POLITECNICO DI TORINO

Corso di Laurea Magistrale in Ingegneria Elettronica

Politecnico
di Torino

Tesi di Laurea Magistrale

Rilevamento ambientale di eventi tramite processamento di campioni sonori

Relatori

Prof. Massimo RUO ROCH

Prof. Marco RE

Candidato

Davide ERRICO

Giugno 2022

A mio nonno Antonio,
che mi guida e protegge sempre.

A zio Oronzino,
che ha sempre creduto in me.

Sommario

Al giorno d'oggi la tecnologia riveste un ruolo fondamentale nella nostra vita quotidiana. L'uso di dispositivi e veicoli connessi è in forte crescita e si cerca costantemente di realizzare sistemi in grado di riconoscere gli eventi che accadono nell'ambiente circostante ed eseguire elaborazioni in base alla loro intelligenza. L'obiettivo di questa tesi consiste nel realizzare un sistema in grado di riconoscere eventi sonori presenti nell'ambiente. Le informazioni rilevate dal sistema possono essere usate in varie applicazioni, in questo caso, nell'ambito di veicoli intelligenti per il monitoraggio di suoni di emergenza nelle città. L'applicazione è stata realizzata su un nodo sensore, ovvero un microcontrollore a bassissimo consumo STM32L496AG, sfruttando i recenti metodi di deep learning basati su reti neurali artificiali costituite da diversi strati sequenziali. Tuttavia, le risorse limitate della piattaforma impongono di ottimizzare l'algoritmo in modo da ridurre il consumo di potenza ed il tempo di esecuzione. Nonostante le tecniche di ottimizzazione introdotte, è necessario realizzare un acceleratore hardware su FPGA della parte critica, ovvero l'inferenza della rete neurale. Gli strumenti di sintesi ad alto livello consentono di ottimizzare e sintetizzare automaticamente il progetto nel minor tempo possibile. Tuttavia, la sintesi mediante linguaggi di descrizione dell'hardware permetterebbe di ottenere risultati migliori in termini di risorse utilizzate e di integrare il modello su piattaforme a basso consumo.

Ringraziamenti

Prima di procedere con la dissertazione, sento il dovere di spendere alcune parole per ringraziare tutti coloro che hanno permesso e favorito il raggiungimento di uno dei più grandi obiettivi della mia vita.

Non è facile citare e trovare le parole giuste per ringraziare, in poche righe, tutte le persone che mi hanno accompagnato fino alla fine di questo percorso: chi con una presenza costante, chi con un supporto morale o materiale, chi con consigli e suggerimenti o solo con parole di incoraggiamento. Spero di non dimenticare nessuno, ma se ciò dovesse accadere, caro lettore, dimmelo e ti ringrazierò di persona, anche per il solo fatto che tu abbia avuto interesse a leggere fino alla fine.

Innanzitutto, vorrei esprimere la mia gratitudine nei confronti del mio relatore, prof. Massimo Ruo Roch, e del mio correlatore, prof. Marco Re, per avermi guidato durante tutte le fasi della composizione di questo lavoro e per avermi dato consigli preziosi che mi hanno consentito di crescere sia personalmente che professionalmente.

Ringrazio i miei genitori per la costante presenza, il sostegno, sia emotivo che economico, e per avermi permesso di intraprendere quest'esperienza indimenticabile senza farmi mai mancare nulla. Sono contento di essere riuscito a ripagare i loro sacrifici rendendoli orgogliosi di me per il grande traguardo raggiunto.

A mia sorella Silvia, la mia complice da sempre e la coinquilina migliore che potessi avere a Torino. Grazie per essermi stata sempre accanto, soprattutto nei momenti difficili, e per avermi sopportato in casa. Ammetto che convivere con un ingegnere non sia per niente facile.

Ringrazio tutti i miei parenti che, anche se lontani, non hanno mai smesso di supportarmi e di essermi vicini virtualmente. In particolare, ci tengo a menzionare gli zii di Torino, Serena e Oronzo, che sono stati il mio punto di riferimento da quando mi sono trasferito nella nuova città, mi hanno trattato come un figlio e

hanno sempre creduto in me.

A Emanuele, l'amico di una vita che mi conosce meglio di chiunque altro. Grazie per essere stato fedele e sempre presente, anche nel momento del bisogno e a migliaia di chilometri di distanza. Abbiamo condiviso i momenti migliori dalla prima elementare fino all'università senza mai separarci. Sono convinto che, anche se un giorno saremo lontani, la nostra amicizia rimarrà indissolubile.

A Chiara e Marta, amiche di serate indimenticabili, di chiacchierate interminabili, di risate, di giornate intense e spensierate a mare. Grazie per avermi supportato, ma soprattutto sopportato tutte le volte che vi ho parlato di progetti e argomenti inerenti all'elettronica.

Un grazie speciale ai miei compagni di università, Davide, Francesco, Luca, Luigi e Thomas, con cui ho trascorso questi anni memorabili. Abbiamo condiviso momenti importanti ed intensi, a volte faticosi, ma li abbiamo superati tutti insieme e mi avete incoraggiato a non mollare mai. Spero di condividere ancora altri momenti insieme a voi e di mantenere il legame forte che abbiamo creato in questi anni.

Allo stesso modo, vorrei ringraziare tutte le persone che ho incontrato durante questo percorso, con cui ho condiviso momenti di studio e divertimento. Sono così tanti i ricordi che mi passano per la testa che è impossibile trovare le parole giuste per onorarli. Grazie per aver reso il mio traguardo davvero speciale!

Indice

Elenco delle tabelle

Elenco delle figure

Acronimi

IoT
Internet of Things

FPGA
Field-Programmable Gate Array

CPU
Central processing unit

GPU
Graphic processing unit

ASIC
Application specific integrated circuit

HDL
Hardware Description Language

RTL
Register Transfer Level

HLS
High-Level Synthesis

hls4ml
High-Level Synthesis for Machine Learning

SED
Sound Event Detection

DCASE

Detection and Classification acoustic scenes and events

DFT

Discrete Fourier Transform

STFT

Short Time Fourier Transform

DCT

Discrete Cosine Transform

MFCC

Mel Frequency Cepstral Coefficient

HMM

Hidden Markov Model

GMM

Gaussian Mixture Model

MLP

Multi Layer Perceptron

CNN

Convolutional Neural Network

PDM

Pulse Density Modulation

DFSDM

Digital Filter for Sigma Delta Modulator

ADC

Analog to Digital Converter

MEMS

Micro Electro Mechanical Systems

SPL

Sound Pressure Level

SMR

Sound Meter

DMA

Direct Memory Access

FAT

File Allocation Table

FatFs

File Allocation Table File System

SDMMC

Secure Digital MultiMediaCards

GLU

Gated Linear Unit

RNN

Recurrent neural network

Capitolo 1

Introduzione

Negli ultimi decenni i dispositivi elettronici hanno presentato un rapido sviluppo, diventando sempre più piccoli, performanti e parte integrante della vita quotidiana. Per questo motivo l'uso di dispositivi e veicoli connessi è in forte crescita: si parla, infatti, di *Internet delle cose* (IoT, acronimo inglese di Internet of Things), caratterizzato da diverse tecnologie interconnesse che lavorano senza il bisogno di interazione umana. Ciò implica che gli oggetti della vita quotidiana sono dotati di sensori, ricetrasmettitori per la comunicazione e una serie di protocolli che li rendono interconnessi tra loro e con gli utenti, diventando parte costitutiva dell'internet [1].

In questa direzione, si cerca costantemente di realizzare dispositivi in grado di riconoscere gli eventi che accadono nell'ambiente circostante ed eseguire delle elaborazioni in base alla loro intelligenza [2]. L'applicazione del paradigma IoT al contesto urbano determina il concetto di *Smart-City*, caratterizzato da un'infrastruttura di comunicazione che consente di migliorare la qualità delle risorse pubbliche.

Tuttavia, rendere i dispositivi consapevoli del contesto (*context-awareness*) è una sfida impegnativa in quanto, attualmente, la maggior parte dei dispositivi è in grado di riconoscere alcuni aspetti dell'ambiente (ad esempio la rumorosità in città), ma non la sorgente sonora, che aprirebbe nuovi scenari in questo ambito. Ad esempio, un dispositivo che può rilevare suoni come l'incidente tra due veicoli, o un'esplosione, potrebbe essere utilizzato per garantire la sicurezza personale. Il rilevamento è basato sulla raccolta di dati del mondo reale da parte di sensori intelligenti presenti nell'ambiente IoT. Un sensore è generalmente costituito da due elementi: uno di rilevamento e un altro di trasduzione.

Raccogliere dati dall'ambiente rappresenta un'attività fondamentale facilmente

accessibile da autorità e cittadini, per rendere più rapida la risposta delle autorità ai problemi della città e aumentare la consapevolezza e la partecipazione dei cittadini [1].

1.1 Architetture IoT

Prima di proseguire con gli obiettivi della tesi è necessario richiamare l'architettura IoT, anche se al momento non esiste un modello di riferimento comune [2]. Uno dei primi modelli proposti è costituito da tre livelli, come riportato in Figura 1.1A:

- *Perception layer* (livello di percezione): livello fisico che contiene tutti i sensori, dove le informazioni sono prelevate dal mondo esterno;

- *Network layer* (livello di comunicazione): responsabile della connessione di dispositivi e server oltre che dell'elaborazione e trasmissione dei dati acquisiti dai sensori;

- *Application layer* (livello applicazione): responsabile di fornire dei servizi agli utenti;

Secondo l'articolo [3], l'architettura IoT è costituita da cinque livelli per rappresentare i passaggi intermedi tra un livello ed il successivo, come riportato in Figura 1.1B.
Il livello di percezione e applicazione mantengono le stesse funzionalità, ma si aggiungono gli altri tre livelli:

- *Transport layer* (livello di trasporto): si occupa del trasferimento dei dati acquisiti dal livello sottostante al livello di elaborazione tramite tecnologie come Wi-Fi, 3G, Bluetooth, NFC;

- *Processing layer* (livello di elaborazione): livello intermedio per il processamento e memorizzazione dei dati tramite tecnologie che includono basi di dati e cloud;

- *Business layer* (livello di attività): gestisce tutto il sistema IoT sulla base dei dati ricevuti dal livello applicazione;

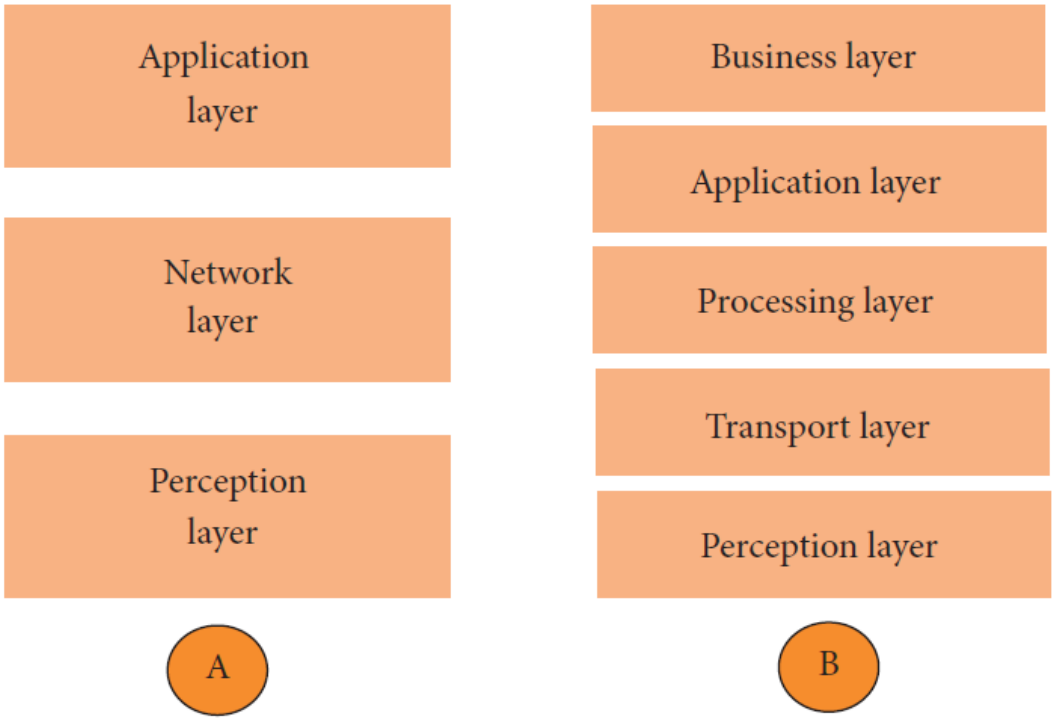


Figura 1.1: *Architetture IoT: A (3 livelli), B (5 livelli) [2]*

1.2 Rilevamento dell'evento sonoro

Questa tesi è incentrata sul rilevamento di eventi sonori nell'ambiente che costituiscono una parte significativa delle caratteristiche del contesto. Infatti, uno dei sensi che utilizziamo spesso nell'interazione con l'ambiente è l'udito. Per questo motivo, gli esseri umani sono in grado di interpretare e assegnare un significato ai suoni. Tuttavia, è difficile realizzare sistemi in grado offrire un'affidabile accuratezza per questo compito. In questa tesi viene proposto e sviluppato un metodo per il rilevamento automatico di eventi sonori (SED, sound event detection o rilevamento dell'evento sonoro). Questo differisce da altre attività di recupero delle informazioni audio, come il riconoscimento vocale, poiché la finalità non è associare l'audio vocale a parole/fonemi, ma identificare audio non vocale.

Il rilevamento dell'evento sonoro consiste nel riconoscere l'evento sonoro generato da una sorgente in un segnale audio continuo, associando un'etichetta testuale chiamata anche *classe*. Gli eventi sonori possono appartenere a due categorie: *monofonici* e *polifonici*. I primi si riferiscono agli eventi isolati per cui si associa

una singola classe all'intera registrazione audio; nel secondo caso si hanno eventi sovrapposti nello stesso intervallo temporale. Quest'ultimo caso è più comune negli ambienti reali, le cui caratteristiche acustiche dipendono dalla presenza di attività umane e della natura. In Figura 1.2 è riportato un esempio di eventi sonori polifonici in un ambiente reale. Per questo motivo, l'intera registrazione audio viene solitamente suddivisa in segmenti di uguale lunghezza, per rilevare la presenza dell'evento in ogni segmento ed infine combinare tutte le predizioni per ottenere l'uscita finale.

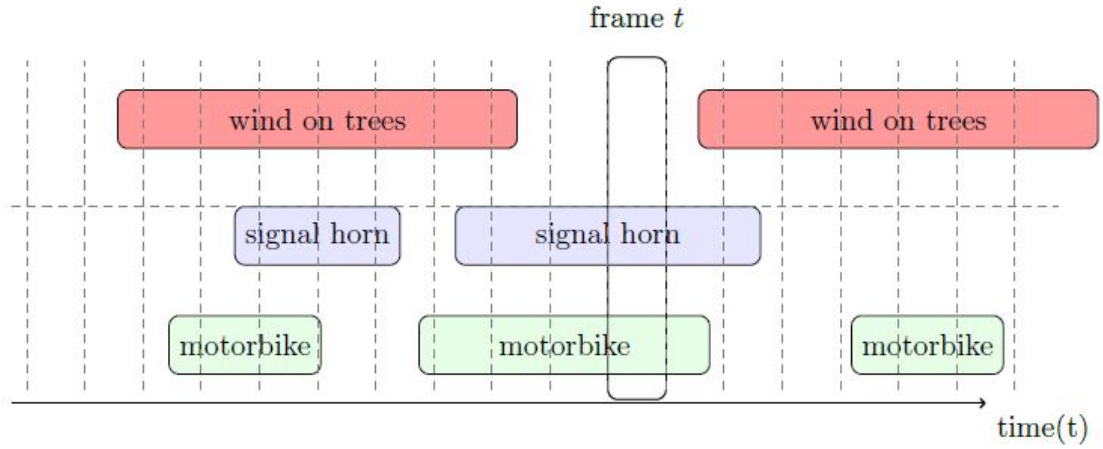


Figura 1.2: *Eventi sonori sovrapposti in una registrazione di un ambiente reale [4]*

1.3 Obiettivi e contributo della tesi

L'obiettivo principale di questa tesi è realizzare un dispositivo integrato in grado di rilevare degli eventi sonori ambientali con le seguenti caratteristiche:

- Presenza di diversi sensori per raccogliere dati da analizzare per garantire la sicurezza delle persone.
- Capacità di pre-processamento locale dei dati raccolti per convertirli in eventi.
- Capacità di elaborazione sia software (microcontrollore) che hardware (FPGA).
- Dimensione adatta per poter essere installato in un'automobile, in una posizione con accesso sia interno che esterno.
- Capacità di auto-alimentarsi, anche con funzionalità ridotte, con solo l'energia prelevata dall'ambiente.

- Inclusione di un modem cellulare per la trasmissione degli eventi.

Le principali domande di ricerca cui si cerca di rispondere durante lo sviluppo del progetto sono elencate di seguito. Una volta scelta la piattaforma che soddisfa i requisiti precedentemente elencati, si valutano preliminarmente le capacità e i limiti della stessa mediante la realizzazione di diversi algoritmi di elaborazione dell'audio al fine di valutare prestazioni, energia dissipata e occupazione di memoria. Si indaga sul tipo di dati prelevati e sulla possibilità di pre-processamento a bordo di questi. Ciò è dovuto al fatto che il rilevamento di eventi sonori richiede molti più dati rispetto, ad esempio, alla misura del livello di pressione sonora. La risorsa più comune per la gestione e memorizzazione di grandi quantità di dati è basata sul cloud, costituito da server, infrastrutture e basi di dati che offrono flessibilità e operazioni in tempo reale. Tuttavia, ciò non è in molti casi sufficiente a causa delle seguenti problematiche [2]:

- Mobilità: le condizioni di comunicazione cambiano con la posizione del dispositivo rendendo difficile la comunicazione con il sistema centrale;

- Latenza: i tempi di trasferimento e comunicazione con il cloud non sono fattibili in applicazioni che richiedono operazioni in tempo reale;

- Scalabilità: l'aumento del numero di dispositivi connessi peggiora la qualità di comunicazione;

- Limiti di elaborazione: i dispositivi possono avere delle limitazioni per elaborare ed interpretare i dati localmente;

- Limiti di potenza: la potenza necessaria per inviare e ricevere informazioni è rilevante. La maggior parte dei dispositivi sono alimentati con batterie, ciò implica che la comunicazione non può essere sempre attiva e sono necessarie soluzione a basso consumo;

Per questo motivo sono presenti vari metodi per codificare l'informazione da inviare al sistema centrale per la memorizzazione e la segnalazione. In Figura 1.3 sono riportati diversi approcci. Nel primo e secondo caso, il nodo sensore invia i dati audio grezzi o una rappresentazione intermedia di essi al server che effettua il rilevamento dell'evento; nell'ultimo caso il rilevamento viene effettuato a bordo ed il server riceve direttamente quest'informazione, consentendo di ridurre i costi e di raggiungere i requisiti dell'applicazione.

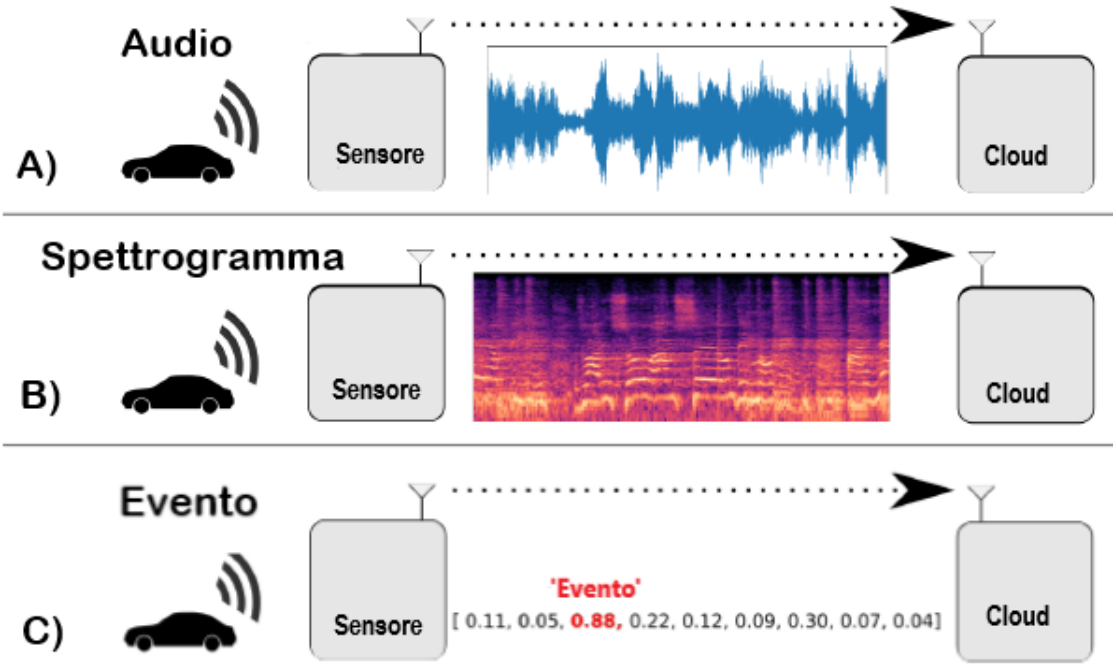


Figura 1.3: *Diverse strategie di trasmissione dei dati.*

Le tecniche utilizzate nel contesto del rilevamento degli eventi sonori sono molteplici e in questa tesi si propongono e si utilizzano i recenti metodi di *deep learning* che includono le reti neurali convoluzionali. Allo stesso tempo, poiché l'applicazione è realizzata su una piattaforma con risorse limitate è necessario trovare il giusto compromesso tra la complessità del modello e la qualità del riconoscimento.
Inizialmente si è scelto di realizzare l'applicazione esclusivamente su microcontrollore e successivamente si è valutata la possibilità di accelerazione hardware mediante FPGA (Field-Programmable Gate Array) per eseguire alcune funzioni in modo più efficiente. Infatti, l'obiettivo non è soltanto quello di ottenere un modello in grado di risolvere un problema di classificazione, ma anche quello di creare un hardware in grado di effettuare il rilevamento più velocemente possibile e consumando il meno possibile. Inoltre, avendo una disponibilità limitata dei componenti, la rete non può essere arbitrariamente grande. Pertanto, sono stati utilizzati degli strumenti per la sintesi automatica che permettono all'utente di risparmiare tempo nella realizzazione della rete neurale nel linguaggio di descrizione dell'hardware (*hardware description language* o HDL) e di concentrarsi maggiormente nei passi di ottimizzazione in modo da ottenere il modello ottimale per FPGA.

In aggiunta, si esaminano le diverse possibilità di gestione della potenza: eseguire funzioni sempre più complesse su dispositivi wireless comporta un aumento dei consumi; oltre a ciò, la richiesta di auto-alimentazione implica la necessità di fornire sempre energia al sistema. Per questo motivo una tecnica promettente consiste nel prelevare energia dall'ambiente e convertirla in energia elettrica.

Le informazioni rilevate dalla piattaforma possono essere utilizzate in svariate applicazioni: in questa tesi lo scopo principale è di utilizzare questo sistema per garantire la sicurezza pubblica attraverso il riconoscimento di eventi sonori di allarme come un incidente fra veicoli, un'esplosione e altre minacce che possono essere identificate e segnalate automaticamente alle autorità competenti. Per questo motivo, un altro quesito fondamentale per la tesi riguarda la scelta delle classi di eventi sonori da riconoscere.

La principale novità di questa tesi consiste nel realizzare un algoritmo di rilevamento di eventi sonori in software utilizzando un microcontrollore, una piattaforma a risorse limitate. Gli algoritmi presenti in letteratura per questo obiettivo sono complessi, anche perché basati su metodi di deep learning sofisticati che permettono di raggiungere un'accuratezza elevata. È stato pertanto necessario analizzare la complessità e le prestazioni del modello sulla piattaforma ed introdurre delle tecniche di ottimizzazione ed accelerazione delle parti critiche della catena per la realizzazione di questa applicazione.

1.4 Sintesi della tesi

Questa tesi è organizzata nel seguente modo. Le informazioni sullo stato dell'arte riguardanti la rappresentazione del suono, il rilevamento degli eventi sonori e le tecniche di classificazione sono riportate nel Capitolo 2. Il Capitolo 3 descrive il lavoro preliminare riguardante alcuni algoritmi di elaborazione dell'audio fino ad arrivare al Capitolo 4 in cui si presenta la realizzazione del modello per il rilevamento degli eventi sonori riportando i risultati ottenuti e le ottimizzazioni introdotte. Nel capitolo 5 si riportano le conclusioni della tesi, mostrando quali risultati sono stati raggiunti e i punti rimasti aperti per i lavori futuri.

Capitolo 2

Stato dell'arte

Al fine di comprendere il sistema realizzato, in questo capitolo si forniscono informazioni di contesto e di ripasso dello stato dell'arte riguardo al rilevamento di eventi sonori, all'estrazione delle caratteristiche audio e alle tecniche di classificazione.

Il punto di partenza è lo schema generale di un sistema di classificazione. I termini *rilevamento* e *classificazione* di un evento sonoro sono utilizzati in modo interscambiabile: La classificazione si riferisce all'assegnazione di un'etichetta dell'evento sonoro presente, mentre il rilevamento indica la presenza dell'evento all'interno del segmento audio. Ciascuna fase sarà commentata in dettaglio al fine di comprendere il metodo adottato nei capitoli successivi.

2.1 Struttura generale del sistema di rilevamento dell'evento sonoro

In Figura 2.1 è riportato un diagramma a blocchi che sintetizza il flusso delle operazioni utilizzate dalla letteratura per questo scopo.
Il primo passo è rappresentato dal set di dati che può essere creato registrando scene audio reali oppure utilizzando le registrazioni realizzate in occasione delle competizioni organizzate dalla comunità DCASE [5]. Segue una fase di pre-elaborazione dei dati durante la quale si estraggono le caratteristiche di interesse che saranno trasferite al classificatore.

2.2 Rappresentazione del suono

L'elaborazione dell'audio avviene nel dominio digitale in modo da poter trattare il segnale senza degradazione. Si considera una sequenza di campioni spaziati di una

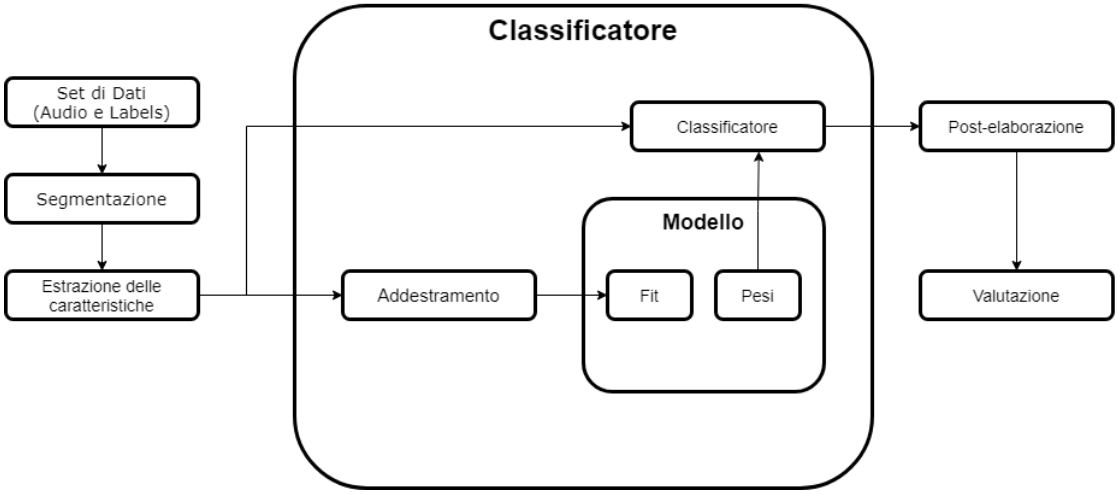


Figura 2.1: *Tipico flusso di operazioni di un sistema di classificazione.*

quantità pari al periodo di campionamento. La frequenza relativa a questo periodo è nota come frequenza di campionamento che può variare da 8 kHz fino a 44.1 kHz. La risoluzione dei campioni è solitamente di 16 bit, anche se sono presenti applicazioni ad alta fedeltà fino a 24 bit [6].

2.2.1 Segmentazione del segnale audio

Nei sistemi di elaborazione del suono in tempo reale il segnale viene suddiviso in segmenti in modo da elaborare porzioni più piccole e produrre prima il risultato invece di generarlo solo alla fine. Tuttavia, quest'operazione causa degli effetti indesiderati poiché crea delle discontinuità. Per risolvere questo problema dovuto alla segmentazione dell'audio si utilizza la tecnica della sovrapposizione, per cui ogni segmento contiene parte del segmento precedente e parte del nuovo. In questo modo, le caratteristiche che sono presenti nel punto di discontinuità vengono considerate nel successivo segmento sovrapposto. La percentuale di sovrapposizione definisce la quantità del precedente segmento che viene ripetuta in quello successivo. Sovrapposizioni del 25% e 50% sono usate comunemente. La Figura 2.2 illustra un esempio di segmentazione di un segnale audio che causa discontinuità nel segnale di uscita.
Il problema delle discontinuità peggiora con l'utilizzo di finestre temporali in quanto le estremità del segmento in analisi sono attenuate ulteriormente come accade quando si analizza il segnale nel dominio della frequenza.

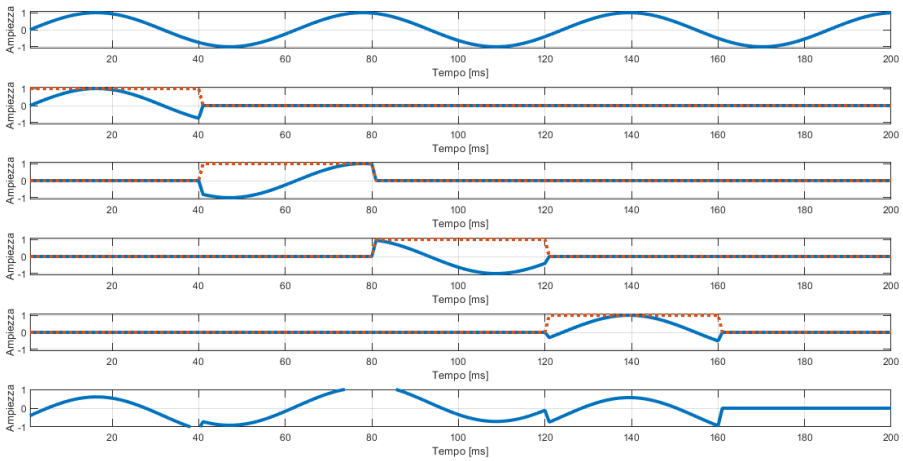


Figura 2.2: *Esempio di un segnale audio diviso in segmenti con conseguenti discontinuità sul segnale ricostruito.*

La trasformata discreta di Fourier (DFT) [7] è usata per trasformare un segnale dal dominio del tempo a quello della frequenza, assumendo che il segnale sia periodico poiché tutti i segnali periodici possono essere rappresentati come una somma infinita di sinusoidi. Nella realtà, però, si può usare solo un numero finito di componenti, generalmente 2048. Considerando ad esempio una frequenza di campionamento di 44.1 kHz, in questo caso, la dimensione del segmento è intorno a 46.4ms (2048/44100); ciò mette in risalto il trade off tra risoluzione nel tempo e in frequenza; infatti, maggiore è il numero di componenti, migliore è la risoluzione in frequenza, ma minore è la risoluzione nel tempo. Formalmente la DFT è definita come nell'equazione(2.1):

$$X_{\mathrm{k}} = \frac{1}{N} \sum_{n=0}^{N-1} x_{\mathrm{n}} e^{2nk\frac{n}{N}} \tag{2.1}$$

per $K = 1...N$, dove N è il numero di campioni contenuti nella finestra, x_n è il campione n-esimo del segnale x nel dominio del tempo e k rappresenta l'indice delle frequenze discrete. L'uscita della DFT è un vettore di numeri complessi secondo la formula generale dell'equazione(2.2), dove $\mathrm{a_k}$ e $\mathrm{b_k}$ sono la parte reale ed immaginaria rispettivamente. Solitamente si analizza il modulo della DFT, calcolato mediante l'equazione(2.3), mentre la fase tramite l'arcotangente come riportato nell'equazione(2.4). Poiché la caratteristica della DFT è simmetrica a

N/2, l'area di interesse per ogni finestra sarà dal bin 0 fino a (N/2 - 1).

$$X_{\mathrm{k}} = a_{\mathrm{k}} + \mathrm{j}\, b_{\mathrm{k}} \tag{2.2}$$

$$|X_{\mathrm{k}}| = \sqrt{{a_{\mathrm{k}}}^2 + {a_{\mathrm{k}}}^2} \tag{2.3}$$

$$\angle X_{\mathrm{k}} = \arctan \frac{b_{\mathrm{k}}}{a_{\mathrm{k}}} \tag{2.4}$$

Tuttavia, la teoria della DFT assume che il segnale sia stazionario, ma la maggior parte dei segnali cambia continuamente, come l'evento sonoro che varia nel tempo. Per questo motivo, è necessario garantire che la finestra di analisi sia dimensionata in modo da poter considerare il segnale stazionario all'interno del periodo di analisi. Quindi, per segnali che tendono a variare molto velocemente, sono necessarie finestre più corte per catturare i dettagli. Ma, come accennato in precedenza, quest'operazione causa discontinuità, come riportato in Figura 2.3:

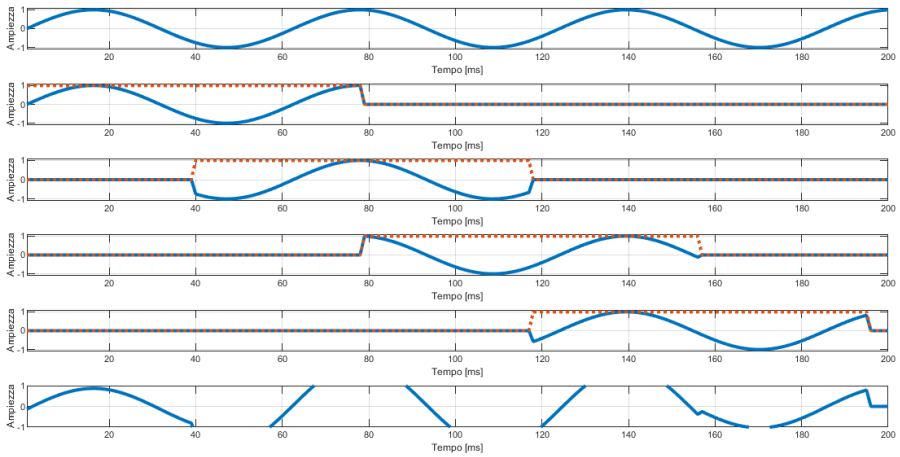


Figura 2.3: *Illustrazione di un segnale audio suddiviso in segmenti con sovrapposizione del 50%. Il segnale ricostruito presenta discontinuità tra i segmenti.*

Questo problema causa la sovrapposizione tra frequenze creandone altre che non sono presenti. Una soluzione consiste nell'utilizzare una funzione che descrive una

finestra tale da attenuare l'inizio e la fine del segmento. Una delle funzioni più comuni è la finestra di *Hamming*, definita con la formula descritta nell'equazione(2.5), dove N è la larghezza della finestra e n è l'indice del campione [8]:

$$w(n) = 0.54 - 0.46cos\left(\frac{2\pi n}{N-1}\right) \tag{2.5}$$

Un altro esempio riguarda la finestra di *Hanning*, chiamata anche *coseno rialzato* poiché può essere vista come il periodo del coseno "rialzato" per annullare i picchi negativi, in questo modo la discontinuità viene attenuata. Un'applicazione di questo tipo di finestra è riportata in Figura 2.4, in cui si può notare che le discontinuità introdotte dalla segmentazione e sovrapposizione dei segmenti sono ridotte nel segnale ricostruito in uscita.

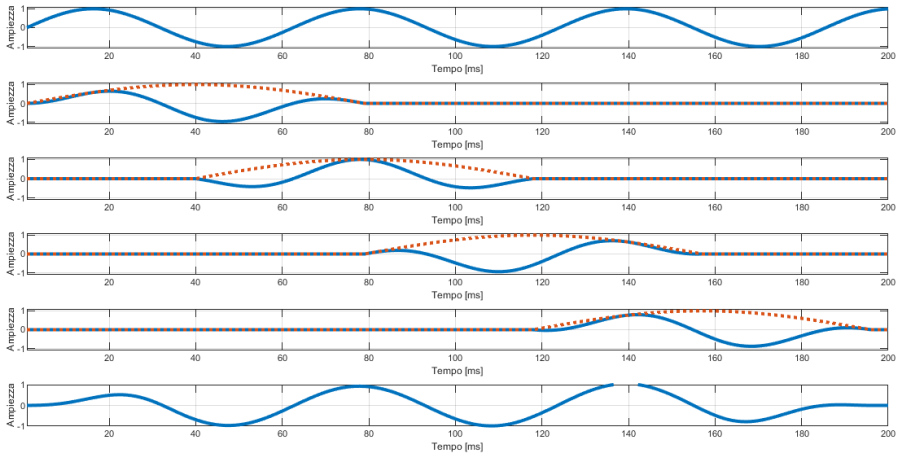


Figura 2.4: *Illustrazione di un segnale audio suddiviso in segmenti con sovrapposizione del 50%, ciascuno dei quali viene moltiplicato con la finestra di Hanning prima di ricostruire il segnale di uscita che non presenta discontinuità.*

2.2.2 Analisi dello spettrogramma

Utilizzando le tecniche descritte in precedenza si ottiene la trasformata di Fourier di breve termine (*short-time Fourier transform* o STFT) che consiste nel suddividere il segnale in segmenti parzialmente sovrapposti tramite un'operazione di finestratura e calcolare la DFT di ciascun segmento. Inserendo i coefficienti DFT di ciascun segmento in colonne separata di una matrice, la STFT può essere rappresentata come una matrice di coefficienti, dove l'indice di colonna rappresenta il tempo e l'indice

di riga è associato con la frequenza del coefficiente corrispondente. Calcolando il modulo di ciascun coefficiente, la matrice risultante può essere trattata come un'immagine, chiamata *spettrogramma*, e rappresenta l'evoluzione del segnale nel dominio tempo-frequenza [9]. L'equazione per calcolare la trasformata discreta di breve termine per ogni segmento è riportata nell'equazione(2.6)[10].

$$X(\omega, m) = STFT(x(n)) := DFT(x(n-m)\omega(n)) = \sum_{n=0}^{L-1} x(n-m)\omega(n)e^{i\omega n} \quad (2.6)$$

dove ω(n) è la finestra di lunghezza L.
In Figura 2.5 è riportato il procedimento per ottenere lo spettrogramma dal segnale audio di partenza.

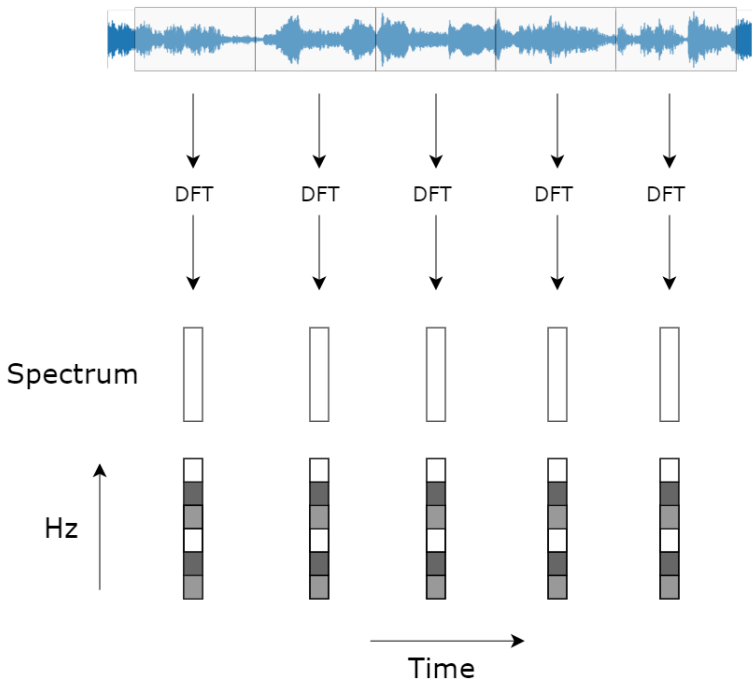


Figura 2.5: *Spettrogramma del segnale suddiviso in segmenti.*

2.2.3 Analisi dello spettrogramma di Mel

Lo spettrogramma derivante dalla STFT è chiamato anche spettro lineare perché l'ampiezza varia linearmente con la frequenza. L'intervallo di frequenze che l'orecchio umano riesce a sentire è compreso tra 20-20000 Hz, ma la relazione tra la percezione e la scala delle frequenze non è lineare [6]. Infatti, gli esperimenti

condotti sull'apparato uditivo umano dimostrano che la percezione è più discriminante alle basse frequenze rispetto alle alte frequenze. Per questo motivo è stata introdotta la scala di Mel che cerca di approssimare la risposta dell'orecchio umano in modo più fine rispetto alla scala lineare. Per convertire la scala lineare nella scala di Mel si utilizza la seguente formula(2.7):

$$M(f) = 2595 \cdot log_{10}\left(1 + \frac{f}{700}\right) \tag{2.7}$$

che permette di mappare la scala lineare alla scala di Mel basata sulla percezione dell'orecchio umano, come mostrato in Figura 2.6:

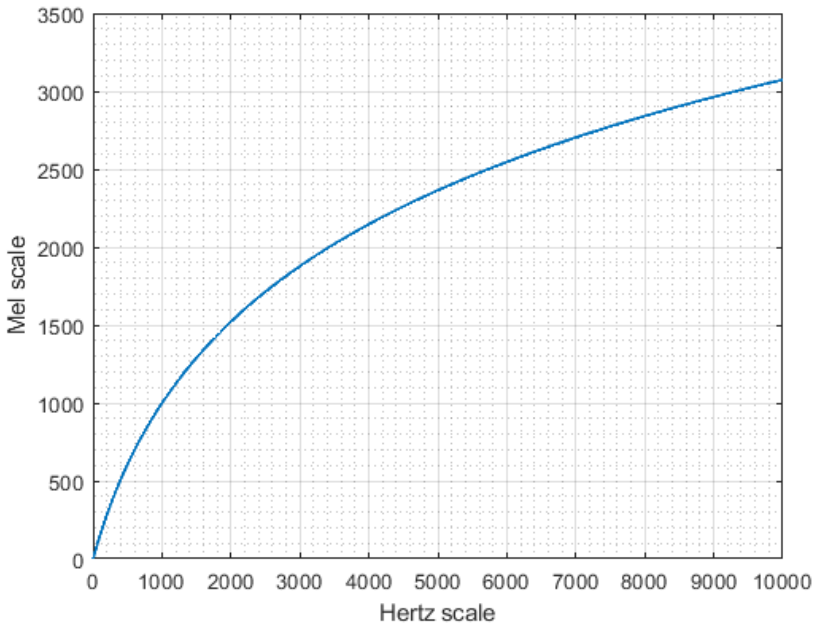


Figura 2.6: *Scala di Mel rispetto alla scala lineare.*

In questo modo si ottiene lo *spettrogramma di Mel* calcolato applicando allo spettro di potenza un banco di filtri, tipicamente 40, pesati secondo la scala di Mel. Ciascun filtro ha una forma triangolare con ampiezza unitaria alla frequenza centrale e decresce linearmente fino a zero fino ad incontrare la frequenza centrale della banda successiva, come mostrato in Figura 2.7[1].

[1] `https://haythamfayek.com/2016/04/21/speech-processing-for-machine-learning.html`

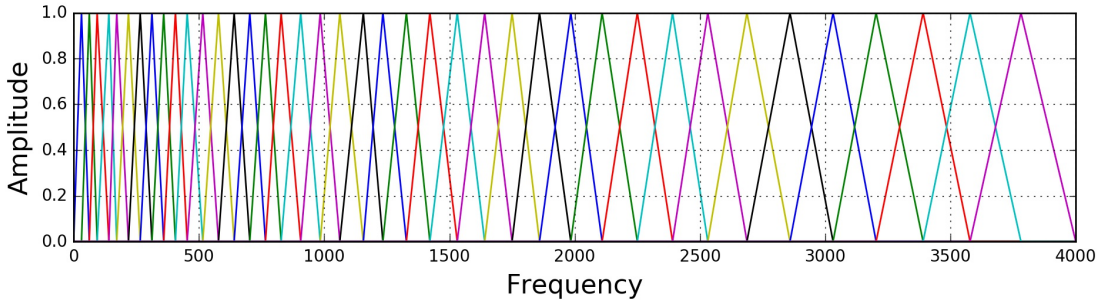


Figura 2.7: *Banco di filtri sulla scala di Mel.*

2.2.4 Analisi del cepstrum

Il concetto di "cepstrum" è stato originariamente introdotto da Bogert et al. in [11] per la caratterizzazione di echi sismici. Nel campo audio, il cepstrum è stato utilizzato inizialmente per l'analisi della voce. Le caratteristiche cepstrali derivano dal logaritmo dello spettro di potenza e catturano le caratteristiche timbriche e l'altezza di un suono. Oggi, le caratteristiche cepstrali sono largamente utilizzate in tutti i campi relativi all'estrazione di informazioni audio, come ad esempio in [12]. Il cepstrum è definito come la trasformata di Fourier del logaritmo del modulo dello spettro del segnale originale [11]. Tuttavia, nella realtà il calcolo è leggermente diverso da questa definizione, ad esempio, la seconda trasformata di Fourier è solitamente sostituita dalla trasformata discreta del coseno (DCT, *Discrete Fourier Transform*). Si tratta di una trasformata simile alla DFT, ma utilizza solo numeri reali ed è in grado di decorrelare i coefficienti del banco di filtri e ridurre la dimensionalità a 13-20 bande.

2.3 Estrazione delle caratteristiche audio

Utilizzando i metodi di analisi tempo-frequenza è possibile estrarre caratteristiche che possono fornire informazioni utili sul segnale in analisi. Quest'operazione di estrazione può essere vista anche come una riduzione dei dati sui cui si basa l'algoritmo di analisi in modo da ricavare le proprietà fondamentali dal segnale originale. Per raggiungere questo obiettivo è importante conoscere il dominio di applicazione per decidere quali sono le caratteristiche di interesse. Esistono diversi tipi di caratteristiche audio che possono essere utilizzate nell'ambito del recupero di informazioni audio [13].

Caratteristiche spettrali

Le caratteristiche utilizzate più comunemente sono quelle cepstrali, in particolare i coefficienti cepstrali di Mel (MFCC)[14]. Sono ottenuti tramite l'analisi cepstrale (2.2.4) ovvero la DCT del logaritmo dei coefficienti ottenuti dallo spettrogramma di Mel. Il nome *cepstrum* è l'anagramma di *spectrum* e indica che queste caratteristiche sono ottenute applicando la trasformata di Fourier al modulo dello spettro del segnale e sono in grado di catturare l'inviluppo dello spettro di un suono e quindi riassumere il suo contenuto spettrale.
Oltre agli MFCC, altre caratteristiche possono essere estratte, come il *flusso spettrale* ed il *centroide* [15]. Il centroide rappresenta il "centro di massa" dello spettro e misura quanto velocemente varia lo spettro tra due segmenti di audio consecutivi. Altre caratteristiche spettrali includono l'*attenuazione spettrale* che identifica la frequenza al di sopra della quale l'ampiezza dello spettro supera una determinata soglia [16] e altre proprietà dello spettro come la planarità e la pendenza.

Caratteristiche temporali

Un'altra categoria di caratteristiche è legata al dominio temporale. Una caratteristica comune è la frequenza di attraversamento dello zero che indica quante volte il segnale cambia segno, utilizzata molto per distinguere segnali vocalizzati da quelli non vocalizzati [17]. Tuttavia, le scene acustiche sono solitamente rumorose, quindi il tasso di attraversamento sarebbe molto elevato.

Caratteristiche spaziali

Nel caso di registrazioni della scena con più microfoni si possono estrarre le caratteristiche dai diversi canali per catturare le caratteristiche della scena. Si sfrutta la differenza interaurale di tempo e spazio che sono legate alla posizione della sorgente sonoro nel campo stereo.

2.4 Tecniche di apprendimento e riconoscimento

La classificazione è una tecnica di apprendimento che ha come obiettivo quello di predire in modo accurato la classe a cui il dato appartiene. Per riassumere le proprietà di eventi sonori dalle caratteristiche estratte sono utilizzati modelli statistici, tra i più importanti il modello di Markov nascosto (HMM o *Hidden Markov Model*)[18] e il metodo di combinazione gaussiana (GMM o *Gaussian mixture model*)[5]. Tuttavia, le tecniche di *Deep Learning*, termine anglosassone che letteralmente significa apprendimento profondo, hanno portato numerosi vantaggi nelle applicazioni di riconoscimento e classificazione [19] e sono le più utilizzate in letteratura [20].

2.4.1 Apprendimento

Nella fase di apprendimento lo scopo è quello di ottenere un modello ottimale in grado di associare i suoni alle classi di interesse. Il modello ha come ingressi sia le caratteristiche estratte per ogni segmento audio che le annotazioni di riferimento che rappresentano le uscite desiderate per quegli ingressi, ovvero contengono l'informazione della classe di suono assegnata a quel dato di ingresso. Il modello quindi suddivide lo spazio delle caratteristiche audio in regioni con un'etichetta assegnata. In Figura 2.8 è riportato lo schema generale della fase di apprendimento e riconoscimento del sistema.

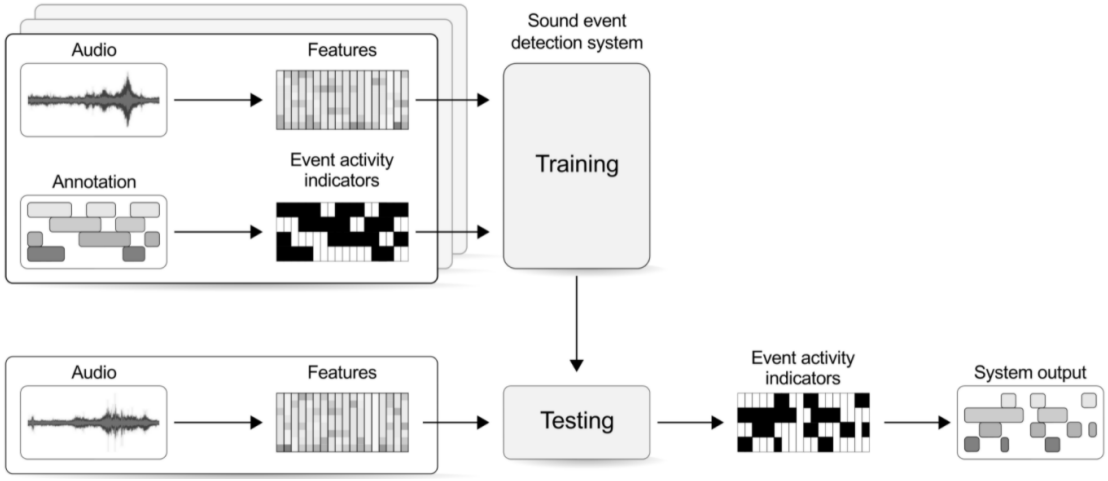


Figura 2.8: *Visione di insieme della fase di apprendimento e classificazione [21]*

Riconoscimento

Il riconoscimento è ottenuto dalla classificazione delle caratteristiche osservate in base alla regione in cui sono presenti. In questo modo, il modello è in grado di classificare i suoni in classi. Infatti, le caratteristiche acustiche entrano in ingresso al classificatore, che utilizza le uscite del modello per ottenere le probabilità di presenza di ciascuna classe per ogni segmento di ingresso. Nella fase di post-elaborazione le probabilità relative alla presenza delle classi sono convertite in uscite binarie che indicano la presenza o assenza della classe all'interno di ciascun segmento di analisi.

Classificazione

La classificazione del suono si distingue in base al numero di classi di uscita: nel caso di una sola classe, l'uscita può essere ottenuta sia selezionando la classe con la probabilità più alta per ogni segmento e calcolando la decisione con un meccanismo di votazione a maggioranza per trovare quella con il maggior numero di occorrenze (*hard voting*), sia combinando le probabilità tramite somma o media e selezionando la classe con la probabilità più alta (*soft voting*); nel caso di classi multiple si utilizza un approccio simile a quello del *soft voting* insieme ad una soglia utilizzata per individuare le classi attive (*binarization*).

2.4.2 Apprendimento profondo

Il deep learning è una sottocategoria del Machine Learning o apprendimento automatico che si avvale di modelli costituiti da più livelli di rappresentazione dell'informazione (da cui il termine "Deep"), costruiti a partire dalla semplificazione dei sistemi neurali biologici. Infatti, si basano su reti neurali organizzate in livelli caratterizzati da nodi, ogni livello è connesso al successivo tramite delle connessioni aventi un peso il cui valore indica il grado di connessione tra due nodi. I pesi sono ottimizzati tramite un processo di retropropagazione durante l'addestramento che minimizza l'errore della rete alterando leggermente il valore dei nodi di quest'ultima.
La forma più comune di apprendimento è quella *supervisionata* in cui i dati di addestramento sono associati ad un'informazione che consente di istruire il sistema per elaborare automaticamente le previsioni sulla classificazione. L'apprendimento *non supervisionato* invece prevede che il sistema riceva soltanto i dati di input senza alcun risultato atteso.
In Figura 2.9 è riportata l'analogia tra il neurone biologico e il suo modello matematico[2].
Nel caso raffigurato a sinistra il neurone riceve i segnali di ingresso dai dendriti e produce i segnali di uscita dall'assone che si dirama connettendosi ai dendriti di altri neuroni. I segnali che viaggiano lungo l'assone interagiscono con i dendriti di un altro neurone in base alla forza sinaptica che nel modello computazionale è associata ad un peso, ad esempio w_0. Il neurone calcola l'uscita come somma pesata degli ingressi e, se questa supera una certa soglia, invia un impulso lungo l'assone. Ciò è modellato tramite una funzione di attivazione che esegue un'operazione matematica sull'uscita.
Le reti neurali sono modellate come un insieme di neuroni connessi in un grafo aciclico, il caso più comune è quello di una rete *completamente connessa* (Fully connected) dove i neuroni di uno strato sono connessi con ciascun neurone del livello

[2] `https://cs231n.github.io/neural-networks-1/`

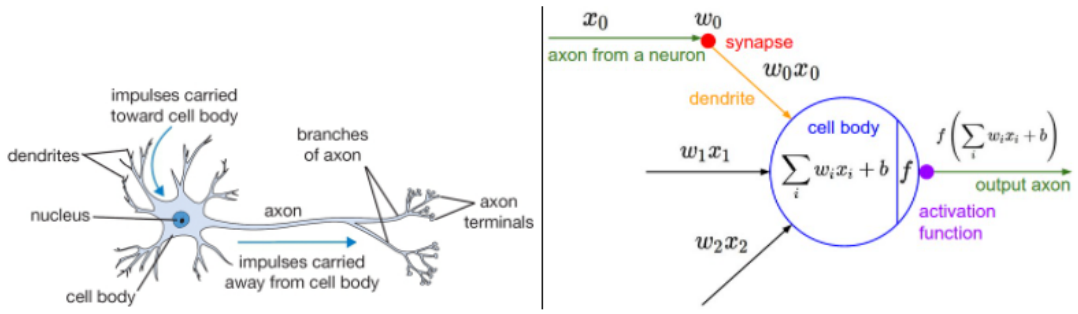


Figura 2.9: *Neurone biologico e il suo modello matematico.*

precedente. Questo tipo di architettura è chiamato anche modello di *propagazione diretta*, poiché le predizioni sono ottenute applicando i dati in ingresso al primo livello e propagando i risultati fino al livello finale [19]. Un esempio tipico ed illustrativo di questo concetto è la rete *Multi-Layer Perceptron* (MLP) raffigurata in Figura 2.10[3].

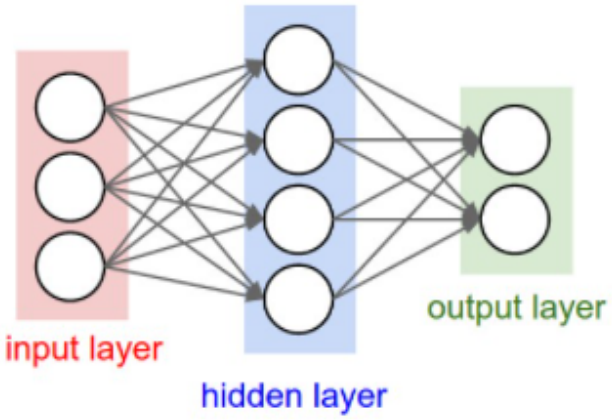


Figura 2.10: *Una rete neurale a due livelli (un livello nascosto di 4 neuroni e un livello di uscita con 2 neuroni), e tre ingressi.*

[3] `https://cs231n.github.io/neural-networks-1/`

2.4.3 Funzioni di attivazione

Per modellare la relazione non lineare esistente tra ingresso ed uscita del neurone si applica una funzione non lineare. Le più comuni sono la funzione *Sigmoide* e la *ReLU* (unità lineare rettificata), riportate rispettivamente in Figura 2.11 e 2.12.

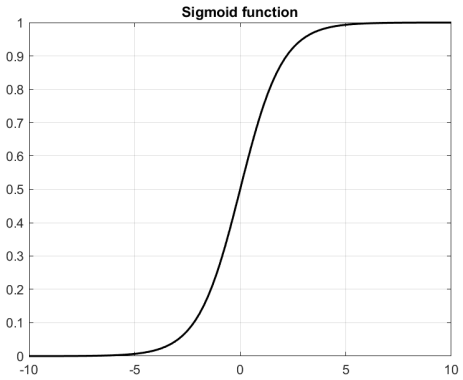


Figura 2.11: Funzione Sigmoide

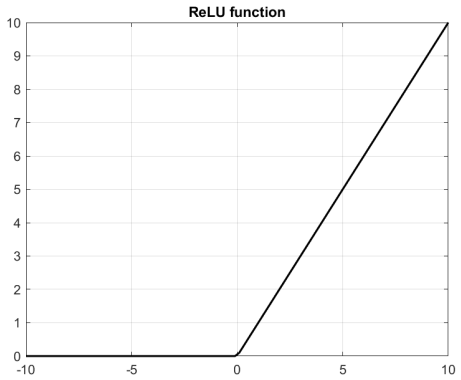


Figura 2.12: Funzione ReLU

Considerando la funzione *sigmoid* definita dall'equazione(2.8)

$$\sigma = \frac{1}{1 + e^{-z}} \tag{2.8}$$

l'uscita del singolo neurono sarà come nell'equazione(2.9):

$$a = \sigma(w_0 a_0 + w_1 a_1 + ... + w_\mathrm{n} a_\mathrm{n} + b) \tag{2.9}$$

dove b è il *bias*, una soglia che permette al neurone di non attivarsi se questa non viene superata.

2.5 Reti neurali convoluzionali

Tra le architetture di apprendimento più conosciute ed usate nelle precedenti competizioni (DCASE) [20] spiccano le reti neurali convoluzionali (*convolutional neural networks* o CNN), un modello di rete neurale costituita da livelli di convoluzione efficaci per l'elaborazione di immagini. Inoltre, può essere applicata anche all'audio trasformando la forma d'onda in un'immagine, ovvero lo spettrogramma.
Questa tipologia di rete è adatta per questo tipo di applicazioni, a differenza delle reti completamente connesse, le quali hanno una scalabilità minore all'aumentare delle dimensioni delle immagini. Infatti, in una rete come la MLP, ogni neurone ha un numero di pesi uguale al prodotto tra altezza, larghezza e profondità

dell'immagine, rendendo non realizzabile la rete per grandi dimensioni. La rete convoluzionale invece ha il vantaggio di disporre i neuroni in un volume 3D e con connessioni condivise in modo da ridurre i parametri necessari (pesi e bias). La differenza tra i due approcci è riportata in Figura 2.13[4].

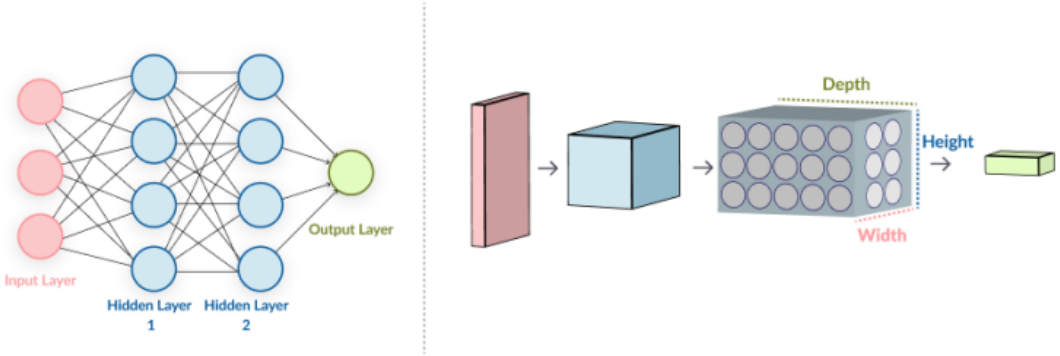


Figura 2.13: *Differenza tra una rete completamente connessa e una rete convoluzionale.*

Un'altra differenza riguarda il processo di elaborazione dell'immagine, in quanto le CNN non analizzano interamente l'immagine, ma la scompongono applicando dei filtri che permettono di riconoscere degli andamenti specifici. L'architettura delle reti convoluzionali differisce dal modello comune poiché gli strati intermedi non sono completamente connessi. La struttura tipica di una rete convoluzionale è rappresentata in Figura 2.14.

Il livello di ingresso riceve i dati che vengono forniti alla rete ed è dimensionato in base alle caratteristiche specifiche del dato in ingresso. Segue il livello convoluzionale, da cui prende il nome la rete, che svolge appunto un'operazione di *convoluzione* in modo da riconoscere delle caratteristiche specifiche nell'immagine. Possono essere presenti più livelli convoluzionali a seconda della complessità delle caratteristiche da riconoscere. Successivamente il livello di *pooling* è in grado di ridurre la dimensionalità eliminando ciò che è superfluo. Infine, il livello di uscita necessario per la classificazione è costituito da un livello completamente connesso che collega tutti i neuroni in modo da classificare le caratteristiche individuate dai livelli precedenti.

[4] `https://cs231n.github.io/convolutional-networks/`

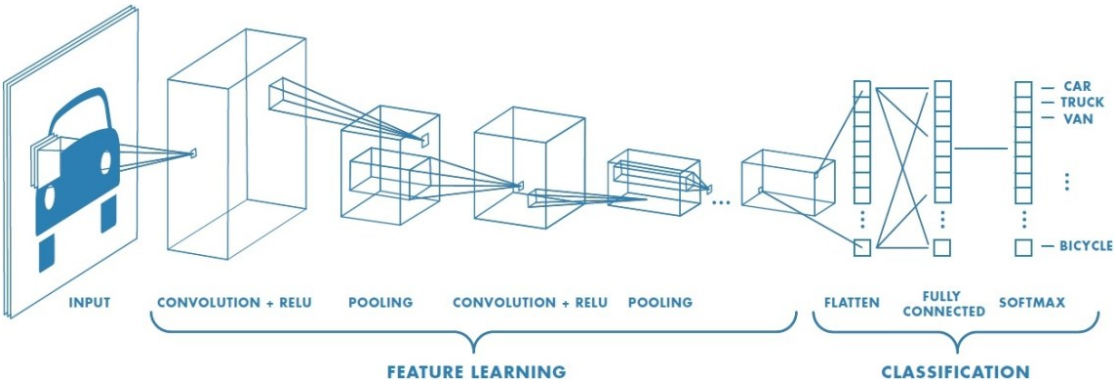


Figura 2.14: *Architettura di una rete neurale convoluzionale [22].*

2.5.1 Livello di convoluzione

Come detto in precedenza, il livello convoluzionale effettua un'operazione principale, ovvero la convoluzione, che consiste nel prodotto di due funzioni, una ritardata rispetto all'altra. Quest'operazione può essere considerata come l'applicazione di un filtro che consiste in una matrice (*kernel*) di dimensioni minori rispetto all'immagine su cui è applicato. L'applicazione della matrice sull'immagine è un prodotto scalare tra i pesi del *kernel* ed il cosiddetto *campo recettivo*, sottoinsieme dell'immagine avente le stesse dimensioni del kernel. Per effettuare la convoluzione sull'intera immagine, il filtro viene spostato di una quantità pari al passo (*stride*) fino al raggiungimento del bordo. Alla fine della scansione si ottiene un'altra matrice chiamata *feature map* che mette in evidenza una particolare caratteristica dell'immagine; perciò, per effettuare il riconoscimento si utilizzano molteplici filtri contemporaneamente, ciò produrrà in uscita un tensore la cui profondità è pari al numero di filtri utilizzati. In Figura 2.15[5] è possibile notare l'operazione di convoluzione tra il kernel ed il campo recettivo.
Ogni filtro coinvolge un numero di valori sinaptici (pesi) pari alla dimensione del kernel, il numero dei parametri non dipende dalla grandezza dell'immagine, ma può essere calcolato secondo l'equazione(2.10):

$$(F \times F \times C + 1) \cdot K \tag{2.10}$$

dove F è la dimensione del filtro, C la profondità dell'immagine e K il numero di filtri.

[5] `https://stanford.edu/~shervine/teaching/cs-230/cheatsheet-convolutional-neural-networks`

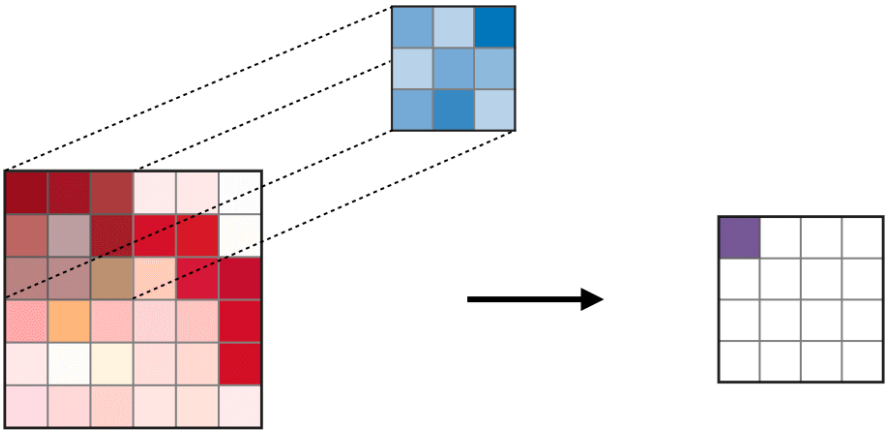


Figura 2.15: *Convoluzione tra kernel e campo recettivo*

2.5.2 Livello di Pooling

L'uscita generata dai livelli convoluzionali permette di avere delle informazioni più dettagliate e consistenti rispetto all'immagine di partenza. Tuttavia, nella maggior parte delle applicazioni non è necessario avere un'elevata risoluzione dell'immagine, per cui è possibile selezionare solo l'informazione utile riducendo le dimensioni delle *feature map* per le elaborazioni successive. Infatti, il livello di pooling ha come scopo quello di ridimensionare le *feature map* lasciando inalterate le caratteristiche di interesse. Esistono diversi meccanismi di pooling, il più utilizzato è il *max pooling* che consiste nell'applicare un filtro solitamente di dimensione 2×2, che si muove sulla *feature map* con un passo della stessa lunghezza. Il filtro di pooling individua i campi recettivi e trova il valore massimo per ognuno, il cui risultato è visibile in Figura 2.16[6]. Nel caso di *average pooling* invece l'uscita è la media dei valori di ingresso.

2.5.3 Livello completamente connesso

Infine, sono presenti i livelli completamente connessi che effettuano la classificazione, quindi generano l'uscita della rete neurale. Questo strato di livelli riceve in ingresso la matrice manipolata dai livelli precedenti e produce un vettore di dimensione N che corrisponde al numero di classi di cui si vuole effettuare la predizione. Analizzando le correlazioni presenti nelle matrici, viene calcolata la probabilità relativa

[6] `https://stanford.edu/~shervine/teaching/cs-230/cheatsheet-convolutional-neural-networks`

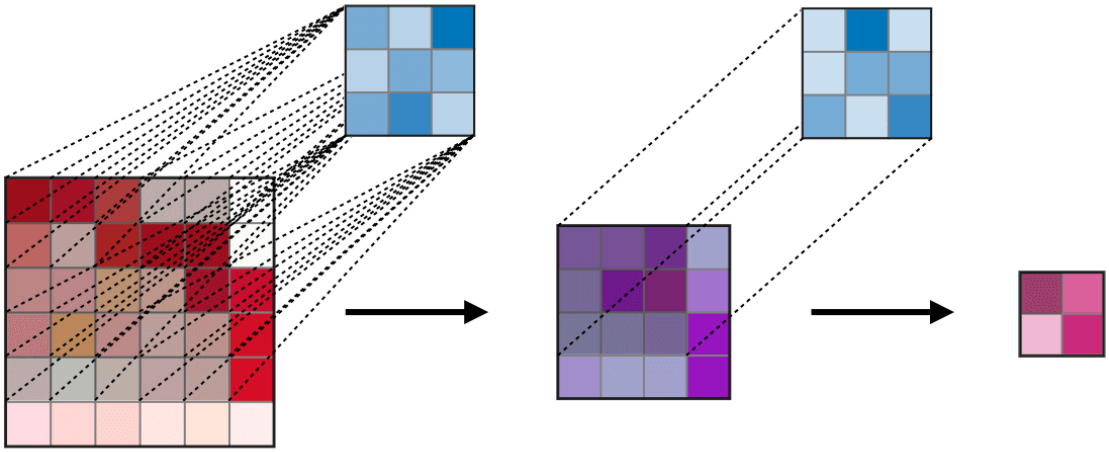


Figura 2.16: *Esempio di max pooling*

di appartenenza ad ogni classe.

Riassumendo, le reti neurali convoluzionali sono dei modelli con caratteristiche particolari e sono notevolmente utilizzati nel campo audio, in particolare per la classificazione ed il riconoscimento degli eventi sonori.

2.5.4 Modelli basati sullo spettrogramma

L'utilizzo delle reti convoluzionali è diffuso in molti articoli, in particolare tra quelli delle competizioni della comunità DCASE. Gli approcci più comuni sono basati sugli spettrogrammi e in particolare sullo spettrogramma di Mel e sui coefficienti cepstrali (MFCC).

L'articolo [20] del 2016 illustra la prima applicazione di una rete convoluzionale che riceve in ingresso due immagini, una relativa allo spettrogramma di Mel e l'altra costituita dalla sua derivata prima, formando una rappresentazione a due canali che combina informazioni sia statiche che dinamiche. Il modello è basato su due livelli convoluzionali, il primo di dimensione 57x6 (frequenza x tempo) e il secondo 1x2 seguito da un livello completamente connesso con 2000 nodi ed infine un ultimo livello che restituisce le probabilità della classificazione. Le prestazioni migliorano da 72.6% del modello di partenza basato sul classificatore GMM a 76% per il modello proposto. Inoltre, è stato scoperto che il metodo probabilistico e finestre temporali più lunghe permettono di ottenere migliori prestazioni.

Un altro esempio è riportato nell'articolo [23] in cui si confrontano diversi modelli di classificazione utilizzando sequenze con lunghezze diverse. Si dimostra che una rete CNN a due livelli permette di raggiungere un' accuratezza superiore (86%) rispetto ad una rete completamente connessa a due livelli, a una rete CNN con un livello e ad un sistema basato sul GMM.

Nell'articolo [24], invece, sono analizzate le prestazioni di diverse architetture di reti convoluzionali con profondità variabile per ottenere un modello con bassa complessità. Utilizzando come caratteristiche di ingresso gli MFCC si raggiunge un'accuratezza di circa 95%.

2.5.5 Addestramento di una rete neurale

L'obiettivo del sistema di classificazione è di effettuare una corretta predizione con dati sconosciuti. Per questo è necessario svolgere una fase di addestramento il cui scopo è di determinare i parametri descritti precedentemente (pesi e bias), che permettono di calcolare i valori di attivazione dei neuroni di ogni livello e quindi consentono alla rete di classificare con maggiore precisione i dati in ingresso. Come già accennato, durante l'apprendimento supervisionato, oltre all'immagine, la rete riceve in ingresso anche la corrispondente etichetta di classificazione che viene confrontata con la predizione effettuata dal modello e nel momento in cui la previsione coincide con l'uscita attesa, la rete lascerà inalterati i propri pesi e bias supponendo che siano corretti per svolgere il compito assegnato. La modifica avviene qualora la predizione non fosse corretta e il modello, per migliorare l'apprendimento e adattarsi all'errore, ricalcolerà queste metriche in modo da essere più accurato nelle classificazioni successive.

I campioni presenti nel set di dati rappresentano un esempio di distribuzione dei dati, per questo motivo è necessario evitare di addestrare il modello su caratteristiche troppo specifiche che non tengono conto della generalizzazione del problema. Si parla in questo caso di *overfitting*, ovvero di adattamento eccessivo del modello che non rispetta i criteri di generalizzazione.

Il set di dati è quindi solitamente suddiviso in diversi sottoinsiemi. Il set di *training* rappresenta i dati utilizzati durante l'addestramento per ottimizzare il modello. Durante la fase di apprendimento, il procedimento di ricalcolo dei parametri viene svolto un numero di volte pari al numero di *epoche* (epochs), ovvero il numero di volte che il set di *training* viene processato dalla rete neurale.

Per valutare la generalizzazione del modello rispetto a dati sconosciuti, le predizioni sono effettuate su un set di *validazione*. Alla fine di ogni epoca, il modello

avrà individuato i pesi più adatti (fino a quell'istante di analisi) per svolgere una corretta predizione della classificazione, calcola un'accuratezza e una funzione di costo delle previsioni in base a pesi e bias utilizzati, e riceve in ingresso il set di dati di validazione che non ha ancora processato. In questo modo si valuta la bontà della fase di addestramento svolta fino a quel momento, però, diversamente dalla fase di training, non vengono alterati i pesi e i bias del modello, ma serve solo per avere una metrica utile per la validazione dei dati attraverso il calcolo dell'accuratezza e della funzione di costo.
Infine, le prestazioni finali sono valutate mediante un set di dati di *test*, che non sono mai stati utilizzati durante la fase di addestramento.

2.5.6 Funzione di costo, discesa del gradiente e algoritmo di retropropagazione

Un elemento fondamentale per la valutazione delle prestazioni del modello è la funzione di costo, solitamente corrispondente alla metrica statistica conosciuta come errore quadratico medio. Tuttavia, questa funzione è uno strumento necessario ma non sufficiente per giungere a delle conclusioni sul risultato dell'apprendimento. Per avere un'informazione più completa in merito alla qualità di apprendimento della rete è necessario aggiungere due tecniche essenziali, ovvero la discesa del gradiente e l'algoritmo della retropropagazione.

Il metodo della discesa del gradiente [25] è un algoritmo iterativo di ottimizzazione in grado di individuare il valore minimo di una funzione a più variabili. Un esempio di funzione di costo a due variabili è riportato in Figura 2.17.
Tuttavia, una generica funzione può dipendere da molte variabili, pertanto, trovare il minimo analiticamente sarebbe molto complicato. Nella realizzazione teorica si parte da un punto casuale all'interno dello spazio multidimensionale e si procede iterativamente fino al raggiungimento del punto più basso della funzione di costo in modo da garantire delle previsioni più accurate. Questa tecnica è applicata agli ingressi della funzione di errore (pesi e bias) ed effettua l'aggiornamento dei pesi nella direzione negativa del gradiente in modo da minimizzare la perdita.

L'algoritmo di retropropagazione rappresenta il metodo di correzione dell'errore con cui si calcola il gradiente e si modificano i valori sinaptici (pesi e bias) del modello al fine di minimizzare la funzione di costo. In base alla differenza tra il risultato ottenuto e quello atteso che ci fornisce una misura di errore della rete, l'algoritmo adatta i pesi attraverso la propagazione a ritroso dell'errore. L'algoritmo è suddiviso in due step [25]: il primo è il *forward pass* in cui gli ingressi stimolano l'attivazione dei neuroni dei diversi livelli e si propagano fino all'uscita dove viene calcolato l'errore; il secondo è il *backward pass* nel quale l'errore calcolato viene

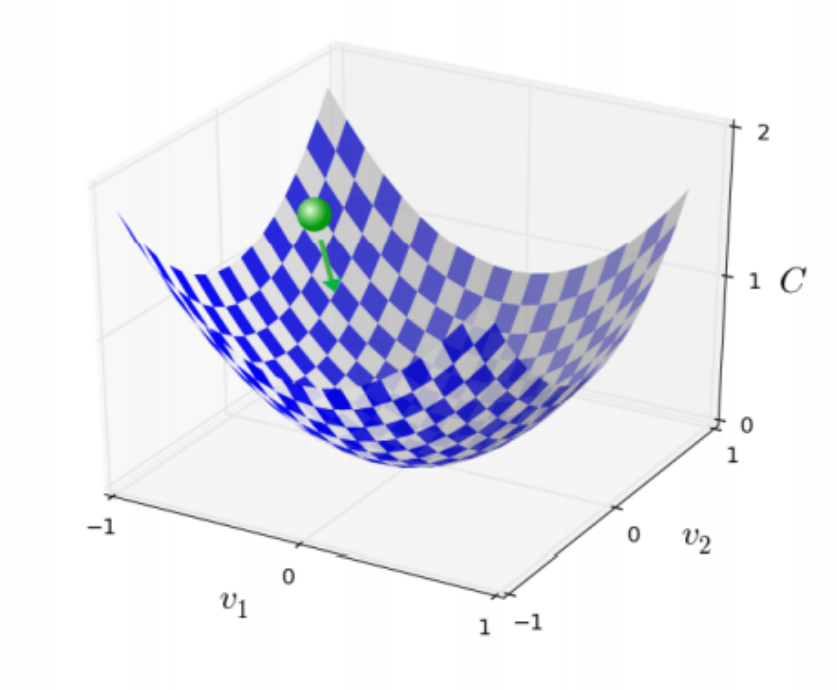


Figura 2.17: *Esempio di funzione a due variabili - discesa del gradiente [25]*

propagato a ritroso e i pesi sono aggiornati in modo da raggiungere un risultato più accurato alla prossima iterazione.

2.6 Metodi di valutazione per il riconoscimento degli eventi sonori

La valutazione riguarda la misura delle prestazioni del modello sulla base di metriche selezionate per garantire l'applicabilità del modello nella vita reale. In questo modo risulta più semplice interpretare il miglioramento del metodo realizzato.

La valutazione dei metodi per il riconoscimento di eventi sonori si avvale di metriche basate sulle predizioni di uscita binarie e sulle uscite desiderate per uno specifico set di test. Le metriche più comuni sono: precision, recall, F1 score ed error rate [21]. Queste sono calcolate a partire da una serie di statistiche intermedie che includono il numero di veri positivi (TP - *true positive*), veri negativi (TN - *true negative*), falsi positivi (FP - *false positive*) e falsi negativi (FN - *false negative*) come riportato in Tabella 2.1. Per quanto riguarda la risoluzione temporale, le metriche possono essere calcolate sia globalmente accumulando le statistiche di tutte le classi oppure rispetto alla singola classe.

Classe attiva	TP	TN	FP	FN
Rilevata	✓		✓	
Reale	✓			✓

Tabella 2.1: Definizione di TP, TN, FP e FN in base ai valori di predizione e reali per un dato segmento

2.6.1 Precision

La precisione è calcolata come il rapporto tra le predizioni corrette rispetto a tutte le predizioni effettuate dal sistema.

$$P = \frac{TP}{TP + FP} \tag{2.11}$$

2.6.2 Recall

Questa metrica è calcolata dal rapporto tra le predizioni corrette rispetto a tutte le predizioni della classe.

$$R = \frac{TP}{TP + FN} \tag{2.12}$$

2.6.3 F1 Score

Si tratta della metrica più utilizzata per il riconoscimento degli eventi sonori ed è ottenuta dalla media armonica di precision e recall.

$$F1 = \frac{2 \cdot P \cdot R}{P + R} \tag{2.13}$$

Il vantaggio è che può essere applicata per il rilevamento di classi multiple, ma non tiene conto dei veri negativi.

2.6.4 Curva ROC

A differenza delle metriche precedenti, la curva ROC (*Receiver Operating Characteristic*) non è ottenuta applicando un'unica soglia per ottenere le predizioni binarie, bensì confrontando il tasso di predizioni corrette (*True positive rate o TPR*) rispetto al tasso di predizioni false (*False positive rate o FPR*). Il *TPR* coincide con il *Recall*, mentre il *FPR* è calcolato come

$$FPR = \frac{FP}{FP + TN} \tag{2.14}$$

L'area sottesa alla curva *ROC*, definita *Area Under Curve* o AUC, rappresenta le prestazioni del sistema in quanto più è grande maggiore è la capacità di distinzione tra le classi.

2.6.5 Error rate

Il tasso di errore (ER) è calcolato a partire da statistiche intermedie che tengono conto degli errori commessi nel rilevamento: insertion (I), deletions (D), e substitutions (S). Mentre i primi due corrispondono ai falsi positivi e falsi negativi, rispettivamente, S tiene conto di quando viene rilevata una classe sbagliata invece di un'altra in un dato segmento. Il tasso di errore viene calcolato quindi come la somma di queste statistiche diviso il numero di eventi considerati attivi nel dato segmento:

$$ER = \frac{I + D + S}{N} \tag{2.15}$$

Il confronto tra l'uscita del sistema e quella di riferimento può essere effettuato con diversi approcci: *segment-based* o *event-based*, ovvero basato sul singolo segmento o sull'istanza dell'evento. Nel primo caso il confronto dipende dalla risoluzione temporale del segmento audio, nel secondo caso si valutano i tempi di inizio e di fine dell'evento. La valutazione *segment-based* è riportata in Figura 2.18.

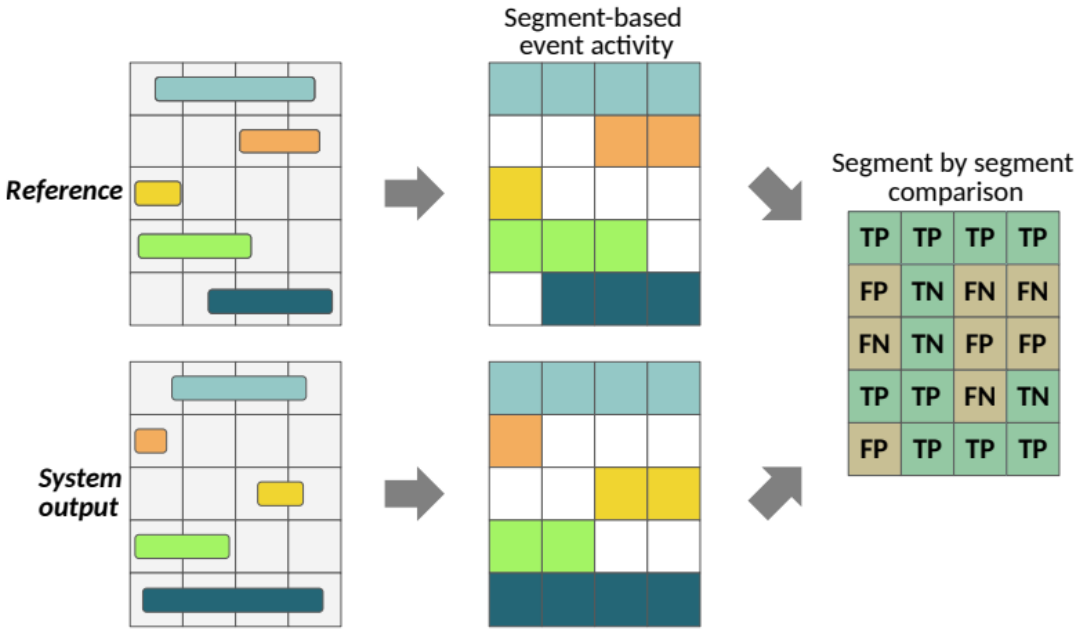


Figura 2.18: *Esempio di confronto segment-based [21]*

2.7 Reti neurali realizzate su piattaforme con risorse limitate

Un microcontrollore è un dispositivo elettronico integrato su un singolo chip, contenente CPU, RAM e memoria non-volatile (FLASH) insieme anche a periferiche per comunicare con il mondo esterno.

Periferiche comuni includono i GPIO (*General Purpose Input Output*) per ingressi/uscite digitali, convertitore analogico/digitale (ADC) per gli ingressi analogici, e connessioni seriali ad alta velocità come I2C e SPI. Per la comunicazione audio sono presenti periferiche specializzate come la I2S(*Inter-IC Sound*) o protocolli PDM(*Pulse density modulation*).

Per la realizzazione di una rete neurale su un microcontrollore è importante valutare la struttura della rete, poiché le risorse in termini di memoria, capacità computazionale e budget di potenza su queste piattaforme sono limitate. Pertanto, sono necessarie operazioni di ottimizzazione della struttura della rete che riguardano il numero di parametri e la complessità computazionale al fine di ottenere modelli efficienti e integrabili su questo tipo di piattaforma.

Queste tecniche di ottimizzazione consistono generalmente nella compressione del numero dei parametri e nella riduzione della precisione di rappresentazione numerica [26]. In questo modo si possono ottenere modelli di dimensioni inferiori che permettono una riduzione della latenza e della potenza dissipata. Bisogna considerare però che le ottimizzazioni generalmente comportano un cambiamento nell'accuratezza del modello, che deve essere considerata durante la fase di sviluppo. Inoltre, questi cambiamenti non si possono prevedere in quanto dipendono da come il modello viene ottimizzato. Pertanto, è necessario trovare il giusto compromesso per evitare di degradare la qualità della rete. Tuttavia, nonostante l'introduzione di queste tecniche, può accadere che le prestazioni non soddisfino i requisiti dell'applicazione. Si valuta quindi la possibilità di introdurre un'accelerazione hardware che consenta di eseguire in modo ancora più efficiente le parti critiche dell'algoritmo.

2.8 Accelerazione hardware

La scelta della piattaforma hardware per accelerare un'applicazione deriva da un compromesso tra prestazioni, consumo di potenza e flessibilità. Da una parte i processori general purpose come le unità di elaborazione centrale (*central processing unit*, CPU) o le unità di elaborazione grafica (*graphic processing unit*, GPU) offrono un alto grado di programmabilità ed un basso rapporto prestazioni/watt; dall'altra

parte gli ASIC (*application specific integrated circuit*) offrono migliori prestazioni per watt al costo di una bassa flessibilità ed un elevato costo di fabbricazione. Le FPGA rappresentano un buon compromesso tra queste scelte, soprattutto per l'accelerazione di reti convoluzionali. Anche se non raggiungono prestazioni migliori rispetto a GPU o ASIC, offrono una maggiore efficienza energetica rispetto alle GPU e una maggiore flessibilità rispetto agli ASIC. Nelle sezioni successive vengono quindi introdotte l'architettura, le risorse riconfigurabili ed il flusso di progetto su FPGA. Infine, si discute sull'importanza delle FPGA per accelerare le reti convoluzionali.

2.8.1 Architettura FPGA

L'architettura di base di un FPGA è riportata in Figura 2.19: si può notare una matrice di blocchi logici configurabili o CLB (*Configurable Logic Blocks*) connessi tra di loro tramite interconnessioni programmabili. L'interfaccia con il mondo esterno è costituita da blocchi di ingresso/uscita ai margini della matrice, ognuno dei quali controlla un pin che può essere configurato come input, output, bi-direzionale o *tri-state*. I CLB sono composti da celle logiche (*logic cell*) che forniscono capacità di

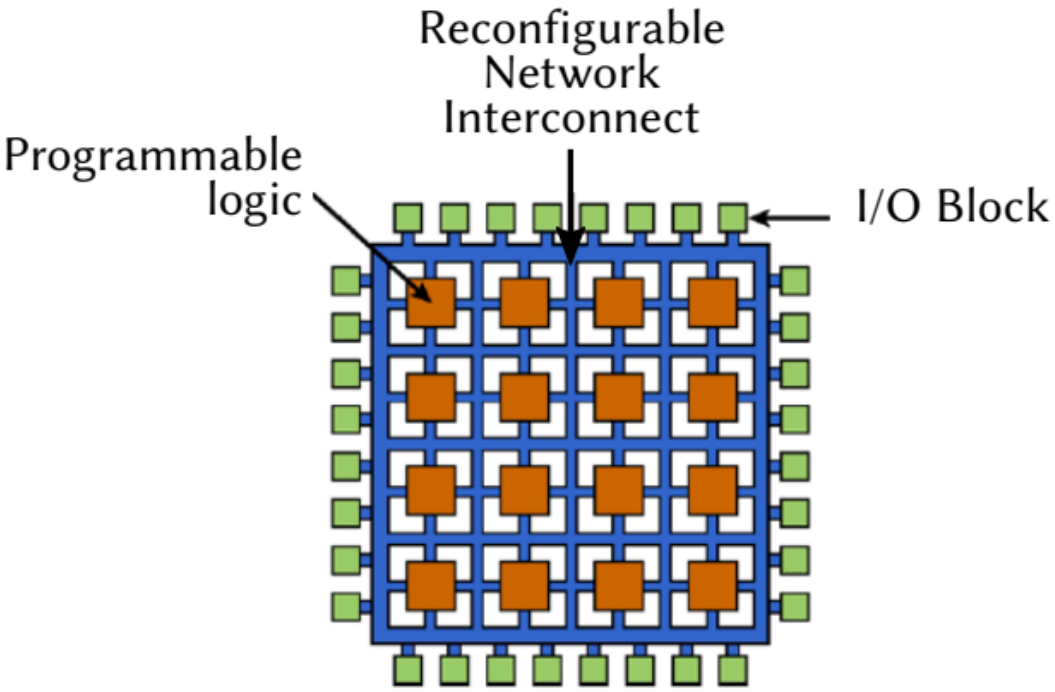


Figura 2.19: *Architettura di base FPGA*

calcolo e memoria. Una cella logica è generalmente costituita da una o più *Look-up table* (LUT) per realizzare funzioni combinatorie e un elemento di memoria per le funzioni sequenziali. Una LUT a N ingressi si comporta come una memoria a 2^N ingressi in cui viene caricata la tabella di verità per realizzare la funzione booleana di interesse. L'insieme dei CLB interconnessi tra di loro permette così di realizzare funzioni complesse. La rete di interconnessioni consiste in linee e matrici di scambio configurate tramite tecnologia programmabile per creare la connessione desiderata. Programmare le matrici di scambio consente di connettere i blocchi desiderati come riportato in Figura 2.20[7]. La rete di interconnessioni ha un notevole impatto

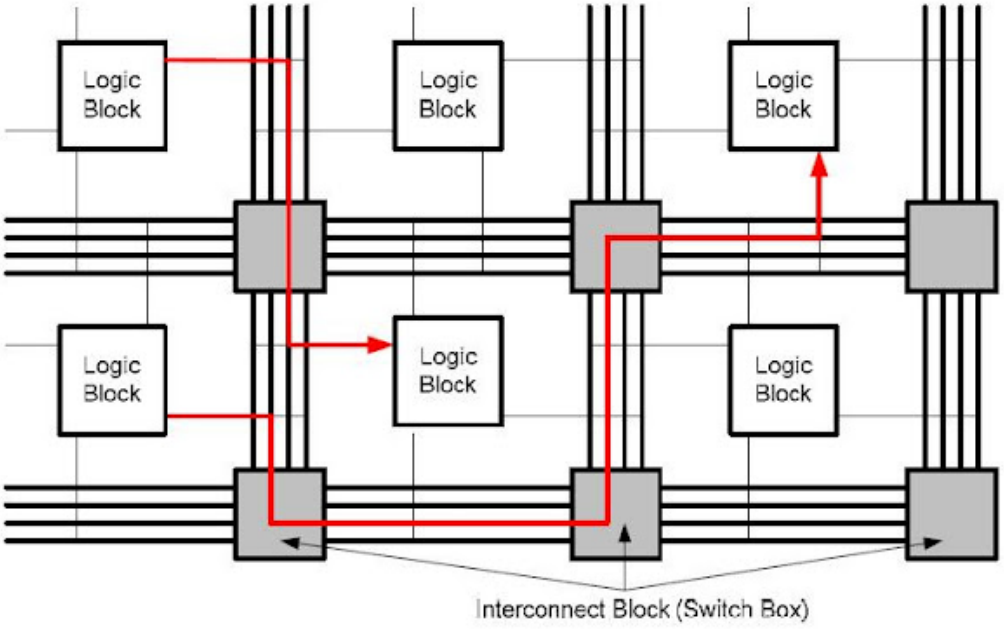


Figura 2.20: *Rete di interconnessione FPGA*

sulle prestazioni, infatti lunghe linee di comunicazioni comportano un ritardo di trasmissione maggiore, diminuendo la frequenza operativa. Per questo motivo le interconnessioni sono gerarchiche e le risorse logiche sono raggruppate in blocchi logici configurabili.
All'interno della matrice sono presenti anche altri elementi come blocchi di memoria, i DCM (*Digital Clock Manager*) che generano il segnale di clock ed altre risorse di calcolo come blocchi DSP (*Digital Signal Processor*), unità aritmetiche ottimizzate per operazioni ripetitive e applicazioni numeriche intensive, costituite da circuiti

[7] `http://fpgabeginners.blogspot.com/2012/08/what-is-fpga.html`

speciali come MAC (*Multiply and Accumulate*) per velocizzare le operazioni più comuni.

2.9 Flusso di progetto FPGA

Generalmente, per mappare l'algoritmo su FGPA si utilizzano linguaggi di descrizione dell'hardware (*hardware description language* o HDL), come VHDL o Verilog, per descrivere la struttura ed il comportamento di circuiti logici digitali. I passi principali del flusso sono riportati in Figura 2.21. La sintesi RTL (*Register Transfer*

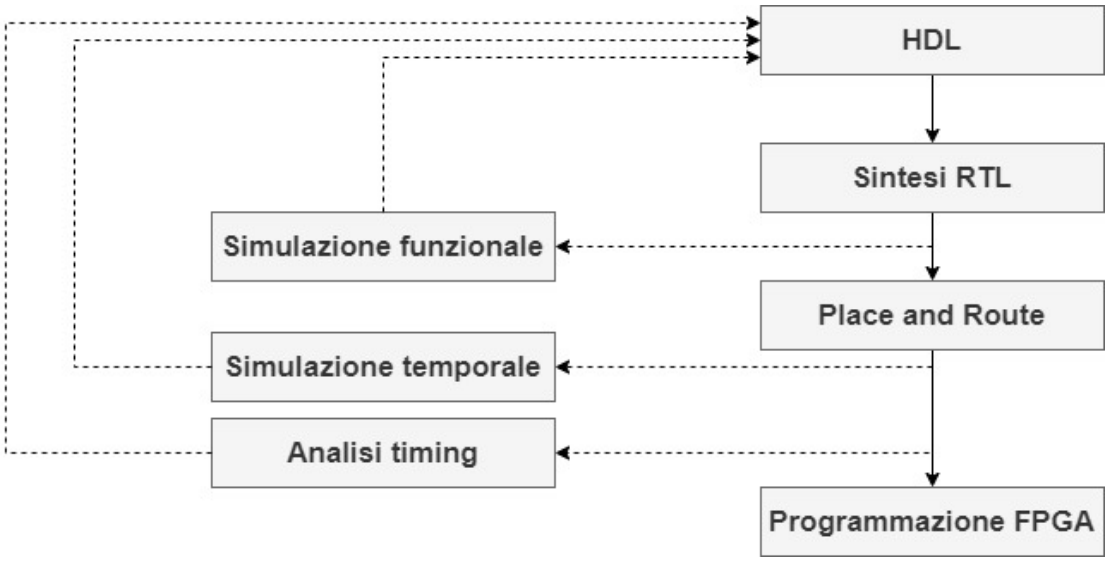


Figura 2.21: *Flusso di progetto FPGA*

Level) è il processo che traduce il codice HDL nella netlist RTL (descrizione del circuito mediante blocchi elementari). Completata la sintesi si esegue una simulazione funzionale per verificare che il comportamento descritto rispetti le funzionalità richieste. Il *Place and Route* consiste nel trasformare la rappresentazione RTL in una rappresentazione a livello di gate logici da piazzare e collegare sul dispositivo. A questo punto si effettua una simulazione per verificare che i tempi di propagazione non interferiscano con la funzionalità del sistema. Inoltre, si possono estrarre i ritardi tra i vari elementi per ottenere la massima frequenza operativa. Infine, la rappresentazione precedente viene trasformata in un file di configurazione per programmare le connessioni sul dispositivo.

2.9.1 Sintesi ad alto livello

Mappare l'algoritmo su FPGA tramite linguaggi di descrizione dell'hardware ha il vantaggio di ottenere risultati migliori in termini di efficienza, risorse e prestazioni. Tuttavia, nel caso di algoritmi complessi come il deep learning, la conversione diventa più difficile sia per la complessità computazionale che per la variabilità delle reti neurali. Per questo motivo, si impiegano molti sforzi nello sviluppo di strumenti di sintesi ad alto livello (*High level synthesis* o HLS). Gli strumenti HLS sono un'alternativa al convenzionale HDL, in quanto sono basati su linguaggi ad alto livello come C/C++, che offrono paradigmi di programmazioni più semplici, e generano a partire da questi la descrizione RTL. La compilazione si basa su direttive o pragma che sono istruzioni che vengono interpretate dal compilatore HLS per la conversione del codice sviluppato nella descrizione dell'hardware e per indicare come il codice C++ deve essere organizzato e distribuito prima della conversione a RTL.

Tra le direttive utilizzate vi è quella riguardante l'interfaccia che si occupa di specificare come le porte di interfaccia nel codice RTL sono realizzate a partire dalla definizione delle funzioni. Mentre nel codice C++ le interfacce di ingressi e uscite delle funzioni sono realizzate automaticamente nella fase di compilazione, nel progetto RTL il passaggio dei dati tra funzioni e moduli è gestito mediante dei protocolli di trasferimento.

Con la direttiva di **srotolamento**, la sintesi logica realizza lo srotolamento del loop, aumentando l'accesso ai dati ed il throughput, ossia la quantità di dati trasmessi nell'unità di tempo. Questa direttiva consente lo srotolamento totale o parziale del loop. Nel primo caso viene creata una copia del loop per ogni iterazione in modo che l'intero loop possa essere eseguito in un'unica iterazione. Nel secondo caso le istanze del loop sono replicate un numero di volte pari ad un parametro configurabile N.

La direttiva **pipeline** consente al compilatore ad alto livello di eseguire il pipelining, ovvero la riduzione dell'intervallo di inizio, espresso come numero di colpi di clock per cui una funzione o un loop può processare nuovi ingressi. Come si può notare in Figura 2.22 l'opzione di default prevede un intervallo di inizio unitario, ma può essere configurato per aumentare il throughput.

Un'altra direttiva riguarda la **partizione dei dati**, ovvero la suddivisione dei vettori in elementi più piccoli al fine di utilizzare elementi di memoria più piccoli come registri al posto di grandi blocchi di memoria. Se è utilizzato insieme alla direttiva di pipelining o srotolamento può aumentare il throughput dei dati.

Il partizionamento può essere di tre tipi: ciclico, a blocco o completo. Nel primo caso il vettore è suddiviso utilizzando l'interleaving, ovvero inserendo un elemento in un nuovo vettore prima di ritornare al primo. Il partizionamento a blocchi suddivide il vettore in parti uguali dato un parametro N. Nell'ultimo caso il vettore viene suddiviso in elementi individuali. In Figura 2.23 sono riportati i casi descritti precedentemente.

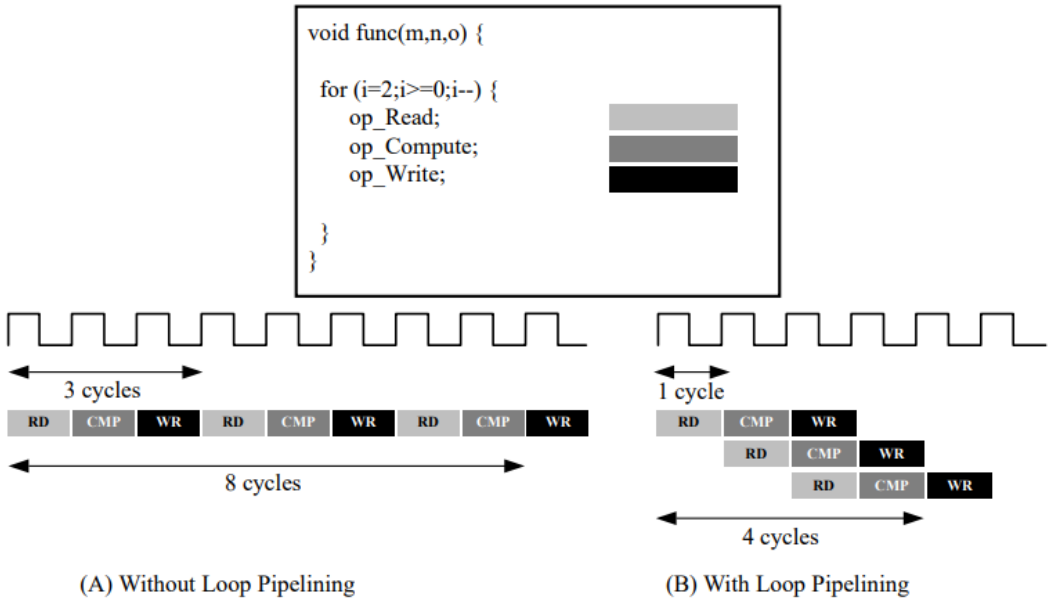


Figura 2.22: *Rappresentazione direttiva Pipelining.* [27]

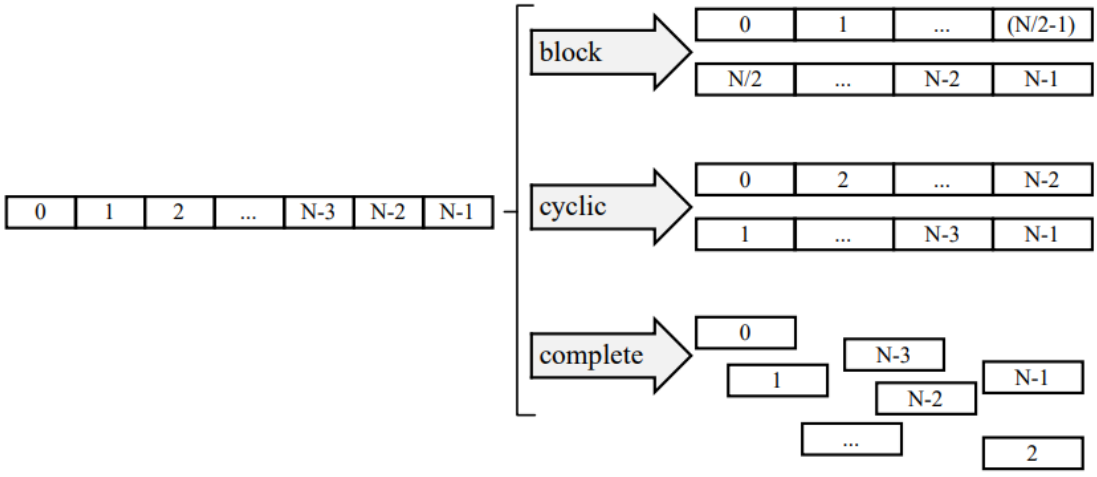


Figura 2.23: *Rappresentazione direttiva Partizionamento.* [27]

Una volta ottenuto il modello della rete neurale è necessario convertire il modello

Python in linguaggio HDL per realizzarlo su FPGA. Per fare ciò sono necessari due passi: il primo riguarda la conversione del modello in codice ad alto livello (HLS), mentre il secondo consiste nella traduzione da HLS a HDL.

2.9.2 HLS4ML: sintesi ad alto livello per il Machine Learning

Esistono numerosi strumenti in grado di realizzare il primo passo della sintesi ad alto livello, ovvero la conversione del modello della rete neurale in linguaggio ad alto livello (HLS) che può essere facilmente convertito in un linguaggio di descrizione dell'hardware. Tra questi vi è hls4ml, un compilatore sviluppato dalla collaborazione tra il Fermi National Accelerator Laboratory, l'istituto di tecnologia di Massachusetts, HawkEye360 (Herndon), CERN e l'università di Illinois a Chicago. Come riportato da [28], l'obiettivo è quello di convertire un modello Python descritto con librerie tradizionali come Keras, Tensorflow o Pytorch in codice C++ seguendo il flusso riportato in Figura 2.24.

In particolare, la conversione del modello è effettuata prendendo in ingresso un file in formato *hdf5* contenente i pesi e i bias ed un file *json* che descrive l'architettura della rete. L'uscita di hls4ml rappresenta un progetto ad alto livello che può essere convertito in HDL come VHDL o Verilog utilizzando delle funzioni che supportano il flusso integrato di Vivado HLS. Tuttavia, Vivado HLS è uno strumento di sintesi RTL che supporta la famiglia di FPGA Xilinx. In letteratura esistono altre piattaforme open-source, come PandA-Bambu (2.9.4), che supportano altri strumenti di sintesi RTL oltre a Xilinx Vivado e consentono di integrare il progetto su diverse piattaforme in base agli obiettivi dell'applicazione. Esplorando questo strumento è quindi possibile scegliere la piattaforma target in base ai requisiti del progetto che includono anche un basso consumo di potenza e di risorse oltre all'accelerazione dell'algoritmo. Nella fase di conversione del modello con hls4ml è possibile introdurre delle tecniche di ottimizzazione per ridurre il numero di risorse necessarie e la latenza. Una di queste tecniche realizzate da hls4ml riguarda la *quantizzazione* [28] che, come già detto in precedenza, consente di passare dalla rappresentazione in virgola mobile di pesi, bias ed ingressi del modello Keras ad una in virgola fissa riducendo la precisione. Questa procedura è realizzata in hls4ml associando alla variabile da convertire il tipo *ap_fixed<a,b>*, dove b rappresenta il numero di bit utilizzati per convertire la parte intera, e a corrisponde al numero totale di bit.

Un'altra tecnica è chiamata *parallelizzazione* [28] che rappresenta il numero di volte in cui una risorsa è utilizzata per effettuare un'operazione. Questo dipende da un parametro configurabile, il fattore di riuso, e il grado di parallelizzazione è inversamente proporzionale al suo valore. Per spiegare meglio il concetto, in

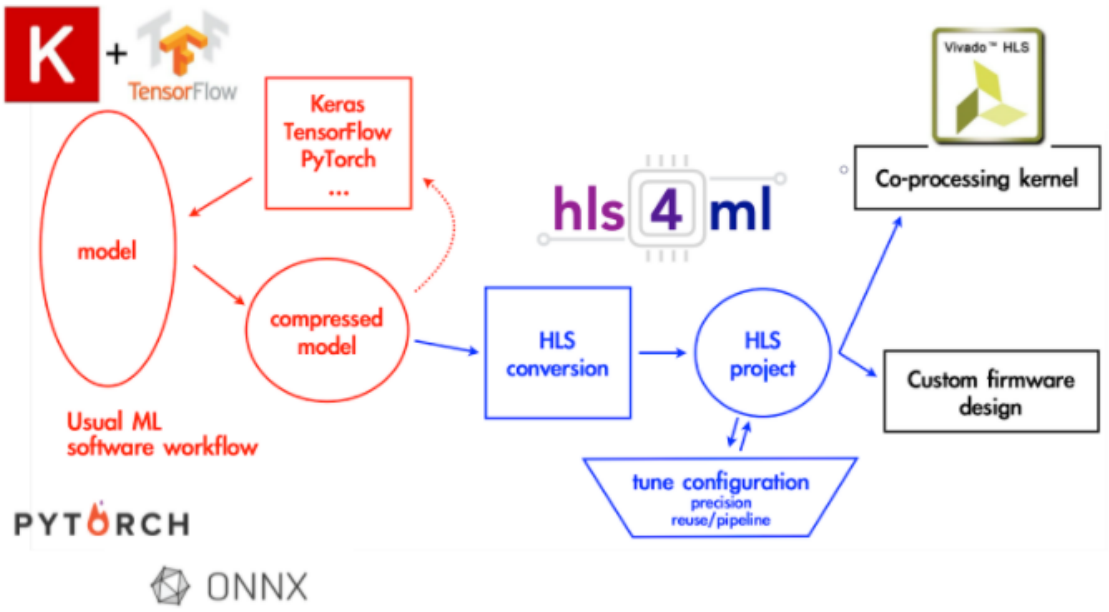


Figura 2.24: *Flusso hls4ml per la conversione del modello della rete neurale nell'implementazione su FPGA; la parte a sinistra in rosso descrive la creazione e ottimizzazione del modello utilizzando infrastrutture di machine learning prima di hls4ml; la parte in blu al centro riassume le operazioni effettuate da hls4ml; infine, a destra in nero sono illustrati i passi per esportare ed integrare il progetto HLS in hardware.* [28]

Figura 2.25 è riportato un esempio in cui se il fattore di riuso è pari ad uno si utilizza il massimo numero di risorse per effettuare la computazione che diventa completamente parallela; all'aumentare del fattore di riuso si utilizzano un numero di risorse inversamente proporzionale allo stesso con il vantaggio di ridurre la complessità del calcolo, ma con un conseguente aumento della latenza. Per tale motivo è necessario configurare questo parametro per trovare il giusto compromesso tra latenza e numero di risorse.

2.9.3 Vivado HLS

Dopo aver considerato la conversione del codice Python in codice HLS e le possibili ottimizzazioni, è possibile spostare l'attenzione alla conversione del codice HLS in codice HDL. Vivado High-Level Synthesis (Vivado HLS) è uno strumento in grado di produrre la microarchitettura RTL a partire dal codice HLS scritto in C/C++. Quest'operazione può essere compiuta in modo automatico dato che hls4ml ha integrato nell'ultima versione Vivado HLS come back-end, per cui è possibile eseguire il flusso di Vivado utilizzando direttamente le funzioni della

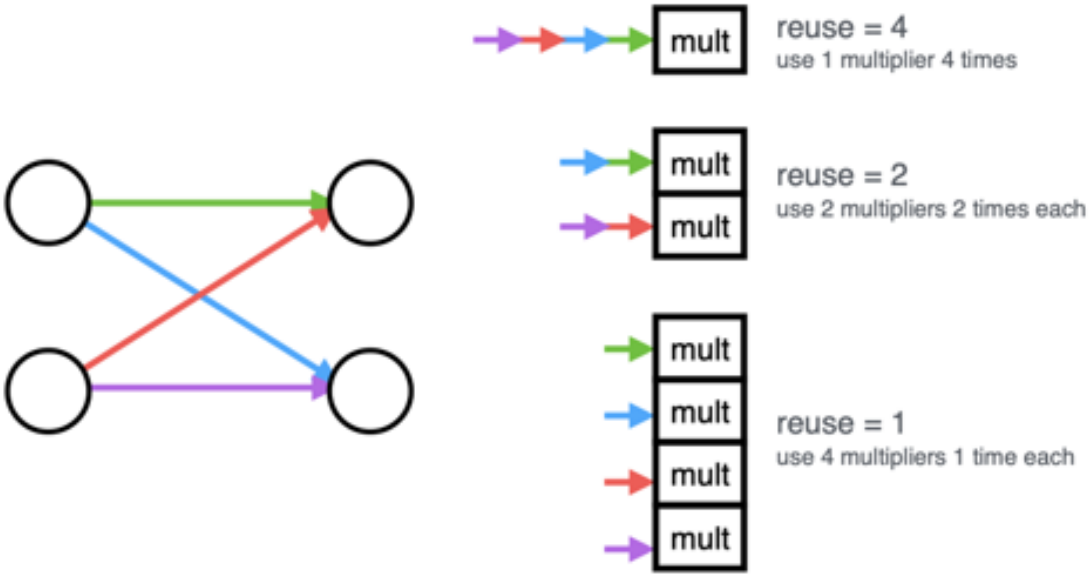


Figura 2.25: *Rappresentazione dell'utilizzo delle risorse per diversi valori del fattore di riuso. A sinistra è riportato il caso di una coppia di neuroni con 4 connessioni che implicano 4 moltiplicazioni. A destra queste risorse sono allocate dalla configurazione completamente seriale (in alto) a quella completamente parallela.* [28]

libreria hls4ml. In questo modo il codice HLS prodotto con hls4ml può essere compilato, ottimizzato in termini di latenza, potenza e throughput, e sintetizzato nell'RTL Verilog o VHDL a partire dalle direttive presenti nel codice ad alto livello. Il flusso principale di Vivado HLS è riportato in [27]:

- Utilizzare linguaggi C/C++ o SystemC per ottenere il codice sorgente ed il testbench;
- Compilare il codice e simularlo (simulazione C);
- Tradurre il codice C nel codice RTL ottimizzato (synthesis);
- Utilizzare il testbench di partenza per eseguire la validazione dell'RTL (co-simulazione);
- Se tutto è corretto, il progetto viene esportato tramite IP che descrive il circuito che può essere realizzato su FPGA;

In particolare, Vivado HLS prende in ingresso il codice ad alto livello per generare un datapath ed un'unita di controllo 'top-level', ovvero solo le funzioni ed i loop

sono associati ad uno stato della macchina a stati. Come detto in precedenza, si sfruttano le direttive presenti nel codice HLS per controllare il modo in cui vengono sintetizzati i costrutti C/C++, mentre funzioni, array e tipi sono sintetizzati utilizzando delle implementazioni specifiche di default. Ad esempio, associa un blocco RTL ad una funzione, esegue l'unrolling del loop ed associa quel blocco per ogni iterazione, oppure assegna memorie ai vettori [27]. La corrispondenza tra il codice C/C++ ed il design hardware avviene tramite dei processi di *scheduling* e *binding*. Il primo impone la temporizzazione delle operazioni nel codice, ovvero in quale colpo di clock deve essere eseguita un'operazione. Il secondo determina il numero di risorse hardware necessarie per eseguire tutte le operazioni. Alla fine del processo Vivado HLS genera dei report che contengono informazioni riguardo alla latenza, all'intervallo di inizio e la complessità del design. La latenza corrisponde al tempo (in termini di colpi di clock) necessario per generare l'uscita a partire dall'applicazione degli ingressi. L'intervallo di inizio indica dopo quanti colpi di clock è possibile applicare un nuovo ingresso. Infine, la complessità si riferisce al numero di risorse utilizzate per realizzare su FPGA il codice HLS convertito:

- Block RAM (BRAM), che corrispondono ai blocchi di memoria
- Digital Signal Processing (DSP), utilizzati per le operazioni MAC;
- Flip Flop (FF), elemento di memoria per registrare un singolo bit;
- Look up table (LUT), tabella che determina l'uscita di una funzione booleana dati gli ingressi.

2.9.4 Bambu

Si tratta di uno strumento open-source per la sintesi ad alto livello sviluppato al Politecnico di Milano a partire dal 2012 [29]. L'approccio di questo strumento è simile al flusso di compilazione: parte da una rappresentazione ad alto livello, scritta in C/C++, per produrre in uscita dopo una serie di passi e ottimizzazioni un codice a basso livello, ovvero la descrizione HDL della corrispondente realizzazione RTL compatibile con gli strumenti di sintesi commerciali. Pertanto, il flusso è modulare ed è costituito da tre fasi: front-end, middle-end e back-end. Nella prima parte Bambu si interfaccia con un compilatore, GCC o Clang, per interpretare e convertire in una rappresentazione intermedia il codice in ingresso. A partire da quest'ultima nella fase middle-end vengono effettuate analisi ed ottimizzazioni indipendenti dal dispositivo per passare alla fase di back-end dove viene la sintesi per generare l'architettura finale tenendo conto delle informazioni del dispositivo e agendo su ogni funzione in modo modulare. In Figura 2.26 è riportato il flusso di compilazione con le operazioni dettagliate dei tre passi precedentemente descritti.

2.10 Accelerazione hardware di reti convoluzionali

Le reti convoluzionali si basano su un algoritmo intrinsecamente parallelizzabile. Questo vantaggio può essere sfruttato al meglio in base all'obiettivo dell'applicazione. Sono presenti diversi tipi di parallelismo nelle reti convoluzionali:

- **Parallelismo nella convoluzione**: la convoluzione tra la matrice ed il filtro può essere eseguita in parallelo in un colpo di clock;
- **Parallelismo nel pooling**: quest'operazione può essere parallelizzata su tutte le matrici contemporaneamente;
- **Parallelismo nelle *feature map* di uscita**: le feature map in uscita sono indipendenti, quindi possono essere calcolate in parallelo;
- **Parallelismo nelle *feature map* di ingresso**: le feature map in ingresso dei livelli precedenti possono essere processate in parallelo per produrre l'uscita;

Altre tecniche di ottimizzazione consistono nella compressione dei parametri e nella riduzione della precisione di rappresentazione, citate precedentemente. Ciò consente di ridurre le risorse hardware per soddisfare le esigenze dell'applicazione. Proprio per la loro natura, le reti convoluzionali risultano adatte per l'elaborazione di immagini, pertanto la realizzazione su CPU non è efficiente essendo quest'ultima basata su un processamento sequenziale. Invece le GPU sono le più adatte per l'addestramento delle reti convoluzionali, ma comportano elevati consumi di potenza. Perciò la realizzazione su FPGA risulta più conveniente, essendo un dispositivo riconfigurabile che può essere programmato per raggiungere gli obiettivi dell'applicazione. Inoltre, consente di sfruttare il parallelismo delle reti convoluzionali con un consumo di potenza molto più basso rispetto alle GPU. Infatti, consente di sfruttare il parallelismo delle risorse per lavorare su diverse parti dell'algoritmo contemporaneamente, oltre a sfruttare tecniche di pipelining per gestire diversi dati simultaneamente e raggiungere una capacità computazionale più alta.

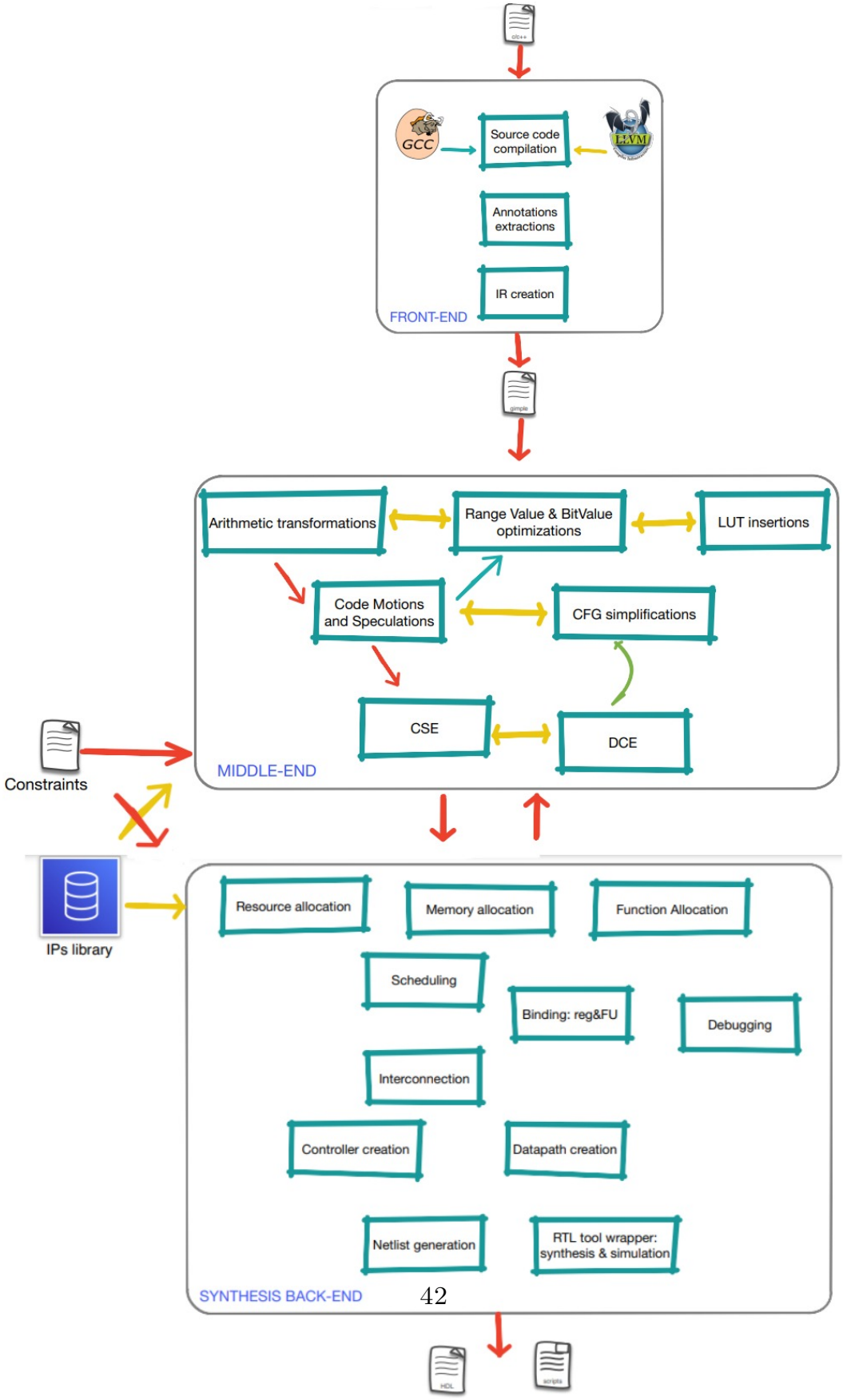


Figura 2.26: *Flusso bambu.* [30]

Capitolo 3

Descrizione del progetto

Questo capitolo ed il successivo presentano il lavoro svolto durante lo sviluppo del progetto della tesi. Il punto di partenza prevede, una volta scelta la piattaforma hardware, la realizzazione di diversi algoritmi di elaborazione di campioni sonori, ovvero la misura del livello di pressione sonora e la classificazione della scena acustica. Per ognuno di questi algoritmi sono state misurate le prestazioni, l'occupazione di memoria e l'energia dissipata al fine di valutare le capacità ed i limiti intrinseci della piattaforma. Nel capitolo successivo verrà presentata la realizzazione dell'applicazione principale della tesi relativa al rilevamento degli eventi sonori.

3.1 Piattaforma Hardware

La piattaforma hardware scelta per questa tesi è il microcontrollore STM32L496G-DISCO [31] di *STMicroelectronics* basato su un processore ARM Cortex M4 con un'unità floating point (FPU), che permette operazioni floating point e istruzioni DSP, e frequenza fino a 80MHz. Si tratta di un microcontrollore a bassissima potenza appartenente alla serie STM32L4. Questa piattaforma consente un'ampia varietà di applicazioni con interfacce utente attraenti che traggono beneficio da molte funzionalità come sensori e connettività ad alta velocità adatte per l'obiettivo della tesi. Sono disponibili 1 MB di memoria Flash e 320 kB di memoria RAM.

Sulla parte superiore della scheda sono presenti due microfoni digitali MEMS omnidirezionali, a basso consumo e con ottima sensibilità (MP34DT01 [32]) per l'ingresso audio.

La scheda dispone di numerose periferiche tra cui un connettore USB, un'interfaccia al modulo di fotocamera digitale (DCMI) per acquisire immagini compresse

e non ad alta velocità e connettori digitali tramite protocolli UART, I2C e SPI. Sono presenti, inoltre, uno schermo LCD e un joystick per navigare all'interno del menu delle applicazioni. Infine, è presente un connettore ST-LINK/V2-1 USB che permette la connessione al PC e l'utilizzo di strumenti di programmazione e debug.

La parte posteriore offre un jack audio e un connettore per micro-SD card che può essere usato per memorizzare il set di dati. È presente anche un amperometro che consente di misurare da 60nA a 50mA per la stima del consumo di corrente da parte del microcontrollore.
Ai lati della scheda i connettori Arduino forniscono la capacità di espansione con un'ampia scelta di schede aggiuntive specializzate.

Figura 3.1: *32L496GDISCOVERY (vista superiore)[33]*

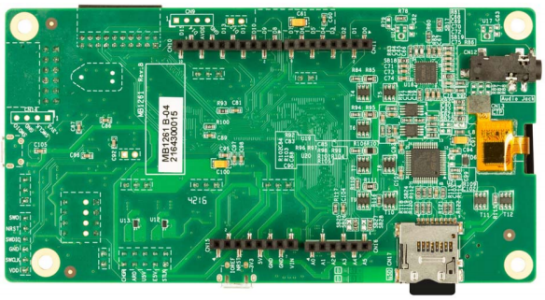


Figura 3.2: *32L496GDISCOVERY (vista inferiore)[33]*

Di fondamentale importanza per lo sviluppo di questo progetto è la presenza di periferiche avanzate come l'interfaccia DFSDM (*Digital filter for Sigma-Delta modulators interface*) che si comporta come un ADC con il vantaggio di avere la parte analogica esternamente al microcontrollore.

Il DFSDM rappresenta la parte digitale dell'ADC $\Sigma\Delta$ collegata alla parte analogica con un'interfaccia seriale ed esegue elaborazioni del segnale fino a 24 bit di risoluzione finale. I microfoni MEMS digitali, che forniscono in uscita dati in formato PDM, possono essere collegati al DFSDM che è in grado di elaborare direttamente il segnale audio per convertire i dati in formato PCM (*Pulse code modulation*) ed effettuare un'operazione di filtraggio in hardware. Infine, i dati di uscita possono essere trasferiti in RAM tramite DMA (*Direct Memory Access*) riducendo in questo modo il carico del processore.

Tutte le caratteristiche descritte finora consentono di ottimizzare l'applicazione ed ottenere una riduzione della potenza dissipata dalla CPU. Inoltre, i dispositivi appartenenti alla serie STM32L4, come questo in esame, supportano otto modalità

a bassa potenza (*LPRun, Sleep, LPSleep, Stop0, Stop1, Stop2, Standby e Shutdown*) che possono essere configurate per la gestione e ottimizzazione della potenza dissipata.
Per quanto riguarda il modem IoT, lo stesso microcontrollore può essere combinato con una scheda di espansione che presenta un modem Quectel BG96, un modulo LTE che supporta tutti gli standard di comunicazione cellulare incluso il NB-IoT [34]. Il soddisfacimento delle richieste per l'obiettivo della tesi derivante dall'ampia varietà di risorse hardware e librerie unite all'efficienza computazionale ed al basso consumo di potenza hanno reso questa piattaforma la perfetta candidata per questo progetto.

3.2 Ambienti di sviluppo

La piattaforma STM32L496 è supportata da uno strumento di sviluppo [35] che fa parte dell'ecosistema software STM32Cube. Si tratta di una piattaforma avanzata di sviluppo che permette la configurazione delle periferiche, la generazione del codice, la compilazione ed il debug. Inoltre, integra tutte le funzionalità di creazione e configurazione offerte dall'interfaccia grafica di *STM32CubeMX*, riportata in Figura 3.3. Dopo la selezione della piattaforma si crea un nuovo progetto con le periferiche

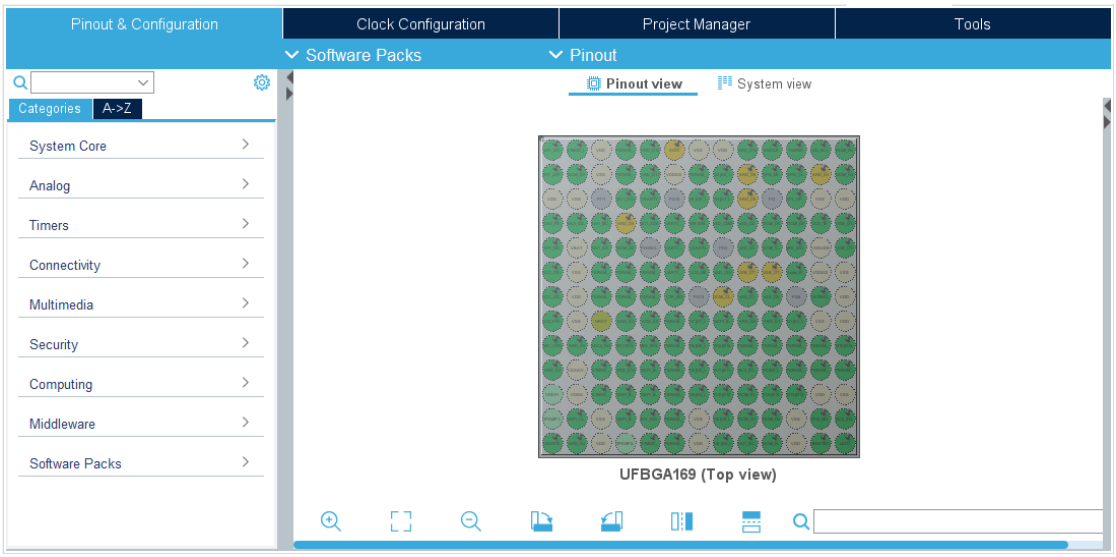


Figura 3.3: *Esempio di configurazione di CubeMX.*

pre-inizialiazzate. L'utente può modificare la configurazione in base all'applicazione e generare il codice. In qualsiasi momento, l'utente può ritornare alla configurazione e rigenerare il codice senza modificare il codice dell'utente.

Sono disponibili, inoltre, dei pacchetti contenenti software di espansione creati da *STMicroelectronics* per STM32Cube che forniscono supporto nella realizzazione dell'applicazione. Alcuni di questi saranno utilizzati e presentati in dettaglio successivamente.

3.3 Misura del livello di pressione sonora

L'orecchio umano è in grado di percepire valori di pressione sonora che coprono un campo di oltre sei ordini di grandezza. Data la difficoltà di elaborare numeri con valori così grandi e considerato che la risposta soggettiva dell'apparato uditivo umano è descritta meglio in termini logaritmici, si esprimono molte delle grandezze utilizzate in acustica sulla scala logaritmica dei decibel.
Il livello di pressione sonora (sound pressure level - SPL) è espresso come

$$L_p(t) = 10\, log_{10} \frac{p(t)^2}{{p_{ref}}^2} \ \ (dB) \tag{3.1}$$

dove $p(t)$ rappresenta la pressione istantanea di un'onda acustica che incide sulla membrana del microfono, mentre p_{ref} è la pressione di riferimento pari a 20 µPa, per la quale si assume convenzionalmente $L_p = 0$. Un problema è la valutazione degli effetti di rumori variabili nel tempo, per cui il livello di pressione come definito in (3.1) risulta inefficiente. A questo scopo è largamente utilizzato il *livello equivalente continuo* L_{eq}.

$$L_{eq} = \frac{1}{T} \int_0^T 10^{\frac{L_p(t)}{10}}\, dt \ \ (dB) \tag{3.2}$$

L_{eq} rappresenta il livello di un ipotetico segnale costante nel tempo il cui contenuto energetico è uguale a quello del segnale variabile in esame [36].

Tipicamente il livello equivalente è riferito alla pressione sonora ponderata A, per cui si utilizza il simbolo $L_{A_{eq}}$. La ponderazione consiste nell'applicare ai livelli di banda d'ottava o di terzi d'ottava una correzione che tiene conto della diversa sensibilità uditiva alle diverse frequenze, poiché riproduce in forma semplificata l'andamento delle curve isofoniche dell'audiogramma normale. Nel dominio discreto, l'integrale viene sostituito dalla sommatoria dei campioni acquisiti nell'intervallo di tempo T [37]:

$$L_{eq,T} = 10\, log_{10} \left(\frac{1}{N} \sum_{i=0}^{N-1} \frac{{p_i}^2}{{p_{ref}}^2} \right) \tag{3.3}$$

3.3.1 Acquisizione dei campioni sonori

L'acquisizione del suono è realizzata mediante i microfoni presenti sulla piattaforma (Sezione 3.1) che convertono le onde di acustiche in segnale digitale. Il microcontrollore acquisisce i campioni provenienti dai microfoni mediante delle periferiche per elaborarli e convertirli nel formato audio standard. I microfoni sono digitali con il trasduttore in tecnologia MEMS che converte la variazione di pressione in una tensione ad essa proporzionale. L'uscita del sensore viene successivamente amplificata e inviata al modulatore PDM che converte il segnale analogico in una sequenza di impulsi la cui densità dipende dall'ampiezza del segnale, come riportato in Figura 3.4. Per acquisire questa sequenza a singolo bit è necessario un protocollo

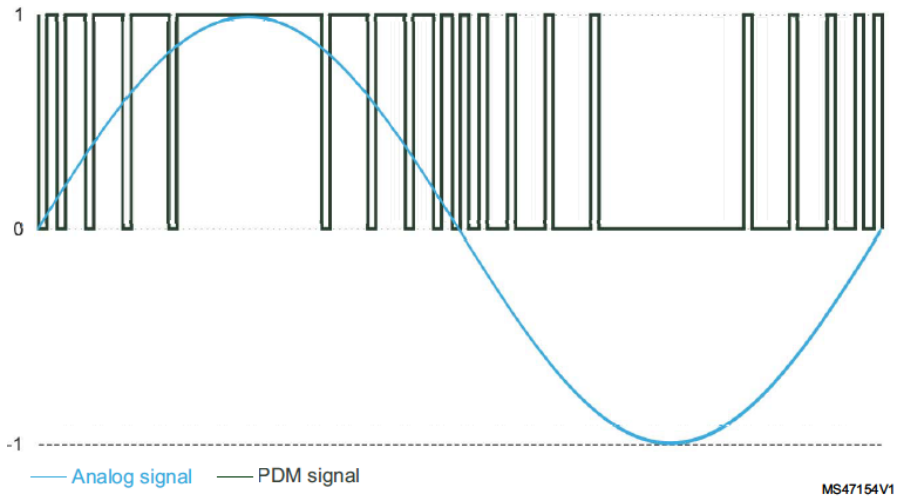


Figura 3.4: *Segnale PDM [38]*

seriale e un segnale di clock in ingresso per controllare il modulatore PDM, la cui frequenza definisce la frequenza di campionamento a cui il segnale in uscita dall'amplificatore è campionato per produrre la sequenza PDM. È presente anche un pin di selezione del canale per impostare il fronte sul quale il dato in uscita è valido. Il diagramma a blocchi di un tipico microfono MEMS digitale è riportato in Figura 3.5.
I microfoni digitali che forniscono i campioni in formato PDM possono essere collegati direttamente al DFSDM (sezione 3.1), una periferica interna al microcontrollore, che fornisce in uscita i campioni filtrati e decimati. Tramite l'interfaccia di *STM32CubeMX* all'interno dell'ambiente di sviluppo è stata abilitata la periferica DFSDM e sono stati selezionati due canali per collegare i due microfoni in modalità stereo. La periferica fornisce un segnale di clock da inviare ai microfoni, impostato ad una frequenza di 2MHz ottenuta con un valore del clock divider pari

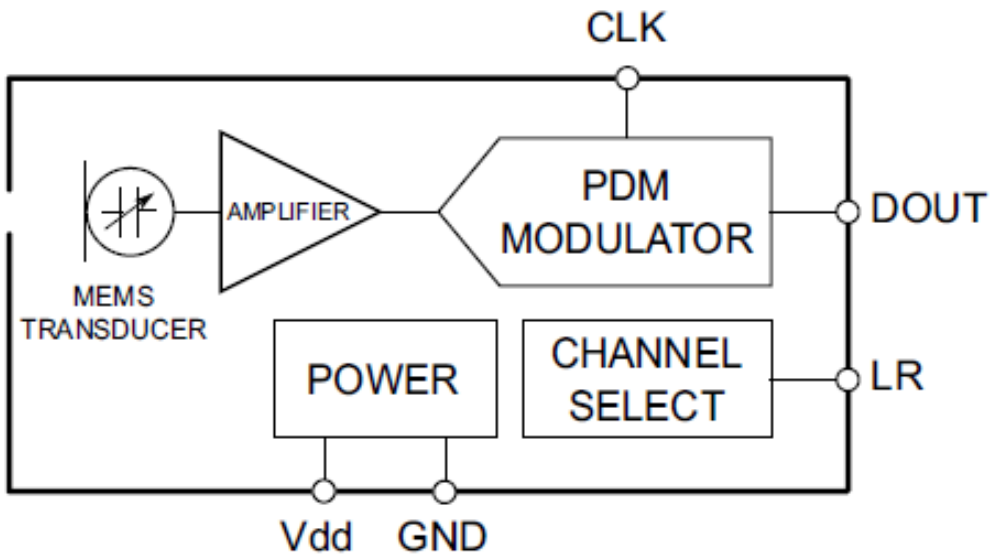


Figura 3.5: *Diagramma a blocchi di un tipico microfono MEMS digitale [38]*

a 40. Ciascun canale della periferica è programmato in modo da ricevere i dati dai microfoni su entrambi i fronti del clock permettendo ai due microfoni di condividere la linea di dato e inviarli ai diversi filtri per essere elaborati. Il filtro decimatore dei due canali è stato impostato con un fattore di decimazione pari a 125, in modo da ottenere una frequenza di campionamento pari a 16kHz. Il trasferimento in memoria dei campioni di ciascun filtro avviene tramite DMA per entrambi i canali mediante un buffer circolare di 960 campioni (60ms), uno per il canale di destra e uno per quello di sinistra. Poiché la massima risoluzione della periferica DFSDM è 24 bit, ma i campioni sono definiti su 32 bit, sono stati aggiunti dei flag che vengono asseriti sia quando viene riempito metà buffer, sia quando viene riempito completamente. Quando ciò avviene i buffer sono passati al DMA spostando i bit dei campioni di otto posizioni in modo da averli su 24 bit.

3.3.2 Descrizione dell'algoritmo

Per realizzare l'algoritmo è stata utilizzata la libreria di *STMicroelectronics* che contiene il software di espansione per l'ambiente di sviluppo *STM32Cube* [35] e consente di integrare il modulo che è in grado di misurare il livello del segnale di ingresso sulla scala logaritmica. In Figura 3.7 è riportata la catena di elaborazione realizzata nel modulo, che prende in ingresso il segnale come sequenza stereo a 16 bit e lo filtra utilizzando la curva di pesatura di tipo A. Successivamente, calcola

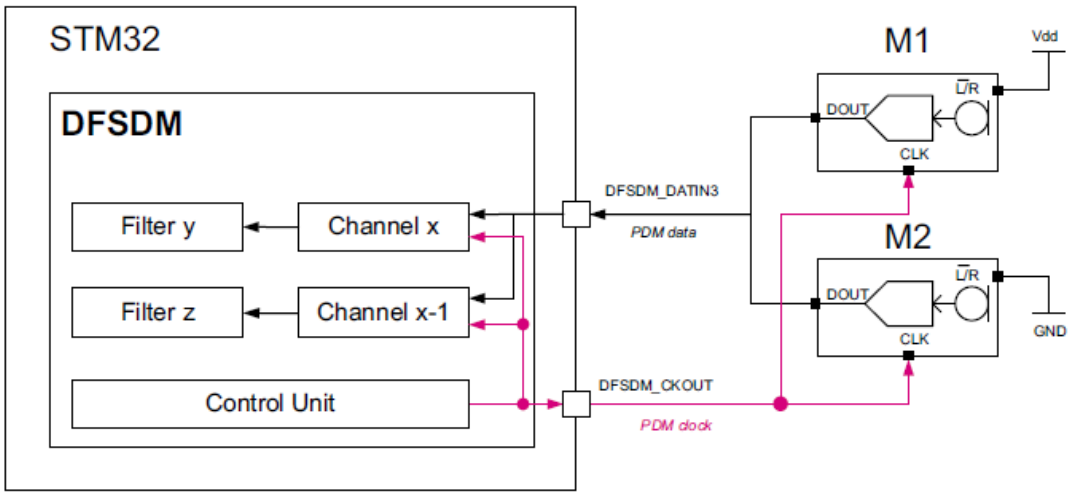


Figura 3.6: *Configurazione stereo DFSDM [38]*

l'energia della sequenza attuale e la converte in scala logaritmica. Infine, il livello viene filtrato per ottenere la media temporale secondo l'equazione(3.3). Il modulo supporta una dimensione massima del buffer pari a 960 campioni, che corrisponde a un segnale stereo di 30ms a 16kHz. Tutte le operazioni sono eseguite con una risoluzione di 32 bit.

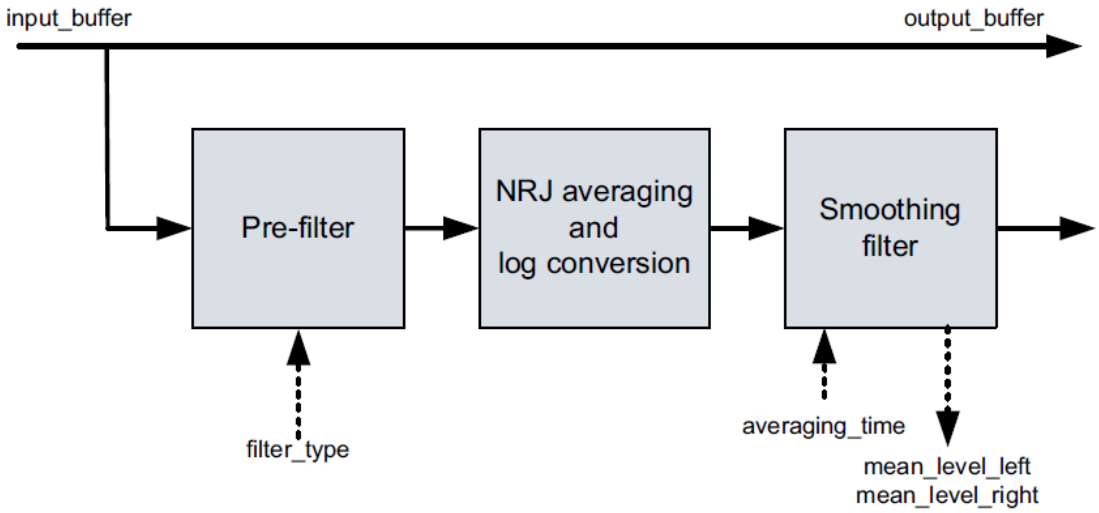


Figura 3.7: *Modulo per il calcolo del livello di pressione sonora [39]*

Per integrare il modulo all'interno del programma è necessario invocare in sequenza alcune funzioni presenti nella struttura principale della libreria: una funzione di reset per inizializzare la memoria del modulo e i parametri statici e dinamici con i valori di default; successivamente si impostano i parametri statici come la frequenza di campionamento e i parametri dinamici per ottenere il tipo di filtro di ponderazione e la costante di tempo del filtro di uscita; infine, si invoca la procedura principale che calcola il livello di pressione sonora. Inoltre, esternamente si definisce la struttura dei buffer di ingresso/uscita insieme ad altre informazioni riguardanti ciascuna sequenza, come il numero di canali e il numero di byte per campione.

In Figura 3.8 è riportata il contenuto dei buffer di entrambi i canali in cui sono memorizzati i campioni sonori in uscita dal DFSDM quando in ingresso ai microfoni vi è un suono che riproduce una sinusoide a 1kHz. In Figura 3.9 è riportata l'uscita

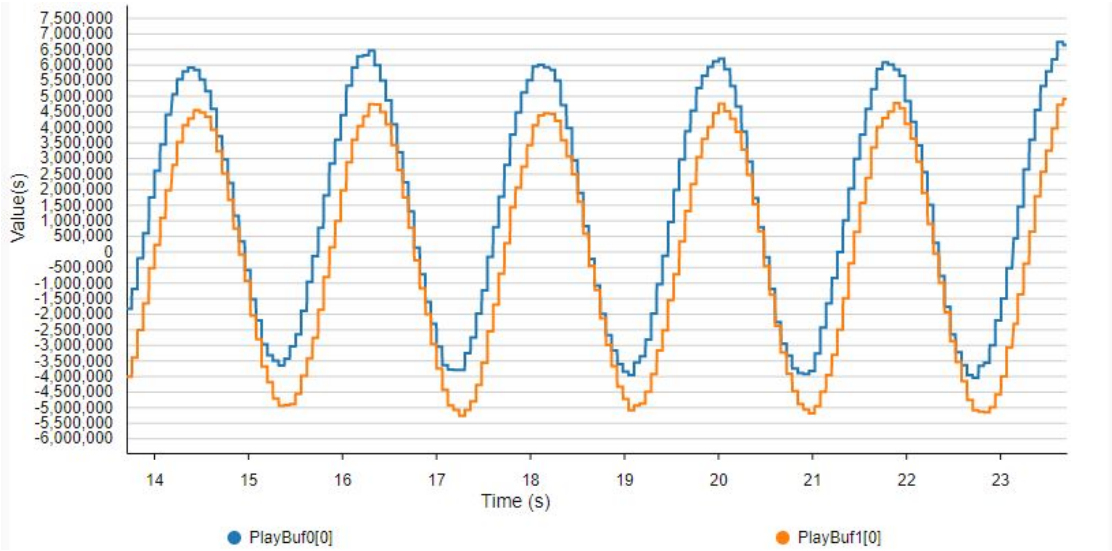


Figura 3.8: *Contenuto in tempo reale dei buffer di uscita del DFSDM con un segnale sinusoidale in ingresso - STM32CubeMonitor.*

del modulo SMR che effettua il calcolo del livello sonoro. Si può notare che il livello sonoro presenta delle variazioni in presenza di suoni esterni, altrimenti rimane piatto.

3.3.3 Complessità dell'algoritmo

A questo punto è stata analizzata la complessità dell'algoritmo, i cui risultati sono riportati in Tabella 3.1. Il tempo totale è stato ottenuto abilitando un timer con una risoluzione di 1µs tramite il quale sono stati misurati il tempo impiegato dalla

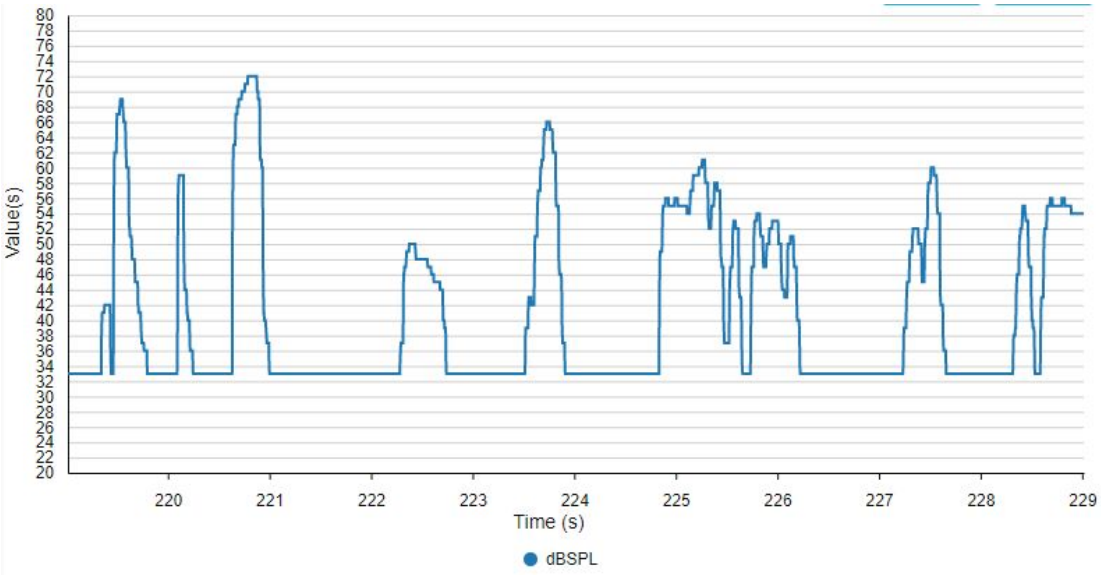


Figura 3.9: *Livello sonoro in uscita dal modulo SMR - STM32CubeMonitor.*

periferica DFSDM per riempire completamente il buffer pari a 525µs, ed il tempo di esecuzione del calcolo del livello di pressione sonora pari a 729µs. Successivamente è stata calcolata l'energia dissipata tramite la misurazione della corrente assorbita dal microcontrollore. Ciò è possibile grazie ad un modulo di espansione *Multi-Function Expander* (MFX) connesso tramite pin GPIO ad un circuito costituito da diverse resistenze che creano una tensione proporzionale al flusso di corrente che viene amplificata prima di essere letta dal modulo MFX.
Come si può notare anche dai valori di occupazione di memoria, la complessità dell'algoritmo è significativamente bassa.

FLASH (KB)	RAM (KB)	Tempo totale (µs)	Energia (µJ)
34.5	63.1	1.3	48.8

Tabella 3.1: Profilazione complessiva dell'algoritmo

3.4 Classificazione della scena acustica

Per classificazione della scena acustica si intende l'abilità di un sistema artificiale di riconoscere il contesto sonoro a partire dalla registrazione di suoni provenienti dall'ambiente circostante [23]. In questa sezione viene presentato un altro algoritmo realizzato utilizzando il microcontrollore (Sezione 3.1) per classificare la scena acustica in base a tre classi di ambienti: interno, esterno ed autoveicolo.

3.4.1 Metodo

Per realizzare l'algoritmo è stato utilizzato un approccio modulare costituito da una sequenza di blocchi ognuno necessario per un determinato scopo, come spiegato nel capitolo 2 dove è riportato il tipico diagramma a blocchi di un sistema di classificazione (Figura 2.1). Il flusso di elaborazione realizzato è rappresentato in Figura 3.10: il segnale audio in ingresso viene suddiviso in segmenti e preprocessato in modo da estrarre le caratteristiche che sono utilizzate dalla rete neurale a valle la cui uscita, pari al numero di scene, rappresenta la probabilità di appartenenza alla rispettiva classe.

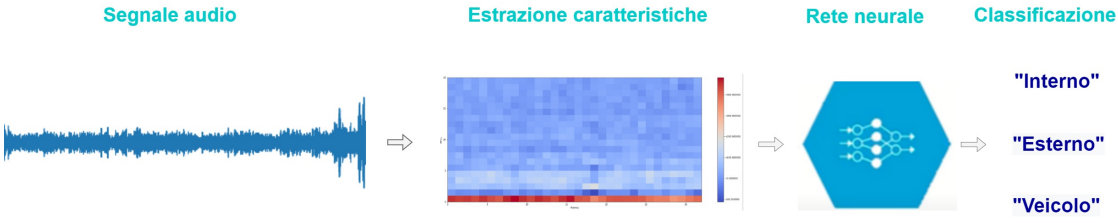


Figura 3.10: *Catena di elaborazione - classificazione della scena acustica*

3.4.2 Set di dati

Il set di dati utilizzato per questo scopo è il *TUT Acoustic Scenes 2016* [5] creato in occasione della competizione organizzata dalla comunità DCASE [40] nel 2016. Il database è costituito da registrazioni di 15 scene diverse appartenenti ad ambienti interni, esterni e interni a veicoli.
La strumentazione utilizzata per registrare le scene comprende dei microfoni auricolari binaurali *Soundman OKM II Klassik/studio A3* a elettrete e un registratore *Roland Edirol R09* a 44.1kHz di frequenza di campionamento e 24 bit di risoluzione.

Le registrazioni sono costituite da audio binaurale della durata di 3-5 minuti.

Successivamente tutto il materiale è stato suddiviso in segmenti di 30 secondi annotando la corrispondente categoria di appartenenza. Sono presenti, pertanto, 1170 segmenti nel database, di cui il 70% è stato selezionato per l'addestramento e il restante 30% per validazione e test del classificatore.

Il sistema iniziale utilizzato per la competizione consiste in un classificatore basato sui coefficienti cepstrali (MFCC) e sul modello Gaussiano (GMM). In particolare, gli MFCC sono calcolati utilizzando segmenti di 40 ms e 50% di sovrapposizione, la finestra di Hamming e 40 bande di Mel. Il classificatore GMM è un modello probabilistico in grado di identificare la presenza di cluster all'interno dell'intera popolazione e di rappresentare la probabilità di distribuzione. Per ogni scena acustica, il modello è addestrato con le caratteristiche descritte in precedenza e le prestazioni sono valutate misurando l'accuratezza intesa come numero di segmenti classificati correttamente sul numero totale di segmenti di test.

3.4.3 Acquisizione dei campioni sonori

Il set di dati è stato memorizzato su una scheda microSD. La piattaforma fornisce un'interfaccia di comunicazione che permette al microcontrollore di comunicare con la microSD grazie ad un modulo integrato SDMMC che viene abilitato tramite l'interfaccia grafica di *STM32Cube*. Quest'ultimo può interfacciarsi con moduli SD I/O per effettuare operazioni di lettura e scrittura ad alta velocità con memorie Flash esterne. Il modulo FatFs fornisce molte funzioni per accedere ai volumi con formato FAT (File Allocation Table) come in questo caso [41]. Queste sono utilizzate per leggere i file audio *.wav* del set di dati per estrarre 1024 campioni per volta fino alla fine di ciascun file. I campioni sono memorizzati quindi in un buffer, in questo modo l'intero segnale audio viene suddiviso in segmenti di 64ms (frequenza di campionamento 16kHz). Inoltre, per evitare i problemi di discontinuità (come spiegato nella sezione 2.2.1), è stata realizzata una sovrapposizione dei segmenti del 50% spostando i campioni del buffer di 512 posizioni ad ogni iterazione.

3.4.4 Estrazione delle caratteristiche audio

Una volta memorizzati i campioni sonori nel buffer, quest'ultimo viene trasferito ad una funzione che si occupa di estrarre le caratteristiche del segmento audio. In questo caso si è scelto di estrarre gli MFCC (sezione 2.3), ovvero i coefficienti cepstrali dallo spettrogramma *Log-Mel*, effettuando una trasformata FFT su 1024 campioni e un banco di filtri di Mel seguito dalla trasformata DCT per ottenere infine 20 coefficienti per ogni segmento. In Figura 3.11 è riportato il diagramma a blocchi delle operazioni necessarie per calcolare i coefficienti. Ciascun blocco è realizzato

tramite delle funzioni in virgola mobile presenti nella libreria di *STMicroelectronics* all'interno del pacchetto FP-AI-SENSING [42].

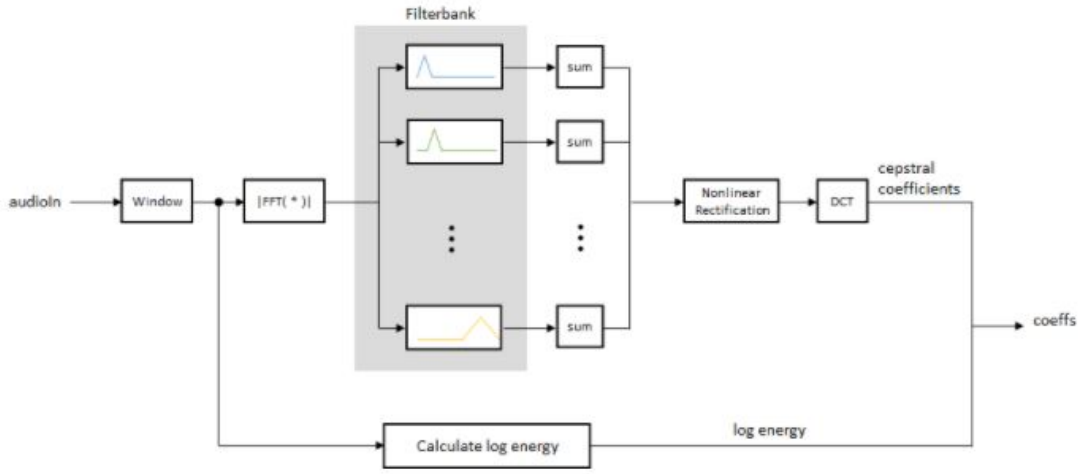


Figura 3.11: *Diagramma a blocchi MFCC*

Poiché ciascun file audio ha una durata di 30s, il numero totale di segmenti è pari a 936. Per ridurre la complessità, la matrice risultante degli MFCC è stata suddiviso considerando 32 segmenti per volta. In questo modo si ottengono 28 matrici per file di dimensione 20 x 32, dove 20 è il numero di coefficienti estratti per ogni segmento e 32 è il numero di segmenti. In Figura 3.12 è riportata la rappresentazione degli MFCC ottenuti a partire dallo spettrogramma Log-Mel del segnale di ingresso. L'intensità del colore è proporzionale al valore del coefficiente.

3.4.5 Modello della rete neurale

Il classificatore è stato realizzato mediante le tecniche di *deep learning*, sia perché sono largamente diffuse e consentono di raggiungere delle prestazioni elevate rispetto ad altri modelli di classificatori, sia perché la piattaforma permette di convertire il modello della rete neurale pre-addestrato grazie al pacchetto di estensione di *X-CUBE-AI* [43] integrato interamente nell'ambiente di sviluppo *STM32CubeMX*. In Figura 3.13 è riportato il flusso delle operazioni necessarie per importare ed eseguire una rete neurale sulla piattaforma. È possibile quindi utilizzare le funzioni generate da *X-CUBE-AI* per eseguire l'inferenza del modello.

Il primo passo consiste nella creazione del modello di riferimento utilizzando software esterni come *Tensorflow*, una libreria software open source per l'apprendimento automatico, che facilita lo sviluppo e l'addestramento del modello grazie

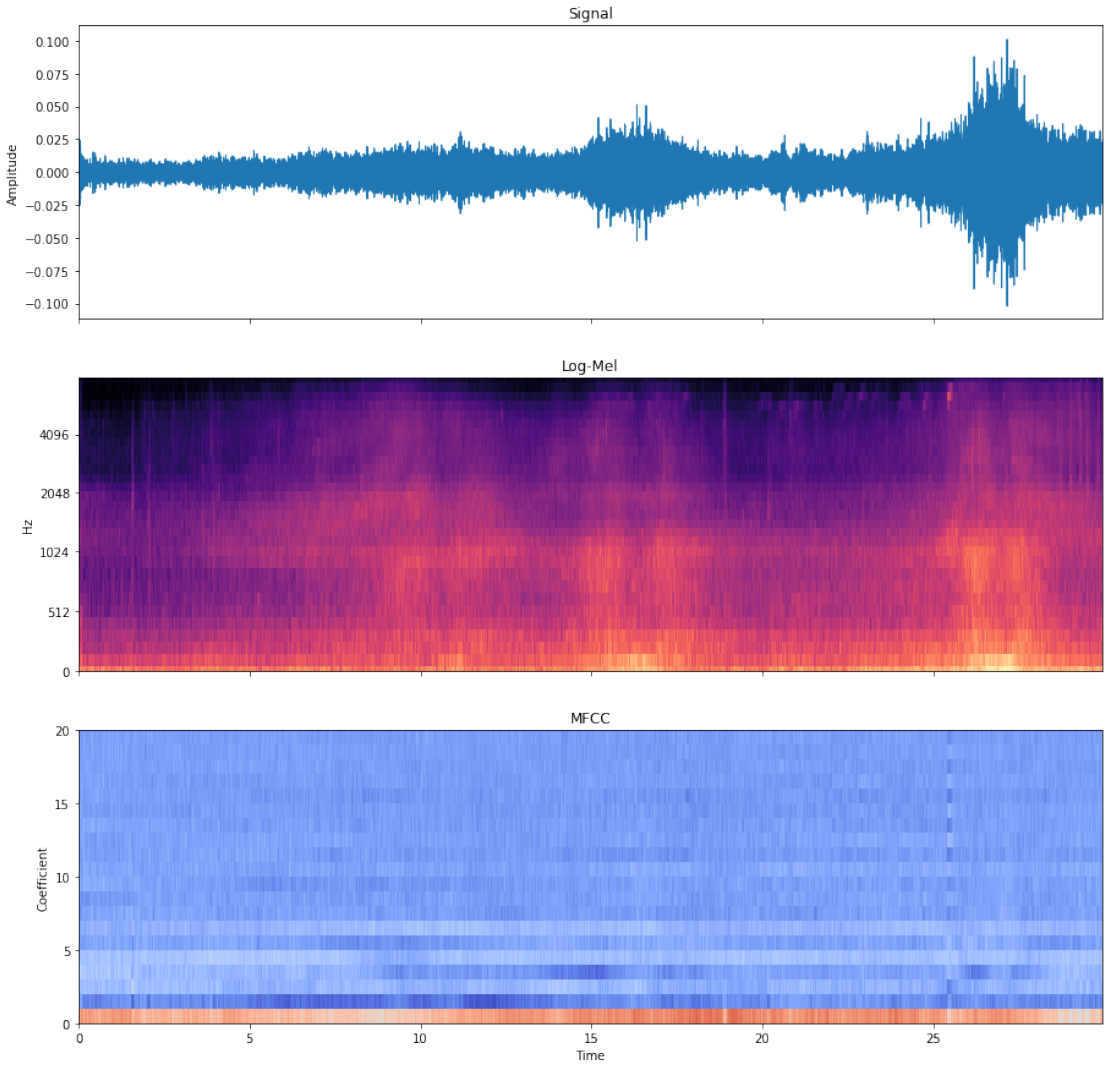


Figura 3.12: *Visualizzazione degli MFCC*

all'utilizzo di un'API (Application Programming Interface) basata su *Keras*, una libreria open source scritta in *Python* per la realizzazione di reti neurali. Successivamente si utilizza l'estensione *X-CUBE-AI* dall'interfaccia *STM32CubeMX* per importare il modello e generare il codice necessario per eseguire l'inferenza.

3.4.6 Flusso di conversione per una rete neurale semplice

Prima di creare il modello per la classificazione della scena acustica, è stata realizzata una semplice rete neurale per acquisire familiarità con il flusso di conversione

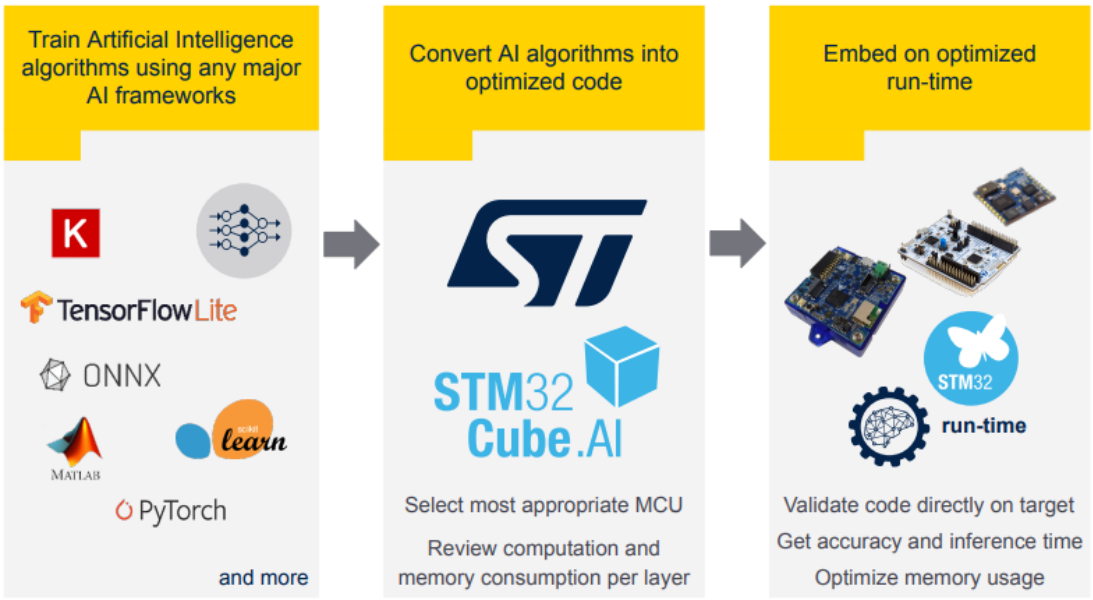


Figura 3.13: *Flusso delle operazioni X-CUBE-AI [44]*

mediante *X-CUBE-AI*.

Il modello consiste in una rete di tre livelli completamente connessi in grado di prevedere l'uscita di una sinusoide in un intervallo compreso tra -1 e 1, dati in ingresso dei numeri compresi tra 0 e 2π.

Per la creazione del modello è stato utilizzato Google Colab, una piattaforma online che permette di scrivere ed eseguire codice Python direttamente dal browser. Dopo aver importato le librerie necessarie, come Numpy per la gestione dei vettori e Keras per le funzioni delle reti neurali, sono stati generati in modo casuale 1000 campioni per il set di dati, di cui il 20% sono usati per il set di validazione e un altro 20% per il set di test. Il restante 60% viene utilizzato per l'aggiornamento dei parametri del modello. Successivamente viene calcolata l'uscita della sinusoide con questi ingressi aggiungendo un rumore gaussiano per rendere la predizione più imprecisa, come riportato in Figura 3.14.

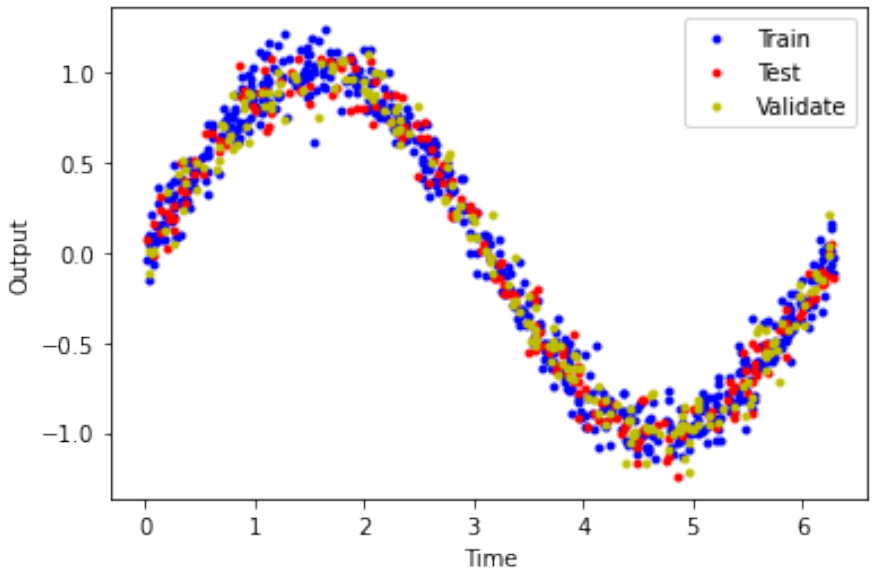


Figura 3.14: *Sinusoide con set di dati casuali*

La rete è costituita da tre livelli completamente connessi e funzione di attivazione ReLU. L'ingresso e l'uscita hanno un solo neurone, mentre i livelli intermedi sono dei livelli di tipo Dense e sono molto diffusi nell'ambito dell'intelligenza artificiale, soprattutto nei modelli più semplici. In Figura 3.15 è riportata la rappresentazione del modello. Il numero di parametri per ogni livello si può calcolare secondo l'equazione (3.4), ovvero come il numero di neuroni del livello precedente sommato al bias moltiplicato per il numero di neuroni del livello successivo. In tabella 3.3 è riportato il riassunto del modello che presenta un numero totale di parametri pari a 321.

$$(N_{in} + 1) \times N_{out} \tag{3.4}$$

Livello	**(Tipo)**	**Dimensione uscita**	**#Param**
dense_9	(Dense)	16	16
dense_10	(Dense)	16	272
dense_11	(Dense)	1	17

Tabella 3.2: Riassunto del modello

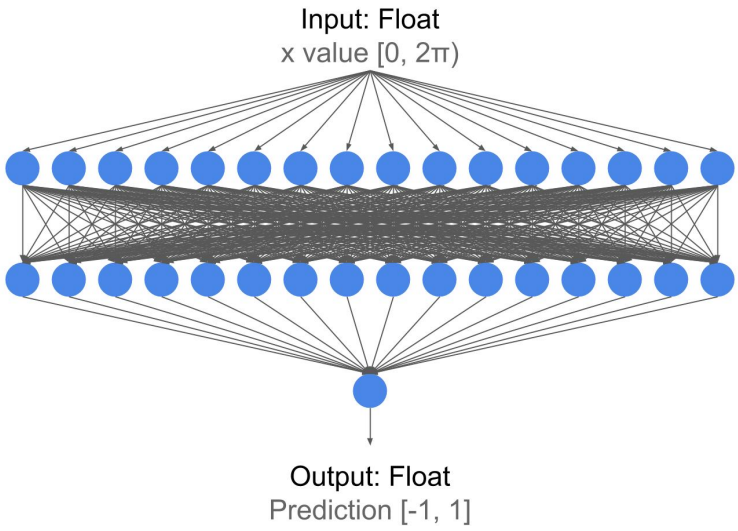
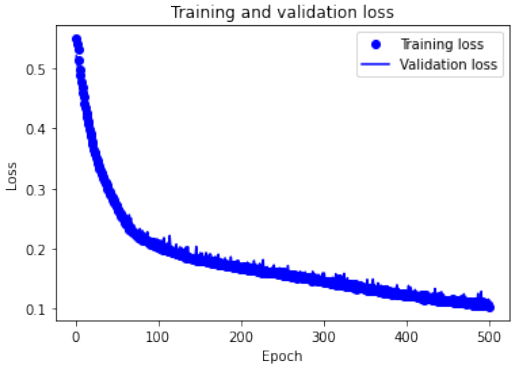


Figura 3.15: *Modello della rete neurale*

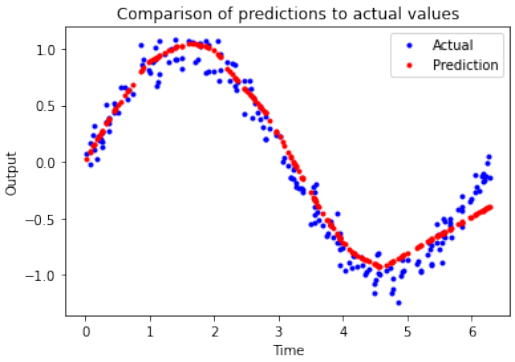
La rete prende in ingresso un elemento per volta ed esegue tutti i calcoli necessari per predire il valore che mappa l'uscita della funzione seno al valore di ingresso. Il modello è stato addestrato utilizzando 500 *epoche* e *batch size* pari a 100. Ciò implica che l'addestramento avviene tramite iterazione di 10 batch di 100 campioni. Dopo l'esecuzione si ottiene un grafico che confronta la funzione di costo dell'addestramento con quella della validazione, come riportato in Figura 3.16a, da cui è possibile notare che la predizione è abbastanza accurata.

Infine, si verifica il comportamento del modello con il set di test. Dopo aver creato il set di test otteniamo un grafico che confronta i valori attuali più rumore con i valori della predizione. Dalla Figura 3.16b si può notare che il modello approssima verosimilmente la sinusoide.

Una volta creato ed addestrato il modello si configura l'interfaccia grafica di *STM32CubeMX* per importare il modello della rete sulla piattaforma tramite la conversione del modello *Keras* nel modello C ottimizzato che può essere eseguito sulla piattaforma. In Figura 3.17 è riportata l'interfaccia di *STM32CubeMX* dopo aver selezionato il pacchetto *X-CUBE-AI*. La generazione del codice si avvale di un ottimizzatore che trova il giusto compromesso tra occupazione di memoria (RAM e ROM) e tempo di esecuzione applicando degli algoritmi di compressione di pesi/bias oppure fondendo alcuni livelli della rete in modo da ridurne la complessità.



(a) *Funzioni di costo del set di addestramento e validazione*



(b) *Confronto delle predizioni con i valori attesi*

Figura 3.16: Funzione di costo e confronto delle predizioni con i valori attesi

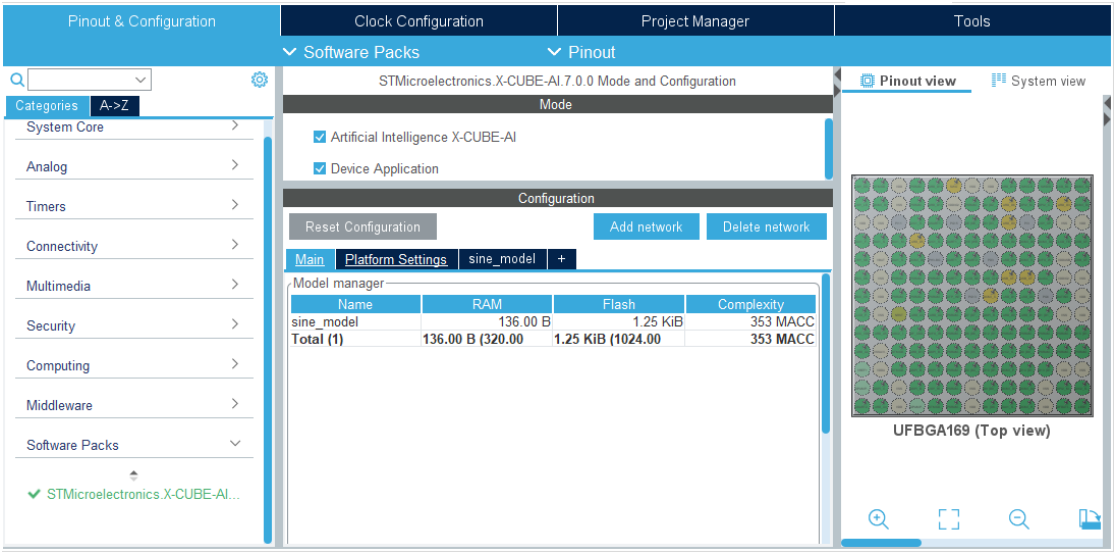


Figura 3.17: *Interfaccia CubeMX con l'estensione di X-CUBE-AI*

Prima di generare il codice è possibile analizzare dall'interfaccia grafica il modello importato per avere informazioni su occupazione di RAM e FLASH, e MACC (*Multiply and Accumulate Complexity*) che indica la complessità del modello in termini di tempo di esecuzione. Inoltre, è possibile eseguire la validazione del modello che permette di confrontare l'accuratezza del modello generato rispetto a quello importato, insieme ad una stima del tempo di inferenza sia con dati casuali che con il set di test.

In Tabella(3.3) è riportato il resoconto ottenuto dall'analisi e dalla validazione del modello importato su *CubeMX* tramite X-CUBE-AI. *STMicroelectronics* stima per un microcontrollore ARM Cortex M4F approssimativamente 9 cicli/MACC. Con 80MHz di frequenza della CPU si ottiene un tempo di esecuzione stimato pari a 85µs. Inoltre, si può notare che l'errore relativo (L2r) tra il modello pre-addestrato e quello importato è inferiore a 0.01 e risulta pertanto accettabile [43].

MACC	**#Parametri**	**RAM (B)**	**FLASH (KB)**	**Errore Relativo**	**Tempo (µs)**
353	321	136	1.25	1.72e-6	85

Tabella 3.3: Resoconto analisi e validazione del modello tramite X-CUBE-AI

Successivamente è stato generato il codice: nel main si includono i file della libreria *X-CUBE-AI* insieme ai file header del modello. Vengono dichiarate alcune variabili come un buffer per l'uscita seriale e un buffer che memorizza i calcoli intermedi per la rete, utilizzando costanti già definite dalla libreria, i vettori di input e output e una struttura di parametri che punta al modello.
Infine, sono state invocate delle funzioni della libreria per inizializzare il modello ed eseguire l'inferenza.

Dopo aver compilato il progetto, il codice è stato eseguito prima in modalità *debug* e, successivamente alla fase di sviluppo del codice, in modalità *release*. In Tabella(3.4) sono riportate le differenze nelle due modalità, si può notare che in modalità release il tempo di inferenza e l'occupazione di FLASH sono ridotti poiché il compilatore effettua ottimizzazioni eliminando le informazioni e variabili utili per il debug per migliorare le prestazioni e la dimensione del codice.

Modalità	**FLASH (KB)**	**RAM (KB)**	**Tempo totale (µs)**
Debug	84.9	17.7	80
Release	53.2	17.7	77

Tabella 3.4: Differenze modalità debug/release per dimensione del codice e tempo di inferenza del modello

3.4.7 Modello per la classificazione della scena acustica

Il modello realizzato è rappresentato in Figura 3.18: si tratta di una rete neurale convoluzionale che ha come ingresso la matrice degli MFCC di dimensioni 20 x 32 che viene processata dai livelli convoluzionali che effettuano una convoluzione tra il campo recettivo ed il filtro (kernel) per estrarre le caratteristiche locali, ovvero la concentrazione di energia sia nel tempo che in frequenza. Tra un livello convoluzionale ed il successivo vi è un livello di pooling che ha il compito di ridurre la dimensione del tensore di uscita per dare più importanza alle caratteristiche piuttosto che alla posizione tempo-frequenza. La classificazione avviene mediante dei livelli completamente connessi che estraggono con la funzione *softmax* la probabilità di appartenenza alle tre classi di uscita. La predizione finale è ottenuta tramite un voto di maggioranza delle probabilità di uscita [20]. In Tabella 3.5 è riportato il

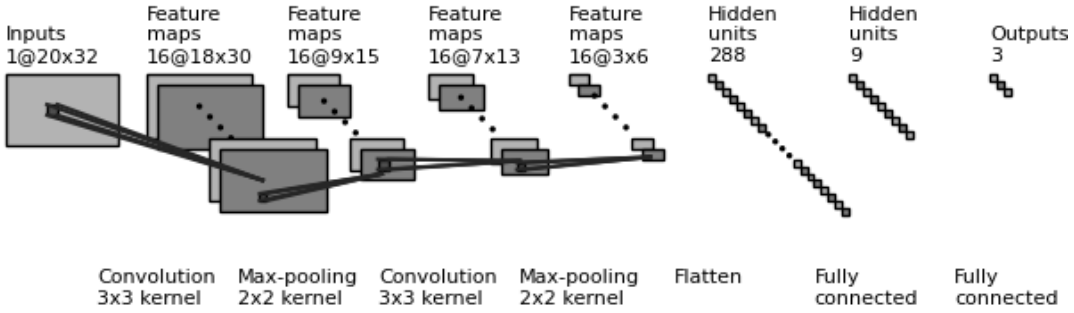


Figura 3.18: *Modello rete neurale per la classificazione della scena acustica*

riassunto riguardante il tipo, la dimensione ed il numero di parametri di ciascun livello.

Livello	**(Tipo)**	**Dimensione uscita**	**#Param**
conv2d	(Conv2D)	(18, 30, 16)	160
max_pooling2d	(MaxPooling2D)	(9, 15, 16)	0
conv2d_1	(Conv2D)	(7, 13, 16)	2320
max_pooling2d_1	(MaxPooling2D)	(3, 6, 16)	0
flatten	(Flatten)	(288)	0
dense	(Dense)	(9)	2601
dense_1	(Dense)	(3)	30

Tabella 3.5: Riassunto del modello per la classificazione della scena acustica

Addestramento della rete neurale

Al fine di addestrare il modello, il set di dati è stato suddiviso in tre sottoinsiemi: *addestramento*, *validazione* e *test*. L'addestramento è stato eseguito in modo graduale, ovvero utilizzando una percentuale di file audio nel set di *addestramento* crescente fino a quando la percentuale di accuratezza converge, come riportato in Figura 3.19. In Figura 3.20, invece, è riportato l'andamento della funzione di costo in base alle previsioni effettuate sul set di validazione a partire dai pesi e bias calcolati tramite il set di addestramento. Si può notare che, all'aumentare del numero di iterazioni (epoche), le funzioni di costo dell'addestramento e della validazione si riducono e si avvicinano sempre di più dimostrando pertanto un miglioramento della qualità del modello. Una volta effettuato l'addestramento, si valuta il comportamento del modello utilizzando il set di test per ottenere le predizioni con dati mai utilizzati prima d'ora.
Una rappresentazione dell'accuratezza del modello è la cosiddetta *matrice di confusione* riportata in Figura 3.21, dove le colonne sono legate alle predizioni, mentre le righe ai valori reali e consente di valutare se nella classificazione vi è "confusione". In questo caso, l'accuratezza totale è elevata ed è pari al 96%.

Inferenza della rete sulla piattaforma

Analogamente a quanto effettuato nella sezione(3.4.6), il modello è stato importato tramite X-CUBE-AI. In Tabella(3.6) è riportato il resoconto ottenuto dall'analisi e dalla validazione del modello importato su *CubeMX* tramite X-CUBE-AI.

MACC	#Parametri	RAM (B)	FLASH (KB)	Errore Relativo	Tempo (µs)
310029	5111	14.7	19.96	1.75e-7	52.5

Tabella 3.6: Resoconto analisi e validazione del modello tramite X-CUBE-AI

Una volta generato il codice, il progetto è stato compilato e il programma è stato eseguito prima in modalità *debug* e successivamente in modalità *release*. In tabella(3.7) sono riportate le differenze nelle due modalità, le considerazioni precedenti sono valide anche in questo caso.

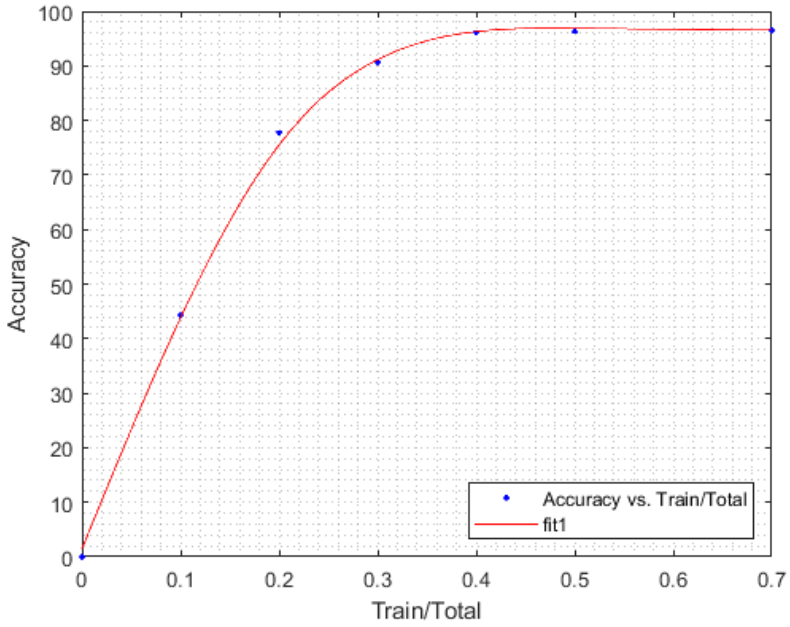


Figura 3.19: *Percentuale di accuratezza del modello rispetto alla percentuale di file nel set di addestramento*

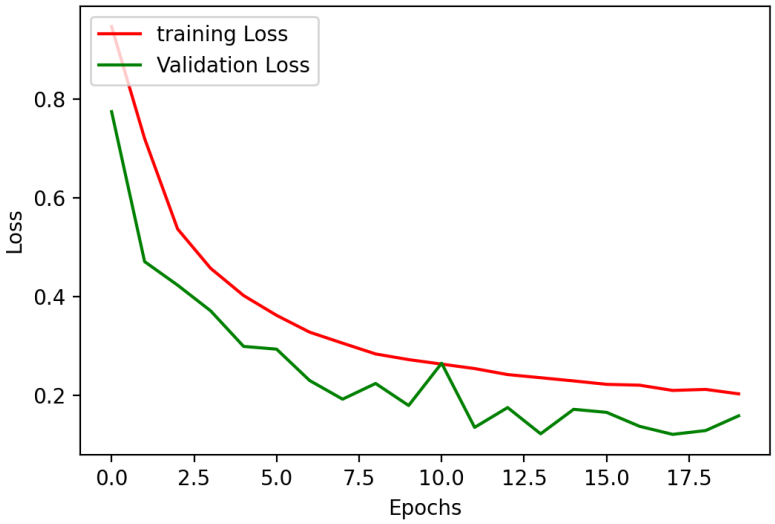


Figura 3.20: *Andamento della funzione di costo al variare del numero di epoche*

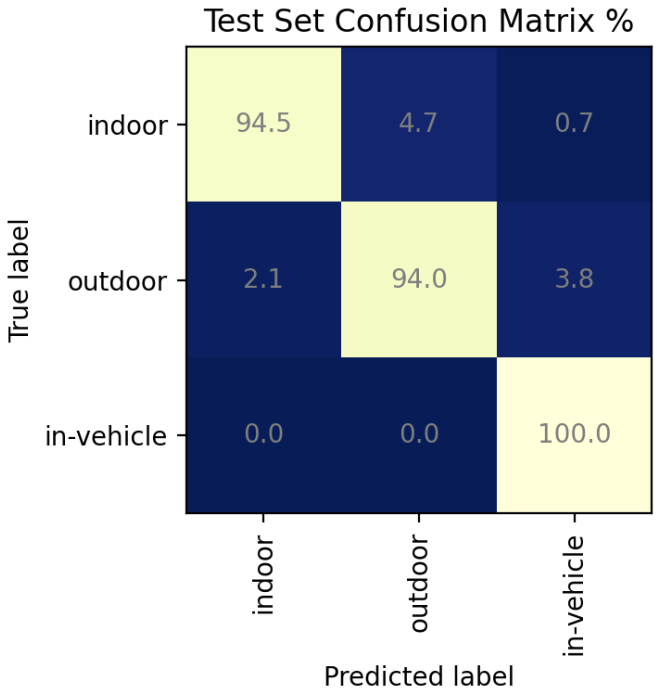


Figura 3.21: *Matrice di confusione per la classificazione della scena acustica*

Modalità	**FLASH (KB)**	**RAM (KB)**	**Tempo totale (ms)**
Debug	230	210	69
Release	217	210	53

Tabella 3.7: Differenze modalità debug/release per dimensione del codice e tempo di inferenza del modello

I valori delle predizioni ottenuti dalla rete durante l'inferenza sono stati quindi confrontati con i valori reali per ottenere l'accuratezza del sistema eseguito sulla piattaforma pari a 93%.

3.4.8 Quantizzazione del modello

Tensorflow permette di minimizzare la complessità di ottimizzazione del modello al fine di ottenere i seguenti vantaggi [45]:

- Riduzione delle dimensioni
- Riduzione della latenza
- Riduzione della potenza

Esistono diversi tipi di ottimizzazione supportati da *Tensorflow* tra cui la quantizzazione che riduce la precisione dei parametri utilizzati dal modello che sono, per impostazione predefinita, numeri in virgola mobile a 32 bit. Ciò risulta in una dimensione e in un tempo di computazione del modello inferiori.

Esistono due forme di quantizzazione: **quantizzazione post-addestramento** e **formazione consapevole della quantizzazione**. La prima è più facile da usare sebbene l'addestramento consapevole della quantizzazione sia spesso migliore per l'accuratezza del modello. Questo tipo di quantizzazione permette all'utente di prendere il modello in virgola mobile già addestrato e quantizzarlo usando solo numeri interi a 8 bit oppure una configurazione mista con i pesi in virgola mobile su 16 bit e i bias in interi a 8 bit. Per la quantizzazione completamente intera, è necessario calibrare o stimare l'intervallo di tutti i tensori a virgola mobile nel modello. Di conseguenza, il convertitore richiede un set di dati rappresentativo per calibrarli, che può essere un piccolo sottoinsieme dei dati di addestramento o di convalida. La quantizzazione a 8 bit approssima i valori in virgola mobile utilizzando la formula riportata nell'equazione(3.5) che consente di convertire la scala dei numeri in virgola mobile nella scala dei numeri interi come rappresentato in Figura 3.22.

$$real\_value = (sint8\_value - zero\_point) \times scale \tag{3.5}$$

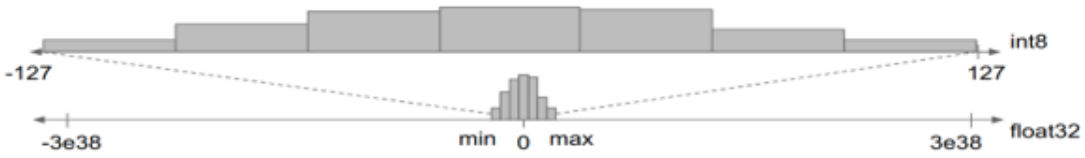
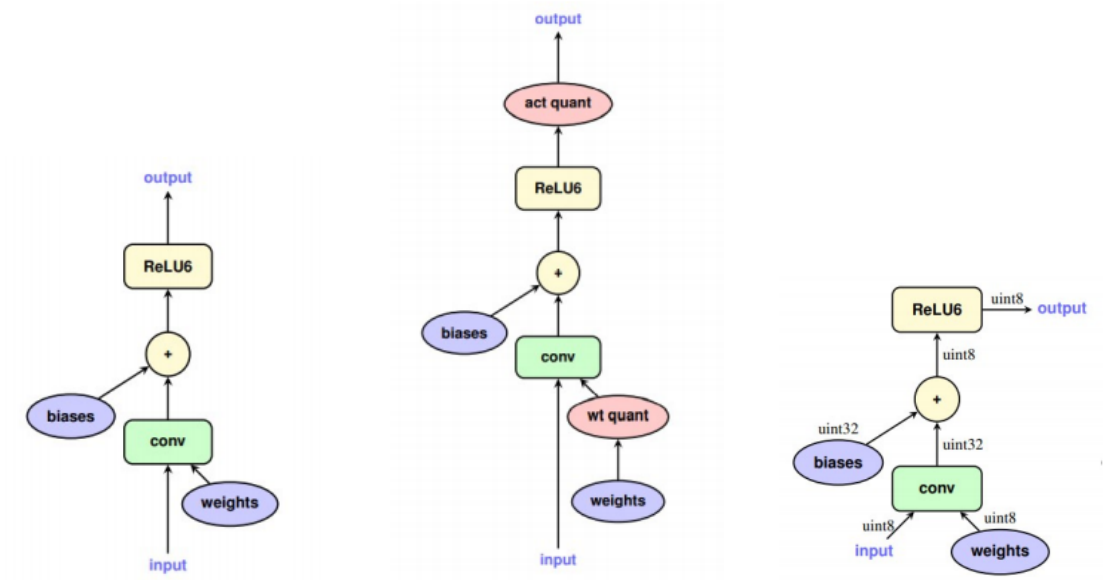


Figura 3.22: *Conversione scala valori virgola mobile nella scala dei valori interi*

Poiché i pesi sono quantizzati dopo l'addestramento, potrebbe esserci una perdita di precisione. Pertanto, è importante controllare l'accuratezza del modello quantizzato

per verificare che qualsiasi degrado rientri nei limiti accettabili.

In alternativa, se il calo di precisione è troppo elevato, si prende in considerazione l'addestramento consapevole della quantizzazione. Tuttavia, ciò richiede modifiche durante l'addestramento del modello per aggiungere nodi di quantizzazione fittizi. Si può notare la differenza in Figura 3.23b, in cui si utilizza la quantizzazione fittizia prima del livello di convoluzione e dopo la funzione di attivazione, rispetto al grafo in Figura 3.23a tradizionale.



(a) *Grafo senza quantizzazione*

(b) *Grafo quantizzazione consapevole*

(c) *Grafo della quantizzazione durante l'inferenza*

Figura 3.23: Passi di quantizzazione consapevole per un tipico livello convoluzionale [46]

Si ottiene pertanto un grafo equivalente a quello utilizzato durante la fase di inferenza riportato in Figura 3.23c. In questo modo è possibile addestrare il modello sfruttando i benefici della quantizzazione e mantenendo l'accuratezza vicina al modello originale, poiché l'errore di quantizzazione viene introdotto come rumore durante l'addestramento per cui il modello risulta più robusto. Infatti, come si può notare in Figura 3.24, la perdita di accuratezza del modello quantizzato dopo l'addestramento non è accettabile, mentre quella del modello addestrato con la quantizzazione consapevole è pressoché coincidente al modello originale.

Nel Listing 3.1 è riportato pertanto un estratto del codice che converte il modello di partenza nel modello quantizzato prima di effettuare l'addestramento.

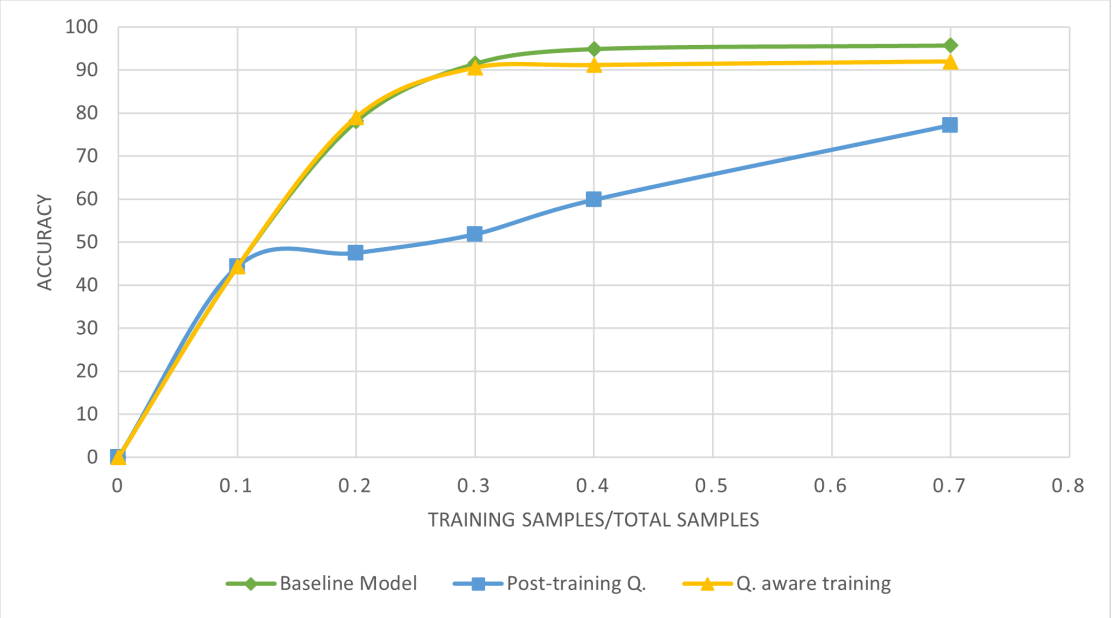


Figura 3.24: *Confronto delle accuratezze ottenute con le tecniche di quantizzazione rispetto al modello originale*

```
print ('Building Model...')
import tensorflow_model_optimization as tfmot
quantize_annotate_layer = tfmot.quantization.keras.
    quantize_annotate_layer

model = tf.keras.Sequential([
tf.keras.layers.Conv2D(16, (3, 3), activation='relu',input_shape=(20,
     32, 1), data_format='channels_last'),
tf.keras.layers.MaxPooling2D((2, 2)),
tf.keras.layers.Conv2D(16, (3, 3), activation='relu'),
tf.keras.layers.MaxPooling2D((2, 2)),
tf.keras.layers.Flatten(),
tf.keras.layers.Dense(9, activation='relu'),
tf.keras.layers.Dense(nclasses, activation='softmax') ])

quant_aware_model = tfmot.quantization.keras.quantize_model(model)
```

Listing 3.1: Modello quantizzato con Qkeras.

Infine, il modello ottenuto con l'addestramento consapevole della quantizzazione è stato importato tramite X-CUBE-AI per eseguire l'inferenza sulla piattaforma. Il risultato ottenuto dalla validazione è riportato in Tabella(3.8): l'accuratezza del modello quantizzato è inferiore di meno del 3% rispetto al modello originale; inoltre, sia l'occupazione di memoria che il tempo di inferenza risultano ridotti significativamente, essendo inferiori di circa quattro volte [47].

Modello	**FLASH (KB)**	**RAM (KB)**	**Tempo totale (ms)**	**Accuratezza (%)**
Originale	19.96	14.7	53	93.55
Quantizzato	5.12	10.42	19	90.96

Tabella 3.8: Confronto tra modello originale e quantizzato tramite X-CUBE-AI

3.4.9 Complessità dell'algoritmo

A questo punto è stata eseguita la profilazione dell'algoritmo per valutare la complessità temporale, l'occupazione di memoria e l'energia dissipata in ogni fase del flusso di elaborazione. Queste misurazioni non includono l'accesso alla scheda microSD poiché questa fase comporterebbe tempi più lunghi ed inoltre non vi è dipendenza dai dati, ovvero il tempo di elaborazione del singolo segmento non dipende dalla lunghezza dell'intero segnale audio.

In Figura 3.25 sono riportati i valori per ciascuna fase: nella fase di pre-processamento si misura il tempo necessario per calcolare una colonna di MFCC per il singolo segmento che consiste nel calcolo della FFT, applicazione del banco di filtri di Mel e DCT. Come si può notare, il basso valore di tempo di elaborazione dimostra i vantaggi di eseguire il pre-processamento a bordo, in quanto il trasferimento dei dati sarebbe più oneroso sia in termini di tempo che di energia. Per quanto riguarda l'inferenza della rete neurale si può notare il notevole impatto della quantizzazione che permette di ridurre di circa quattro volte il tempo di inferenza; ciò si riflette anche sull'energia dissipata, anche in questo caso calcolata considerando l'alimentazione pari a 1.8V.
In Tabella 3.9 è riportata la profilazione complessiva dell'algoritmo. Come si può notare, sia il tempo totale che l'energia risultano dimezzati grazie alla quantizzazione del modello. Ciò dimostra che, al costo di una piccola perdita di accuratezza nella

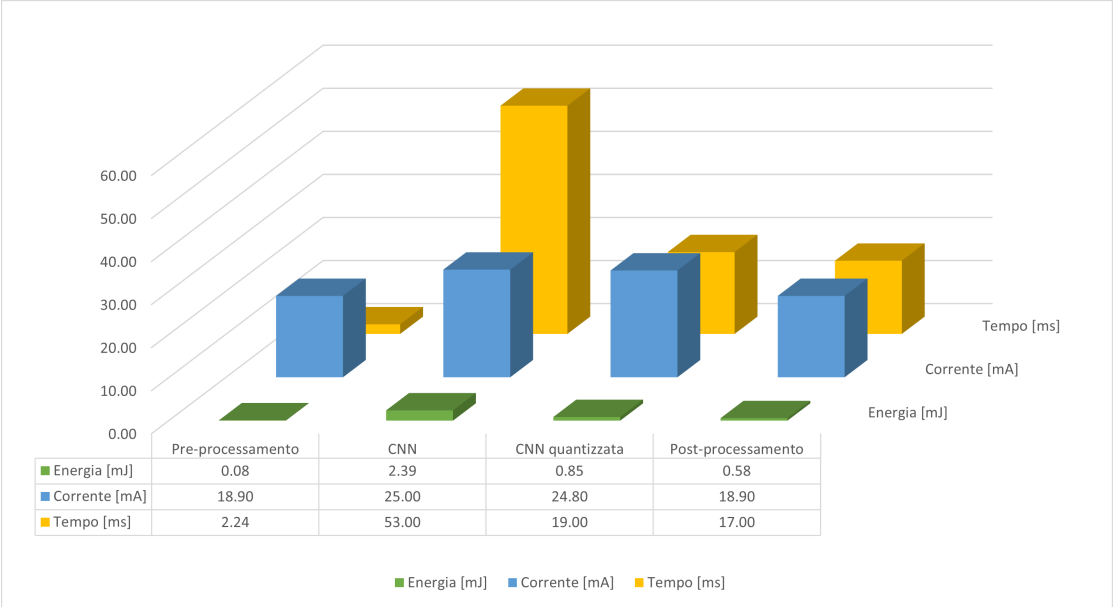


Figura 3.25: *Profilazione di ciascuna fase dell'algoritmo*

predizione, la quantizzazione risulta fondamentale per le applicazioni realizzate su piattaforme con risorse limitate.

Modello	FLASH (KB)	RAM (KB)	Tempo totale (ms)	Energia (mJ)
Originale	217	205	72.24	3.07
Quantizzato	223	194	38.24	1.53

Tabella 3.9: Profilazione complessiva dell'algoritmo

Capitolo 4

Rilevamento degli eventi sonori

In questo capitolo viene presentata la realizzazione dell'algoritmo principale di questa tesi. Il punto di partenza riguarda la scelta del set di dati, successivamente si descrive ogni passo della catena di elaborazione fino alla valutazione della complessità e delle prestazioni dell'algoritmo.

4.1 Metodo

Il metodo proposto è caratterizzato da un approccio simile a quello adottato per la classificazione della scena acustica (sezione 3.4.1). Come detto in precedenza, in letteratura i termini relativi al rilevamento e alla classificazione dell'evento sonoro sono intercambiabili, poiché gli eventi sonori sono definiti associando etichette predefinite, chiamate anche classi.

La prima fase prevede l'estrazione delle caratteristiche tempo-frequenza (spettrogramma) da ciascuna registrazione audio. Successivamente si utilizza una rete neurale convoluzionale (CNN) per associare alle caratteristiche acustiche la predizione binaria relativa alla presenza delle classi di eventi.

4.2 Set di dati

La scelta del set di dati è di fondamentale importanza per lo sviluppo dell'applicazione. Come detto, l'obiettivo è di realizzare un sistema in grado di riconoscere suoni ambientali di emergenza o pericolo. In letteratura sono presenti diversi set di dati creati per il rilevamento degli eventi sonori, molti dei quali sono stati creati dalla

comunità DCASE. In particolare, in occasione della competizione del 2017 è stato utilizzato un set di dati adatto per *smart cars*, *smart cities* ed applicazioni affini. Quest'ultimo contiene un sottoinsieme di eventi sonori estratti da una collezione di due milioni di sequenze audio di 10s annotate manualmente (**AudioSet**[1]). Il sottoinsieme consiste in 17 eventi sonori appartenenti a due classi di suoni: allarme e veicolo [48]. La classe dei suoni di allarme è risultata adatta per lo scopo di questa tesi, infatti contiene suoni come urla di spavento che indicano una situazione di emergenza.

Una particolarità di questo set di dati riguarda le annotazioni, in quanto non sono indicati l'istante di inizio e di fine dell'evento, ma soltanto le classi degli eventi presenti nella registrazione. Si parla infatti di *weak labels*, ovvero di annotazioni deboli, per distinguerle dalle cosiddette *strong labels* che contengono anche le indicazioni temporali dell'evento. Questa scelta è dovuta al fatto che annotare manualmente la presenza di ogni evento all'interno della registrazione risulta oneroso. Tuttavia, ciò aumenta la difficoltà nello sviluppo dell'applicazione e comporta un peggioramento nelle prestazioni del sistema nel rilevare la sequenza temporale dell'evento. Il set di training è costituito da 51172 file audio, di cui almeno 30 per ogni evento sonoro; il set di test invece contiene 488 file, con almeno 30 sequenze audio per classe. In aggiunta, il set di test contiene le annotazioni temporali che consentono di valutare le prestazioni del sistema.

4.3 Acquisizione dei campioni sonori

Analogamente all'algoritmo precedente, il set di dati è stato salvato sulla scheda microSD e utilizzando il modulo FatFs si accede al volume per la lettura dei file audio *.wav*. I campioni estratti sono memorizzati in un buffer di dimensione pari a 1024 con una sovrapposizione pari a 360 campioni ad ogni iterazione. In questo modo, poiché ciascun file audio ha una durata di 10s, il numero totale di segmenti è pari a 240.

4.4 Estrazione delle caratteristiche audio

Ciascun buffer è trasferito ad una funzione che calcola lo spettrogramma *Log-Mel* tramite una serie di funzioni del pacchetto FP-AI-SENSING [42] che in sequenza effettua la FFT, applica il banco di filtro di Mel a 64 bande e calcola il logaritmo dell'uscita. Una volta processato l'intero file audio si ottiene uno spettrogramma

[1] `https://research.google.com/audioset/`

di dimensione 240x64 (tempo-frequenza) ad esempio come riportato in Figura 4.1. Queste caratteristiche audio mostrano buone prestazioni per il rilevamento di eventi sonori ([4], [48], [49]). Inoltre, in questo caso, lo spettrogramma log-Mel consente di raggiungere prestazioni migliori rispetto agli MFCC [50].

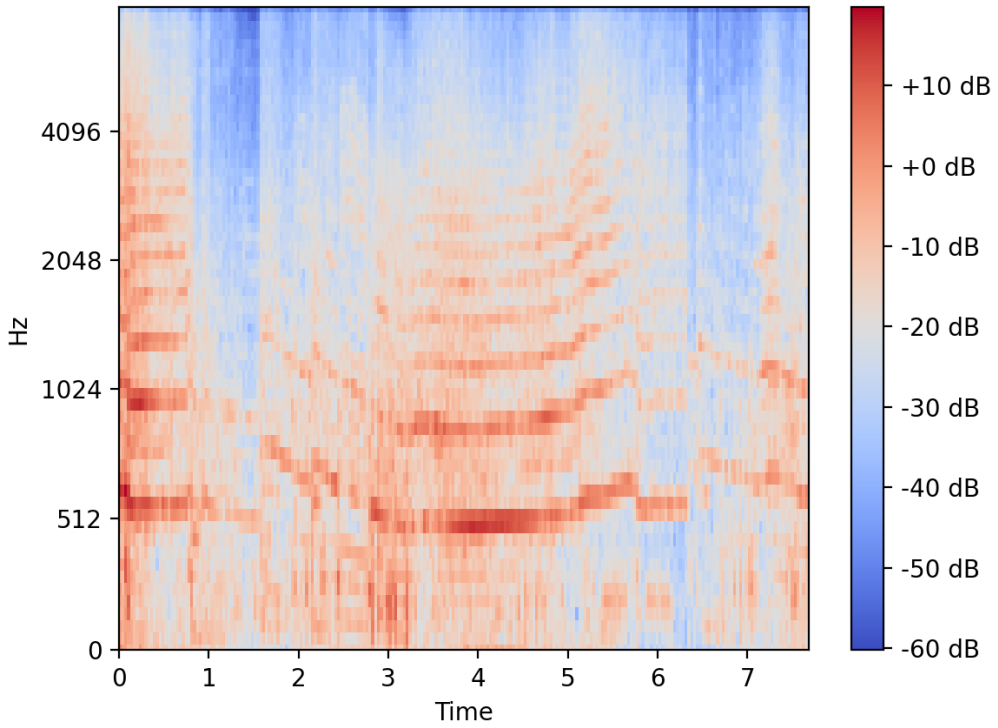


Figura 4.1: *Spettrogramma Log-Mel*

4.5 Modello della rete neurale

Il modello di partenza è riportato in Figura 4.2 e consiste in una rete convoluzionale con unità lineari (GLU o *gated linear units*) e livelli ricorrenti (RNN o *Recurrent neural network*) applicati allo spettrogramma in ingresso. Le unità lineari sono utilizzate al posto della funzione di attivazione ReLU per controllare l'informazione tra un livello ed il successivo: la sequenza audio è convertita nello spettrogramma e trattata come un'immagine dalla rete convoluzionale. Tuttavia, a differenza delle immagini, gli eventi sonori sono concentrati soltanto in una porzione dello spettrogramma, pertanto, queste unità servono per eliminare le parti dello spettrogramma

che non contengono informazione utile. Le unità lineari sono definite come in (4.1) [50]:

$$Y = (W * X + b) \odot \sigma(V * X + c) \tag{4.1}$$

dove σ è la funzione sigmoide, X l'ingresso, W e V i filtri della convoluzione, b e c i bias.
Pertanto, l'uscita è il prodotto tra matrici dell'uscita lineare $(W * X + b)$ modulata da $\sigma(V * X + c)$.
I livelli ricorrenti sono utilizzati per tenere conto della correlazione tra i vari segmenti dello spettrogramma. Infatti, le reti ricorrenti prevedono una retroazione che consente di basare le decisioni sul passato e sulla posizione reciproca degli elementi in sequenza.
Successivamente, vi sono i livelli completamente connessi: oltre al livello feed-forward con la funzione di attivazione sigmoide utilizzata per ottenere la classificazione in ogni segmento audio, è aggiunto un livello in parallelo con funzione di attivazione softmax. Quest'ultima viene utilizzata per la localizzazione temporale dell'evento in quanto assegna un peso maggiore alla classe con probabilità più alta ed un peso minore alle altre classi. Le due uscite sono quindi moltiplicate tra di loro per ottenere le probabilità di presenza dell'evento per ogni classe in ogni segmento. Infine, per ottenere la classificazione finale del file audio si effettua una media sull'intera sequenza temporale dello spettrogramma. Il modello di cui

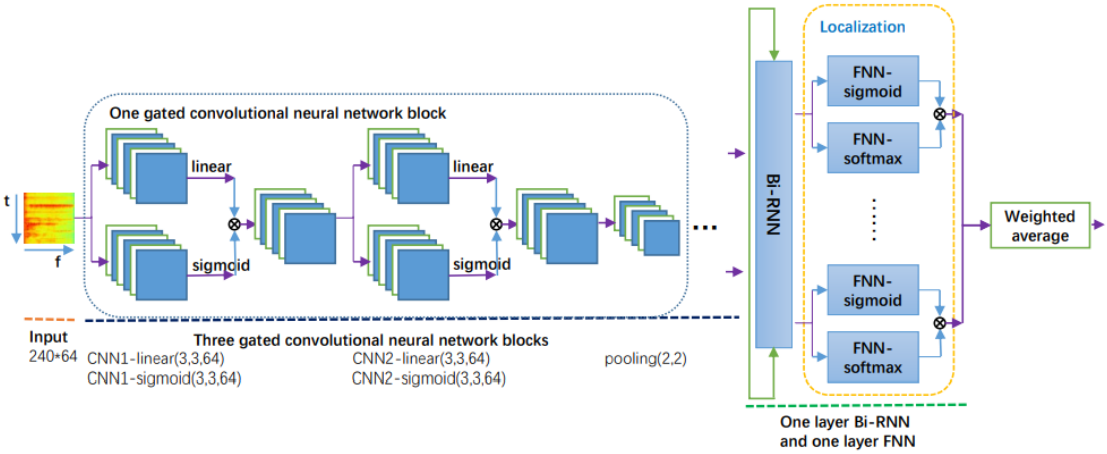


Figura 4.2: *Diagramma proposto dai vincitori della competizione DCASE 2017 per il rilevamento degli eventi sonori [50]*

sopra è stato realizzato con *Tensorflow* con un procedimento analogo a quello della classificazione della scena acustica che prevede l'addestramento della rete con il set di *training* e successivamente la valutazione della qualità con il set di *test*. Tuttavia, come si può notare dalla Figura 4.2 il modello è notevolmente complesso: infatti il

modello Keras pre-addestrato è stato importato su *CubeMX* tramite X-CUBE-AI e dall'analisi del modello sono emersi i risultati riportati in Tabella 4.1.

MACC	**#Parametri**	**RAM (MB)**	**FLASH (MB)**
790,062,480	1,758,097	7.56	6.71

Tabella 4.1: Resoconto analisi del modello [50] tramite X-CUBE-AI

È evidente dalla Tabella 4.1 che questo modello è irrealizzabile, soprattutto per l'elevato numero di MACC che non risulta adatto per il tipo di applicazione dato che implicherebbe un tempo di inferenza non accettabile. Pertanto, è stato necessario trovare un compromesso tra la complessità del modello e l'accuratezza al fine di poter importare il modello sulla piattaforma ed ottenere dei risultati ragionevoli. Si è pensato quindi di ridurre la complessità del modello ed il numero di classi di eventi da riconoscere ad un'unica classe: l'urlo. Nella sezione successiva sarà descritto quindi il modello semplificato e si valuterà la scalabilità dei parametri derivante da queste modifiche.

4.5.1 Creazione del modello

In Figura 4.3 è riportato il modello semplificato utilizzato per rilevare la presenza dell'urlo all'interno della registrazione audio, mentre in Tabella 4.2. Per diminuire la complessità del modello originale (Figura 4.2) sono stati ridotti i livelli convoluzionali e sono stati eliminati i livelli di attenzione, ovvero le unità lineari (GLU) e i livelli ricorrenti (RNN). Inoltre, il pooling si effettua soltanto lungo l'asse delle frequenze, in modo da mantenere l'uscita con dimensionalità pari al numero di segmenti dello spettrogramma ed ottenere quindi la probabilità di presenza della classe in ogni singolo segmento. Infine, l'uscita è costituita solamente da un livello completamente connesso con funzione di attivazione sigmoide per ottenere la probabilità dell'evento in ogni segmento dello spettrogramma che viene convertita in uscita binaria mediante l'applicazione di una soglia ottimale ricavata tramite il calcolo del valore massimo della metrica F1. In questo caso la funzione softmax, utilizzata nel caso della classificazione multi-etichetta, non è necessaria in quanto si vuole riconoscere la presenza di un'unica classe.
Il modello è stato quindi addestrato nuovamente applicando una funzione di costo con entropia binaria tra le probabilità della predizione e i valori attesi [50]. La

funzione di costo può essere definita come:

$$E = -\sum_{n-1}^{N}(P_n log O_n + (1 - P_n) log(1 - O_n)) \tag{4.2}$$

dove E è l'entropia, mentre P_n e O_n sono le probabilità ed i valori attesi per ogni segmento audio. L'andamento della funzione di costo è riportato in Figura 4.4 che dimostra la convergenza tra il set di addestramento e validazione indicando quindi un miglioramento della qualità del modello all'aumentare del numero di epoche e l'assenza di overfitting.

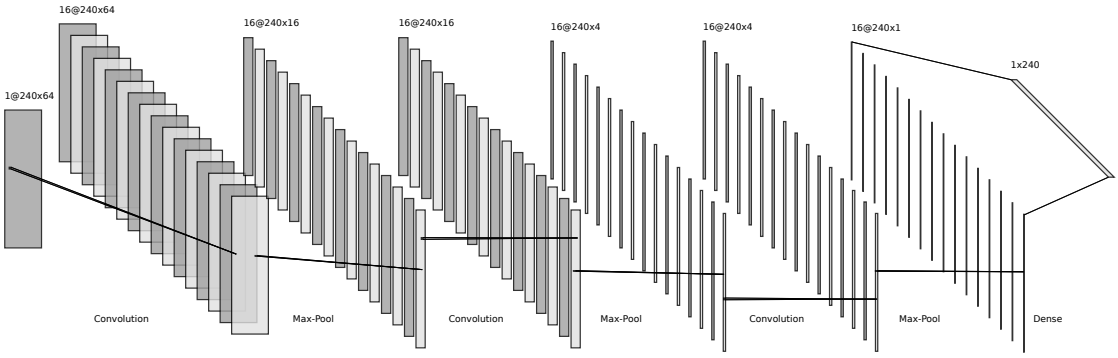


Figura 4.3: *Modello rete neurale per il rilevamento dell'evento sonoro*

Livello	**(Tipo)**	**Dimensione uscita**	**#Param**
in_layer	(Input)	(264, 64, 1)	
conv2d	(Conv2D)	(240, 64, 16)	144
max_pooling2d	(MaxPooling2D)	(264, 16, 16)	
conv2d_1	(Conv2D)	(240, 16, 16)	2304
max_pooling2d_1	(MaxPooling2D)	(240, 4, 16)	
conv2d_2	(Conv2D)	(240, 4, 16)	2320
max_pooling2d_2	(MaxPooling2D)	(240, 1, 16)	
reshape_1	(Reshape)	(240,16)	
dense	(Dense)	(240,1)	17

Tabella 4.2: Riassunto del modello per il rilevamento dell'evento sonoro

Come si può notare dalla Tabella 4.3 la complessità del modello si riduce notevolmente, soprattutto in termini di MACC. Infatti, dalla validazione del modello risulta un tempo di esecuzione pari a 1085ms, che è ancora elevato per l'applicazione,

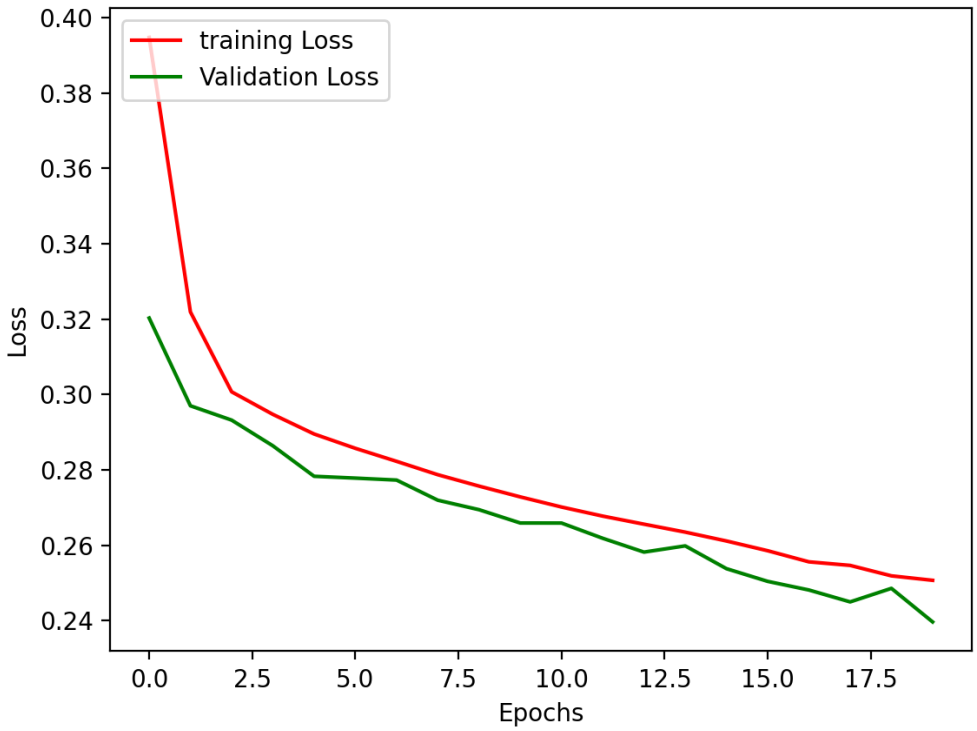


Figura 4.4: *Andamento della funzione di costo al variare del numero di epoche*

MACC	#Parametri	RAM (KB)	FLASH (KB)	Errore Relativo	Tempo (ms)
13,922,656	4,785	304.9	18.82	1.39e-7	1085

Tabella 4.3: Resoconto analisi e validazione del modello tramite X-CUBE-AI

ma sicuramente inferiore a quello del modello originale. Tuttavia, la riduzione della complessità del modello necessaria per realizzare l'algoritmo sulla piattaforma ha comportato un peggioramento della qualità del modello rispetto al caso iniziale con il set di addestramento originale. Infatti, come si può notare nella prima colonna della Tabella 4.4 in cui sono riportate le metriche descritte nella sezione 2.6, l'accuratezza del modello semplificato si riduce rispetto a quella del modello originale: ciò è dovuto all'elevato numero di falsi positivi.
In Figura (4.5) è riportata una rappresentazione dello spettrogramma, dell'uscita

	Modello semplificato		**Modello originale**	
Metriche	**Training originale**	**Training nuovo**	**Training** originale	**Training nuovo**
PRECISION (%)	25.5	37.8	22.0	55.5
RECALL (%)	79.2	61.1	45.1	64.2
F1 (%)	38.6	46.7	29.6	59.5
ACC (%)	64.8	80.5	73.1	87.8

Tabella 4.4: Confronto delle prestazioni tra modello semplificato ed originale al variare del set di training

della rete neurale e della sua conversione in uscita binaria, e dell'uscita attesa per un file del set di test. Come si può notare dall'uscita di riferimento, l'urlo non è presente, ma vi sono numerosi falsi positivi.

Per tale motivo il set di addestramento è stato modificato includendo solo i file audio adatti per quest'applicazione, ovvero quelli contenenti urla legate ad una situazione di pericolo, ed eliminando tutti i file relativi alle altre classi del set di dati. Il set di addestramento è stato ridotto quindi a circa cento file audio utili, ad ognuno dei quali è stata associata un'etichetta binaria per indicare la presenza dell'urlo nella porzione della registrazione ed è stato ripetuto analogamente alla fase precedente l'addestramento ed il test del modello semplificato. Come si può notare nella seconda colonna della Tabella 4.4, le prestazioni migliorano, in particolare si riducono i falsi positivi e quindi aumentano la precisione e l'accuratezza. Inoltre, è evidente la differenza tra le Figure (4.5) e (4.6), infatti in quest'ultima è riportata la predizione dello stesso file audio, ma non sono più presenti i falsi positivi dell'immagine precedente. In Figura 4.7 si può notare il corretto riconoscimento dell'urlo. Inoltre, è importante evidenziare che addestrando nuovamente il modello originale con il nuovo set di training si ottengono prestazioni superiori a quelle riportate dall'articolo [50], come si può notare nell'ultima colonna della Tabella 4.4.

4.6 Quantizzazione del modello

Anche in questo caso è stato necessario ricorrere alla quantizzazione per ridurre la complessità dell'algoritmo. Sono state effettuate quindi sia la quantizzazione post-addestramento sia la quantizzazione consapevole usando solo numeri interi a 8 bit. Nel primo caso la perdita di accuratezza non è elevata e l'errore di quantizzazione è pari al 10% circa, mentre nel secondo caso è praticamente irrilevante. Il risultato ottenuto dalla validazione è riportato in Tabella 4.5.

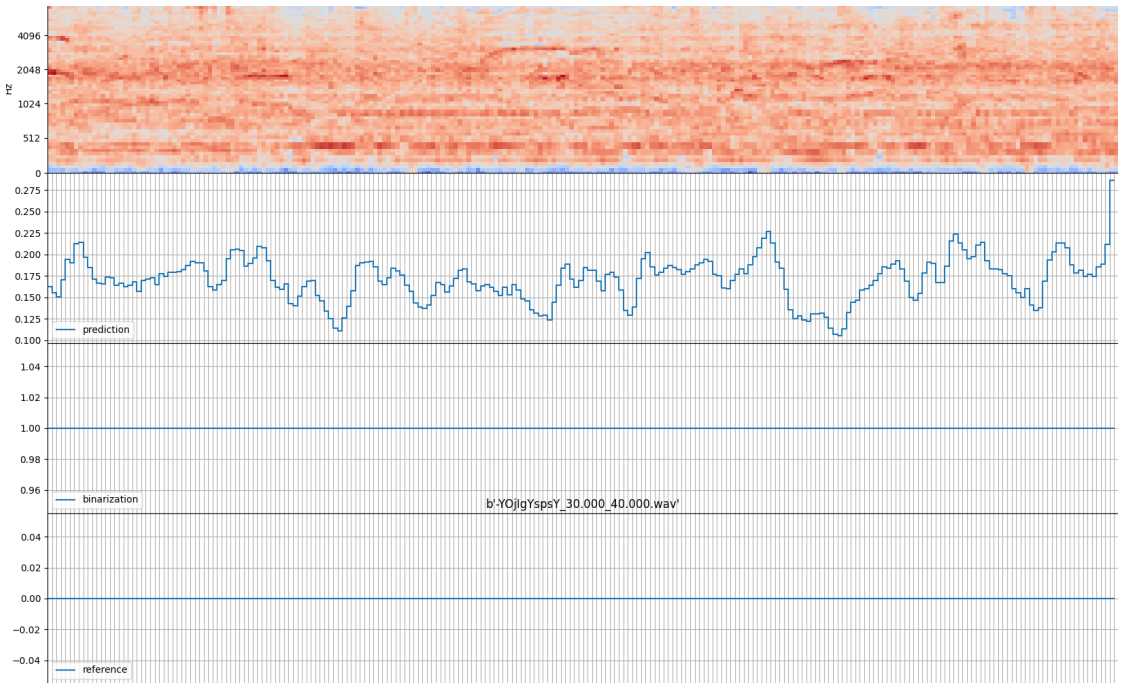


Figura 4.5: *Predizione del modello semplificato addestrato con il set di dati originale*

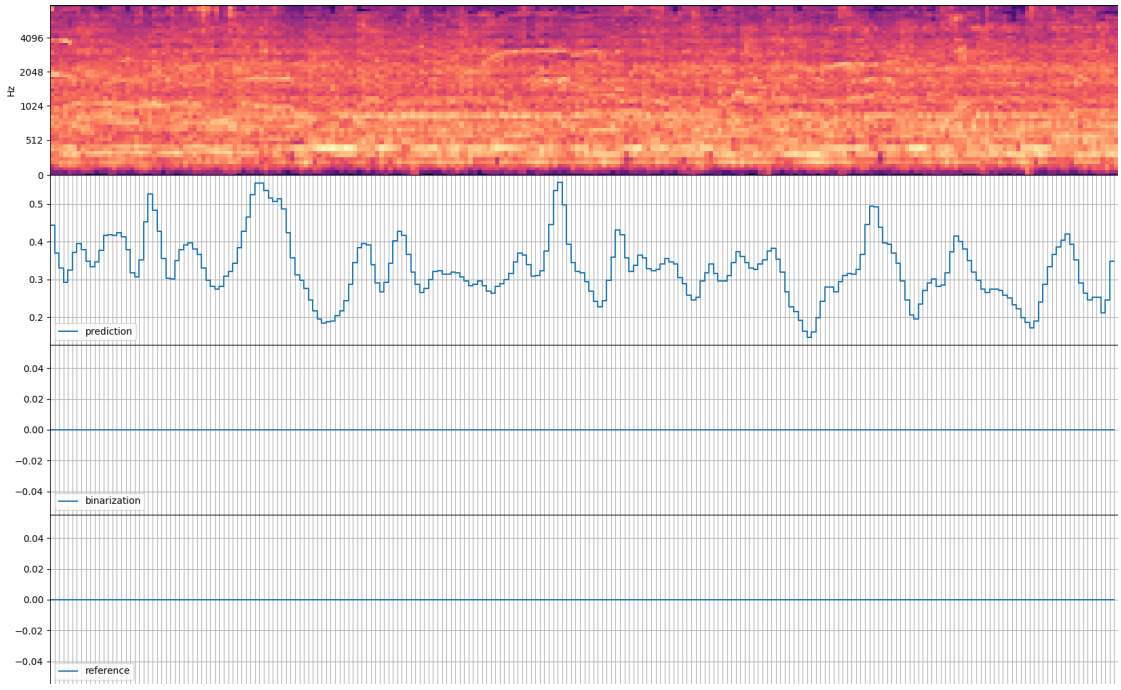


Figura 4.6: *Predizione del modello semplificato addestrato il set di dati nuovo*

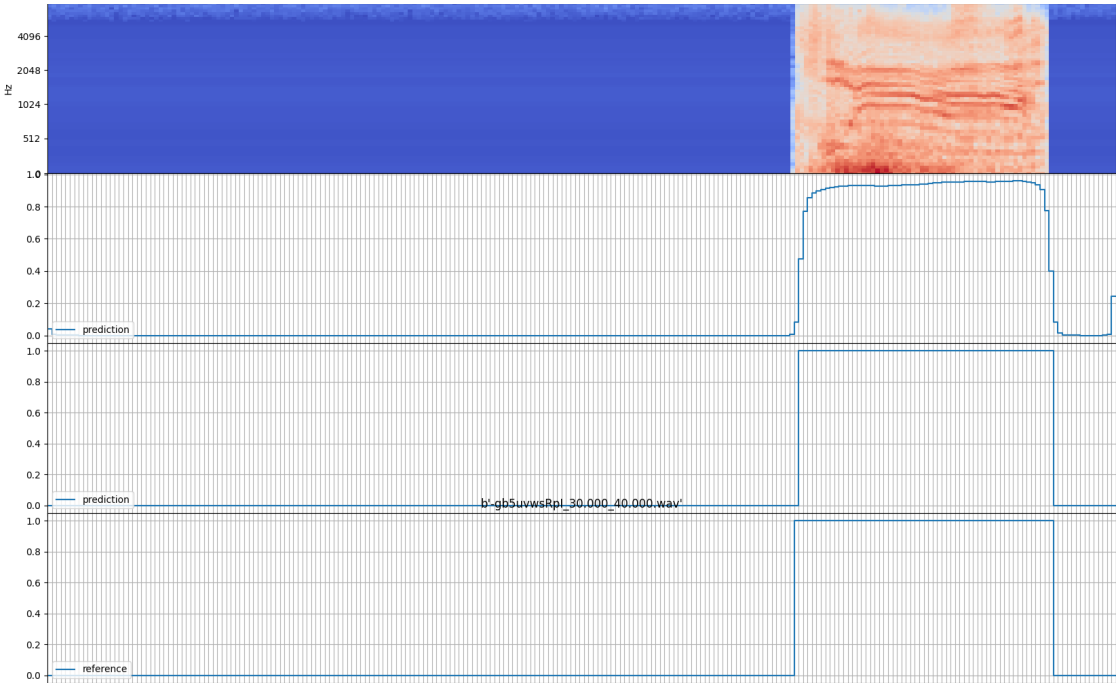


Figura 4.7: *Rilevamento dell'urlo - modello semplificato con il nuovo set di training*

Modello	FLASH (KB)	RAM (KB)	Tempo totale (ms)	Accuratezza (%)
Originale	18.82	304.9	1085	80.5
Quantizzato	4.85	80.92	717	80.3

Tabella 4.5: Confronto tra modello originale e quantizzato tramite X-CUBE-AI

In Figura 4.8 sono riportati i valori per ciascuna fase: nella fase di pre-processamento si misura il tempo necessario per calcolare una colonna dello spettrogramma. Segue poi l'inferenza della rete neurale si può notare l'impatto della quantizzazione che permette di ridurre il tempo di inferenza. Ciò si riflette anche sull'energia dissipata, calcolata considerando sempre l'alimentazione pari a 1.8V. Anche in questo caso, in seguito alla quantizzazione, l'occupazione di memoria si riduce di circa quattro volte, mentre il tempo di esecuzione è 1.5 volte inferiore. Tuttavia non è ancora accettabile per questa applicazione, in quanto l'obiettivo è di ottenere un tempo di

esecuzione pari alla lunghezza temporale del singolo segmento audio, ovvero 64ms. Pertanto, è necessario introdurre un'accelerazione tramite FPGA per raggiungere questo obiettivo.

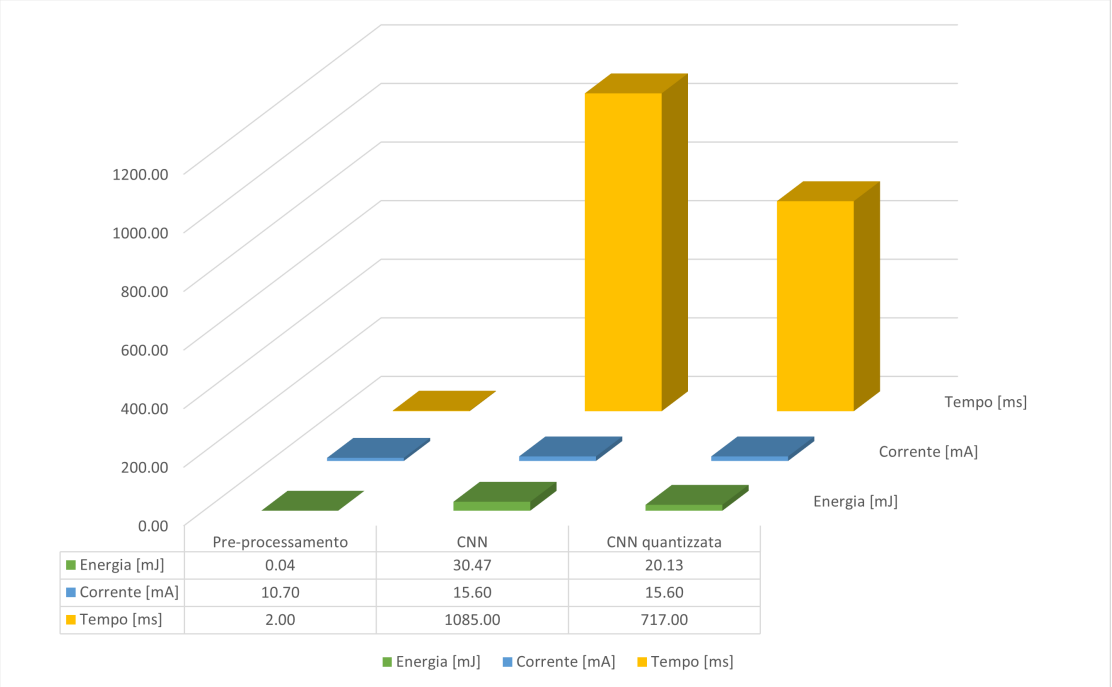


Figura 4.8: *Profilazione di ciascuna fase dell'algoritmo*

4.7 Accelerazione con FPGA

Questa sezione descrive l'utilizzo dello strumento hls4ml per convertire il modello Python della rete neurale in codice C++ e successivamente utilizzare il flusso integrato con Vivado per ottenere il codice HDL. Una volta generato il progetto ad alto livello è stata eseguita la simulazione per verificare il comportamento del codice C++ e generare le uscite che, se la conversione (inclusa la compressione e la quantizzazione di pesi e bias) è avvenuta correttamente, coincidono con quelle del modello Python di partenza. Una volta verificato che le uscite sono corrette il passo successivo consiste nella sintesi del modello, dove il codice C++ è convertito in codice HDL. Completata la sintesi è possibile ottenere una stima del numero di risorse utilizzate e della latenza.

4.7.1 Conversione del modello con hls4ml

Il punto di partenza per effettuare la conversione è il modello Keras descritto nella sezione precedente. Per trasferire la configurazione della quantizzazione introdotta precedentemente a hls4ml si utilizza QKeras [51], una libreria sviluppata da Google che permette di addestrare in modo consapevole la rete neurale sfruttando dei quantizzatori per specificare quanti bit utilizzare in ogni livello. Inoltre, la libreria è scritta in modo da applicare modifiche minime al codice precedente.
Il risultato è riportato nel Listing 4.1 da cui si può notare che le modifiche necessarie consistono nell'aggiunta di una `Q` a monte del nome del livello originale Keras e nella definizione del tipo di quantizzazione, ad esempio i parametri `kernel_quantizer` e `bias_quantizer`. In particolare, queste modifiche sono state applicate solo ai livelli che effettuano dei calcoli sui dati in ingresso, mentre i livelli che effettuano solo delle manipolazioni, come i livelli `Reshape` e `MaxPooling2D`, restano invariati. In questo caso kernel e bias sono quantizzati con QKeras usando 8 bit totali e 6 bit per la parte intera. Il parametro `alpha` può essere usato per cambiare la scala assoluta dei pesi mantenendoli discreti all'interno dell'intervallo. Le funzioni di attivazione sono definite da una versione quantizzata della funzione ReLU che viene passata come parametro al livello `QActivation` specificando il numero di bit di precisione e della parte intera.
A questo punto il modello con questa nuova configurazione è stato riaddestrato e salvato prima di passare alla fase di conversione in C++. Successivamente sono state utilizzate le funzioni di hls4ml per estrarre la configurazione dal modello QKeras e convertirlo in codice C++.

Listing 4.1: Modello quantizzato con Qkeras.

```
from tensorflow.keras.layers import Input, Activation
from qkeras import quantized_bits
from qkeras import QDense, QActivation
from qkeras import QConv2D

    x = Input(shape=(n_time, n_freq), name='in_layer')
    x = Reshape((n_time, n_freq, 1))(x)
    x = QConv2D(16, (3, 3), padding="same", use_bias=False,
        kernel_quantizer=quantized_bits(8,6,alpha=1),
        bias_quantizer=quantized_bits(8,6,alpha=1),
        name="conv2d_1")(a1)
    x = QActivation("quantized_relu(8,6)",name="act_1")(a1)
    x = MaxPooling2D(pool_size=(1, 4))(a1)
    x = QConv2D(16, (3, 3), padding="same", use_bias=False,
        kernel_quantizer=quantized_bits(8,6,alpha=1),
        bias_quantizer=quantized_bits(8,6,alpha=1),
        name="conv2d_2")(a1)
    x = QActivation("quantized_relu(8,6)",name="act_2")(a1)
    x = MaxPooling2D(pool_size=(1, 4))(a1)
```

```
    x = QConv2D(16, (3, 3), padding="same", use_bias=True,
        kernel_quantizer=quantized_bits(8,6,alpha=1),
        bias_quantizer=quantized_bits(8,6,alpha=1),
        name="conv2d_3")(a1)
    x = QActivation("quantized_relu(8,6)",name="act_3")(a1)
    x = MaxPooling2D(pool_size=(1, 4))(a1)
    x = Reshape((240, 16))(a1)
    x = QDense(1,
        kernel_quantizer="quantized_bits(8,6,1)",
        bias_quantizer="quantized_bits(8,6)",
        kernel_regularizer=l1(0.001),
        name="dense")(a1)
    out = Activation("sigmoid", name="act_4")(a1)
```

Un'altra caratteristica di hls4ml è che permette di profilare numericamente il modello al fine di individuare se la precisione utilizzata è esatta. In particolare, con questo metodo è possibile visualizzare come i parametri del modello QKeras sono stati interpretati durante la conversione in termini di pesi e predizioni. In questo modo è possibile individuare eventuali errori di rappresentazione derivanti dalla quantizzazione e verificare l'efficacia del modello. Un esempio è riportato in Figura 4.9 che rappresenta la distribuzione delle predizioni del modello QKeras, ottenuta utilizzando il formato *boxplot*. Ciascun rettangolo rappresenta la distribuzione dei valori dai valori di quantile più basso al più alto del vettore contenente le attivazioni. La linea centrale del rettangolo corrisponde alla media, mentre le linee nere rappresentano l'intervallo dei valori. Come si può notare, la distribuzione delle predizioni del modello QKeras si adatta correttamente alla precisione del modello hls4ml perché i rettangoli colorati sono all'interno degli intervalli grigi. Pertanto, è possibile verificare che la precisione utilizzata è adeguata, altrimenti sarebbe stato necessario aumentare la precisione e quindi il numero di bit utilizzati, oppure modificare il modello Keras.

Inoltre, è possibile confrontare le predizioni tra il modello senza quantizzazione ed il modello QKeras. Per fare ciò è stata utilizzata la curva ROC (*Receiver Operating Characteristic*) riportata in Figura 4.10. Si può notare che la capacità di distrinzione tra la presenza e assenza di urlo è elevata dati i valori di area sottesa alle curve *ROC* e soprattutto la perdita di accuratezza nel modello QKeras non è eccessiva, come si era visto già in precedenza.

Prima di procedere con la sintesi RTL a partire dal modello C++ è necessario simulare il progetto generato da hls4ml per verificare che le uscite siano coerenti con il modello QKeras di partenza. Per fare ciò è stato utilizzato il testbench generato da hls4ml che acquisisce i dati del set di test da un file e li passa in ingresso alla funzione principale che realizza il modello della rete neurale. Infine, salva le uscite su un file e le confronta con quelle ottenute dal modello QKeras utilizzando una

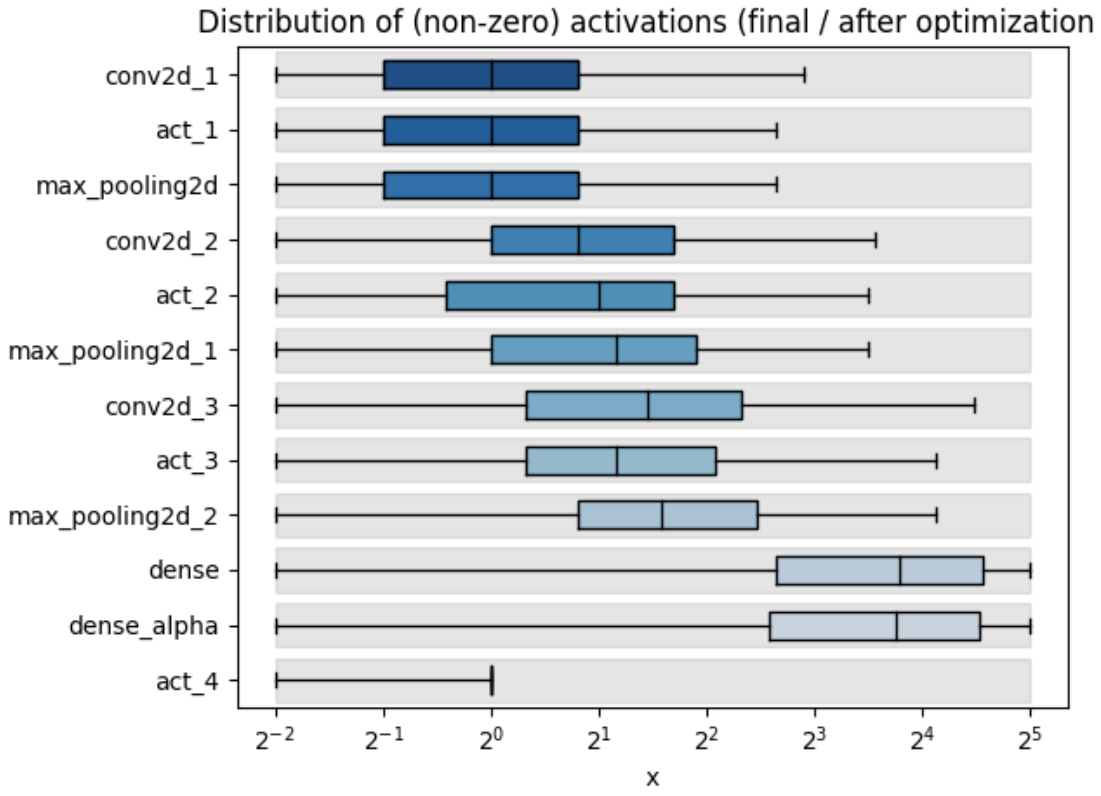


Figura 4.9: *Distribuzione Attivazioni*

funzione che calcola la differenza e la rappresenta normalizzata tra 0 e 1 in un grafico in formato *boxplot*. Se la differenza è maggiore o uguale al valore originale, allora la differenza normalizzata viene impostata pari a 1. Se invece la differenza è minore del valore originale, allora la differenza normalizzata sarà pari al rapporta tra la differenza ed il valore originale. In Figura 4.11 è riportato il grafico risultante da cui si può notare che le uscite dei due modelli coincidono.

4.7.2 Sintesi RTL con Vivado HLS

Una volta verificata la correttezza del modello C++, il prossimo passo consiste nella sintesi del modello, dove il codice ad alto livello viene convertito in codice HDL. È possibile selezionare la piattaforma di interesse, in questo caso è stata impostata quella di default xcu250-figd2104-2L-e, ed il periodo di clock fissato a 5n sec. Alla fine di questo processo è possibile avere informazioni sul numero di risorse e la latenza. Nella Tabella 4.6 sono riassunti i risultati latenza sia in colpi di clock che assoluta. Si può notare che quella assoluta è di circa 50ms,

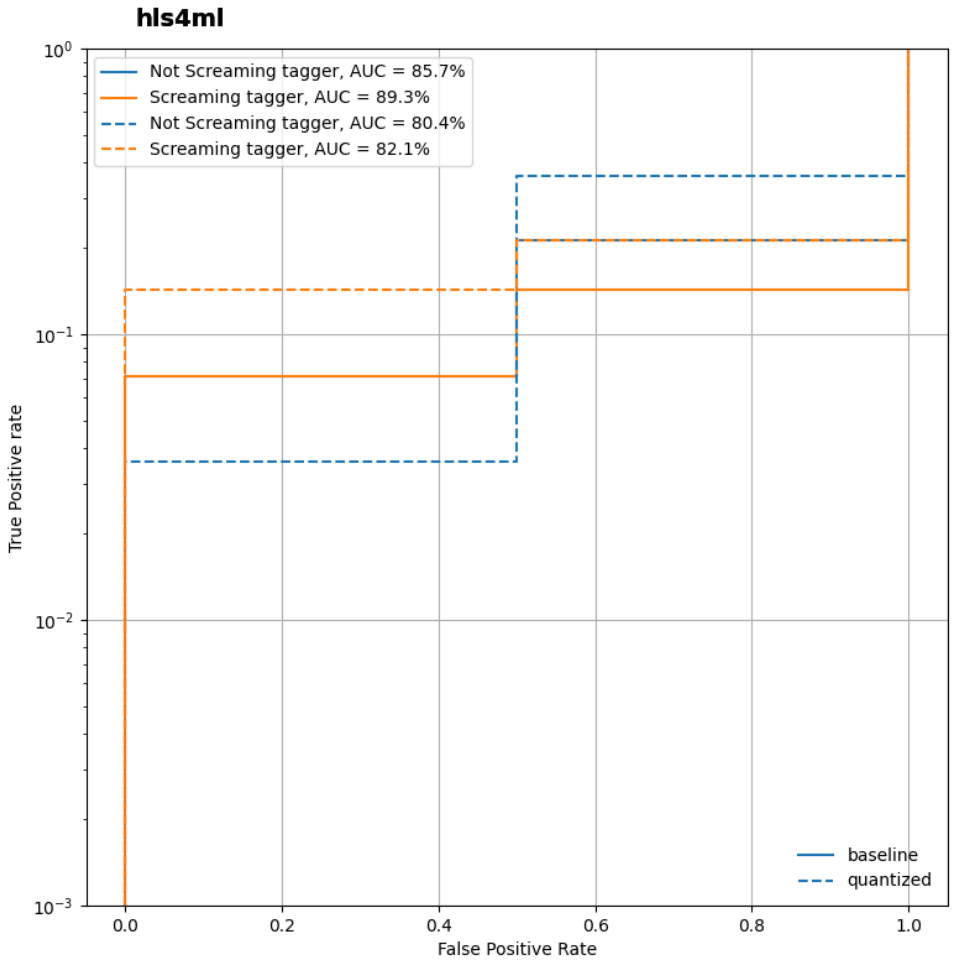


Figura 4.10: *Curva ROC*

significativamente inferiore rispetto a quella ottenuta con l'inferenza del modello quantizzato sul microcontrollore e rispetto all'obiettivo di 64ms.

Nella Tabella 4.7 è riportata la stima di utilizzo delle risorse, dove si indica come ciascun elemento viene mappato rispetto ai componenti disponibili. Si può notare che, anche se il totale è inferiore al numero di risorse disponibili, la complessità risulta comunque elevata. Ciò implica che, pur riuscendo a mappare la rete su questa piattaforma e a raggiungere l'obiettivo di timing prefissato, non sarebbe possibile realizzare la sintesi su un'altra piattaforma a basso consumo, dato che

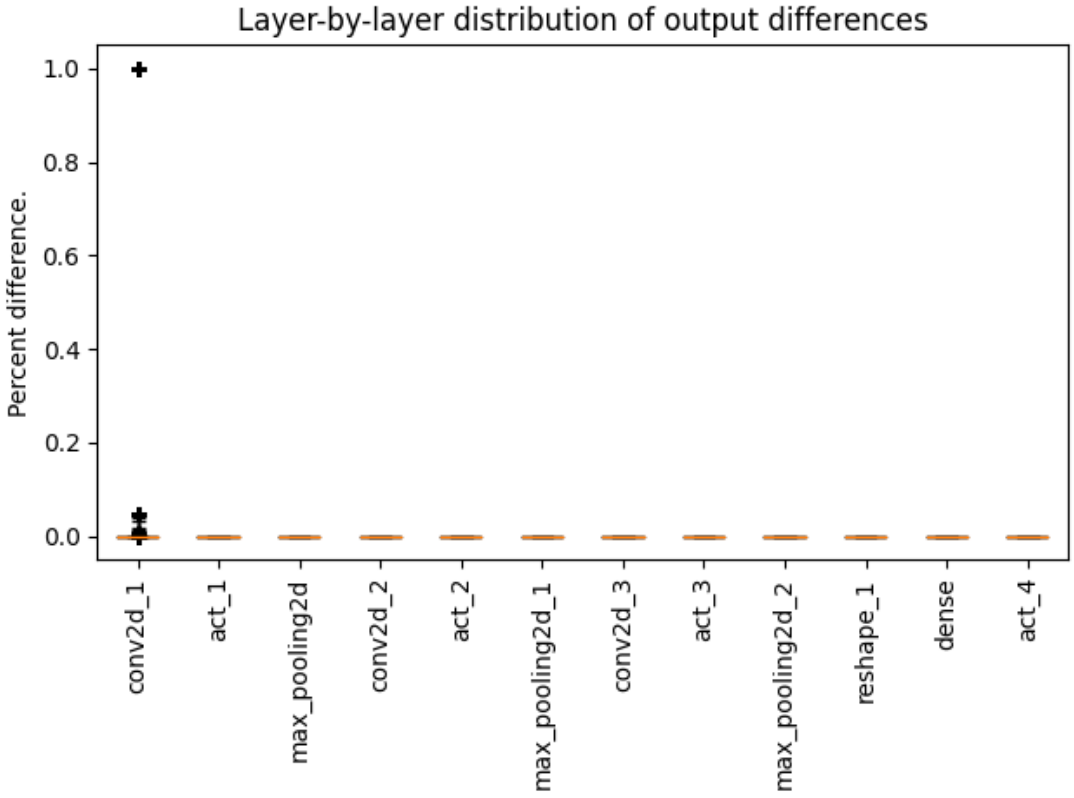


Figura 4.11: *Differenza normalizzata tra il modello QKeras ed il modello C++*

Latency (cycles)		**Latency (absolute)**		**Interval (cycles)**	
min	max	min	max	min	max
10054940	10059523	50.275 ms	50.298 ms	31945	10058005

Tabella 4.6: Stima della latenza

l'obiettivo dell'applicazione è di raggiungere un consumo di potenza molto ridotto. Pertanto, si può dire che, nonostante la sintesi ad alto livello consente di ottimizzare e sintetizzare automaticamente il progetto nel minor tempo possibile, gli strumenti di sintesi automatica non permettono di raggiungere gli stessi risultati della sintesi effettuata manualmente ed andrebbero messi a punto per la particolare applicazione.

Name	**BRAM_18K**	**DSP48E**	**FF**	**LUT**	**URAM**
DSP	-	-	-	-	-
Expression	-	-	0	32	-
FIFO	782	-	18126	47457	-
Instance	162	243	268510	236508	0
Memory	-	-	-	-	-
Multiplexer	-	-	-	36	-
Register	-	-	6	-	-
Total	944	243	286642	284033	0
Available	5376	12288	3456000	1728000	1280
Utilization (%)	17	1	8	16	0

Tabella 4.7: Stima di utilizzo delle risorse

Capitolo 5

Conclusioni

L'obiettivo di questo studio consiste nel realizzare un metodo di rilevamento di eventi sonori basato sulle reti neurali utilizzando un microcontrollore. In questo capitolo saranno presentati i risultati ottenuti e i lavori futuri che andrebbero svolti per risolvere alcuni punti rimasti aperti.

Il risultato finale di questo progetto è stato quello di ottenere un'accuratezza di riconoscimento dell'urlo pari a **80.5%**, ottenuta semplificando il modello di partenza utilizzato dai vincitori della competizione DCASE 2017 e migliorando la strategia di addestramento della rete neurale utilizzando solo i file audio che contengono l'urlo di interesse. Inoltre, si è scoperto che addestrando il modello originale con il nuovo set di training si ottiene un'accuratezza pari a **87.8%** che risulta superiore a quella ottenuta addestrando il modello con il set di training di partenza (vedi Tabella 4.4).
Successivamente è stata introdotta la quantizzazione per comprimere il modello e ridurre la latenza dell'inferenza: utilizzando il metodo di quantizzazione consapevole è possibile ridurre al minimo la perdita di accuratezza dovuta alla riduzione della precisione, infatti l'accuratezza del modello quantizzato risulta praticamente identica a quella del modello floating point. Nonostante ciò, il tempo di esecuzione risulta comunque inaccettabile per l'applicazione. Per questo motivo, è stato necessario ricorrere a strumenti di sintesi ad alto livello che permettono di ottenere automaticamente l'architettura ottimale mappata su FPGA. Utilizzando hls4ml è stato possibile ottenere una stima della latenza che risulta essere intorno a 50ms. Tuttavia, il numero di risorse utilizzate risulta ancora elevato (vedi Tabella 4.7). Quest'ultima osservazione suggerisce di esplorare nei lavori futuri altri strumenti di sintesi ad alto livello come Bambu per confrontare i risultati ottenuti dalla sintesi su altre piattaforme e vedere se sono presenti differenze in termini di ottimizzazione dell'architettura.

5.0.1 Lavori futuri

Successivamente ai risultati ottenuti, rimangono alcuni punti aperti che andrebbero considerati in futuro:

- **Bambu**
  Durante questo lavoro è stato utilizzato hls4ml per la sintesi ad alto livello. Tuttavia, esistono altri strumenti open-source come Bambu che consente di effettuare la sintesi su diverse piattaforme. In questo modo si potrebbero confrontare i risultati dei due compilatori per individuare eventuali differenze nella sintesi. Comunque, si prevede che la complessità dell'architettura sintetizzata non vari significativamente, dato che il flusso di compilazione e le tecniche di ottimizzazione dei compilatori sono simili. Per questo motivo, risulta necessario realizzare l'acceleratore utilizzando linguaggi di descrizione dell'hardware per ottenere il migliore risultato in termini di complessità e prestazioni.

- **Sintesi del modello su FPGA a basso consumo**
  Come già detto, gli strumenti di sintesi ad alto livello sono utili perché consentono di generare automaticamente l'architettura e di ottenere in breve tempo una stima della sua complessità. Tuttavia, per avere un design ottimale, la strada migliore da seguire è quella di descrivere l'algoritmo con i linguaggi di descrizione dell'hardware. In questo modo è possibile ottimizzare la rete per effettuare la sintesi su una piattaforma a basso consumo.

- **Acquisizione dei campioni sonori**
  Il metodo utilizzato prevede il processamento dei campioni sonori dal set di file audio memorizzati sulla scheda microSD. Al fine di impiegare l'applicazione in una situazione reale è necessario utilizzare i microfoni presenti sulla piattaforma. Per questo motivo il codice sviluppato andrebbe integrato come spiegato nella sezione 3.3.1 in modo da acquisire i campioni sonori da una scena acustica reale.

Bibliografia

[1] Andrea Zanella, Nicola Bui, Angelo Castellani, Lorenzo Vangelista e Michele Zorzi. «Internet of Things for Smart Cities». In: *IEEE Internet of Things Journal* 1.1 (2014), pp. 22–32. DOI: 10.1109/JIOT.2014.2306328 (cit. alle pp. 1, 2).

[2] Pallavi Sethi e Smruti Sarangi. «Internet of Things: Architectures, Protocols, and Applications». In: *Journal of Electrical and Computer Engineering* 2017 (gen. 2017), pp. 1–25. DOI: 10.1155/2017/9324035 (cit. alle pp. 1–3, 5).

[3] Shivangi Vashi, Jyotsnamayee Ram, Janit Modi, Saurav Verma e Chetana Prakash. «Internet of Things (IoT): A vision, architectural elements, and security issues». In: *2017 International Conference on I-SMAC (IoT in Social, Mobile, Analytics and Cloud) (I-SMAC)*. 2017, pp. 492–496. DOI: 10.1109/I-SMAC.2017.8058399 (cit. a p. 2).

[4] Emre Cakir, Toni Heittola, Heikki Huttunen e Tuomas Virtanen. «Polyphonic sound event detection using multi label deep neural networks». In: *2015 International Joint Conference on Neural Networks (IJCNN)*. 2015, pp. 1–7. DOI: 10.1109/IJCNN.2015.7280624 (cit. alle pp. 4, 73).

[5] Annamaria Mesaros, Toni Heittola e Tuomas Virtanen. «TUT database for acoustic scene classification and sound event detection». In: *2016 24th European Signal Processing Conference (EUSIPCO)* (2016), pp. 1128–1132 (cit. alle pp. 9, 17, 52).

[6] Ian McLoughlin. «Introduction». In: *Applied Speech and Audio Processing: With Matlab Examples*. Cambridge University Press, 2009, pp. 1–6. DOI: 10.1017/CBO9780511609640.002 (cit. alle pp. 10, 14).

[7] Tae Hong Park. *Introduction to Digital Signal Processing: Computer Musically Speaking*. Nov. 2009. ISBN: 978-981-279-027-9. DOI: 10.1142/6705 (cit. a p. 11).

[8] Julius O. Smith. *Spectral Audio Signal Processing*. online book, 2011 edition. http://ccrma.stanford.edu/~jos/sasp/ (cit. a p. 13).

[9] A. Pikrakis e T. Giannakopoulos. *Introduction to Audio Analysis: A MATLAB Approach*. Elsevier Science & Technology, 2014 (cit. a p. 14).

[10] Douglas L. Jones e Ivan Selesnick. *The DFT, FFT, and Practical Spectral Analysis*. 2007. URL: `http://cnx.org/content/col10281/1.2/` (cit. a p. 14).

[11] B. Bogert, M. Healy e J. Tukey. «The quefrency analysis of time series for echoes: cepstrum, pseudo-autocovariance, cross-spectrum, and saphe-cracking». In: New York: Proceedings of the Symposium on Time Series Analysis, 1963, pp. 209–243 (cit. a p. 16).

[12] Mingchun Liu e Chunru Wan. «Feature Selection for Automatic Classification of Musical Instrument Sounds». In: *Proceedings of the 1st ACM/IEEE-CS Joint Conference on Digital Libraries*. JCDL '01. Roanoke, Virginia, USA: Association for Computing Machinery, 2001, pp. 247–248. ISBN: 1581133456. DOI: `10.1145/379437.379663`. URL: `https://doi.org/10.1145/379437.379663` (cit. a p. 16).

[13] Daniele Barchiesi, Dimitrios Giannoulis, Dan Stowell e Mark D. Plumbley. «Acoustic Scene Classification: Classifying environments from the sounds they produce». In: *IEEE Signal Processing Magazine* 32.3 (2015), pp. 16–34. DOI: `10.1109/MSP.2014.2326181` (cit. a p. 16).

[14] H. Terasawa, M. Slaney e J. Berger. «Perceptual distance in timbre space». In: Limerick, Ireland: Proceedings of Eleventh Meeting of the International Conference on Auditory Display, 2005, pp. 61–68 (cit. a p. 17).

[15] Dinesh Vij, Naveen Aggarwal, Bhaskaran Raman, K.K. Ramakrishnan e Divya Bansal. *Acoustic Scene Classification Based on Spectral Analysis and Feature-Level Channel Combination*. Rapp. tecn. DCASE2016 Challenge, set. 2016 (cit. a p. 17).

[16] J. H. Foleiss e T. F. Tavares. «Mel-band features for DCASE 2016 acoustic scene classification task». In: *DCASE* (set. 2016) (cit. a p. 17).

[17] Bachu R., Adapa B.K., Kopparthi S. e Buket Barkana. «Separation of Voiced and Unvoiced Speech Signals using Energy and Zero Crossing Rate». In: mar. 2008 (cit. a p. 17).

[18] Annamaria Mesaros, Toni Heittola, Antti Eronen e Tuomas Virtanen. «Acoustic event detection in real-life recordings». In: lug. 2014 (cit. a p. 17).

[19] Yann LeCun, Y. Bengio e Geoffrey Hinton. «Deep Learning». In: *Nature* 521 (mag. 2015), pp. 436–44. DOI: `10.1038/nature14539` (cit. alle pp. 17, 20).

[20] Daniele Battaglino, Ludovic Lepauloux e Nicholas Evans. «Acoustic scene classification using convolutional neural networks». In: *DCASE 2016, Workshop on Detection and Classification of Acoustic Scenes and Events, September 3rd, 2016, Budapest, Hungary*. Budapest, 2016 (cit. alle pp. 17, 21, 25, 61).

[21] Annamaria Mesaros, Toni Heittola e Tuomas Virtanen. «Metrics for Polyphonic Sound Event Detection». In: *Applied Sciences* 6 (mag. 2016), p. 162. DOI: 10.3390/app6060162 (cit. alle pp. 18, 28, 30).

[22] Y. Lecun, L. Bottou, Y. Bengio e P. Haffner. «Gradient-based learning applied to document recognition». In: *Proceedings of the IEEE* 86.11 (1998), pp. 2278–2324. DOI: 10.1109/5.726791 (cit. a p. 23).

[23] Michele Valenti, Aleksandr Diment, Giambattista Parascandolo, Stefano Squartini e Tuomas Virtanen. «A Convolutional Neural Network approach for acoustic scene classification». In: mag. 2017. DOI: 10.1109/IJCNN.2017.7966035 (cit. alle pp. 26, 52).

[24] Yerin Lee, Soyoung Lim e Il-Youp Kwak. «CNN-Based Acoustic Scene Classification System». In: *Electronics* 10 (feb. 2021), p. 371. DOI: 10.3390/electronics10040371 (cit. a p. 26).

[25] Michael A Nielsen. «Neural networks and deep learning». In: *Determination press* 25 (2015) (cit. alle pp. 27, 28).

[26] *TensorFlow model optimization*. Tensorflow.org. URL: https://www.tensorflow.org/model_optimization/guide (cit. a p. 31).

[27] *Xilinx. Vivado Design Suite User Guide: High-Level Synthesis*. Xilinx. URL: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2017_4/ug902-vivado-high-level-synthesis.pdf (cit. alle pp. 36, 39, 40).

[28] Farah Fahim et al. *hls4ml: An Open-Source Codesign Workflow to Empower Scientific Low-Power Machine Learning Devices*. 2021. arXiv: 2103.05579 [cs.LG] (cit. alle pp. 37–39).

[29] R. Nane et al. «A Survey and Evaluation of FPGA High-Level Synthesis Tools». In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 35.10 (ott. 2016), pp. 1591–1604. ISSN: 0278-0070. DOI: 10.1109/TCAD.2015.2513673 (cit. a p. 40).

[30] Fabrizio Ferrandi, Vito Giovanni Castellana, Serena Curzel, Pietro Fezzardi, Michele Fiorito, Marco Lattuada, Marco Minutoli, Christian Pilato e Antonino Tumeo. «Invited: Bambu: an Open-Source Research Framework for the High-Level Synthesis of Complex Applications». In: *2021 58th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 2021, pp. 1327–1330. DOI: 10.1109/DAC18074.2021.9586110 (cit. a p. 42).

[31] *DS11585: STM32L496xx datasheet*. STMicroelectronics (cit. a p. 43).

[32] *MP34DT01-M: MEMS audio sensor omnidirectional digital microphone*. STMicroelectronics (cit. a p. 43).

[33] *UM2160 User manual: Discovery kit with STM32L496AG MCU*. STMicroelectronics (cit. a p. 44).

[34] *LTE Cellular to Cloud Pack with STM32L496AG MCU*. STMicroelectronics. URL: https://www.st.com/en/evaluation-tools/p-l496g-cell02.html#overview (cit. a p. 45).

[35] *STM32CubeIDE Integrated Development Environment for STM32*. STMicroelectronics. URL: https://www.st.com/en/development-tools/stm32cubeide.html (cit. alle pp. 45, 48).

[36] Silvia Santini e Andrea Vitaletti. «Wireless Sensor Networks for Environmental Noise Monitoring». In: *GI/ITG KuVS Fachgespraech Drahtlose Sensornetze* (gen. 2007) (cit. a p. 46).

[37] Santini S. e Vitaletti A. «First experiences using wireless sensor networks for noise pollution monitoring». In: Glasgow: Proceedings of the 2008 Workshop on Real-World Wireless Sensor Networks (REALWSN) (cit. a p. 46).

[38] *AN5027 Application Note: Interfacing PDM digital microphones using STM32 MCUs and MPUs*. STMicroelectronics (cit. alle pp. 47–49).

[39] *UM2031 User manual: Sound meter library software expansion for STM32Cube*. STMicroelectronics (cit. a p. 49).

[40] *DCASE2016 task 1, acoustic scene classification*. 2016. URL: http://dcase.community/challenge2016/task-acoustic-scene-classification (cit. a p. 52).

[41] *UM1721 User manual: Developing applications on STM32Cube™ with FatFs*. STMicroelectronics (cit. a p. 53).

[42] *STM32Cube function pack for ultra-low power IoT node with artificial intelligence (AI) application based on audio and motion sensing*. STMicroelectronics. URL: https://www.st.com/en/embedded-software/fp-ai-sensing1.html#overview&secondary=st_all-features_sec-nav-tab (cit. alle pp. 54, 72).

[43] *UM2526 User Manual: Getting started with X-CUBE-AI Expansion Package for Artificial Intelligence (AI)*. STMicroelectronics (cit. alle pp. 54, 60).

[44] *X-CUBE-AI Data Brief: Artificial Intelligence (AI) software expansion for STM32Cube*. STMicroelectronics (cit. a p. 56).

[45] *Model optimization*. Tensorflow.org. URL: https://www.tensorflow.org/lite/performance/model_optimization (cit. a p. 65).

[46] Benoit Jacob, Skirmantas Kligys, Bo Chen, Menglong Zhu, Matthew Tang, Andrew Howard, Hartwig Adam e Dmitry Kalenichenko. *Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference*. 2017. arXiv: 1712.05877 [cs.LG] (cit. a p. 66).

[47] *Quantization aware training*. Tensorflow.org. URL: https://www.tensorflow.org/model_optimization/guide/quantization/training (cit. a p. 68).

[48] A. Mesaros, A. Diment, B. Elizalde, T. Heittola, E. Vincent, B. Raj e T. Virtanen. «DCASE 2017 Challenge setup: Tasks, datasets and baseline system». In: Munich - Germany: DCASE 2017 - Workshop on Detection, Classification of Acoustic Scenes e Events, nov. 2017. URL: https://hal.inria.fr/hal-01627981 (cit. alle pp. 72, 73).

[49] Emre Cakir, Giambattista Parascandolo, Toni Heittola, Heikki Huttunen e Tuomas Virtanen. «Convolutional Recurrent Neural Networks for Polyphonic Sound Event Detection». In: *IEEE/ACM Transactions on Audio, Speech, and Language Processing* 25.6 (giu. 2017), pp. 1291–1303. ISSN: 2329-9304. DOI: 10.1109/taslp.2017.2690575. URL: http://dx.doi.org/10.1109/TASLP.2017.2690575 (cit. a p. 73).

[50] Yong Xu, Qiuqiang Kong, Wenwu Wang e Mark D. Plumbley. «Large-Scale Weakly Supervised Audio Classification Using Gated Convolutional Neural Network». In: *2018 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. 2018, pp. 121–125. DOI: 10.1109/ICASSP.2018.8461975 (cit. alle pp. 73–75, 78).

[51] Claudionor N. Coelho et al. «Automatic heterogeneous quantization of deep neural networks for low-latency inference on the edge for particle detectors». In: *Nature Machine Intelligence* 3.8 (giu. 2021), pp. 675–686. ISSN: 2522-5839. DOI: 10.1038/s42256-021-00356-5. URL: http://dx.doi.org/10.1038/s42256-021-00356-5 (cit. a p. 82).