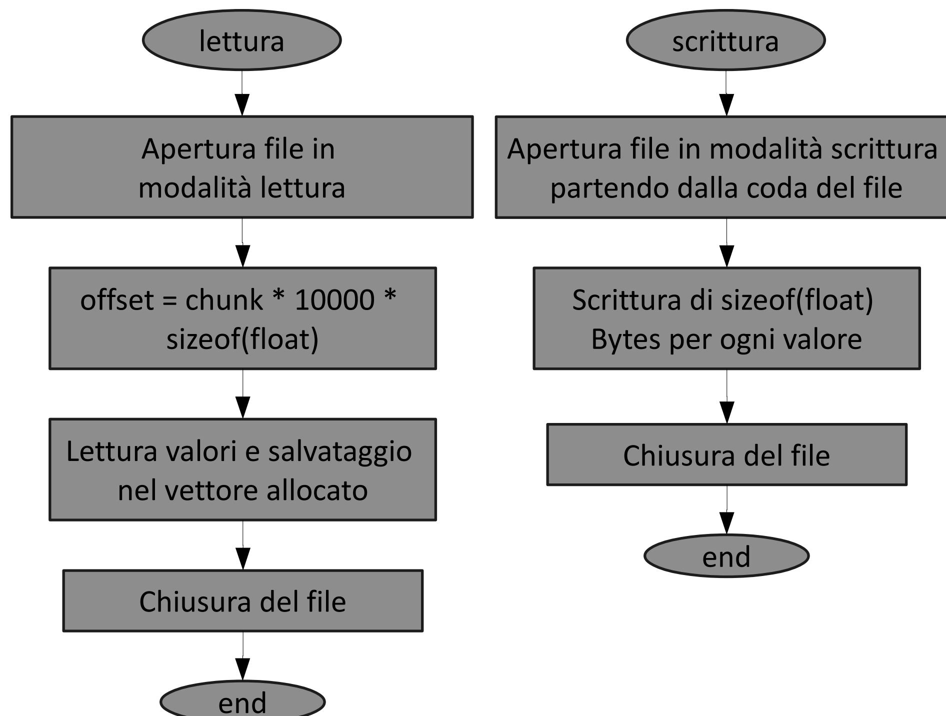

Figura 4.6: Diagramma per funzioni di lettura e scrittura su SD

4.4.3 Creazione delle finestre ed inferenza

Nella funzione dedicata all'inferenza, ogni chunk di dati viene processato per generare gli input da fornire alla rete neurale. Questo processo consiste nella suddivisione del blocco in finestre scorrevoli di 99 elementi, ogni finestra viene analizzata e viene prodotto l'output corrispondente. Il metodo di avanzamento della finestra scorrevole garantisce una sovrapposizione dei dati per una risposta accurata e coerente, una descrizione del processo è fornita in Figura 4.7.

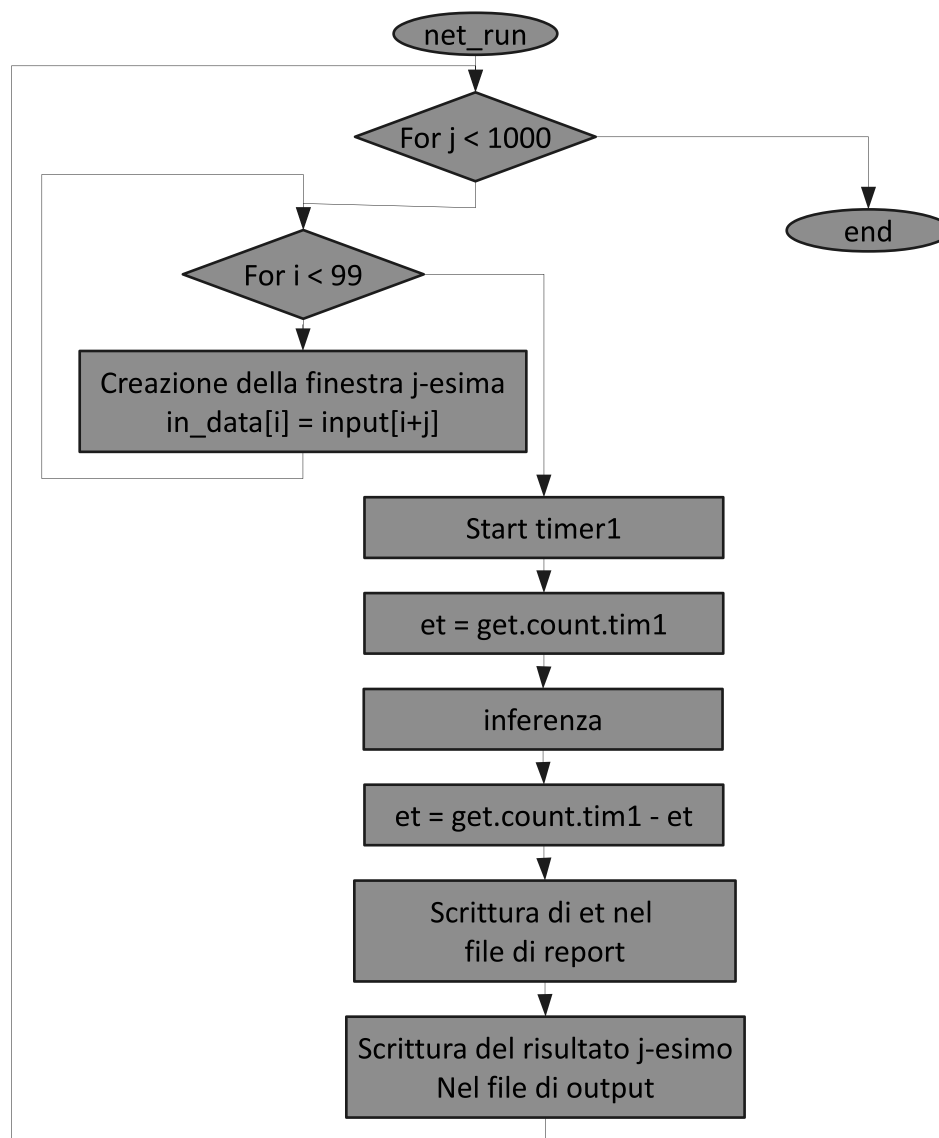

Figura 4.7: Diagramma di flusso inferenza

Capitolo 5

Risultati Ottenuti

In questo capitolo si procede ad analizzare i risultati ottenuti dalle diverse reti neurali implementate e testate su piattaforma embedded, con particolare attenzione alle prestazioni raggiunte rispetto alla versione originale della rete sviluppata in ambiente Python su PC (attraverso TensorFlow Lite 4.1). [19]

Tale comparazione tra le prestazioni ottenute su piattaforma embedded e quelle ottenute su PC consente di definire le differenze significative in termini di accuratezza e precisione dei risultati, permettendo una valutazione dell'efficacia dell'esecuzione su dispositivi a basso consumo e memoria limitata (3.1).

Per eseguire l'inferenza da PC si ricorre ad un codice Python strutturato in maniera simile a quella descritta in 4.5, con la differenza che non è necessaria la suddivisione in chunk, grazie alla maggiore capacità di memoria disponibile rispetto al dispositivo embedded, una descrizione del codice implementato è visibile in 5.1.

Codice 5.1: Codice python che esegue l'inferenza su PC

```
1 #libreria per gestire gli array
2 import numpy as np
3 #libreria per il machine learning
4 import tensorflow as tf
5 #modulo per calcolare il tempo di inferenza
6 import time
7 # carico modello originale e dati
8 interpreter = tf.lite.Interpreter(model_path='converted_model
     .tflite')
9 interpreter.allocate_tensors()
10 input_details = interpreter.get_input_details()
11 output_details = interpreter.get_output_details()
12 data = np.fromfile('input.bin', dtype=np.float32)
13 data = data.astype(np.float32)
14 output = []
15 lunghezza_finestra = 99
16 #crea il file di report dove si scrivono i tempi di inferenza
17 with open('report_originale_py.txt', 'w') as report:
18 #ciclo nel quale si creano le finestre
19     for i in range(len(data) - lunghezza_finestra + 1):
```

```
20 #standardizzazione dell'input e creazione della j-esima
   finestra
21         finestre = (data[i:i + lunghezza_finestra] - 522.0) /
               814.0
22         input_data = finestre.reshape(1,99)
23         start_time = time.perf_counter()  # start time
24         interpreter.set_tensor(input_details[0]['index'],
              input_data)
25         interpreter.invoke()
26 #output finestra j-esima e standardizzazione
27         output_data = interpreter.get_tensor(output_details
              [0]['index']) * 3999.0
28         end_time = time.perf_counter()  # stop time
29         elapsed_time = end_time - start_time
30         report.write(f"ciclo {i}: {elapsed_time:.8f} secondi\
              n")
31         output.append(output_data)
32 # Salva l'output
33 float_vector = np.array([arr.item() for arr in output], dtype
   =np.float32)
34 float_vector.tofile('output_originale_py.bin')
```

Nei successivi paragrafi si fa inoltre riferimento a parametri utilizzati per valutare le prestazioni delle diverse reti neurali:

- **Errore assoluto**, ottenuto attraverso:

$$\Delta[i] = |output_{py}[i] - output_{stm}[i]| \tag{5.1}$$

- Errore assoluto medio, **MAE**, ottenuto come:

$$MAE = \frac{\sum_{i=1}^{N} \Delta[i]}{N} \tag{5.2}$$

  con N pari alla lunghezza del vettore Δ

- Errore quadratico medio, **MSE**:

$$MSE = \frac{\sum_{i=1}^{N} \Delta[i]^2}{N} \tag{5.3}$$

- **deviazione standard**:

$$\sigma = \frac{\sqrt{\sum_{i=1}^{N} (\Delta[i] - \overline{\Delta})^2}}{N} \tag{5.4}$$

  con $\overline{\Delta}$ pari alla media dell'errore assoluto

5.1 Risultati dell'inferenza su PC

Durante l'inferenza su PC, la rete neurale ha mostrato prestazioni elevate, grazie alla capacità di poter processare l'intero set di dati sfruttando a pieno le risorse computazionali e la memoria disponibile.

L'inferenza è stata eseguita con un tempo medio ogni finestra pari a 65 μs evidenziando un' efficienza considerevole nell'elaborazione di una grande quantità di dati in tempo reale, essendo i valori di ingresso dei campioni di potenza attiva forniti con un periodo di 6 secondi (come illustrato in 2.3). Il PC è quindi riuscito ad elaborare l'intera finestra di 7098 campioni, pari a circa $7098 \cdot 6 = 42'588\,\mathrm{s} \approx 12$ ore di misurazioni in un tempo di elaborazione totale di circa $0,000065 \cdot 7098 \approx 0,46$ secondi, producendo l'output visibile in Figura 5.1.

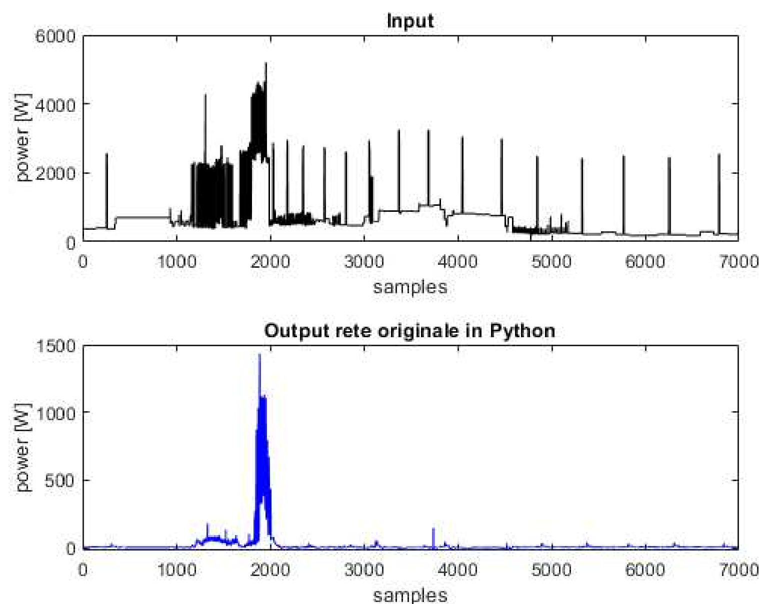

Figura 5.1: Risultati inferenza da PC rete originale

Anche la rete ottimizzata ha mostrato un notevole miglioramento in termini di tempi di inferenza che sono scesi da 65 μs a 42 μs, con un tempo totale di elaborazione di 0,3 secondi circa). Questa ottimizzazione ha portato ad una perdita di precisione relativamente limitata, mantenendo tuttavia l'andamento qualitativo dell'output sostanzialmente invariato (come illustrato in Figura 5.2).

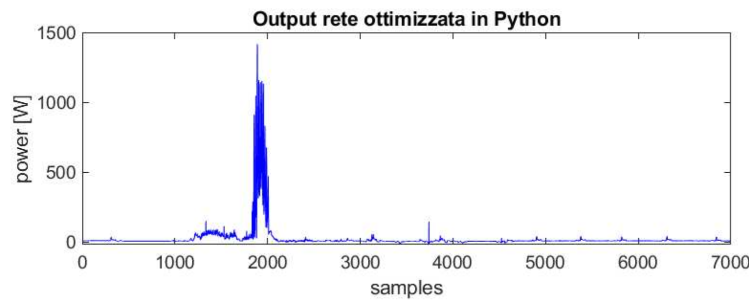

Figura 5.2: Risultati inferenza da PC rete ottimizzata

Si evidenzia nel confronto tra le due reti:

- **Errore assoluto** descritto in Figura 5.3

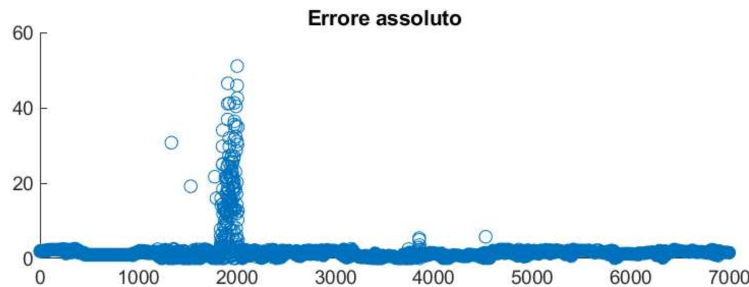

Figura 5.3: Errore assoluto tra il modello originale e ottimizzato in Python

- Errore medio, **MAE**, pari a $1,7048\,\mathrm{W}$
- **MSE** pari a $11,4873\,\mathrm{W}^2$
- **Deviazione standard** pari a $0,035012\,\mathrm{W}$

5.2 Risultati dell'inferenza su piattaforma embedded

Durante l'inferenza su piattaforma embedded, la rete neurale ha evidenziato un comportamento stabile per i modelli originali e quelli ottimizzati o compressi con fattori di compressione moderati. I risultati si mantengono in linea con le aspettative, dimostrando che l'implementazione è robusta per applicazioni pratiche. Tuttavia, con l'aumento dei fattori di compressione, si osserva un calo significativo delle prestazioni, come illustrato in Figura 5.4.

Dal punto di vista dei tempi di inferenza, si nota un notevole rallentamento rispetto all'inferenza su PC (come prevedibile data la differenza della potenza di calcolo tra i due dispositivi). Tuttavia, la rete riesce ad elaborare l'intero vettore di input in tempi molto ragionevoli, vicini alle esigenze di un elaborazione real-time. Il tempo medio di inferenza per una finestra è dell'ordine dei 40 ms. Considerando l'intero vettore di input contenente 7098 valori (equivalenti a circa 12 ore di letture), si

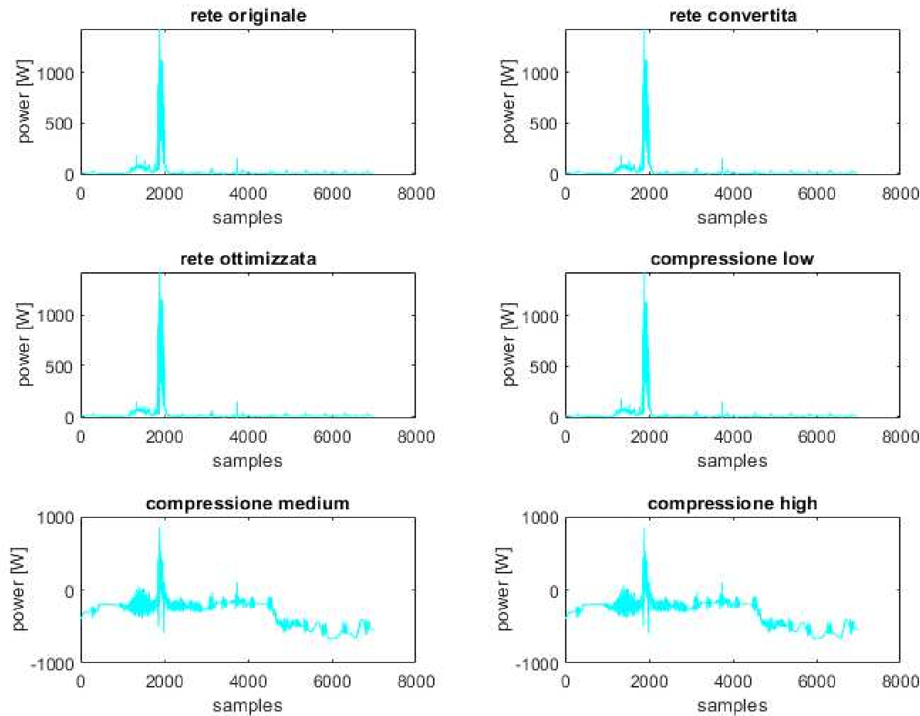

Figura 5.4: Risultati inferenza sul Microcontrollore

ottiene un tempo di elaborazione totale di $0.040 \cdot 7098 = 283,92\,\mathrm{s} \approx 5\,\mathrm{min}$, che si estende a circa 7 minuti per via delle operazioni accessorie fatte (apertura, lettura e scrittura su SD).

5.3 Confronto tra esecuzioni su PC e microcontrollore

5.3.1 Differenze nei tempi di inferenza

Le differenze nei tempi di inferenza tra l'esecuzione su PC e su piattaforma embedded sono significative e rispecchiano le differenze nelle caratteristiche hardware. La rete neurale implementata in TensorFlow Lite ha dimostrato di poter processare ciascuna finestra in un tempo di circa 65 μs, sfruttando le elevate capacità di calcolo e la grande frequenza di clock del computer. Questo ha consentito l'elaborazione di un intero set di 7098 campioni, corrispondenti a circa 12 ore di misurazioni in un tempo complessivo di circa 0,5 secondi.

Nel caso del microcontrollore, l'inferenza di una singola finestra richiede mediamente 40 ms, tempo decisamente superiore rispetto a quello ottenuto su PC. Questo rallentamento è sicuramente dovuto alle limitate capacità computazionali, alla memoria ristretta ed alle minori frequenze di clock della piattaforma embedded rispetto ai sistemi più complessi come il PC. Di conseguenza, l'elaborazione dell'intero set di

input richiede circa 5 minuti, estesi a 7 includendo le operazioni accessorie effettuate, in particolare in tabella 5.5 si riportano i tempi di inferenza tra i vari modelli testati.

modello	Originale	Convertito	Ottimizzato	Fattore low	Fattore medium	Fattore high
Tempi di inferenza di 1 finestra	35 ms	35 ms	35 ms	42,1 ms	33,9 ms	33,9 ms

Figura 5.5: Differenze nei tempi di inferenza tra i modelli testati

Dalla tabella si evidenzia, come già rilevato in fase di analisi dei modelli (paragrafo 4.3.1), e come verrà comfermato nei paragrafi successivi, che:

1. Il modello originale ottenuto tramite Keras viene convertito internamente, durante la preparazione delle reti in STMcubeMX, in un modello Tensorflow lite, determinando risultati pari a quelli ottenuti dal modello convertito esternamente all'ambiente STM.

2. Il modello ottimizzato, con alcuni layer ottimizzati in interi ad 8 bit (come visibile in figura 4.1), viene in realtà riconvertito ad un modello con pesi e funzioni di attivazione in virgola mobile a 32 bit, determinando tempi di inferenza identici ai modelli precedenti, ma risultati meno precisi.

3. I modelli compressi, nei quali come visto in figura 4.3, oltre ai pesi, circa un 25% delle operazioni sono effettuate in float ad 8 o 4 bit, non corrisponde un guadagno in termini di tempi di inferenza. Questo è probabilmente dovuto dalle troppe conversioni (float32 to float8 e viceversa) effettuate, che vanno a "silenziare" il poco guadagno nei tempi ottenuto dalla compressione.

5.3.2 Impatto dell'ottimizzazione e compressione

In Figura 5.6 viene riportato un grafico contenente i valori calcolati dei parametri descritti in 5 per i vari modelli testati rispetto all'inferenza eseguita su PC del modello convertito in TensorFlow Lite. Da tali risultati si può evidenziare che, come atteso, i modelli originali presentano errori molto limitati, trascurabili rispetto al valore misurato (errore assoluti massimi dell'ordine di 10^{-3} W su valori di potenza di picco misurata di 1400 W). Per quanto riguarda il modello ottimizzato e quello compresso di un fattore low, si evidenzia un comportamento simile, ancora accettabile se si vanno ad effettuare analisi che non necessitano di elevata precisione dei risultati (errori assoluti massimi dell'ordine di 40 W in corrispondenza del picco di potenza consumata), specialmente tenendo in considerazione il notevole guadagno in termini di consumo di memoria descritto nella Figura 4.3. Nei modelli con fattori di compressione più spinti si può visualizzare, sia nell'andamento qualitativo del risultato, sia dagli errori calcolati, un notevole peggioramento della precisione che rende questi modelli non utilizzabili in ambienti pratici.

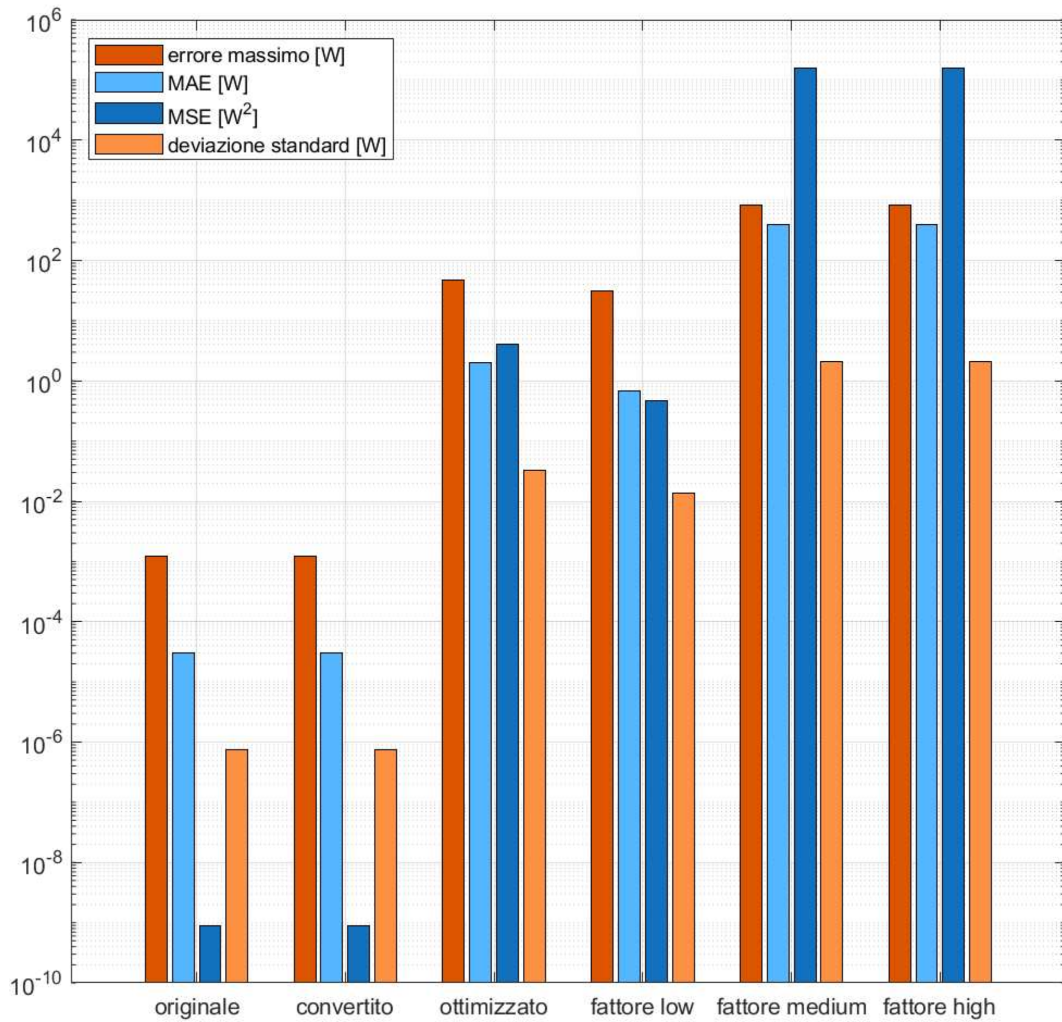

Figura 5.6: Errori modelli processati in STM rispetto al PC

5.4 Quantizzazione della rete tramite Developer cloud

Essendo i modelli valutati fino ad ora creati attraverso delle ottimizzazioni o compressioni del modello originale, per cercare di ottenere un risultato con un netto miglioramento nei tempi di inferenza si procede in questa sezione con la quantizzazione vera e propria della rete in interi ad 8 bit attraverso il developer cloud fornito da STM [18].

Developer Cloud è una piattaforma online avanzata, messa a disposizione da ST-Microelectronics per l'ottimizzazione dei modelli destinati alle piattaforme embedded. Per lo scopo del progetto viene utilizzata la sua funzione di applicare una quantizzazione post-training del modello. Questa procedura permette quindi di ridurre la precisione dei parametri del modello da 32 bit ad 8 bit, mantenendo comunque un'elevata accuratezza. Di seguito si descrivono i passaggi eseguiti sulla piattaforma [18]:

1. **Caricamento del modello originale**: il modello pre-addestrato originale, descritto nella sezione 4.1 viene caricato nella piattaforma.

2. **Definizione delle tipologie di input ed output**: per la quantizzazione si mantengono input ed output nel formato float32. La quantizzazione post-training crea un modello in grado di effettuare sia un recast degli input in interi nell'intervallo da -128 a +127, sia di riconvertire gli output in formato float32.

3. **Caricamento del dataset di calibrazione**: Per poter eseguire una quantizzazione precisa dei pesi e calibrata in base ai dati che verranno forniti in input si utilizza un dataset di calibrazione. Tale dataset è ottenuto creando alcune finestre di 99 valori partendo dal vettore di input originale, simulando i valori in ingresso tipici dell'applicazione.

4. **Procedura di quantizzazione e salvataggio**: una volta configurati i parametri necessari la piattaforma esegue la quantizzazione e genera un modello qantizzato in formato TensorFlow Lite che viene salvato e testato.

Analizzando tramite Netron tale rete si può osservare che tutti i pesi sono stati convertiti in interi ad 8 bit, inoltre si osservano 2 nuovi strati aggiunti al modello. Questi strati gestiscono la conversione tra il formato float32 ed il formato int8 (Figura 5.7).

5.4.1 Impatto della quantizzazione

Come aspettato inizialmente, il processo di quantizzazione produce un modello più efficiente di quelli analizzati nei paragrafi precedenti, facendo notevolmente abbassare i tempi medi di ogni inferenza. In particolare, analizzando tale modello in STMcubeMX si ottiene:

- **Memoria FLASH scritta**: 292 Kbytes

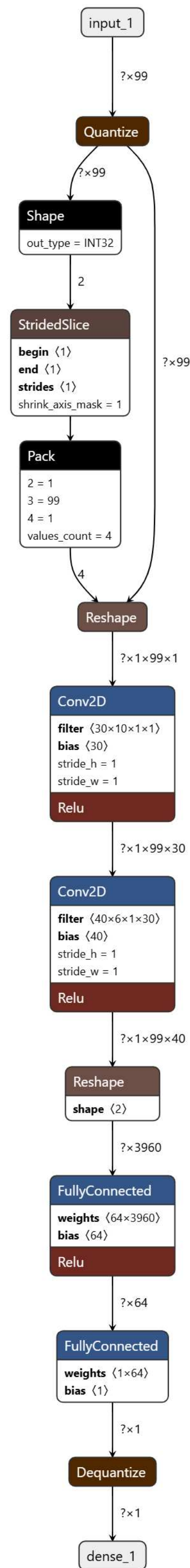

Figura 5.7: Analisi della rete quantizzata

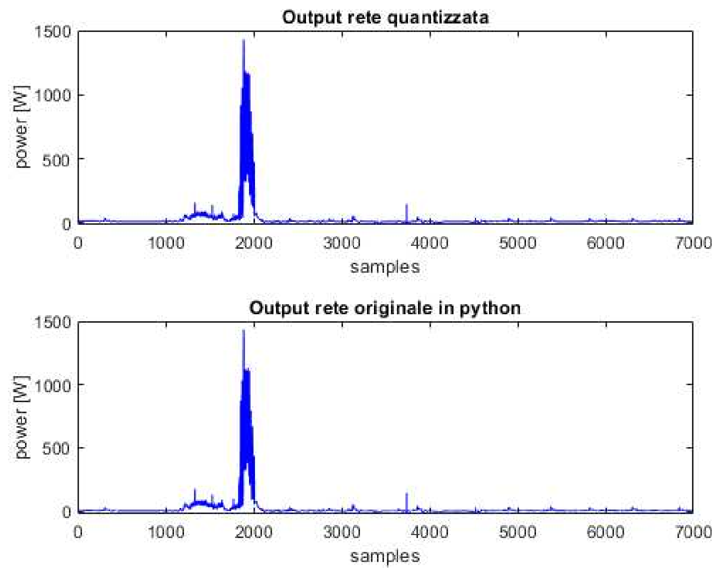

Figura 5.8: Andamento output della rete quantizzata

- **Memoria RAM scritta**: $31,4\,$Kbytes
- **MACC**: $996'339$
- **Formato pesi**: int8

Questo indica che il processo di quantizzazione post-training del modello influisce notevolmente sulla memoria occupata dai pesi, determinando una riduzione di quasi un fattore 4 della memoria FLASH scritta.

Testando tale modello su piattaforma embedded si ottiene inoltre, confrontando il risultato con quello ottenuto dal modello originale (descritto in 5.1), un andamento visualizzabile in Figura 5.8, qualitativamente molto vicino a quello aspettato. Andando quindi ora ad analizzare il risultato fornito in modo quantitativo si ottiene:

- **Errore assoluto** mostrato in Figura 5.9

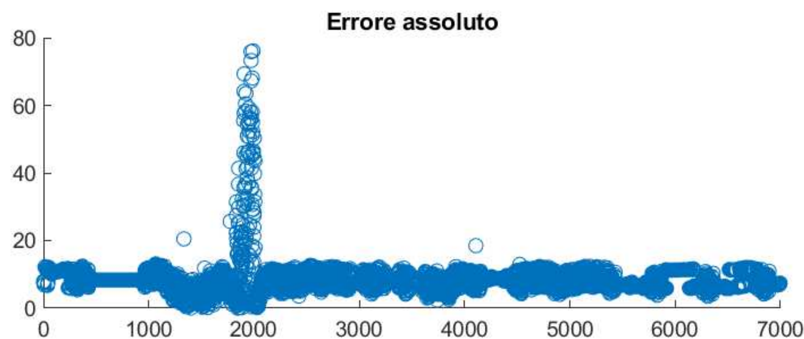

Figura 5.9: Errore assoluto rete quantizzata

- Errore medio, **MAE**, pari a $8,5514\,$W

- **MSE** pari a $95,0698\,\mathrm{W}^2$
- **Deviazione standard** di $0,055989\,\mathrm{W}$
- **Tempo necessario per ogni inferenza**, compresa quindi la quantizzazione e dequantizzazione, di $23,786\,\mathrm{ms}$

Da questi risultati si può visualizzare come l'alternativa in cui la rete viene quantizzata tramite la piattaforma on-line fornita da STM risulti valida e produca un risultato in un tempo di inferenza minore di circa un 32% rispetto a quelli dei modelli visti nei paragrafi precedenti. Tuttavia, è essenziale in questo caso che i valori in input seguano dei pattern simili a quelli inseriti tramite il dataset di calibrazione.

Una calibrazione adeguata è necessaria per determinare le soglie di quantizzazione dei pesi e delle funzioni di attivazione. Sfruttando un dataset di calibrazione che rappresenti adeguatamente i dati reali si riesce ad ottenere una maggiore precisione del modello quantizzato riducendo il rischio che esso porti ad errori di saturazione per valori non previsti (valori fuori dal range di calibrazione). In assenza di una calibrazione accurata, infatti i valori non compresi nel range potrebbero portare alla saturazione dei dati, forzando i risultati della quantizzazione entro i limiti imposti (-128 e +127). Per questo motivo si inserisce come dataset di calibrazione un insieme di finestre ottenute dal seguente codice 5.2

Codice 5.2: Codice python che crea il dataset di calibrazione

```python
import numpy as np

print("Carico il file di input")
data = np.load("input-DA-UTILIZZARE.npy")

# Impostazioni per le finestre
window_size = 99

print("Creo il dataset di calibrazione")
windows = []
for i in range(0, len(data) - window_size + 1, 1):
    windows.append((data[i:i + window_size]-522.0)/814.0)

# Converte la lista di finestre in un array NumPy
windows = np.array(windows)

print("Salvo il set di dati in un file .npz")
np.savez("input-DA-UTILIZZARE.npz", windows=windows)

input("premere invio per terminare")
```

5.5 Sintesi dei risultati ottenuti

Tra le varie versioni portate nel dispositivo embedded della rete CNN originale, si possono delineare quelle ottimizzate maggiormente per determinate applicazioni, in particolare si può subito dire che:

- La conversione della rete Keras originale in TensorFlow Lite è un'operazione effettuata internamente dal tool X-cube-AI, per questo motivo dai modelli originale e convertito si ottengono gli stessi risultati.

- L'ottimizzazione di default tramite TensorFlow Lite, che produce un modello ibrido (con pesi sia in float32 che in formati compressi), non permette di ottenere dei vantaggi durante l'esecuzione embedded. Questo è dovuto dal fatto che X-cube-AI riporta internamente tutti i pesi in float32, introducendo nel risultato soltanto l'errore di quantizzazione senza miglioramenti in tempi di inferenza o memoria FLASH scritta.

- La compressione dalla rete direttamente all'interno del tool X-cube-AI è una valida alternativa nel caso di un fattore di compressione basso. Il risultato, invece, sia nell'andamento che nell'accuratezza, è molto discostante dall'originale nel caso di fattori 'medium' ed 'high'

Di seguito viene riportata una tabella riassuntiva dei modelli considerati più validi (Figura 5.10).

	Inferenza in Python	Inferenza su microcontrollore		
	Originale	**Originale**	**Fattore 'low'**	**Quantizzato**
Dimensione file originale	3105 Kbyte	3105 Kbyte	1024 Kbyte	262 Kbyte
Memoria FLASH scritta	/	1055 Kbyte	296 Kbyte	292 Kbyte
Memoria RAM scritta	/	30,6 Kbyte	30,6 Kbyte	31,4 Kbyte
Tipologia di input ed output	Float 32	Float 32	Float 32	Float 32
Pesi	Float 32	Float 32	Float 8	Int 8
Errore assoluto massimo	/	$12{,}207 \times 10^{-4}$ W	30,828 W	76,106 W
Errore medio, MAE	/	$2{,}980 \times 10^{-5}$ W	0,686 W	8,551 W
MSE	/	$8{,}882 \times 10^{-10}$ W^2	$0,471$ W^2	95,0698 W^2
Deviazione standard	/	$7{,}419 \times 10^{-7}$ W	0,014 W	0,056 W
Tempo di inferenza	65 µs	35 ms	42,1 ms	23,8 ms

Figura 5.10: Tabella riassuntiva conclusiva dei modelli di maggiore interesse

Dalla tabella si può subito osservare che i vari modelli sono dei buoni compromessi, la scelta dipende dall'applicazione che si deve andare ad eseguire, in particolare:

- **Modello originale**: È il modello che sicuramente riporta dei risultati molto affidabili ed implementa una rete robusta nel dispositivo embedded. Anche se i tempi di inferenza si dilatano molto rispetto a quelli dell'inferenza su PC, rimane un'ottima soluzione nel caso descritto, dove i campioni dei valori in input vengono presi con una frequenza di $\frac{1}{6} \approx 0,167\,\text{Hz}$ ed i risultati forniti dal chip ad una frequenza di circa $\frac{1}{0{,}035} = 28,5\,\text{Hz}$, rendendo l'applicazione quasi real-time.
- **Modello compresso di fattore 'low'**: Rappresenta una versione del modello più leggera in termini di memoria occupata (ridotta di quasi un fattore 4) ma meno precisa della precedente. I tempi di inferenza tendono ad aumentare probabilmente per via delle conversioni interne effettuate tra i vari strati (descritti in Figura 4.3 e Figura 5.5). Questo modello rimane valido per applicazioni con reti più complesse che richiedono quindi una maggiore memoria disponibile, ad esempio se si vogliono implementare modelli multi-elettrodomestico.
- **Modello quantizzato**: Rappresenta una versione nella quale si perde maggiormente la robustezza e la precisione dei risultati, ma si guadagna notevolmente nei tempi di inferenza, ottenendo un modello che produce output ad una frequenza di $\frac{1}{0{,}0238} \approx 42\,\text{Hz}$. Questo modello è quindi valido per applicazioni in cui si conosce il pattern generale di andamento dell'input (per via di quanto discusso in 5.4) e nelle quali si vuole ottenere un vincolo real-time più stringente, concedendo però una limitazione nella precisione del risultato.

Capitolo 6

Conclusione

Il lavoro svolto in questa tesi ha dimostrato l'efficacia dell'implementazione di una rete neurale creata per il *Non Intrusive Load Monitoring* (NILM) su piattaforma embedded (microcontrollore STEVAL-STWINKT1B) di STMicroelectronics. L'obiettivo principale era realizzare un porting efficace della rete sull'hardware dalle risorse computazionali limitate, mantenendo comunque prestazioni accettabili. Attraverso un approccio metodico è stato possibile effettuare un confronto dettagliato tra vari modelli, applicando tecniche di scalabilità volte alla riduzione della memoria occupata dalla rete neurale e dei tempi di inferenza, ottenendo un bilanciamento tra costo computazionale e prestazioni del modello. Questo approccio ha dimostrato il potenziale per sviluppare sistemi di monitoraggio energetico su dispositivi a basso consumo, rendendoli particolarmente promettenti per applicazioni pratiche.

Tra le soluzioni testate in ambiente STM, oltre agli ottimi risultati ottenuti dal modello originale (descritto nella sezione 4.1), sono risultate particolarmente efficaci le tecniche di compressione tramite STMcubeMX (come descritto nella sezione 4.3.1) oppure di quantizzazione offerte dal pacchetto developer cloud di STM (descritto nella sezione 5).

In particolare si parte dal modello originale in cui sono stati riscontrati degli errori commessi molto limitati (massimo errore di 10^{-3} W) e tempi di inferenza quasi real-time. Attraverso le operazioni di compressione e quantizzazione si riesce a ridurre di quasi un 75% la memoria occupata dal modello, anche se contemporaneamente si perde di precisione nel risultato ottenuto (l'errore massimo commesso sale a 30 W nel caso di modello compresso e 70 W nel caso di modello quantizzato). Si nota inoltre una riduzione notevole del tempo di inferenza nel caso della rete quantizzata (con un guadagno di circa il 30%).

Per gli sviluppi futuri, un aspetto cruciale sarà l'integrazione di modelli multi-appliance, che permetterebbero la disgregazione simultanea di più dispositivi con un'unica rete neurale. Inoltre, l'adozione di tecniche di quantizzazione avanzate e nuove architetture di rete potrebbe migliorare ulteriormente le prestazioni senza sacrificare la precisione. Tuttavia, lo sviluppo di modelli multi-appliance è strettamente vincolato alla quantità di memoria disponibile nella piattaforma embedded ed ai tempi di inferenza, che devono rispettare il vincolo imposto dal campionamento del segnale in ingresso. In questo contesto, il tempo necessario per ogni inferenza

deve essere quantomeno minore del periodo di campionamento del segnale in input, che rappresenta il consumo di potenza attiva aggregato.

In definitiva, l'implementazione di reti neurali profonde per il task del NILM su piattaforma embedded è una strada promettente per lo sviluppo di soluzioni energetiche alternative, aprendo nuove possibilità in ambito domestico. Il bilanciamento tra precisione, tempo di esecuzione e prestazioni rimane una sfida chiave, influenzata da molti fattori, ma le opportunità offerte dall'integrazione con sistemi di *smart home* potrebbe rendere questa tecnologia interessante nella gestione energetica futura. La flessibilità dimostrata da questo approccio consente lo sviluppo della tecnica descritta in base alle specifiche esigenze applicative, fornendo la possibilità di sviluppare soluzioni flessibili in ambito domestico ed industriale.

Bibliografia

[1] Iosif Mporas Pascal A. Schirmer. Non-intrusive load monitoring: A review. *IEEE TRANSACTIONS ON SMART GRID, VOL. 14, NO. 1*, 2023.

[2] Emanuele Principi Giulia Tanoni and Stefano Squartini. Multilabel appliance classification with weakly labeled data for non-intrusive load monitoring. *IEEE TRANSACTIONS ON SMART GRID, VOL. 14, NO. 1*, 2023.

[3] Eurostat. "electricity and heat statistics: Consumption of electricity and derived heat. `https://ec.europa.eu/eurostat`, 2021.

[4] R. von Mises. Reti neurali convoluzionali (cnn): come funzionano e cosa sono. `https://www.bnova.it/intelligenza-artificiale/reti-neurali-convoluzionali-cnn/`, 2024.

[5] J. Kelly and W. Knottenbelt. The u.k.-dale dataset, domestic appliance-level electricity demand and whole-house demand from five u.k. homes. *Sci. Data, vol. 2, Art. no. 150007*, 2015.

[6] D. Murray et al. An electrical load measurements dataset of united kingdom households from a two-year longitudinal study. *Sci. Data, vol. 4, no. 1, Art. no. 160122*, 2017.

[7] Stwin sensortile wireless industrial node development kit and reference design for industrial iot applications. `https://www.st.com/en/evaluation-tools/steval-stwinkt1b.html`.

[8] Stlink-v3 compact in-circuit debugger and programmer for stm32. `https://www.st.com/en/development-tools/stlink-v3minie.html`.

[9] Integrated development environment for stm32. `https://www.st.com/en/development-tools/stm32cubeide.html`. Nel sito, oltre al software, sono presenti le guide per l'utilizzo.

[10] Stm32cube initialization code generator. `https://www.st.com/en/development-tools/stm32cubemx.html`. Nel sito, oltre al software, sono presenti le guide per l'utilizzo.

[11] Ai expansion pack for stm32cubemx. `https://www.st.com/en/embedded-software/x-cube-ai.html`. Tool per il machine learning e guida per l'utilizzo.

[12] Python. `https://www.python.org/`.

[13] Mathworks. Matlab. `https://it.mathworks.com/products/matlab.html`.

[14] Tensorflow. `https://www.tensorflow.org/tutorials?hl=it`.

[15] Chaoyun Zhang, Mingjun Zhong, Zongzuo Wang, Nigel Goddard, and Charles Sutton. Sequence-to-point learning with neural networks for non-intrusive load monitoring. *Proceedings of the AAAI Conference on Artificial Intelligence, 32(1).*, 2017.

[16] Netron. `https://netron.app/`.

[17] `https://github.com/Luca-m3/repository`. In questo link si possono trovare i codici utilizzati nel progetto.

[18] St edge ai developer cloud. `https://stm32ai-cs.st.com/home`.

[19] Intro to tensorflow lite. `https://vittoriomazzia.com/tensorflow-lite/`.