FACOLTÀ DI INGEGNERIA

CORSO DI LAUREA MAGISTRALE IN INGEGNERIA ELETTRONICA

IMPLEMENTAZIONE DI UNA TINY RECURRENT NEURAL NETWORK SU PIATTAFORMA EMBEDDED PER IL RICONOSCIMENTO DI ATTIVITÀ FISICHE

IMPLEMENTATION OF A TINY RECURRENT NEURAL NETWORK ON AN EMBEDDED SYSTEM FOR HUMAN ACTIVITY RECOGNITION

Relatore:
LAURA FALASCHETTI

Studente:
SARA COLAZILLI

ANNO ACCADEMICO 2022-2023

SOMMARIO

SOMMARIO 2

INTRODUZIONE 4

CAPITOLO 1 RETI NEURALI 6

1.1 Percettrone 8

1.2 Topologia 9

1.3 Reti Neurali Ricorrenti 12

1.3.1 Long Short-Time Memory Recurrent Neural Network 12

1.4 Apprendimento 15

1.5 Stima di una Rete Neurale 17

1.6 Funzione di Attivazione 19

1.7 Classificazione 23

1.8 Valutazione 26

CAPITOLO 2 CASO DI STUDIO 29

2.1 Dataset 32

2.1.1 Suddivisione Dati 35

2.2 Pre-Processing dei dati 36

2.2.1 Pulizia dei dati 36

2.2.2 Oversampling 37

2.2.3 Finestratura dei dati 39

2.3 Analisi della Componente Principale (PCA) 40

CAPITOLO 3 IMPLEMENTAZIONE RNN 43

3.1 Architettura della RNN 43

3.2 Ambiente di Sviluppo 50

3.3 Risultati 51

3.3.1 Prestazioni della RNN al variare del Pre-Processing 52

3.3.2 Confronto con lo Studio di Riferimento 55

CAPITOLO 4 PORTING SU DISPOSITIVO EMBEDDED 57
4.1 Ambiente di sviluppo 58
4.2 Implementazione 61
4.3 Quantizzazione 62
4.3.1 TensorFlow Lite 63
4.3.2 QKeras 67
4.4 Tiny RNN 70
CONCLUSIONI 77

INTRODUZIONE

Al giorno d'oggi la tecnologia riveste un ruolo fondamentale nella nostra vita quotidiana. L'uso di dispositivi connessi è in forte crescita e si cerca costantemente di realizzare sistemi in grado di riconoscere gli eventi che accadono nell'ambiente circostante e capaci di eseguire elaborazioni complesse in base alla loro intelligenza. In questo contesto le tecniche di apprendimento automatico, sia quelle tradizionali di Machine Learning (k-NN, SVM, …) che quelle più recenti di Deep Learning (DNN, RNN), si sono dimostrate efficaci in compiti di riconoscimento e classificazione. In particolare il Deep Learning, basato sulle Reti Neurali, consente maggiore flessibilità, migliori prestazioni e gestione di dati complessi, ha però lo svantaggio di richiedere un elevato consumo energetico: la fase di training si basa solitamente su una grande quantità di dati elaborati da costosi algoritmi e, sebbene la fase di inferenza richieda meno potenza di elaborazione, è comunque un processo costoso.

Una soluzione ampiamente adottata è il Cloud Computing, ovvero l'utilizzo della potenza di calcolo di GPU disponibili nel Cloud, che però comporta la trasmissione dei dati raccolti a un server di rete, richiedendo quindi una connettività permanente e causando problemi di privacy e latenza non deterministica. In molte applicazioni è quindi necessario eseguire i calcoli sul dispositivo stesso, ma implementare reti neurali su dispositivi embedded è un compito impegnativo: questi dispositivi hanno una quantità di memoria molto ridotta, si basano su core di elaborazione generici che non implementano tecniche di parallelismo, lavorano a una frequenza generalmente molto più bassa rispetto alle GPU, e sono dotati di piccole celle di batteria. Pertanto l'implementazione di una rete neurale su microcontrollore deve affrontare importanti problemi in termini di vincoli di tempo reale, consumo energetico e occupazione di memoria.

Obiettivo di questo lavoro è esaminare un campo di applicazione di particolare interesse, quale il riconoscimento di attività fisiche a partire da biosegnali, e sviluppare una rete neurale che sia implementabile su dispositivi a basso costo e basso consumo, preservando buone prestazioni in termini di accuratezza e qualità del riconoscimento.

Il lavoro è articolato come segue:

- Nel primo capitolo verrà fornita una panoramica sulle reti neurali: vedremo il loro funzionamento, il modello biologico su cui si basano, le possibili applicazioni e i vantaggi e svantaggi che il loro utilizzo comporta. Inoltre, vedremo le diverse tipologie di reti neurali disponibili, soffermandoci sulle Reti Neurali Ricorrenti (RNN).
- Nel secondo capitolo verrà introdotto il caso di studio: descriveremo il set di dati e il pre-processamento necessario per poterli utilizzare in una RNN; vedremo la tecnica PCA di estrazione delle features, che si dimostrerà fondamentale per ottenere la convergenza della rete.
- Nel terzo capitolo verrà presentata l'architettura della rete: descriveremo la scelta degli iperparametri e analizzeremo i risultati ottenuti dalle simulazioni su PC, faremo inoltre un confronto con uno studio di riferimento, che lavora sullo stesso dataset e utilizza un classificatore bayesiano per il riconoscimento delle attività.
- Nel quarto capitolo verrà realizzato il porting su piattaforma embedded e verranno proposte diverse strategie di quantizzazione e semplificazione della rete, verranno infine esaminati i risultati ottenuti, traendo una serie di conclusioni.

Capitolo 1
RETI NEURALI

Negli ultimi anni il Machine Learning (ML) e l'intelligenza artificiale hanno visto una nuova ondata di entusiasmo alimentata dalla crescente quantità di dati e potenza di calcolo disponibili, nonché dalla scoperta di nuovi algoritmi di apprendimento che garantiscono un miglioramento delle prestazioni nei compiti assegnati. Prima della nascita dell'Intelligenza Artificiale, le macchine riuscivano ad eseguire delle istruzioni precise date in sequenza; con l'Intelligenza Artificiale c'è stata una evoluzione del metodo di apprendimento con cui l'intelligenza diventa abile in un compito o azione. Se per Intelligenza Artificiale si intende l'insieme delle strategie adottate dai sistemi informatici per svolgere un compito umano, con Machine Learning si indica una sottoclasse dell'Intelligenza Artificiale che migliora le proprie operazioni nel tempo imparando dai dati. Le Reti Neurali Artificiali e il Deep Learning sono a loro volta metodi specializzati nell'ambito della disciplina del Machine Learning, che traggono ispirazione dal funzionamento del cervello umano [1].

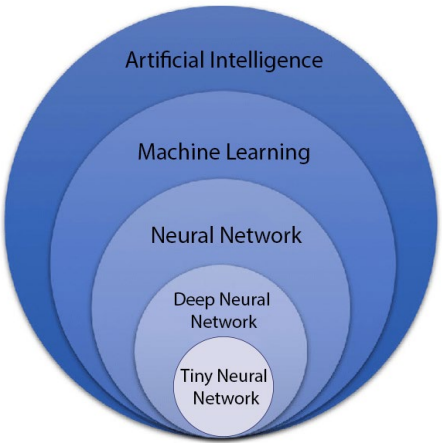

Figura 1.1 *– Intelligenza Artificiale*

Le Reti Neuronali biologiche sono circuiti complessi di neuroni interconnessi tra loro che, sottoposti a stimoli esterni, innescano una serie di impulsi nervosi che provocano mutazioni alla struttura del reticolo neuronale stesso, mediante le quali la rete riconosce, elabora e si adatta agli stimoli. Analogamente la Rete Neurale Artificiale impara ad approssimare una funzione di risoluzione di un determinato problema, come ad esempio la classificazione di immagini, tramite una progressiva modifica delle connessioni della rete al fine di migliorare l'elaborazione dell'input (stimoli) e massimizzare la precisione della funzione che si intende imparare. Le Reti Neurali sono quindi in grado di apprendere qualsiasi funzione sfruttando esclusivamente l'uso di un numero considerevole di stimoli rappresentativi del problema [2]. Il risultato del processo di apprendimento è un modello che, realizzando un mapping degli input sugli output, riesce a fornire il migliore output per ciascun dato passato in input.

Il Deep Learning è una sottocategoria del Machine Learning 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. Il Deep Learning è quindi costituito da modelli complessi che consentono il processamento di una grande quantità di dati grezzi. Nonostante la quantità di dati richiesta, e la complessità della messa a punto degli iperparametri da parte del programmatore, le reti neurali rappresentano uno strumento universale per la risoluzione di problemi ampiamente ed estremamente utile in tutte quelle applicazioni in cui le informazioni da elaborare sono estremamente complesse e/o la definizione della funzione risolutrice non sia possibile attraverso l'analisi del problema.

Le Tiny Neural Network sono reti neurali artificiali di dimensioni ridotte ed efficienti dal punto di vista energetico, ideali quindi per l'implementazione su dispositivi con risorse limitate, come microcontrollori e sistemi embedded. L'ottimizzazione delle dimensioni e delle risorse di una rete neurale può essere cruciale in applicazioni in cui il consumo energetico, la latenza o le restrizioni di spazio sono importanti, come nell'Internet delle cose (IoT), nelle applicazioni mobili o nei sistemi embedded. Tuttavia, la riduzione delle dimensioni della rete può comportare una diminuzione delle capacità di apprendimento e delle prestazioni rispetto a modelli più grandi e complessi. Pertanto, la scelta del tipo di rete neurale da utilizzare dipende dalle esigenze specifiche dell'applicazione.

1.1 Percettrone

Una rete neurale artificiale è formata da un insieme di neuroni artificiali, che simulano il funzionamento dei neuroni biologici, e da una serie di connessioni pesate, che rendono la rete un sistema adattivo, che cambia la propria struttura, ovvero i pesi delle connessioni, in base a informazioni che attraversano la rete durante una prima fase di apprendimento.

Il percettrone, la prima base teorica di un neurone artificiale, è stato modellato sul neurone biologico ed è l'unità di calcolo fondamentale della rete neurale. È formato da:

- Un insieme di connessioni, ciascuna delle quali è caratterizzata da un peso (efficacia sinaptica)
- Un sommatore che somma i segnali in input pesati dalle rispettive sinapsi, producendo in output una combinazione lineare degli input
- Una funzione di attivazione che ha il compito di limitare l'ampiezza dell'output di un neurone

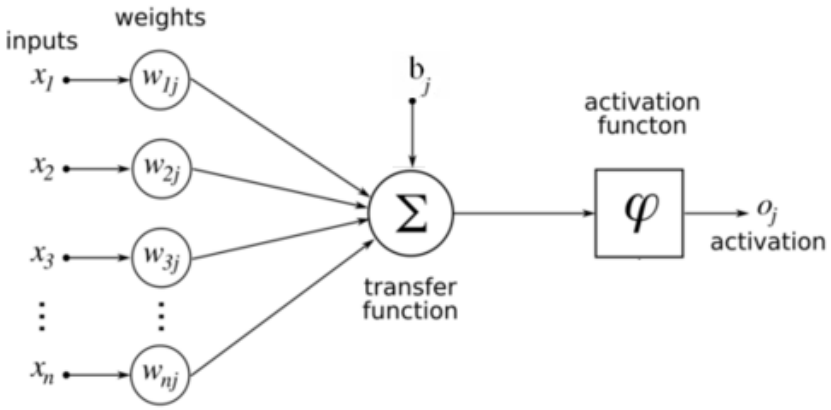

Figura 1.2 Percettrone

L'idea è che l'insieme delle sinapsi (i pesi w) possa in qualche modo essere appreso e sia in grado di controllare, in base al segno e all'intensità, l'influenza di un neurone sugli altri.
Il neurone artificiale riceve in ingresso un vettore di input numerici moltiplicati scalarmente per un vettore di pesi. Uno dei pesi viene chiamato bias ed è collegato ad un input fittizio di valore 1. Tale somma viene modellata da una funzione di attivazione, non lineare (in caso contrario la rete neurale si ridurrebbe ad un modello lineare generalizzato) che tipicamente coincide con la funzione sigmoide:

$$\sigma = \frac{1}{1+e^{-x}} \qquad (1\text{-}1)$$

La formula matematica che riassume tale funzionamento è:

$$y = \varphi\left(\sum_{i=1}^{N} w_i x_i + b\right) \qquad (1\text{-}2)$$

dove y è l'output, φ è la funzione di attivazione, N è il numero di input, x_i e w_i sono l'input i-esimo e il suo corrispondente peso, e b è il bias. L'apprendimento di un neurone è quindi rappresentato da un ribilanciamento di pesi e bias al fine di approssimare correttamente la funzione desiderata.

Il singolo neurone artificiale rappresenta la topologia più semplice che si possa avere. Grazie alla semplicità dell'interfaccia input/output del neurone, esso può essere poi collegato con altri neuroni per costituire topologie più complesse, le quali permettono di risolvere problemi di difficoltà crescente.

1.2 Topologia

Una rete neurale artificiale è caratterizzata, come visto, da un numero elevato di neuroni artificiali che si scambiano informazioni, e viene comunemente rappresentata da un grafo orientato pesato con:

- nodi → singoli neuroni
- archi → connessioni pesate tra i neuroni
- direzione degli archi → verso di propagazione del segnale

I neuroni sono organizzati in strati, chiamati layer, in modo che gli output di un determinato livello divengano gli input del livello successivo. I layer più comunemente usati sono i fully-connected, in cui ogni neurone è collegato con tutti i neuroni a monte e/o a valle. Nonostante nel tempo si siano definite reti sempre più articolate, le principali sono:

- **Single Layer Feedforward Network**

  Presenta un input-layer per la ricezione dei dati, un output-layer costituito da un numero di neuroni pari al numero di valori in uscita che si desiderano, ed un singolo layer nascosto, o hidden-layer, di dimensione arbitraria. Il nome deriva dal fatto che il segnale inserito nella rete (feed) viene propagato in avanti (forward) tra un layer e l'altro. In Figura 1.3 è rappresentato il diagramma di una semplice rete neurale con P ingressi x_j, uno strato nascosto composto da M neuroni:

  $$z_m = \sigma(\alpha_{0m} + \alpha_m^T x) \qquad (1\text{-}3)$$

  e una singola uscita

  $$y = \sum_{m=0}^{M} \beta_m z_m \qquad (1\text{-}4)$$

  - α input weights
  - β output weights
  - σ funzione di attivazione

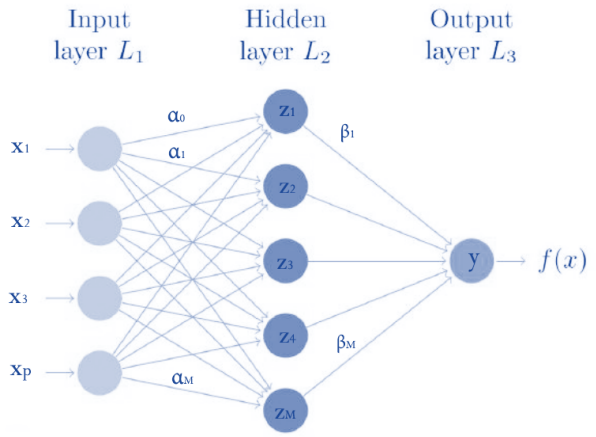

Figura 1.3 – Esempio di una feedforward Network

- **Multiple Input-Multiple Output SLFN:**

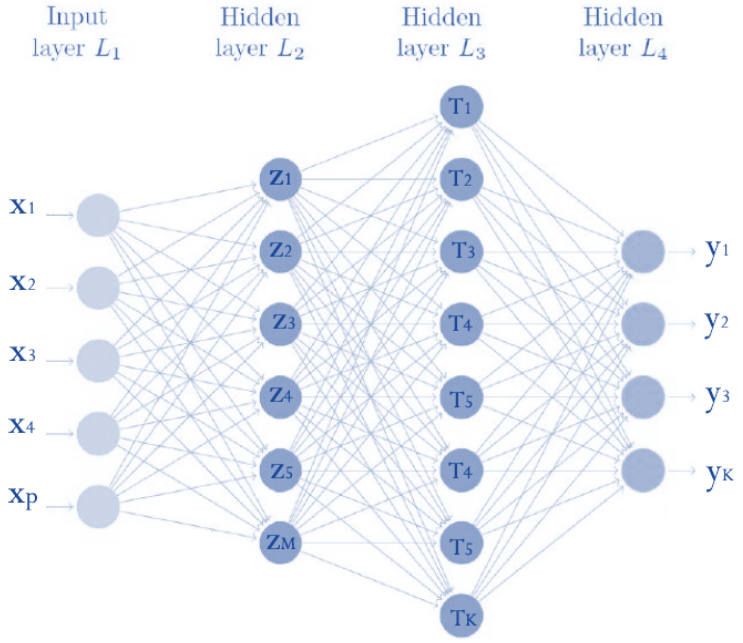

Figura 1.4 – Esempio di una MIMO SLFN

In Figura 1.4 vediamo rappresentata una rete neurale con k nodi di output (uno per classe). Analogamente a prima:

$$z_m = \sigma(\alpha_{0m} + \alpha_m^T x) \qquad m = 1, \dots, M$$
$$T_k = \sigma(\beta_{0k} + \beta_k^T z) \qquad k = 1, \dots, K$$
$$Y_k = G_k(T) \qquad k = 1, \dots, K \quad T = [T_1 \dots T_k] \qquad (1\text{-}5)$$

G_k dipende dal problema di approssimazione: in caso di regressione, ci si riconduce al caso SNFL a singola uscita: $Y_k = G_k(T) = T_k$, $G(\cdot)$ è una funzione identità.

Per problemi di classificazione in K classi, si sceglie G(k) in modo da normalizzare le variabili in uscita, ovvero in modo che $\sum_{k=1}^{K} G_k(T) = 1$, in particolare si sceglie la funzione softmax:

$$G_k(T) = \frac{e^{T_k}}{\sum_{l=1}^{K} e^{T_l}} \tag{1-6}$$

La funzione softmax restituisce una probabilità per ogni classe, e la somma totale sarà sempre pari a 1.

- **Deep Neural Network (DNN)**

  Quando sono presenti molteplici hidden-layer o si ha una composizione a cascata di Reti Neurali, si parla di Deep Neural Network (DNN). Il motivo dietro la nascita delle DNN risiede nel fatto che, nonostante il Teorema dell'Approssimazione Universale affermi che sia possibile approssimare qualsiasi funzione con una rete feedforward con un singolo hidden-layer, il numero dei suoi neuroni crescerebbe esponenzialmente con la difficoltà del problema [2].; usando invece una DNN è possibile ottenere medesimi risultati aggiungendo più hidden-layer di dimensione ridotta: ogni layer infatti elabora ed estrae feature degli output del layer precedente, offrendo un maggiore grado di astrazione. Tuttavia l'addestramento delle DNN risulta estremamente più difficile dal punto di vista pratico a causa di molte problematiche correlate alla profondità e alla complessità stessa della rete.

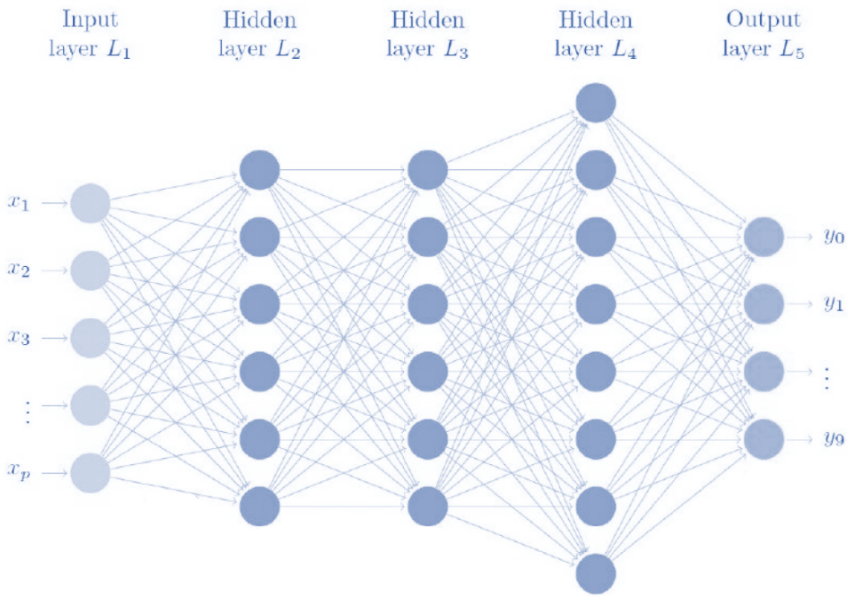

Figura 1.5 – Esempio di una Deep Neural Network

1.3 Reti Neurali Ricorrenti

Le Recurrent Neural Network (RNN) sono una particolare forma di rete neurale artificiale adatta per sequenze o serie temporali in cui i dati mostrano una dipendenza l'uno dall'altro. A differenza delle reti feedforward tradizionali, in cui si ha il passaggio di informazioni tra layer, in una RNN si ha il passaggio delle informazioni all'interno dello stesso layer tramite un anello di retroazione: l'uscita di una cella viene utilizzata come ingresso alla cella stessa, con la cella che ha così due ingressi, il passato, che le dà una memoria a breve termine, e il presente.

Andando a dispiegare la retroazione di una RNN, si ottiene la sequenza in Figura 1.6 [3].

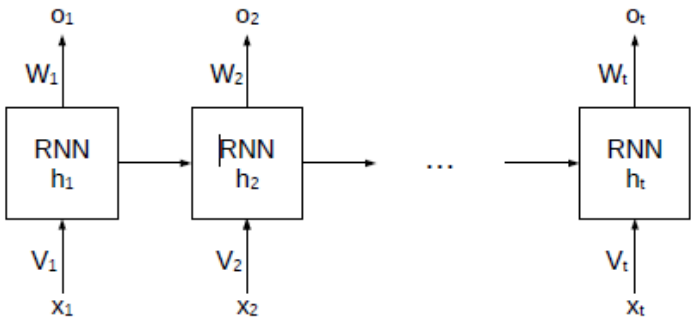

Figura 1.6 *Struttura di una RNN*

Le RNN introducono il concetto di "memoria": sono in grado di memorizzare gli stati o le informazioni degli input precedenti per generare l'output successivo della sequenza, riescono quindi ad analizzare sequenze nel tempo e prevedere andamenti futuri.

In questa architettura ogni nodo produce uno stato corrente nascosto h_t e una uscita o_t a partire dall'input corrente x_t e lo stato precedente h_{t-1} nel modo seguente

$$h_t = f(W_h h_{t-1} + V_h x_t + b_n) \quad \textbf{(1-7)}$$

$$o_t = f(W_o h_t + b_o) \quad \textbf{(1-8)}$$

dove W e V sono i pesi per gli strati nascosti nelle connessioni ricorrenti, b denota il bias per gli stati nascosti e di uscita, e f è una funzione di attivazione.

1.3.1 *Long Short-Time Memory Recurrent Neural Network*

Le Long Short-Term Memory (LSTM) sono un tipo speciale di RNN, in grado di apprendere dipendenze temporali a lungo termine, hanno infatti la capacità di imparare a "dimenticare" parte del loro stato nascosto durante il processo di inferenza.

Tutte le reti neurali ricorrenti hanno la forma di una catena di moduli ripetuti della rete neurale. Nelle RNN standard, questo modulo ripetitivo ha una struttura molto semplice, con un singolo strato $tanh$.

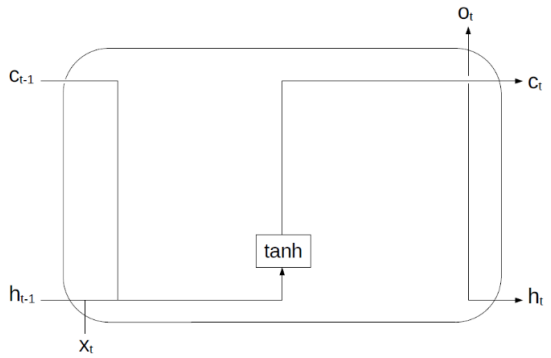

Figura 1.7 *Unità RNN*

Nelle LSMT l'unità che viene ripetuta è una cella di memoria costituita da quattro strati [4].

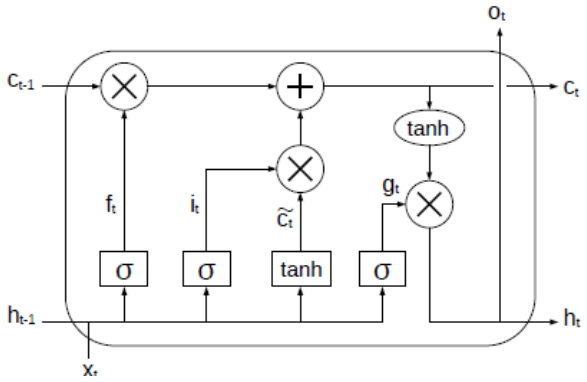

Figura 1.8 *Unità LSTM*

La rete ha tre input:

- x_t: input nell'istante attuale
- h_{t-1}: uscita nell'istante precedente
- c_{t-1}: stato nell'istante precedente, è la memoria dell'istante precedente

L'elemento principale delle LSTM è lo stato della cella, ovvero la linea orizzontale lungo la parte superiore del diagramma: lo stato c_t è semplicemente una somma tra il precedente stato c_{t-1} e l'elaborazione dell'input attuale "modulato" tramite delle maschere in modo che solo parte di esso venga ricordato.

Poiché la funzione d'attivazione per l'aggiornamento dello stato è di fatto l'identità, la derivata sarà 1 e quindi il gradiente nella backpropagation non svanirà né esploderà, ma rimarrà costante attraverso tutti gli istanti temporali della rete unfolded.

LSTM ha la capacità di rimuovere o aggiungere informazioni allo stato della cella tramite delle strutture dette gate, i quali applicano alla concatenazione dell'input x_t e dell'output h_{t-1} un'attivazione sigmoide, ottenendo un vettore in uscita i cui elementi appartengono al range (0; 1). Usare una sigmoide invece che una semplice binary-step risulta più efficace sia

perché è differenziabile, sia perché indica la quantità del dato da ricordare invece di offrire un'uscita dicotomica che indichi solamente se ricordare o non ricordare.

Indicando con:

- $W_{x_f}, W_{x_i}, W_{x_0}, W_{x_g}$ le matrici dei pesi per il corrispondente vettore di input connesso
- $W_{h_f}, W_{h_i}, W_{h_0}, W_{h_g}$ le matrici dei pesi dello stato a breve termine del passo temporale precedente
- b_f, b_i, b_0, b_g termini di bias

I gate sono:

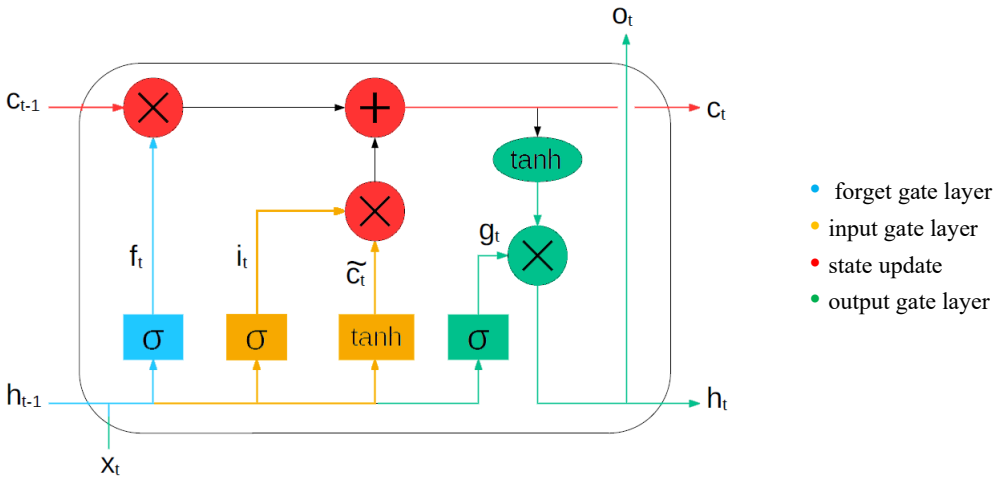

Figura 1.9 *Gate in LSTM*

- **forget gate layer,** (in azzurro in *Figura 1.9*): viene usato per mascherare lo stato c_{t-1} in modo che solo parte dello stato venga considerato allo step t. Decide quali informazioni della memoria a lungo termine rimuovere considerando h_{t-1} e x_t e ponendo in uscita un numero tra 0 (non mantenere nessuna informazione) e 1 (mantenere tutta l'informazione) per ogni C_{t-1}

$$f_t = \sigma\left(W_x^T f \cdot x_t + W_h^T f \cdot h_{t-1} + b_f\right) \quad \textbf{(1-9)}$$

- **input gate layer** (in arancio in Figura 1.9): decide quali nuove informazioni mantenere nello stato della cella; è costituito da due strati:
  - strato sigmoide decide quali valori aggiornare

$$i_t = \sigma\left(W_x^T i \cdot x_t + W_h^T i \cdot h_{t-1} + b_i\right) \quad \textbf{(1-10)}$$

- strato tanh crea un vettore di nuovi candidati, che possono essere aggiunti allo stato

$$\widetilde{c_t} = \tanh\left(W_x^T c \cdot x_t + W_h^T c \cdot h_{t-1} + b_c\right) \quad \textbf{(1-11)}$$

- **Aggiornamento dello stato** c_{t-1} al nuovo stato c_t (in rosso in *Figura 1.9*): si moltiplica il vecchio stato per f_t e si aggiunge $i_t \otimes \widetilde{c_t}$

$$c_t = f_t \otimes c_{t-1} + i_t \otimes \widetilde{c_t} \quad \textbf{(1-12)}$$

- **Output gate layer** (in verde in Figura 1.9): definisce cosa mandare in uscita, è una versione filtrata dello stato della cella:
  - strato tanh forza i valori dello stato tra -1 e 1
  - strato sigmoide decide quale parte dello stato viene mandato in uscita

$$o_t, h_t = g_t \otimes \tanh(c_t) \quad \textbf{(1-13)}$$

Il vantaggio principale dell'LSTM è la possibilità di riconoscere pattern a lungo e breve termine in una sequenza, tuttavia, avendo una struttura più complessa, è computazionalmente più costoso e richiede maggiori tempi di training.

1.4 Apprendimento

Una rete neurale può essere addestrata per approssimare qualsiasi funzione, perciò i possibili task che essa può svolgere sono molti. Alcune categorie d'interesse sono:

- **Classificazione:** identificazione di categorie o classi di appartenenza dei dati, basandosi su un insieme di osservazioni la cui categoria risulta già nota.

  Può essere:
  - Binaria: la classe da predire è una sola, il risultato del classificatore indica appartenenza oppure non appartenenza alla classe.
  - Multi-class: le classi sono più di una e ogni osservazione ha una sola classe
  - Multi-label: le classi sono più di una e ogni osservazione può avere più classi
  - Crisp: si assegna una sola classe fra quelle possibili (comporta discretizzazione degli output)
  - Probabilistica: si assegna all'oggetto da classificare una probabilità di appartenenza a ciascuna delle classi possibili; concettualmente è più valida in quanto permette di fornire una stima dell'incertezza del classificatore stesso oltre che della sua correttezza.

- **Denoising:** riproduzione di dati puliti a partire da dati corrotti.
- **Estrazione di feature:** riduzione della dimensionalità e/o realizzazione di una rappresentazione sub-simbolica omogenea dei dati a partire da osservazioni eterogenee.

Tutti i problemi di apprendimento possono essere quindi formulati come un mapping fra degli input e degli output: un modello di machine learning deve apprendere qual è il migliore output da produrre per ciascun dato passato in input. Il processo di addestramento può avvenire in tre modalità differenti:

- **Apprendimento supervisionato**: la rete è addestrata con un insieme di dati etichettati, ovvero si forniscono al modello sia dei dati di input che l'output ad essi associato. In questo apprendimento la rete deve dunque imparare a inferire la relazione che lega gli ingressi con i valori attesi, cercando di minimizzare l'errore di previsione. È il più comunemente utilizzato, soprattutto per risolvere problemi di classificazione.
- **Apprendimento non supervisionat**o: i dati impiegati per l'apprendimento non hanno un valore atteso. Lo scopo è inferire una funzione che riesca a descrivere la struttura interna o estrarre le caratteristiche interessanti di un insieme di dati in ingresso. Questo tipo di apprendimento permette di estrarre feature e/o ignorare dati ridondanti e questo lo rende adatto anche per l'addestramento di reti per la compressione delle informazioni.
- **Apprendimento per Rinforzo**: il sistema prende decisioni sequenziali interagendo con l'ambiente, e in base alle azioni che compie riceve feedback in forma di ricompense o penalizzazioni. L'obiettivo del sistema è sviluppare una strategia che gli permetta di massimizzare la ricompensa nel lungo termine.

La minimizzazione dell'errore avviene tramite l'applicazione di un algoritmo noto come gradient-descent. L'obiettivo finale è quello di addestrare la rete al fine di dotarla di una sufficiente capacità di generalizzazione che le permetta, basandosi sugli esempi con cui è stata addestrata, di prevedere correttamente il valore atteso di ingressi laddove non è noto.

1.5 Stima di una Rete Neurale

Stimare una rete neurale è un compito complesso, trattandosi di una funzione gerarchica $f(x, \theta)$ del vettore di input x e della collezione di pesi $\theta = (\alpha, \beta)$

$$\{\alpha_{0,m}, \alpha_m \quad m = 1, \dots, M\} \quad M \times (P + 1) \text{ Pesi ingresso}$$

$$\{\beta_{0,k}, \beta_k \quad k = 1, \dots, K\} \quad K \times (M + 1) \quad \text{ Pesi uscita}$$

Definiamo la funzione errore:

$$R(\theta) = \sum_{i=1}^{N} R_i \tag{1-14}$$

Con errore quadratico medio:

$$R_i = \sum_{i=1}^{k} \left(y_{ik} - f_k(x_i)\right)^2 \tag{1-15}$$

Lo scopo della rete è quello di minimizzare l'errore di previsione dell'output rispetto al valore atteso dei dati, è quindi necessario calcolare tale errore e procedere a ritroso per calibrare i pesi in base a quanto essi hanno inciso sull'output errato.

Il principio alla base della minimizzazione dell'errore è la discesa del gradiente, o gradient-descent, un algoritmo iterativo di ottimizzazione in grado di individuare il valore minimo di una funzione a più variabili che si basa sul concetto che la direzione di massima discesa verso un minimo locale o globale di una funzione di costo R_i corrisponde a quella determinata dall'opposto del suo gradiente [5].

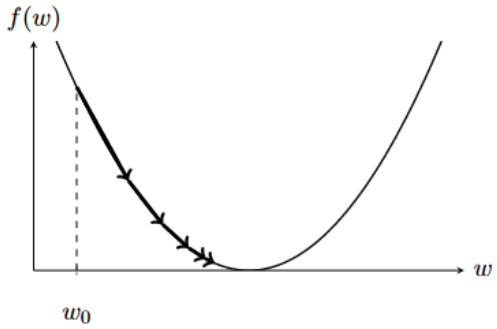

Figura 1.10 – Discesa del gradiente di una funzione

Per minimizzare la funzione, si parte quindi da pesi casuali e si procede poi iterativamente (ogni iterazione corrisponde ad un'epoca) calcolando il valore in uscita della funzione, e calibrando il valore del parametro al fine di minimizzare la funzione.

L'algoritmo, che riassumiamo di seguito, termina quando viene verificata una stopping-rule, per esempio si trova il minimo o il valore della funzione non decresce per N iterazioni consecutive:

1. ***Scelta dei valori iniziali dei pesi***

$$\alpha_m = \alpha_m^{(r)} \tag{1-16}$$

$$\beta_k = \beta_k^{(r)} \tag{1-17}$$

2. ***Forward step*:** passo in avanti lungo tutta la rete per il calcolo delle uscite della rete neurale $f_k(x_i)$
3. ***Calcolo degli errori δ_{ki}*** a partire dalla derivata della funzione di errore:

$$\frac{\delta R_i}{\delta \beta_{k_m}} = \delta_{ki} z_{mi} \;\; con\; \delta_{ki} = -2\big(y_{ik} - f_k(x_i)\big) G_k'\big(\beta_k^T z_i\big) \tag{1-18}$$

4. ***Back propagation step***: propagazione dell'errore all'indietro, ovvero calcolo dell'errore dell'hidden layer:

$$\frac{\delta R_i}{\delta \alpha_{m_l}} = \delta_{mi} x_{il} \;\; con\; \delta_{mi} = \sum_{k=1}^{K} 2\big(y_{ik} - f_k(x_i)\big) G_k'\big(\beta_k^T z_i\big) \beta_{k_m} \sigma'(\alpha_m^T x_i)$$

$$\rightarrow \delta_{mi} = \sigma'(\alpha_m^T x_i) \sum_{k=1}^{N} \delta_{ki} \beta_{km} \quad \begin{matrix} m = 1, \dots M \\ i = 1, \dots, N \end{matrix} \;\; noti \;\; \left[\delta_{ki}, \alpha_m^{(r)}\right] \tag{1-19}$$

5. **Stima delle derivate**

$$\frac{\delta R_i}{\delta \beta_{k_m}} = \delta_{ki} z_{mi}$$

$$\frac{\delta R_i}{\delta \alpha_{m_l}} = \delta_{mi} x_{il} \tag{1-20}$$

6. **Aggiornamento dei parametri**

$$\begin{aligned} \beta_{km}^{(r+1)} &= \beta_{km}^{(r)} - \gamma_r \sum_{i=1}^{N} \frac{\delta R_i}{\delta \beta_{km}^{(r)}} \\ \alpha_{ml}^{(r+1)} &= \alpha_{ml}^{(r)} - \gamma_r \sum_{i=1}^{N} \frac{\delta R_i}{\delta \alpha_{ml}^{(r)}} \end{aligned} \tag{1-21}$$

γ_r è il learning rate, un parametro che determina la velocità e l'accuratezza dell'aggiornamento: se troppo grande l'algoritmo è soggetto ad oscillazioni ed è incapace di raggiungere un punto di minimo, se troppo piccolo si avranno alti tempi di convergenza.

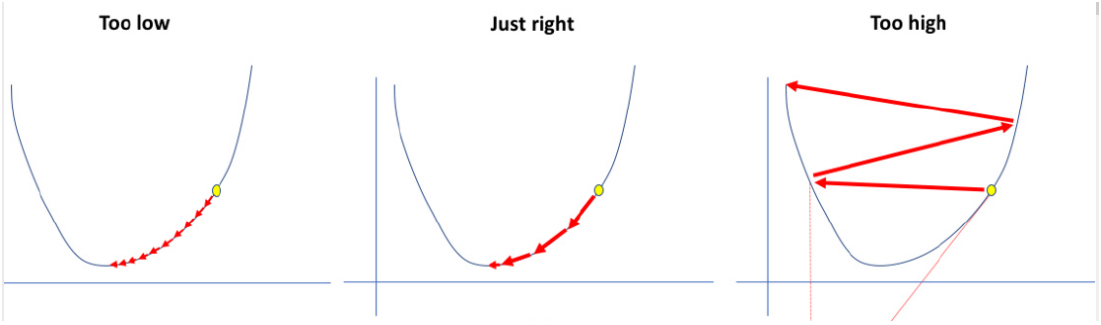

Figura 1.11 *– Convergenza dell'algoritmo per diversi Learning Rate*

1.6 Funzione di Attivazione

La funzione di attivazione viene applicata all'output di un neurone e determina se il neurone viene attivato o meno. Viene utilizzata per aggiungere non linearità alla rete e consentire l'apprendimento di modelli complessi, è necessario, inoltre, che sia continua e differenziabile, in modo da poter applicare il procedimento di back propagation.

Ogni activation-function ha delle caratteristiche peculiari che la rendono più adatta a certi domini:

- **Sigmoide:** È la versione continua di una binary-step (gradino unitario), una funzione che assume valore 0 per s < 0 ed 1 per s ≥ 0 e che può essere usata per classificazioni binarie crisp.

$$y = \frac{1}{1+e^{-s}} \tag{1-22}$$

A piccole variazioni dell'input *s* corrispondono simili variazioni in uscita solo quando l'input si trova approssimativamente intorno al range [−3, 3], dal momento che la funzione sigmoide schiaccia asintoticamente il resto degli input verso valori in output pari 0 o ad 1, come si può osservare in figura:

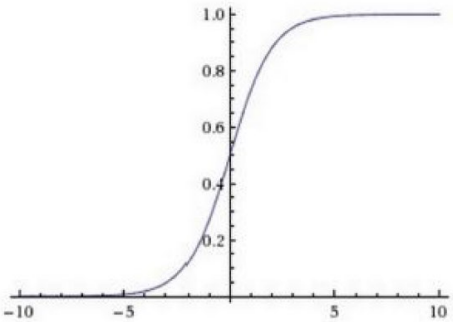

Figura 1.12 – Funzione Sigmoide

Ciò implica che se l'input s si trova fuori da $[-3, 3]$ la funzione satura, causando un gradiente prossimo allo 0 e portando quindi ad una possibile stasi dell'apprendimento. Inoltre, non essendo centrata sull'origine, gli output di un layer con sigmoide sono sempre positivi e quindi tutti i pesi di un neurone a valle, durante la backpropagation, tendono a crescere o decrescere tutti insieme.

Comporta una serie di svantaggi:

- dissolvenza del gradiente a causa della saturazione dei neuroni che presentano valori di output agli estremi del codominio della funzione di attivazione, in questo caso (0; 1). É facile infatti notare che:

$$\sigma(z) = 1 \qquad z \to \infty$$
$$\sigma(z) = 0 \qquad z \to -\infty \tag{1-23}$$

  Tale saturazione diventa problematica durante la fase di back propagation, in quanto il gradiente locale assume valori prossimi allo zero, che per la regola della catena vanno a moltiplicare tutti i gradienti calcolati in precedenza e quelli successivi, conducendo così all'annullamento del gradiente globale

$$\sigma'(z) = \sigma(z)(1 - \sigma(z))$$
$$\sigma'(z) = 0 \quad z \to \infty$$
$$\sigma'(z) = 0 \quad z \to -\infty \tag{1-24}$$

  In pratica, si ha un flusso utile del gradiente solo per valori di input che rimangono all'interno di una zona di sicurezza, cioè nei dintorni dello zero.

- gli output della funzione sigmoide non sono centrati intorno allo zero**,** e di conseguenza gli strati processati successivamente riceveranno anch'essi valori con una distribuzione non centrata sullo zero. Ciò comporta che gli input in ingresso ai neuroni saranno sempre positivi, e pertanto il gradiente dei pesi associati diventerà sempre positivo o sempre negativo, determinando una dinamica a zig zag degli aggiornamenti dei pesi che rallenta in maniera significativa il processo di convergenza.
- l'operazione exp(·) al denominatore è molto costosa dal punto dal punto di vista computazionale

- **Tangente iperbolica:** è una versione scalata della sigmoide, con codominio (-1; 1):

$$y = 2 \cdot sigmoid(2s) - 1 \tag{1-25}$$

$$\tanh(s) = \frac{e^s - e^{-s}}{e^s + e^{-s}} \quad \textbf{(1-26)}$$

È preferibile alla sigmoide perché centrata sull'origine, e la sua derivata è maggiore, ciò garantisce una convergenza verso il minimo della funzione di perdita più rapida. Rimane il problema della saturazione dei neuroni, anzi viene addirittura accentuato, dal momento che la zona di sicurezza risulta ancora più ristretta.

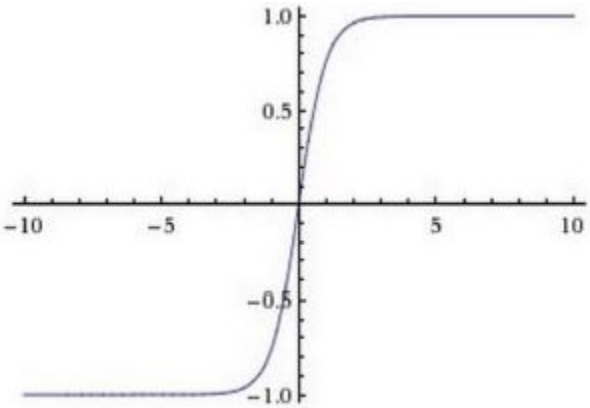

Figura 1.13 – Funzione Tangente Iperbolica

- **Softmax:** è una particolare sigmoide definita dall'equazione:

$$y_i = \frac{e^{s_i}}{\sum e^{s_i}} \quad \textbf{(1-27)}$$

con y_i e s_i rappresentanti l'output e la somma pesata dell'i-esimo neurone del layer. Vincola i neuroni di un layer normalizzandone il vettore di output in modo che $0 \leq y_i \leq 1 \ \forall i$ e che $\sum y_i = 1$.

Il vettore di output può essere visto come una distribuzione discreta di probabilità, tale funzione viene quindi utilizzata per classificazioni multiclass, con ingresso appartenente ad una delle possibili classi.

- **ReLu (**Rectified Linear Unit): è definita come

$$ReLU(s) = \max(0, s) \quad \textbf{(1-28)}$$

se il gradiente calcolato è positivo, viene lasciato passare, perché la derivata locale per il quale viene moltiplicato è pari a 1, e non va a saturare i neuroni, risolve quindi il problema della dissolvenza del gradiente con conseguente miglioramento delle prestazioni. Se invece il gradiente accumulato ha segno negativo, viene azzerato (la derivata locale è nulla lungo tutto il semiasse negativo) con la conseguenza che i pesi non vengono aggiornati e il neurone "muore". Valori elevati del learning rate

amplificano questo problema, dal momento che cambiamenti più consistenti dei pesi si traducono in una maggiore probabilità che questi affondino nella "zona morta".

$$ReLU(s) = \begin{cases} 1 & se\ s > 0 \\ 0 & se\ s < 0 \end{cases} \tag{1-29}$$

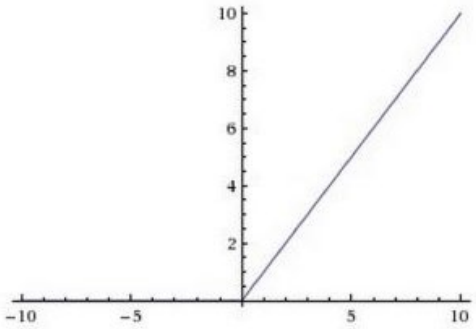

Figura 1.14 *– Funzione ReLU*

- **Leaky-ReLu:** è un tentativo di risolvere il problema della disattivazione dei neuroni comportato dalla ReLU classica, e consiste nell'introdurre una piccola pendenza negativa (α) nella regione dove la ReLU è nulla.

$$LeakyReLU(s) = \max(\alpha s, s) \tag{1-30}$$

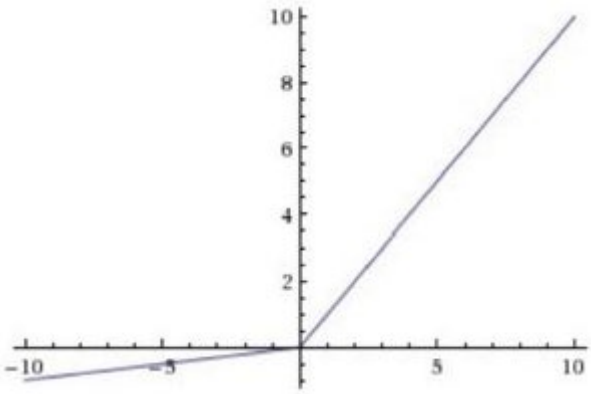

Figura 1.15 *– Funzione Leaky ReLU*

1.7 Classificazione

La classificazione ha come obiettivo quello di predire in modo accurato la classe a cui il dato appartiene. Per stimare l'accuratezza del classificatore nelle nuove osservazioni si divide l'insieme dei dati in due set, tali che la loro intersezione risulti vuota:

- **training-set:** 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, ovvero il numero di volte che il set di training viene processato dalla rete neurale
- **test-set** rappresenta i dati sconosciuti alla rete, su cui vengono fatte le predizioni e utilizzato per valutare la generalizzazione del modello

Per consentire l'ottimizzazione degli iperparametri che definiscono l'implementazione, ovvero l'architettura, la complessità e le caratteristiche del classificatore, solitamente il training-set viene suddiviso in due sottoinsiemi:

- **training-set** in senso stretto
- **validation-set**, che funge da set di test per la scelta dei migliori iperparametri del classificatore

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.

Obiettivi del tuning sono la convergenza, ovvero la capacità della rete di predire correttamente il risultato in uscita, e la generalizzazione.

Convergenza: i due fattori che incidono maggiormente sulla convergenza sono:

- learning-rate: indica la velocità di convergenza della funzione di costo; viene gradualmente ridotto, in modo che la funzione di perdita scenda rapidamente lungo la superficie degli errori. È necessario calibrarlo con cura:
    - se troppo basso la convergenza è lenta ed è probabile che l'algoritmo si fermi in uno dei punti critici;
    - se troppo alto, può portare all'overshooting, ovvero all'oscillazione intorno ad un minimo, oppure ad una divergenza.

- Gradiente: calcolato tramite chain rule, dipende da loss-function, activation-function e pesi.
    - Se troppo basso (ad esempio a causa di sigmoidi saturate) si può avere il vanishing gradient.
    - Se troppo alto, si può avere l'exploding gradient.

Generalizzazione: è la capacità di ottenere predizioni attendibili su nuovi dati che non sono stati usati per addestrare la rete neurale. Una rete progettata per generalizzare bene produce mappature input/output corrette anche se l'input è lievemente differente dagli esempi usati in fase di training; se la rete però viene addestrata con troppi esempi, essa rischia di memorizzare il training set e cadere nell'"over fitting" o "overtraining". Una rete "sovra-addestrata" perde quindi la capacità di generalizzazione. Fattori che incidono sulla generalizzazione sono:

- Dimensione del batch: definisce il numero di campioni da elaborare prima di aggiornare i parametri del modello. Alla fine del batch le previsioni vengono confrontate con le variabili di output previste e viene calcolato un errore. Un set di dati di training può essere suddiviso in uno o più batch.
- Numero di Epoche: definisce il numero di volte in cui l'algoritmo di apprendimento riprocessa l'intero set di dati di addestramento. Un' epoca è composta da uno o più batch.
- Qualità degli ingressi e loro dimensionalità: se le classi non sono equamente rappresentate nel dataset, un classificatore potrebbe essere orientato a riconoscere meglio le classi dominanti.

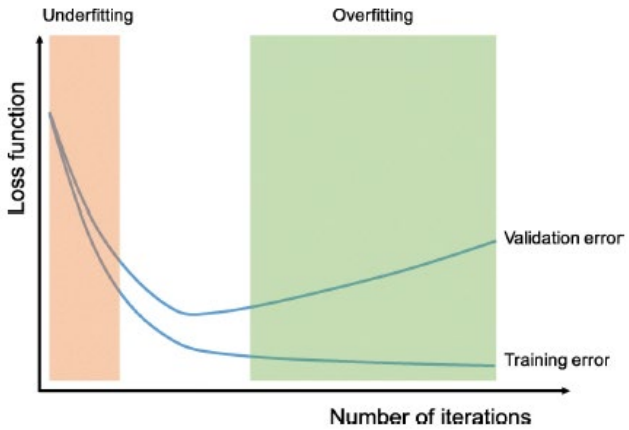

Figura 1.16 - Zone di overfitting e underfitting in funzione del costo

Non si ha buona generalizzazione in caso di:

- **Over-fitting**: rete troppo complessa e/o addestrata per troppe epoche, che deriva anche dalle caratteristiche irrilevanti dei dati (rumore), divenendo poco generale,

ovvero poco preciso per le nuove osservazioni. Può essere causato dall'utilizzo di un numero elevato di epoche

- **Under-fitting**: rete troppo generale, non riesce ad eseguire un adeguato pattern recognition che garantisca dei buoni risultati di classificazione.
  Può essere corretto tramite un aumento della complessità del classificatore, un aumento del dataset e un aumento del numero di epoche

Queste problematiche possono essere gestite in diversi modi:

- **Undersampling**: si sottocampionano le classi maggioritarie, è possibile solo se il training-set è ampio
- **Oversampling**: si sovracampionano le classi minoritarie
- **Early stop**: per evitare che la rete si specializzi a mappare esattamente i dati di training nell'output, si usa un validation-set su cui calcolare periodicamente la loss durante l'addestramento in modo da interrompere quest'ultimo quando la loss riprende a salire.
  Può accadere che la loss oscilli, raggiungendo in un secondo momento una loss inferiore. Per raggiungere la massima generalizzazione si può eseguire un intero ciclo di training e a posteriori trovare l'epoca con il più basso errore nel validation-set.
- **Data augmentation**: per ridurre l'overfit si sintetizzano nuovi dati tramite interpolazione, trasformazioni, aggiunta di rumore gaussiano con media 0 e varianza molto piccola.
- **Dropout**: si rimuovono casualmente e temporaneamente alcune connessioni durante la fase di training in modo che i neuroni siano meno sensibili ai pesi dei nodi a monte. Quando un input arriva ad un determinato neurone, c'è una probabilità p che il neurone propaghi un output ed una probabilità 1-p che il suo output sia 0. la probabilità p diviene un ulteriore iperparametro. È possibile imporre il dropout su qualsiasi layer, compreso l'input-layer.

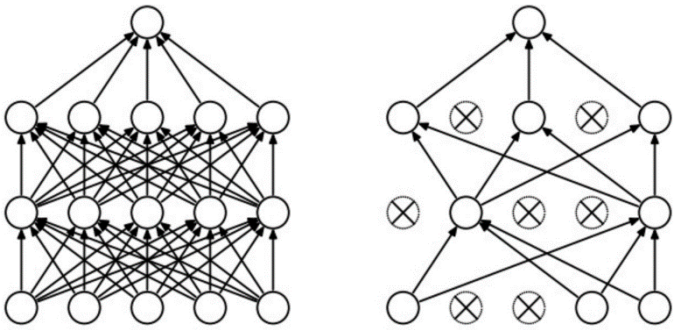

Figura 1.17 **-** Dropout

- **Riduzione della dimensionalità**: si rimuovono gli attributi inutili o ridondanti semplificando il dominio d'ingresso e facilitando il training: con un vettore di ingresso limitato, il numero di neuroni necessari ad ogni layer risulta inferiore.

1.8 Valutazione

Per valutare le prestazioni di un modello predittivo vengono utilizzate una serie di metriche, che forniscono diverse informazioni a cui dare diversa importanza a seconda dell'ambito di utilizzo del modello e dal problema specifico in esame. Prima di entrare nel dettaglio delle metriche, diamo alcune definizioni.

I risultati per ogni osservazione possono essere divisi in quattro categorie:

- **true positive (TP)**: il valore reale è positivo e la previsione è corretta
- **true negative (TN)**: il valore reale è negativo e la previsione è corretta.
- **false positive (FP)**: il valore reale è negativo, ma è stato previsto come positivo, è conosciuto come errore di prima specie.
- **false negative (FN)**: il valore reale è positivo, ma è stato previsto come negativo, è conosciuto come errore di seconda specie

In un task di classificazione binaria in cui si ha una classe positiva e una classe negativa il calcolo di TP, TN, FP e FN è immediato. In un task di classificazione multiclasse il calcolo avviene considerando singolarmente ciascuna classe.

I conteggi di queste quattro categorie vengono solitamente rappresentati attraverso una matrice di confusione, utilizzata per valutare le prestazioni di una classificazione: nelle righe si hanno le classi reali mentre nelle colonne le classi predette dal classificatore; nella cella corrispondente ad una particolare riga e colonna si ha il numero di occorrenze che rientrano in quella specifica casistica. Dato che le classi riportate nelle righe e nelle colonne della matrice hanno lo stesso ordine, nella diagonale principale della matrice si hanno le classificazioni corrette, ovvero le classificazioni in cui la classe reale corrisponde alla classe predetta dal classificatore. Le classificazioni errate si trovano fuori diagonale.

		Classe predetta	
		Appartenenza	Non appartenenza
Classe attesa	Appartenenza	TP	FN
	Non appartenenza	FP	TN

Figura 1.18 - *matrice di confusione binaria*

Dalla matrice di confusione si possono calcolare i valori di TP, TN, FP e FN per l'i-esima classe:

- TP, corrisponde al valore riportato nella cella della matrice in cui classe reale e predetta coincidono. Ovvero, il valore della cella corrispondente all'i-esima colonna e all'i-esima riga;
- FP, corrisponde alla somma dei valori sulla colonna i-esima ad eccezione del valore TP;
- FN, corrisponde alla somma dei valori sulla riga i-esima ad eccezione del valore TP;
- TN, corrisponde alla somma di tutti i valori riportati nella matrice di confusione ad eccezione dei valori che si trovano sulla riga e sulla colonna i-esime.

Dalla matrice di confusione ricaviamo quattro metriche che consento la valutazione e identificazione del modello migliore:

Accuracy - L'accuratezza misura la capacità del modello di classificare correttamente i dati. È la metrica più semplice da calcolare, ma può essere fuorviante in presenza di classi sbilanciate verso i veri positivi o i veri negativi. Ad esempio un classificatore per riconoscere l'occorrenza di una malattia che si riscontra nell'1% delle persone, che fornisce sempre come predizione la non-malattia, avrebbe comunque un'accuratezza del 99%. Matematicamente, rappresenta il rapporto tra il numero di predizioni corrette e il numero totale di predizioni

$$Accuracy = \frac{TP + TN}{(TP + FN + TN + FP)} \quad \textbf{(1-31)}$$

La metrica di accuratezza considerano solo il numero di previsioni corrette (veri positivi e veri negativi) fatte dal modello. Non tiene conto dell'importanza relativa dei diversi tipi di errori, come falsi positivi e falsi negativi.

Precision - La precisione misura la capacità del modello di classificare i positivi come positivi e i negativi come negativi. È il rapporto tra il numero di predizioni positive corrette e il numero totale di predizioni positive:

$$Precisione = \frac{TP}{(FP + TP)} \quad \textbf{(1-32)}$$

Dato che i falsi positivi comportano un peggioramento della metrica, viene utilizzata come riferimento quando è necessario ridurre il numero di falsi positivi, e in genere quando le classi sono molto squilibrate. Ottimizzare la precisione comporta il rischio di mancata diagnosi in pazienti malati (un numero elevato di FN).

Recall (o sensitivity) – La sensibilità misura la capacità del modello di classificare i positivi tra tutti i positivi esistenti all'interno di un set di dati, è infatti il rapporto tra il numero di predizioni positive corrette e il numero totale di casi positivi.

$$Sensibilità = \frac{TP}{(FN + TP)} \quad \textbf{(1-33)}$$

È una metrica da considerare quando è necessario ridurre il numero di falsi negativi. L'ottimizzazione della sensibilità aiuta a ridurre al minimo la possibilità di non rilevare un cancro, al costo di predire il cancro in pazienti sani (un numero elevato di FP).
Precisione e sensibilità sono due metriche inversamente correlate: in generale, aumentare la precisione di un modello (previsioni effettivamente positive su tutte le previsioni positive fatte) ne ridurrà la sensibilità (previsioni effettivamente positive su tutte le previsioni positive del set di dati) e viceversa. Ad esempio, se il problema consiste nell'identificare tutti i casi positivi, allora la sensibilità è la metrica più importante. Se il problema consiste nell'identificare tutti i casi positivi e negativi, allora la precisione è la metrica più importante.

F1-Score – Detta anche balanced F1-score, è usata per bilanciare precisione e sensibilità, fornendone la media armonica. Rispetto alla media standard, quella armonica attribuisce un peso maggiore ai valori piccoli, in questo modo un classificatore ottiene un alto punteggio F1 score solo quando precision e recall sono molto alte. È spesso considerata più completa rispetto all'accuratezza, in quanto tiene conto della capacità del modello di identificare sia i positivi che i negativi
Viene spesso utilizzato come valore singolo che fornisce informazioni di alto livello sulla qualità dell'output del modello.

$$Score\ F1 = \frac{2 * precisione * sensibilità}{(precisione + sensibilità)} \quad \textbf{(1-34)}$$

Capitolo 2
CASO DI STUDIO

Il riconoscimento dell'attività umana tramite sensori indossabili è uno degli argomenti più interessanti nei settori della ricerca sanitaria, dell'ambient assisted living, dello sport e del fitness. Questi sensori, che possono essere integrati in smartwatch e smartphone, generano segnali che possono essere utilizzati per raccogliere informazioni quali posizione e movimento del corpo, frequenza cardiaca (HR) e affaticamento muscolare dell'individuo che svolge l'attività. Riconoscere accuratamente i modelli di attività umana pone diversi problemi. Innanzitutto, l'accuratezza del processo di classificazione può essere fortemente influenzata dalla posizione dei sensori, quindi il loro posizionamento ottimale è fondamentale per ottenere previsioni corrette.

Tipicamente viene utilizzato il segnale fotopletismografico (PPG) il quale, a differenza dei segnali ECG e sEMG, che necessitano l'applicazione di elettrodi metallici sulla pelle, richiede un contatto più semplice nei siti periferici del corpo, consentendo la stima della frequenza cardiaca in maniera non invasiva. Il principio di funzionamento del fotopletismografo è piuttosto semplice: il sensore, da applicare sulla pelle, è costituito da un LED che emette un fascio di luce ad una certa lunghezza d'onda e con un'intensità nota. La radiazione emessa, dopo aver attraversato i tessuti sottostanti, viene acquisita da un fotorilevatore che ne quantifica l'attenuazione convertendo l'intensità della luce in un segnale elettrico proporzionale ad essa. L'attenuazione è strettamente correlata alla variazione di riflettività dei vasi sanguigni causata dal battito cardiaco.

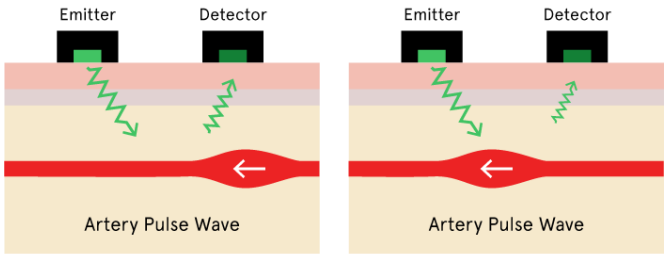

Figura 2.1 *– PPG: principio di funzionamento*

Seppur versatili e facilmente rilevabili, i segnali PPG hanno lo svantaggio di essere gravemente danneggiati dagli artefatti di movimento (MA), che derivano principalmente da un movimento relativo tra la sorgente luminosa/rilevatore PPG e la pelle dell'utente durante il movimento [6]. Al fine di ridurre gli MA, si sono rivelate molto utili numerose tecniche di elaborazione del segnale basate su dati derivati da diversi tipi di sensori, in particolare dall'accelerometro, che si è dimostrato efficace nel monitorare azioni che coinvolgono movimenti ripetitivi, come camminare, correre, andare in bicicletta [7].

In letteratura ci sono diversi studi in cui, tramite le tecniche tradizionali di Machine Learning, vengono sviluppati modelli efficaci nella classificazione delle attività umane che impiegano questa coppia di segnali. Partiremo da uno di questi studi [8] , che adotta una tecnica basata sulla decomposizione dei valori singolari (SVD) e sulla trasformata di Karhunen-Loève (KLT) per l'estrazione delle caratteristiche, e sulla classificazione bayesiana per il riconoscimento delle classi. Lavoreremo sullo stesso dataset e utilizzeremo le tecniche di Deep Learning per implementare una RNN, per poter fare un confronto tra le due strategie in termini di prestazioni e affidabilità.

In

Figura 2.2 viene rappresentato il flusso di lavoro seguito: dopo una prima fase che prevede la pre-elaborazione dei dati grezzi estratti dal dataset del caso di studio, la rete viene progettata, addestrata e testata. Il modello risultante viene ottimizzato per ridurre l'impronta di memoria e la latenza di inferenza: si tentano gli schemi di quantizzazione implementati tramite TensorFlow Lite e la libreria QKERAS, tuttavia l'unico approccio che garantisce compatibilità con l'estensione X-Cube_AI è la riduzione delle dimensioni del modello tramite opportuno tuning degli iperparametri. Il passaggio successivo prende come input il modello ottimizzato e lo traduce in una rappresentazione in linguaggio C che può essere compilata sulla piattaforma indossabile.

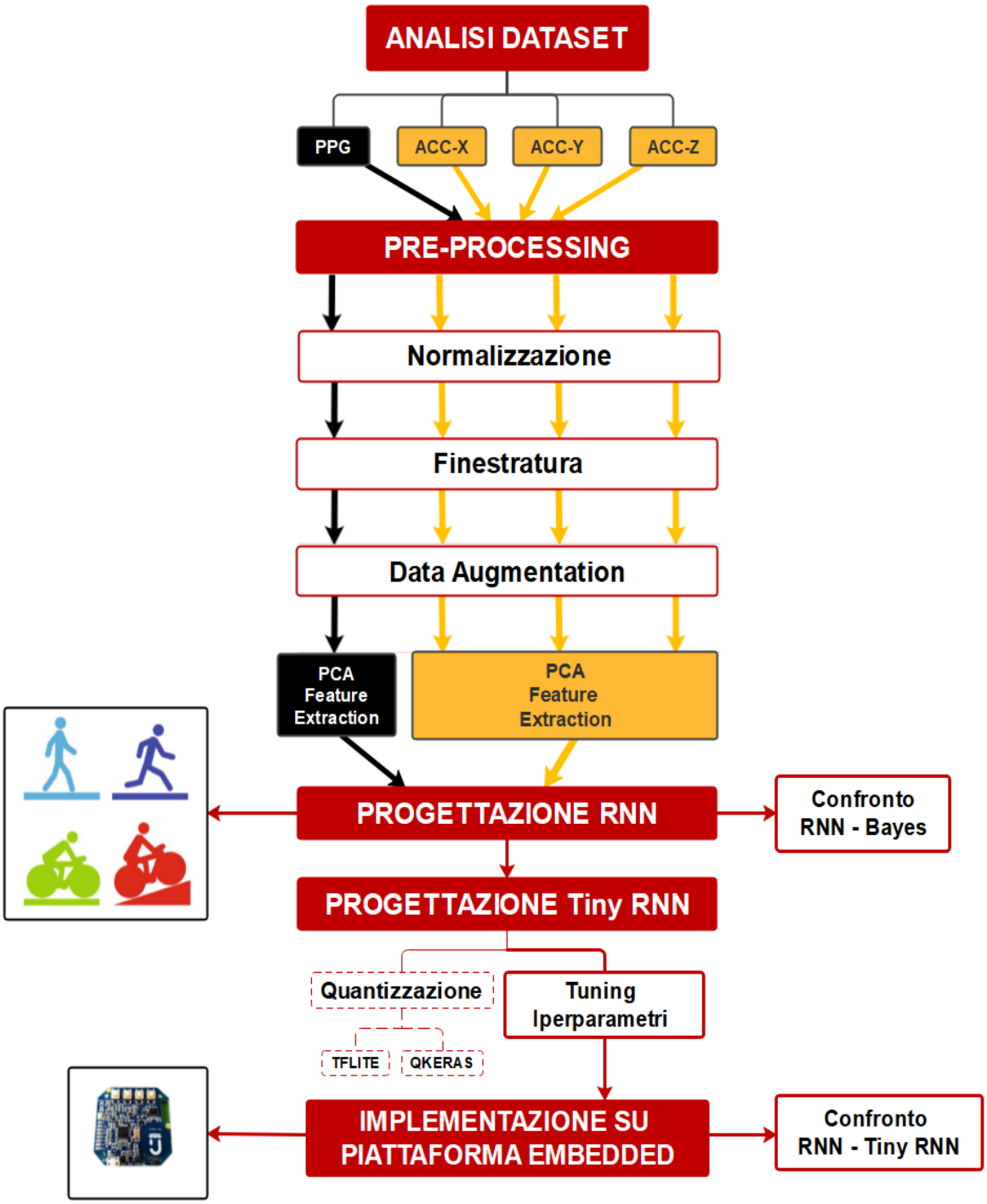

Figura 2.2 *– Flusso di Lavoro*

2.1 Dataset

Il dataset di riferimento è stato raccolto da Deleram Jarchi e Alexander J. Casson ed è reso disponibile da Physionet [9]; è costituito da 19 registrazioni relative a 4 tipologie di attività svolte da 8 partecipanti (3 maschi, 5 femmine), di età compresa tra 22 e 32 anni (media 26,5). Ciascuna registrazione contiene 13 segnali:

#	Segnale	Unità	Descrizione
1	ECG	mV	segnale ECG registrato dal torace gold standard per la frequenza cardiaca.
2	PPG	mV	PPG registrato dal polso
3	Gyro 1	deg/s	Dati del giroscopio sul primo asse.
4	Gyro 2	deg/s	Dati del giroscopio sul secondo asse.
5	Gyro 3	deg/s	Dati del giroscopio sul terzo asse.
6	Low noise accelerometer 1	m/s^2	±2 g Dati dell'accelerometro sul primo asse
7	Low noise accelerometer 2	m/s^2	±2 g Dati dell'accelerometro sul secondo asse
8	Low noise accelerometer 3	m/s^2	±2 g Dati dell'accelerometro sul terzo asse
9	Wide range accelerometer 1	m/s^2	±16 g Dati dell'accelerometro sul primo asse
10	Wide range accelerometer 2	m/s^2	±16 g Dati dell'accelerometro sul secondo asse
11	Wide range accelerometer 3	m/s^2	±16 g Dati dell'accelerometro sul terzo asse
12	Tempi di campionamento per tutti i segnali tranne ECG	s	Tempi di campionamento per la sincronizzazione
13	Picchi R	s	Tempi dei picchi ECG R identificati manualmente, riferiti al tempo 0 s all'inizio della registrazione.

Tabella 2-1 Segnali all'interno di ciascun record

Il database è costituito da più record di dati, uno per partecipante e attività di esercizio. Le misurazioni sono state effettuate utilizzando un'unità ECG a canale singolo (registratore Actiwave ed elettrodi autoadesivi in cloruro d'argento (Ag/AgCl)) posizionata sul torace, e un'unità di misurazione inerziale (unità Shimmer 3 GSR+ Shimmer Sensing) posizionata sul polso sinistro mentre i partecipanti svolgevano diverse attività su tapis roulant e cyclette. Nello specifico è stato chiesto di eseguire uno o più tipi di esercizio tra quelli disponibili, per un massimo di 10 minuti:

- Camminare su un tapis roulant a un ritmo normale
- Correre su un tapis roulant, a un ritmo stabilito dal partecipante
- Pedalare su una cyclette con impostazione a bassa resistenza
- Pedalare su una cyclette con impostazione ad alta resistenza

L'obiettivo era introdurre una serie di artefatti di movimento rappresentativi nei segnali cardiaci raccolti, senza necessariamente seguire una routine di esercizi prestabilita. Pertanto, ogni partecipante era libero di impostare il ritmo del tapis roulant e la frequenza della pedalata sulla bicicletta in modo da sentirsi a proprio agio e anche di modificare le impostazioni o interrompere l'esercizio in qualsiasi momento.

I dati sono archiviati nel formato Physionet WaveForm DataBase (WFDB) [10] in modo tale che ogni record sia costituito da due file, uno con estensione .hea, che fornisce informazioni sull'intestazione, inclusi i nomi dei segnali e le unità, e uno con estensione .dat, che contiene la registrazione vera e propria.

Sono disponibili in totale diciannove registrazioni, con una combinazione della stessa persona che svolge attività diverse e di persone diverse che svolgono attività diverse. Un esempio dei segnali raccolti durante la camminata è mostrato nella Figura 2.3. Un esempio dei segnali raccolti durante la bicicletta a bassa resistenza è mostrato nella Figura 2.4.

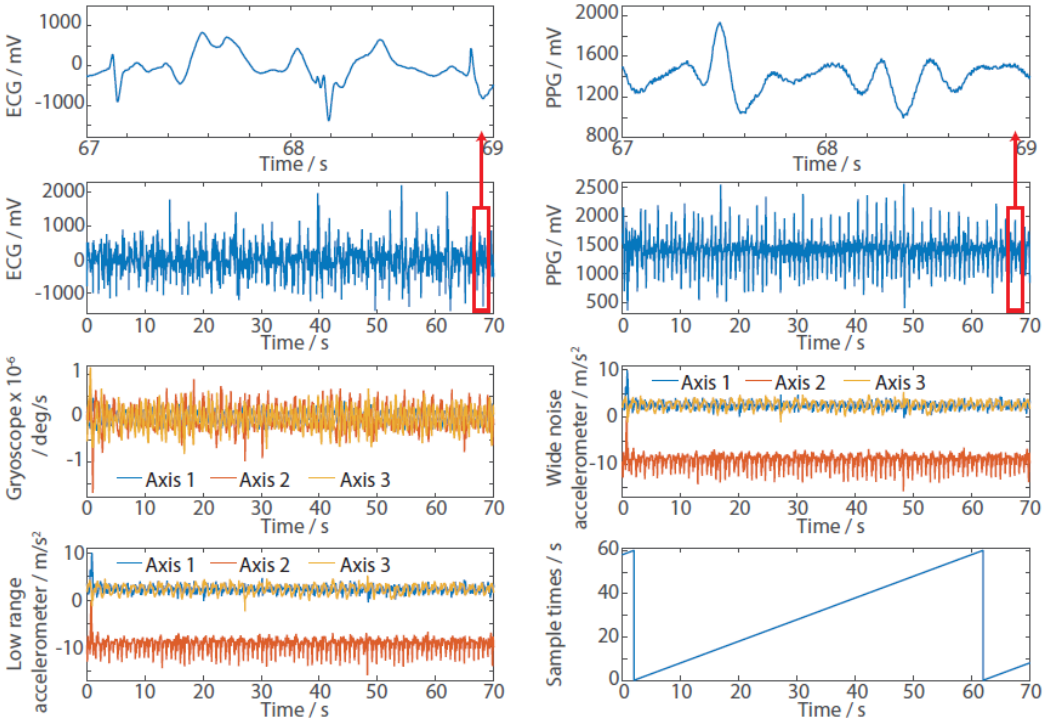

Figura 2.3 – Settanta secondi di dati dal record s1_walk

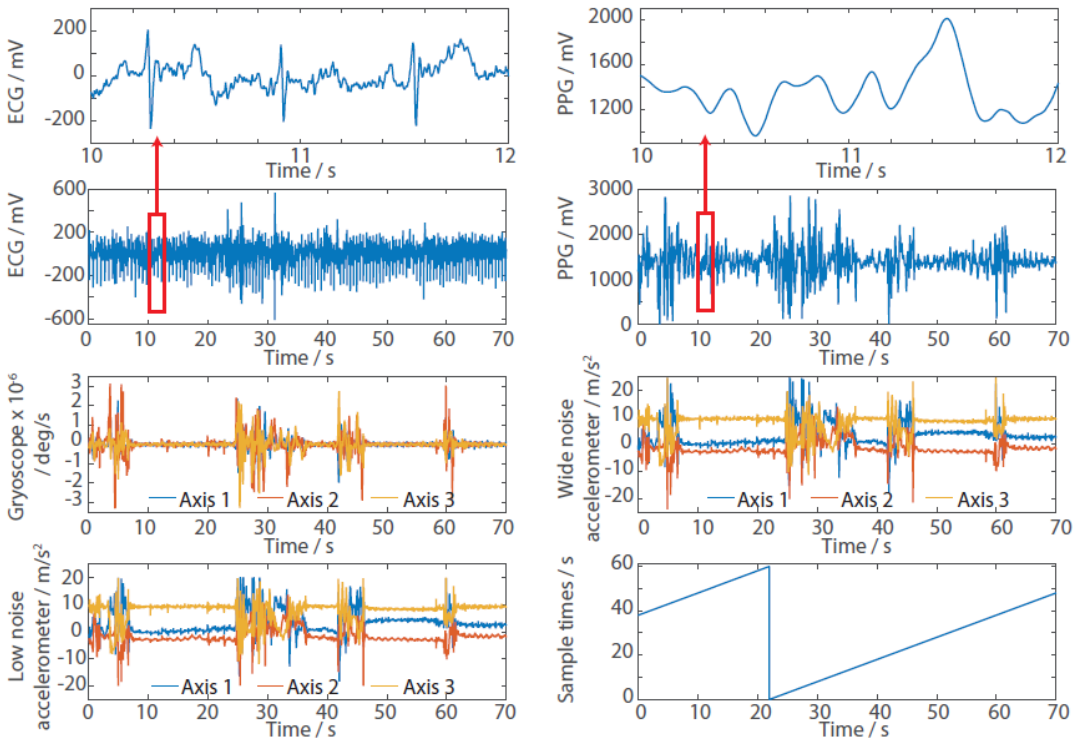

Figura 2.4 - Settanta secondi di dati dal record ***s6_low_resistance_bike***

La maggior parte dei partecipanti ha trascorso tra i 4 e i 9 minuti per ciascuna attività, la durata di ciascun record di dati è riportata nella **Tabella 2-2**.

Soggetto	Walk	Run	High Bike	Low Bike
S1	9:48	-	9:39	9:48
S2	6:39	-	5:41	6:54
S3	4:47	5:07	4:54	4:41
S4	-	4:52	-	-
S5	-	5:08	4:40	-
S6	5:36	5:02	4:40	-
S8	6:42	4:47	-	-
S9	3:40	-	-	-

Tabella 2-2 Dettagli Dataset

Tutti i segnali sono stati campionati a 256 Hz.

Per i record di camminata e corsa, il database contiene i segnali PPG e di movimento originari dopo la segmentazione nell'attività appropriata; non viene applicato alcun filtro oltre a quello integrato nell'hardware Shimmer.

Per i record di ciclismo, per ridurre le grandi quantità di rumore ad alta frequenza, le tracce PPG sono state filtrate con un filtro passa basso digitale IIR Butterworth di secondo ordine con cut-off di 15 Hz e ritardo di gruppo pari a zero [11].

2.1.1 *Suddivisione Dati*

È prassi comune, durante l'addestramento di una rete neurale, suddividere i dati di training in due set: dati effettivamente utilizzati per adattare i pesi della rete (Training set) e dati di convalida (Validation Test) per monitorare le prestazioni della rete durante le varie epoche di addestramento.

Poiché il numero di soggetti è ridotto e soggetti diversi presentano inevitabilmente differenze sostanziali nei loro dati, la distribuzione statistica dei dati potrebbe non essere abbastanza uniforme e quindi la scelta di una singola partizione dei dati di addestramento e convalida potrebbe non portare a risultati rappresentativi. Si è quindi deciso di suddividere il dataset per soggetti e non per campioni. Tale scelta, in combinazione alla limitatezza del numero di soggetti disponibili, ha costretto a non poter definire un Validation Test da utilizzare durante la fase di training.

Per la distribuzione dei soggetti nei due gruppi Training Test e Test Set, si è utilizzata la suddivisione proposta in [8]: poiché non tutti i soggetti hanno svolto tutte le attività, si sono suddivisi i soggetti in modo da avere almeno due segnali per ciascun tipo di attività sia nel training set che nel test set:

- * Training Set: {S1, S2, S4, S8}
- + Test Set: {S3, S5, S6, S9}

	Soggetti							
Attività	**S1**	**S2**	**S3**	**S4**	**S5**	**S6**	**S8**	**S9**
High Bike (H)	*	*	+					
Low Bike (L)	*	*	+		+	+		
Run (R)			+	*	+	+	*	
Walk (W)	*	*	+			+	*	+

Tabella 2-3 Suddivisione Dataset

2.2 Pre-Processing dei dati

Prima di progettare la rete, è stata considerata una fase di preelaborazione dei dati, per ridurre la dimensione del problema e combinare opportunamente i dai in ingresso, in modo da migliorare le prestazioni del learning e ottimizzare la convergenza della rete.

2.2.1 *Pulizia dei dati*

Come detto, il segnale PPG è gravemente danneggiato da artefatti da movimento (MA), principalmente dovuti al movimento tra la sorgente luminosa/rilevatore PPG e il polso del soggetto, si è quindi ritenuto non sufficiente l'utilizzo del solo segnale PPG per il riconoscimento di attività. Da analisi dei rimanenti 12 segnali disponibili per ciascun record, si è deciso di utilizzare i segnali registrati dall'accelerometro low noise sui 3 assi, in modo da avere un segnale di ingresso alla rete a 4 dimensioni: PPG, ACC_X, ACC_Y, ACC_Z.

Prima di concatenare i segnali registrati dall'accelerometro low noise e il segnale PPG, è stato necessario realizzare una pulizia preliminare dei dati, in particolare:

- Eliminazione dei punti non acquisiti (NaN) presenti negli istanti iniziali di ciascun segnale, sia PPG che ACC
- Per il segnale PPG è stata implementata una Standardizzazione Statistica, in modo da uniformare le ampiezze, compensare le rapide variazioni e quindi isolare l'andamento del segnale dagli artefatti da movimento.
  Utilizzando la formula:

  $$\boldsymbol{PPG_{std}} = \frac{\boldsymbol{PPG-\mu}}{\boldsymbol{\sigma}} \qquad \textbf{(2-1)}$$

  dove μ e σ sono rispettivamente la media originale e la deviazione standard, si fa in modo che la media e la deviazione standard risultanti siano rispettivamente 0 e 1.
  Per garantire che i dati possano essere elaborati in tempo reale durante il porting nel sistema integrato, μ e σ vengono calcolati in modo indipendente per ogni finestra dei dati in arrivo.
- Per i segnali dell'accelerometro, essendo più regolari del segnale PPG, si è semplicemente sottratto il valore medio secondo la formula:

  $$ACC_0 = ACC0 - \mu \qquad \textbf{(2-2)}$$

  dove μ è la media originale. In questo modo si elimina l'offset fisso approssimativamente costante (e ininfluente ai fini dell'analisi) dovuto alla proiezione dell'accelerazione di gravità attraverso i tre assi spaziali.

Poiché l'offset può cambiare bruscamente durante lo stesso esercizio, a causa del cambiamento di posizione inconsapevole del soggetto, si sottrae il valore medio nelle singole finestre di dati, individualmente. Test preliminari hanno dimostrato che la normalizzazione dei valori di accelerazione in base alla loro deviazione standard ha un effetto negativo sull'accuratezza finale.

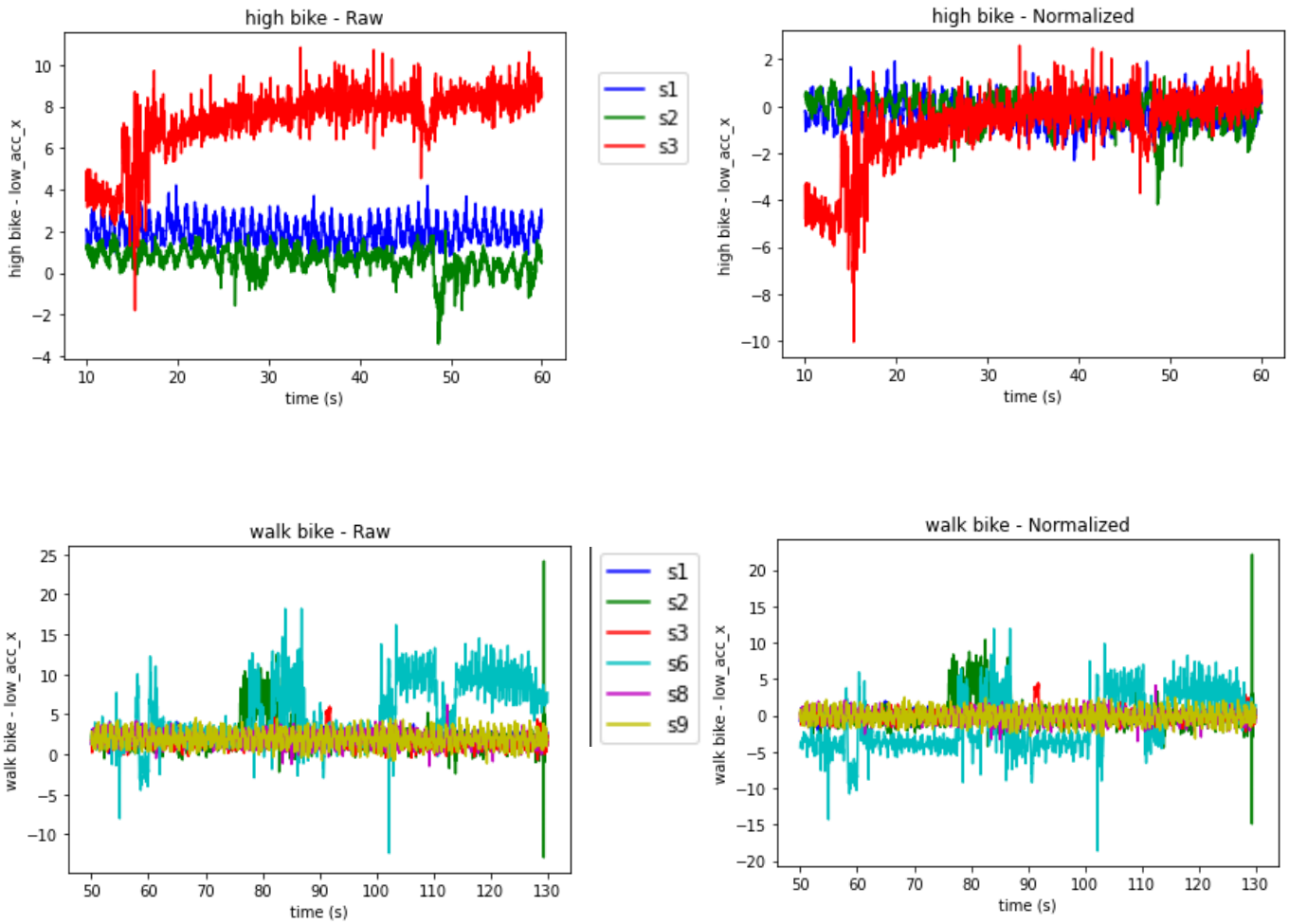

Figura 2.5 Standardizzazione Dati

In *Figura 2.5* vengono mostrati i segnali di accelerazione (originali a sinistra e standardizzati a destra) relativi a tutti i soggetti che hanno svolto uno specifico esercizio.

2.2.2 *Oversampling*

Per garantire un buon addestramento della rete è necessario avere un dataset ampio e vario, tuttavia nel nostro caso la quantità di dati disponibili è limitata e addirittura il numero di input appartenenti alle diverse classi non è equamente rappresentato, con il rischio che la rete possa

risultare sbilanciata verso una classe specifica. Si ricorre allora alla tecnica dell'oversampling [12], ovvero si incrementa il numero dei dati relativi alle classi con meno occorrenze, in modo che i dati utilizzati per l'addestramento siano distribuiti in modo più uniforme tra le diverse classi. Il sovracampionamento è stato applicato sia al sottoinsieme di soggetti utilizzati per l'addestramento, che ai soggetti del test.

La *Tabella 2-4* mostra un esempio di distribuzione dei dati prima e dopo l'aumento dei dati, per finestre di 10s (2560 campioni) con una sovrapposizione di 2s (512 campioni).

	Train		**Test**	
Classe	**Originale**	**Sovracampionato**	**Originale**	**Sovracampionato**
Walk	492	679	136	444
Low Bike	451	679	412	444
High Bike	280	679	444	444
Run	679	679	407	444
Totale	1902	2716	1399	1776

Tabella 2-4 Sovracampionamento

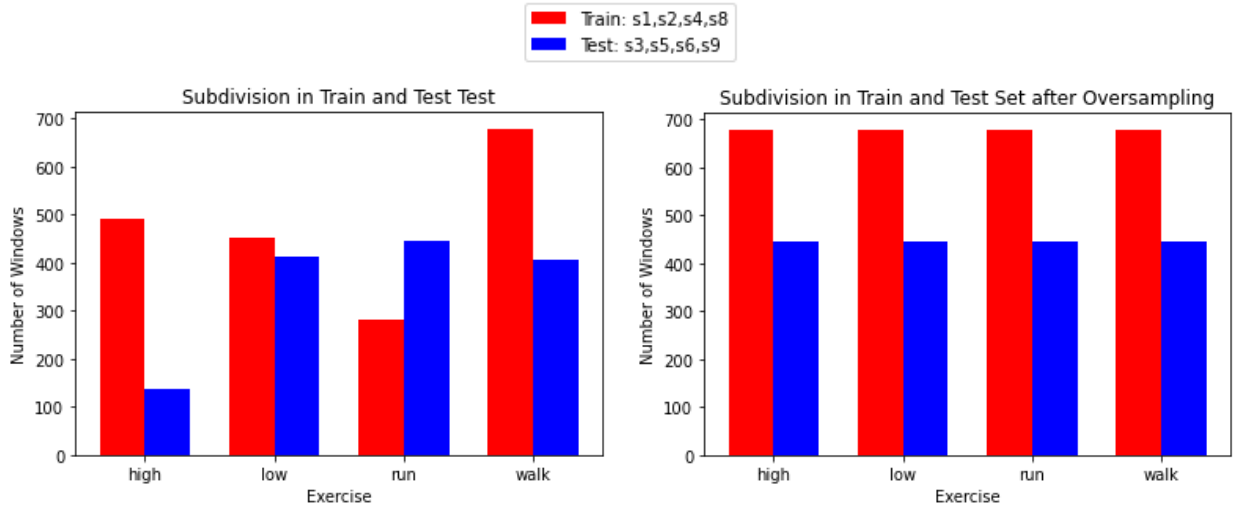

Figura 2.6 *Sovracampionamento Train e Test Set*

X-Data_Train_Oversampling

Soggetto	Classe	Campioni
S1	0	290
S1	1	285
S1	3	289
S2	0	202
S2	1	166
S2	3	194
S4	2	141
S8	2	139
S8	3	196
S1	0	187
S1	1	228
S4	2	141
S8	2	139
S4	2	119

X-Data_Test_Oversampling

Soggetto	Classe	Campioni
S3	0	136
S3	1	142
S3	2	149
S3	3	139
S5	1	135
S5	2	149
S6	1	135
S6	2	146
S6	3	163
S9	3	**105**
S3	0	136
S3	0	136
S3	0	36
S3	1	32
S3	3	37

Tabella 2-5 *Dettaglio campioni oversampling*

In *Tabella 2-5* vengono visualizzati i Training Set e Test Set finali, con il dettaglio dei soggetti e dei campioni ripetuti necessari per garantire una più uniforme distribuzione dei dati.

2.2.3 *Finestratura dei dati*

Per poter essere processati da una RNN, i dati in ingresso devono essere suddivisi in finestre di lunghezza fissa, eventualmente con sovrapposizione tra finestre adiacenti. Indicando con $\boldsymbol{w}$ il numero di campioni in una finestra e $\boldsymbol{o}$ il numero di campioni sovrapposti tra finestre adiacenti, l'n-esima finestra di dati corrisponde a campioni nel seguente intervallo:

$$n[(w-o),\ n(w-o)+w-1] \quad n \geq 0 \tag{2-3}$$

Ovviamente, la dimensione della finestra temporale influenza le prestazioni dei modelli di classificazione in modi diversi. Innanzitutto deve essere sufficientemente grande da catturare "l'impronta digitale" dell'attività umana per poter essere adeguatamente riconosciuta ma non deve essere troppo grande per includere attività consecutive. Per quanto riguarda l'HAR, in letteratura sono state utilizzate diverse lunghezze di finestra: a partire da 1 s fino a 30 s. Naturalmente, la dimensione della finestra dipende strettamente anche dalle attività da

riconoscere e, inoltre, influisce anche su altri aspetti come la dimensione del modello e il calcolo inferenziale e i costi energetici.

Da una serie di test preliminari, riassunti in *Tabella 2-6*, si è verificato che la migliore accuratezza è ottenibile con una finestra di 2560 campioni, corrispondente a una lunghezza di 10 secondi, e una sovrapposizione dell'80%, corrispondente ad una lunghezza di 8 secondi.

Finestra (campioni)	2048 (8s)			2304 (9s)			2560 (10s)			2816 (11s)		
Overlap (s)	7	6	4	8	7	5	9	**8**	6	10	9	7
Campioni Train	5456	2728	1368	5444	2728	1368	5432	**2716**	1360	5420	2716	1360
Fit Accuracy %	96,1	93	89,2	95,7	92,5	87,4	95,6	**93,3**	92,6	96,3	94,3	93,2
Test Accuracy %	62,3	70,5	61,6	66,6	63,3	64,2	70,2	**86**	68,6	71,7	68,3	70,2

Tabella 2-6 – *Risultati finestratura*

Indicando con N il numero di finestre, la matrice in input alla RNN ha dimensione: $\mathrm{N} \times w \times 4$

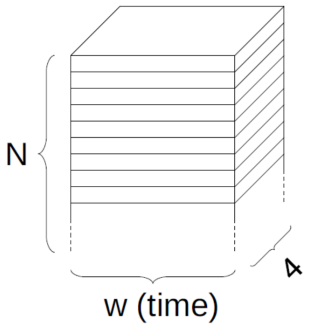

Figura 2.7 *Struttura Dati in Ingresso*

2.3 Analisi della Componente Principale (PCA)

Nelle predizioni dei modelli giocano un ruolo fondamentale le features, ovvero le caratteristiche legate ai dati, che vengono utilizzate per valutare i dati passati in input ed effettuare le predizioni. A causa della maledizione della dimensionalità, il numero di dati di training necessari per addestrare una rete neurale cresce esponenzialmente con la dimensione del vettore di input; dall'altra parte l'insieme degli dati disponibili risulta estremamente limitato, per tale ragione si è scelto di applicare una feature-extraction per ridurre la dimensionalità e scegliere gli attributi maggiormente significativi per i dati da fornire in ingresso alla rete. Nel nostro caso abbiamo utilizzato la tecnica PCA (Principal Component Analysis), comunemente utilizzata per ridurre la dimensionalità dei dati isolando le caratteristiche più rilevanti del segnale, con il risultato di ridurre il costo computazionale e

migliorare il rapporto segnale/rumore dei dati risultanti, e quindi migliorare l'accuratezza finale [13].

La decomposizione PCA si realizza tramite:

1. Decomposizione SVD della matrice di dati in ingresso $A\ (n\ \times\ m)$:

$$A = USV^T \tag{2-4}$$

   dove $S\ (n\ \times\ n)$ è una matrice diagonale contenente i valori singolari, e U, V hanno rispettivamente dimensione $m \times n,\ n \times n$ e contengono i vettori singolari.

2. Riduzione della dimensionalità selezionando i primi p valori singolari ($p\ <\ n$), purché ordinati in ordine decrescente, di S e di conseguenza troncando V alle prime p colonne (V_p). Questa decomposizione assicura la conservazione della maggior parte della varianza della matrice originaria per un dato rango p.

   Si ha quindi:

$$A^* = AV_p \quad (m \times p) \tag{2-5}$$

   che contiene le componenti principali di A e ha rango inferiore.

Data la natura dei segnali, si è deciso di applicare la decomposizione al tensore costituito dai tre segnali di accelerazione, che hanno contenuto informativo simile e coerente, e poi separatamente al singolo segnale PPG. Avremo quindi, rispettivamente:

- Tensore tridimensionale $N \times w \times 3$ per i 3 segnali ACC
- Matrice bidimensionale $N \times w$ per il segnale PPG

con N il numero di finestre di dati, w la lunghezza della finestra.

Un problema che sorge nel caso delle reti neurali è che la matrice dei dati di input, una volta che i dati originali sono stati suddivisi in finestre, è un tensore A tridimensionale, con dimensione (in questo caso) $N\ x\ w\ x\ 3$.

Per poter applicare SVD e PCA è necessario modellare A in una matrice bidimensionale [13]

Da risultati sperimentali si è visto che i risultati migliori si ottengono con:

1. Trasposizione $A \in \mathbb{R}^{N\times w\times 3} \rightarrow A' \in \mathbb{R}^{N\times 3\times w}$
2. Incolonnamento prime due dimensioni $A' \in \mathbb{R}^{N\times 3\times w} \rightarrow A'' \in \mathbb{R}^{3N\times w}$
3. Selezione delle prime p componenti principali, che si ottengono semplicemente come $A''' = A''V_p \in \mathbb{R}^{3N\times p}$

Per poter essere utilizzata come ingresso alla RNN, la matrice viene riconvertita in un tensore tridimensionale, trasponendo la seconda e la terza dimensione, e si concatena con la matrice PPG:

4. $A''' \in \mathbb{R}^{3N \times p} \rightarrow A^* \in \mathbb{R}^{N \times p \times 3}$ **(2-6)**

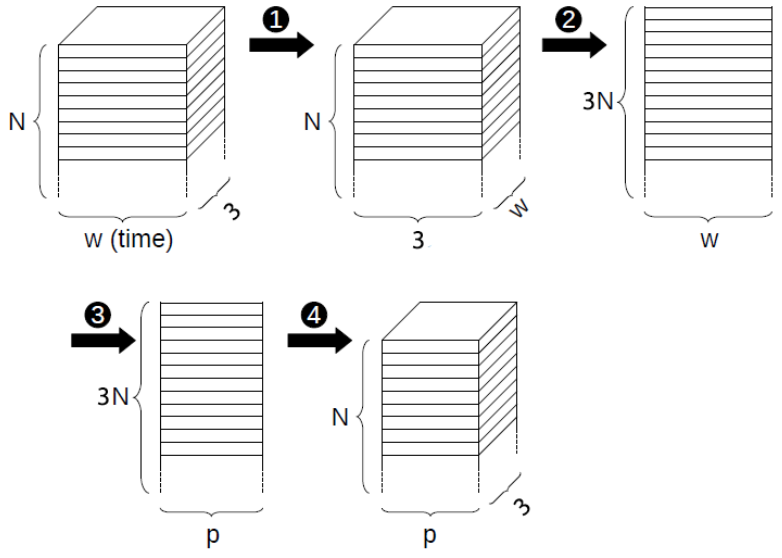

Figura 2.8 ***Fasi per realizzare la PCA*** [13]

Una prima serie di test è stata eseguita utilizzando i dati originali, successivamente è stata applicata la decomposizione PCA alla matrice di dati di input, utilizzando come iperparametro p, ovvero il numero di componenti principali selezionate dal segnale originale.

Da test preliminari, si è visto che i risultati migliori si ottengono con p=50. Si nota che l'aggiunta di ulteriori funzionalità non comporta un aumento delle prestazioni misurate. Ciò dimostra che non tutte le caratteristiche forniscono contenuti informativi originali utili al processo di classificazione.

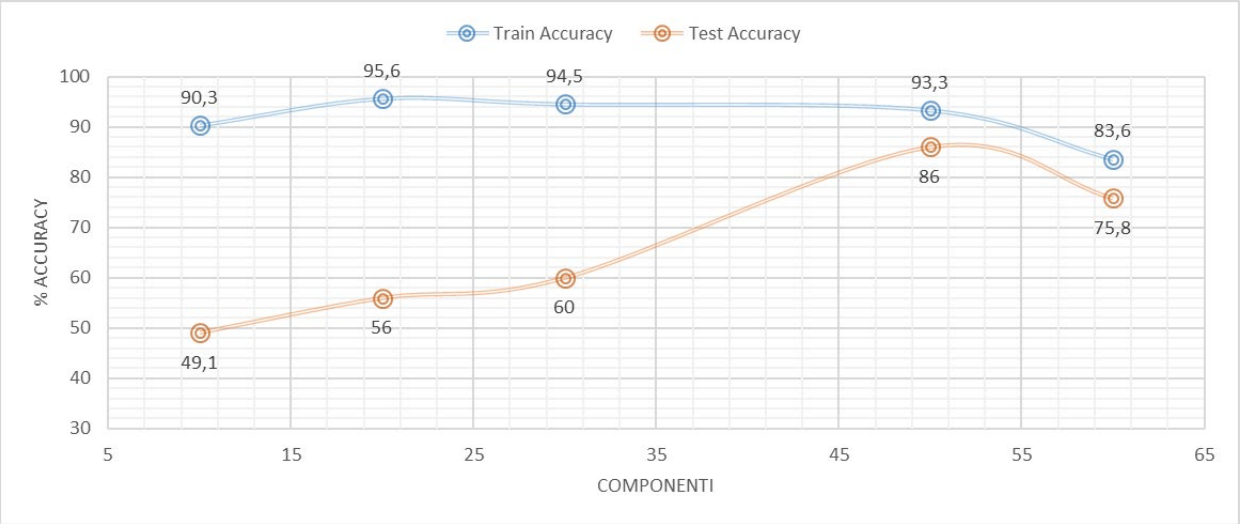

Figura 2.9 *Tuning parametro p*

Capitolo 3
IMPLEMENTAZIONE RNN

3.1 Architettura della RNN

Le reti neurali sono sistemi complessi le cui prestazioni e esigenze computazionali dipendono da diversi parametri. Questi parametri possono essere suddivisi in iperparametri e parametri apprendibili [14]. Con il termine iperparametri si intendono quei parametri che vengono utilizzati per controllare la struttura della rete e il processo di apprendimento in modo da poter essere suddivisi, a loro volta, in iperparametri strutturali e iperparametri algoritmici. Gli iperparametri strutturali, detti anche iperparametri modello, descrivono la struttura e la topologia della rete e sono rappresentati, ad esempio, dal numero di strati, dal numero di neuroni in ogni strato, dal grado di connettività, dalla funzione di trasferimento dei neuroni, ecc. Poiché modificano direttamente la struttura della rete, influenzano sia la sua efficacia, la sua complessità computazionale, sia la sua impronta di memoria. I parametri algoritmici sono invece la velocità di apprendimento, la dimensione del training set, l'algoritmo di addestramento e così via, e vengono utilizzati per controllare il processo di apprendimento. Questi parametri non fanno parte del modello e non hanno alcuna influenza sulle sue prestazioni ma influenzano la velocità e la qualità del processo di apprendimento [15].

Infine, i parametri addestrabili sono rappresentati da pesi e bias delle connessioni neuronali e vengono modificati durante la fase di apprendimento dagli algoritmi di training. Il numero di questi parametri dipende strettamente dal tipo di rete e dalla scelta degli iperparametri strutturali e influenza direttamente la complessità computazionale e l'impronta di memoria del modello.

La rete è stata implementata usando il seguente pattern:

- Init: nell'inizializzazione della rete, viene definito il modello, passando ad esso tutti gli iperparametri strutturali. Qui si definisce la struttura della rete ed il grafo computazionale, partendo dagli input e giungendo fino agli output, alla loss-function e all'ottimizzatore.
- Train: riceve in ingresso i dati di training e procede con l'addestramento della rete usando un early-stop. In particolare si avvia una sessione di Tensorflow, si eseguono

N epoche di training, ognuna con selezioni casuali di mini-batch, fino a che non si verifica una condizione di terminazione in base alla loss del set. Restituisce in uscita le metriche ottenute sul training-set.

- Test: riceve in ingresso i dati di test, avvia una sessione di Tensorflow e produce gli output corrispondenti. Restituisce le metriche ottenute sul test-set.

L'RNN implementata si basa su architetture comunemente utilizzate per dataset relativi a sequenze temporali [16]; le funzioni d'attivazione, gli ottimizzatori del gradiente, l'inizializzatore dei pesi, la funzione di perdita cross-entropy e la cella LSTM sono quelle già implementate in Tensorflow, in quanto risultano più stabili ed efficienti [17].

La rete consiste in una serie di strati LSTM e strati completamente connessi:

- `LSTM`: cella LSTM implementata da Tensorflow, con il parametro units a definire la dimensionalità dello spazio di uscita (più è grande maggiori sono le capacità di generalizzazione ma anche la dimensione complessiva della rete), e il parametro `return_sequence` settato a true per garantire che lo strato LSTM successivo abbia il corretto formato in ingresso.
- `Dropout`: livello che rimuove casualmente e temporaneamente alcune connessioni durante la fase di training in modo che i neuroni siano meno sensibili ai pesi dei nodi a monte. Il parametro p definisce la probabilità che il neurone propaghi un output, e conseguentemente la probabilità 1-p che il suo output sia 0. Consente la riduzione dell'overfitting, ovvero la condizione per cui il modello deriva anche dalle caratteristiche irrilevanti dei dati (rumore), divenendo poco generale, ovvero poco preciso per le nuove osservazioni.
- `Dense`: livello Fully Connected di dimensione 4 che esegue la classificazione in una delle 4 classi. Il neurone n-esimo produce un'uscita y_n dipendente dagli input $x_1, \dots, x_m$, e dai pesi ω_{nj} del neurone associati a ciascun input, in particolare:

$$y_n = \phi\left(\sum_{j=1}^{m} \omega_{nj} x_j + b_n\right) \quad \textbf{(3-1)}$$

dove ϕ è la funzione di attivazione e b_nè un valore di polarizzazione.

Tramite una serie di test si è definita la struttura ottimale della RNN: si è valutata la variante LSTM bidirezionale, che considera la sequenza di dati in entrambe le direzioni, in modo da avere contesto non solo del passato ma anche del futuro, ma i piccoli miglioramenti delle prestazioni non giustificano l'aumento della complessità computazionale e dell'occupazione della memoria che il LSTM bidirezionale comporta. Si è valutato l'utilizzo di uno strato di Batch Normalization, che normalizza la media e la deviazione standard dei dati a livello

globale, operando su singoli batch di dati man mano che l'addestramento procede, ma si è visto che si ottengono migliori risultati con la normalizzazione dell'intero dataset in fase di pre-processing.

Stage	Iperparametri	Rete #1	Rete #2	Rete #3	Rete #5	Rete #6
BatchNorm	Presence	**no**	no	si	no	no
LSTM1	Units	**96**	96	96	96	96
DROP1	Rate	**0,3**	0,2	0,2	0,2	0,2
LSTM2	Units	**96**	96	96	96	96
DROP2	Rate	**0,3**	0,2	0,2	0,2	0,2
LSTM3	Units	**96**	-	96	96	96
DROP3	Rate	**0,3**	-	0,2	0,2	0,2
Training	Optimizer	**Adam**	Adam	Adam	Adam	Adam
	Batch Size	**32**	32	20	16	64
Train Acc	%	**93,3**	94,2	78,8	92,5	92
Test Acc	%	**86**	68,47	58,7	68,9	67,7

Tabella 3-1 *Selezione Strati*

Il cuore della struttura è quindi rappresentato dalla cascata di tre strati LSTM, ciascuno seguito da uno strato di dropout per ridurre l'overfitting. La dimensione dell'input è determinata dal numero di segnali e dalla dimensione w della finestra scorrevole: quando si utilizzano i dati originali senza PCA, l'input è una serie temporale di dimensione $w \times 4$, quando si utilizza PCA, i dati di input sono le componenti principali $p \times 4$.

Dopo aver definito i singoli strati impiegati nell'architettura della *RNN*, si scelgono gli iperparametri della rete eseguendo una serie di training su un set di possibili valori discreti per ogni iperparametro, e ottenendo una stima della performance della rete mediante specifiche metriche di valutazione; nel nostro caso si sono ottimizzate:

- **LSTM units**: numero di unità nascoste, corrisponde alla quantità di informazioni che il livello ricorda tra time steps (hidden layer). L'hidden layer può contenere informazioni da tutte le fasi temporali precedenti, indipendentemente dalla lunghezza della sequenza. Se il numero di unità è troppo grande, il livello potrebbe adattarsi eccessivamente ai dati di addestramento e provocare overfitting

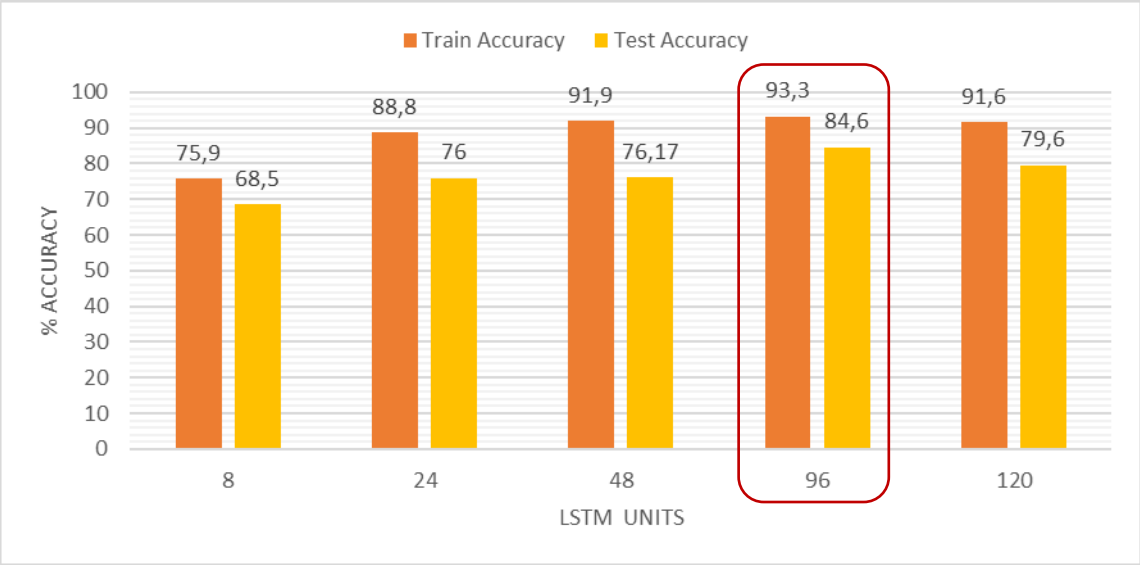

Figura 3.1 *Tuning LSTM Units*

- **Drop rate**: frazione delle unità (neuroni) che vengono ignorate durante l'addestramento

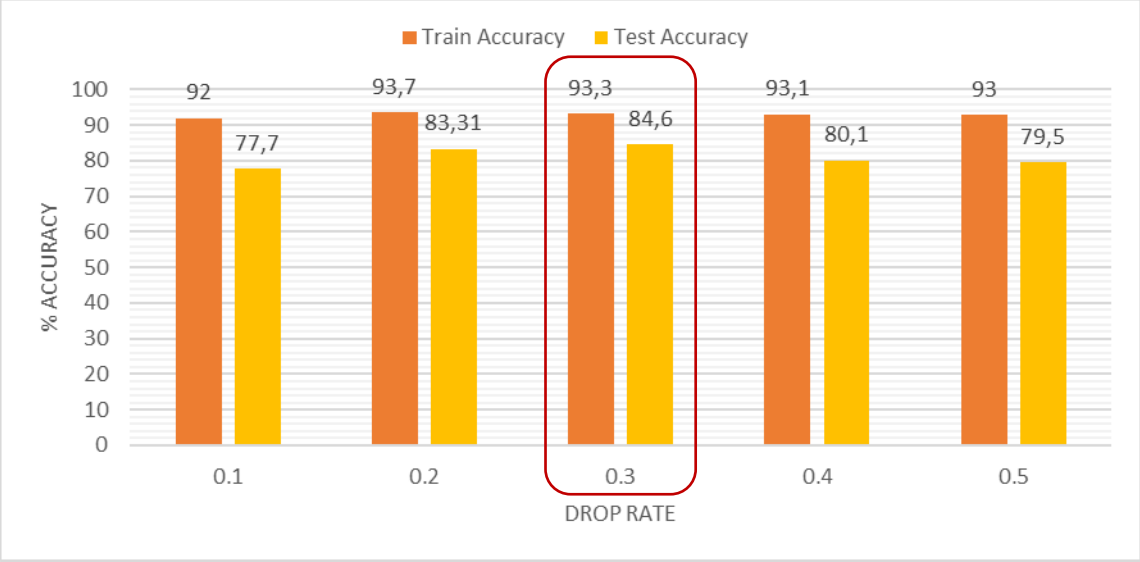

Figura 3.2 *Tuning Drop Rate*

Definita la struttura, si ha l'implementazione del modello tramite il metodo `compile()`

```
  model.compile(loss = keras.losses.SparseCategoricalCrossentropy(from_logits =
True),    optimizer = 'adam', metrics = ['accuracy'])
```

Il metodo compile riceve in ingresso i parametri:

- **loss fuction**: errore di previsione, rappresenta l'errore che deve essere minimizzato tramite il training. La rappresentazione specifica dell'errore dipende dalla particolare funzione associata alla rete; nel caso in esame si utilizza la Sparse Categorical

Crossentropy, una misura che in statistica valuta quanto una distribuzione probabilistica P approssima bene una distribuzione nota Y: nel caso delle NN, P è l'insieme degli output di un layer, Y è il vettore della classe attesa. Questa misura si fonda sul concetto di entropia H(Y), intesa come il costo in bit per codificare un'informazione Y: più l'informazione è improbabile maggiore è l'entropia e maggiore è il numero di bit necessari per codificare l'informazione.

L'entropia da sola non è sufficiente in quanto non tiene conto di quanto la distribuzione P si avvicina o si allontana dalla distribuzione di riferimento Y, per cui l'ulteriore fattore tenuto in conto è l'entropia relativa H(Y|P), che indica invece quanto P diverge da Y.

$$CROSS\ ENTROPY = H(Y) + H(Y|P) \tag{3-2}$$

- **ottimizzatore**: determina come vengono aggiornati i pesi durante l'addestramento, si utilizza l'ottimizzatore Adam

Successivamente si ha la fase di Training, realizzata tramite il metodo `fit()`:

```
history = model.fit(x_data_train, y_data_train, epochs = epochs,
batch_size=32, callbacks = callbacks)
```

Il modello riceve in ingresso i dati di training e procede con l'addestramento della rete, utilizzando i parametri:

- **Bach size**: definisce il numero di dati che attraversano la rete neurale prima che i pesi vengano aggiornati [18]. Alla fine di ogni batch, i risultati predetti vengono confrontati con quelli reali e viene calcolato l'errore.

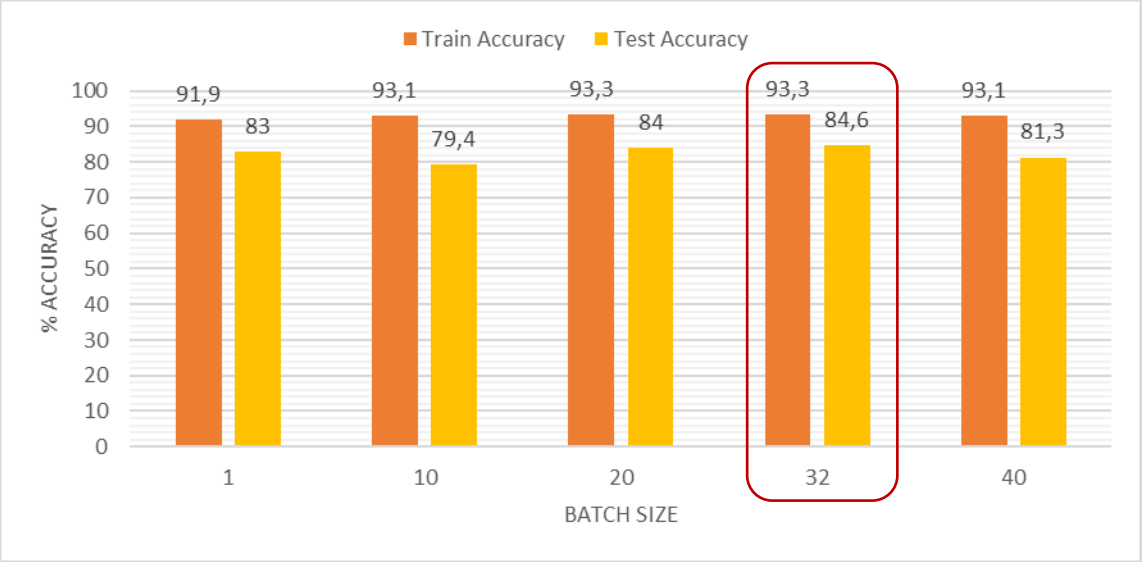

Figura 3.3 *Tuning Batch Size*

- **Epoche**: definisce quante volte l'algoritmo di apprendimento, che consente l'aggiornamento dei parametri del modello, viene eseguito sull'intero dataset, con l'obiettivo di minimizzare l'errore e aumentare l'accuratezza. Viene scelto in modo da trovare un compromesso tra minimizzazione dell'errore e tempo di calcolo [19].

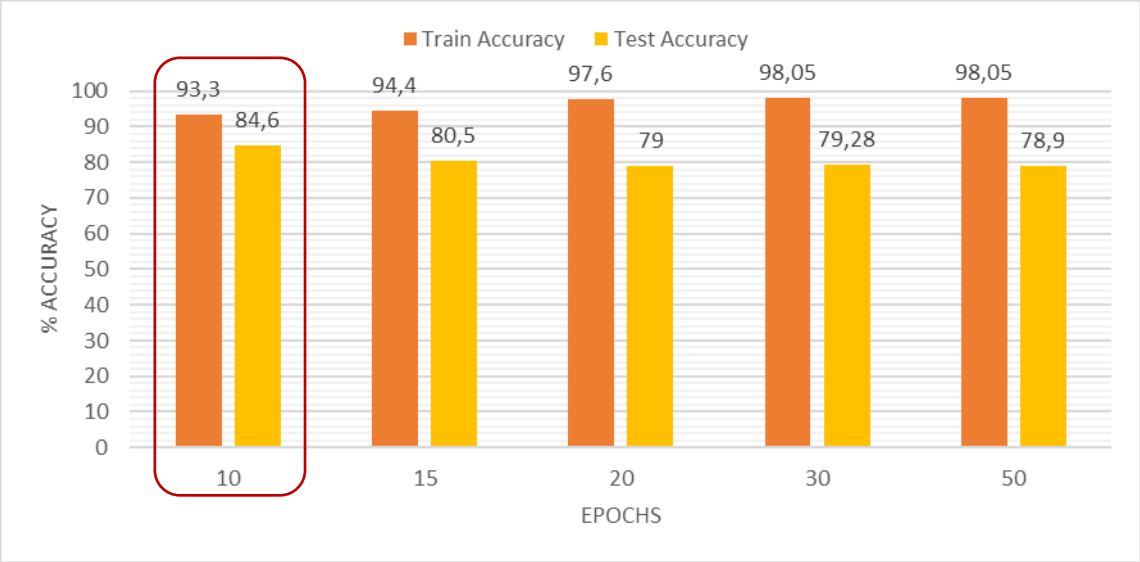

Figura 3.4 *Tuning Epochs*

La rete è stata quindi addestrata e testata con i seguenti parametri:

- Finestre di 2560 campioni e sovrapposizione dell'80%.
- Suddivisione dei dati in Training Set e Test Set, senza il Validation Set:
  - Training Set: {S1, S2, S4, S8}
  - Test Set: {S3, S5, S6, S9}
- Data Augmentation per uniformazione dei dati disponibili per le diverse classi di esercizi
- Struttura della rete come definito nel paragrafo precedente
- Un totale di 10 epoche di training

La fase di valutazione della rete neurale si realizza tramite:

- il metodo `evaluate()`: valuta le prestazioni complessive del modello su un set di dati di test restituendo le metriche di valutazione "loss" e "accuracy".
- il metodo `predict()`: genera le previsioni di output relative al set di campioni forniti in input.

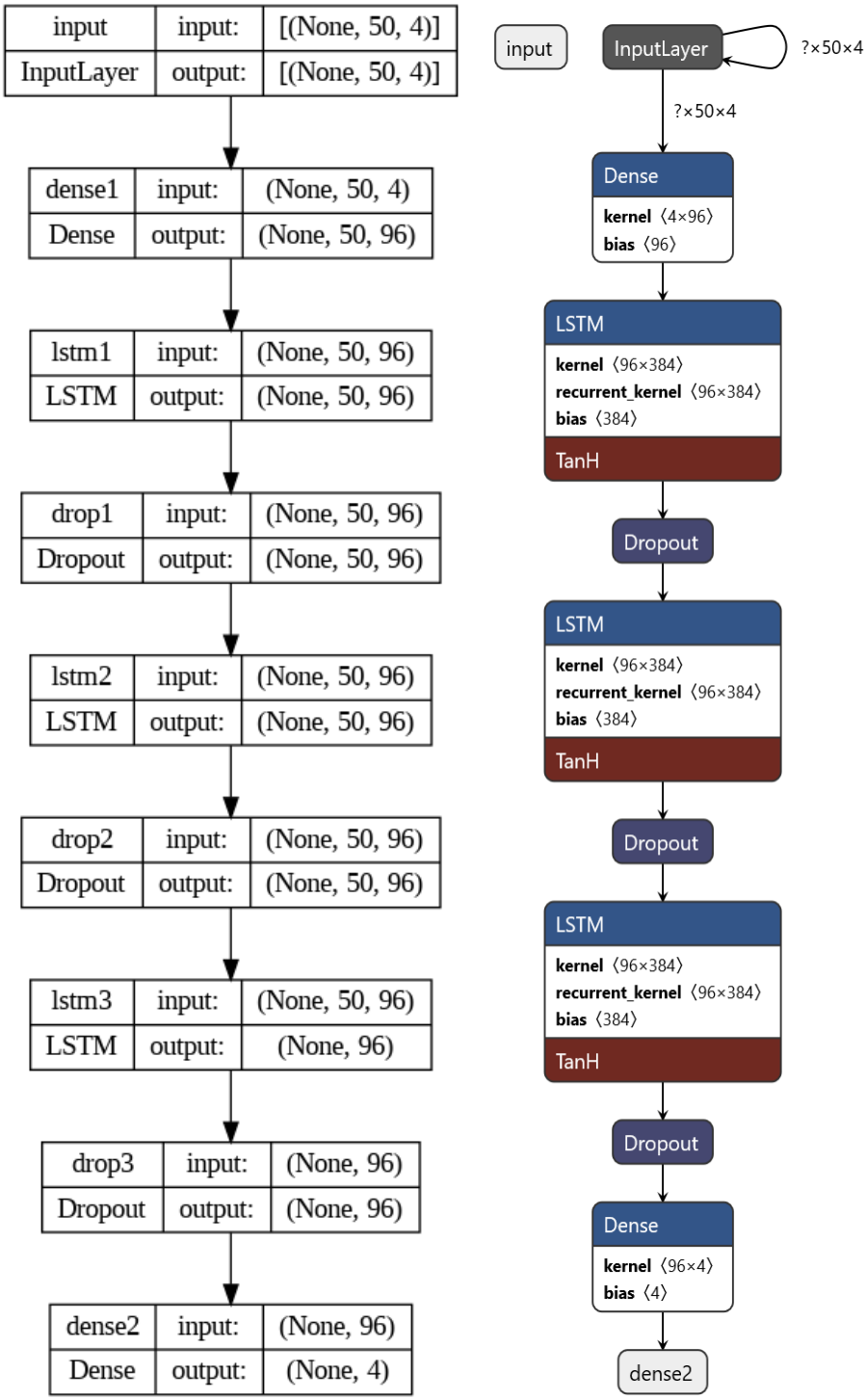

Figura 3.5 *Architettura RNN*

3.2 Ambiente di Sviluppo

Una volta eseguita un'analisi approfondita del dominio dei dati e scelta la migliore rappresentazione di quest'ultimi, è stato implementato il modello della rete.
L'RNN è stata sviluppata in Python utilizzando Tensorflow, una libreria open-source sviluppata da Google per la creazione e lo sviluppo di reti neurali, che dispone di una ampia comunità di ricercatori che lavorano costantemente al progetto, in modo da mantenere sempre aggiornata la piattaforma allo stato dell'arte attuale nell'ambito del Machine Learning.
Colab è fondamentalmente un ambiente notebook Jupyter gratuito eseguito interamente nel cloud, non richiede configurazione e supporta le librerie di Machine Learning più popolari.

Un punto di forza di questa libreria consiste nell'offrire una vasta gamma di API a diversi livelli di profondità, per estendere le funzionalità della libreria e permettere al programmatore sia di usufruire di modelli già ben assemblati, sia di poter lavorare a basso livello per definire i propri modelli o calibrarne di esistenti. In particolare è stata utilizzata la libreria Keras, che funge da interfaccia ad alto livello per TensorFlow, garantendo modularità e semplicità di utilizzo. Keras consente la creazione del modello attraverso la configurazione e interconnessione di nodi, ciascuno dei quali accetta uno o più tensori in ingresso, producendo un tensore in output.
I tipi di nodi possono essere i seguenti:

- Constant: sono i valori costanti del modello.
- Variable: sono i pesi addestrabili della rete; mantengono il proprio valore tra le varie sessioni e sono condivisibili nel grafo computazionale.
- Placeholder: sono delle variabili metasintattiche che non assumono alcun valore fino a che non vengono inizializzate tramite dati esterni, dunque usati per passare alla rete input e valori attesi, oltre che possibili ulteriori strutture di supporto.
- Operation: sono delle operazioni eseguibili sui nodi precedentemente definiti.

La loss-function ad esempio è un nodo al penultimo step del grafo computazionale, seguita da un ottimizzatore che riceve in ingresso il tensore della loss ed esegue una backpropagation sulle variabili. Un tool fondamentale e reso disponibile da Tensorflow è la Tensorboard, un server locale che permette di osservare la struttura del modello a diversi livelli di dettaglio ed ottenere grafici temporali sugli output di qualsiasi nodo valutato durante una sessione. Ciò permette di ottenere facilmente gli andamenti della loss oppure dei pesi e dei gradienti applicati ad essi.

3.3 Risultati

Le performance della rete sono state valutate utilizzando le metriche introdotte in 1.8, nello specifico, nella fase di scelta del numero delle epoche di addestramento, si sono valutate l'accuratezza, la perdita e la matrice di confusione, quest'ultima fondamentale per ottenere un riepilogo dei risultati di previsione della rete per ciascuna delle classi da identificare.

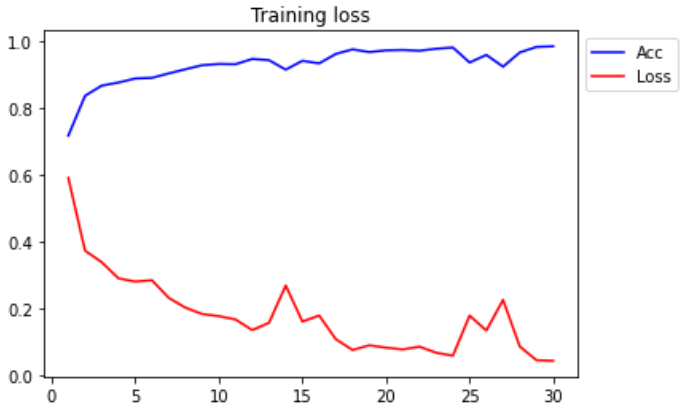

Figura 3.6 *Andamento di Loss e Accuracy rispetto alle epoche*

La Figura 3.6 mostra l'andamento dell'accuratezza e della loss rispetto alle epoche di training: si nota un miglioramento della accuracy sul training set, con un valore ottimale che si aggira sulle 20 epoche ma, come verificato sperimentalmente, aumentare il numero di epoche comporta un overfitting e un peggioramento delle prestazioni sul test set, con addirittura una errata classificazione per l'attività "Low Bike", che viene confusa con l'attività "High Bike".

Utilizzando gli iperparametri identificati nella fase di tuning, si è quindi verificato che il numero ottimale di epoche di addestramento è pari a 10, con cui si ottiene:

- Accuracy sul train set: 93, 3%
- Accuracy sul test set: 84,6%
- Matrice di confusione con tutte le classi correttamente previste

Si nota che i valori più bassi di riconoscimento si hanno per le classi meno rappresentate dal set di test e per le quali l'oversampling è stato più consistente. In particolare l'attività di Low Bike viene confusa con quella di High Bike, ciò è imputabile al fatto che le due attività, entrambe relative al pedalare su una bicicletta, comportano gli stessi movimenti, differendo solo per la quantità di sforzo esercitato. In questi casi, quindi, la classificazione fa affidamento quasi esclusivamente sul segnale PPG che, come atteso, si rivela limitato per il corretto riconoscimento delle attività.

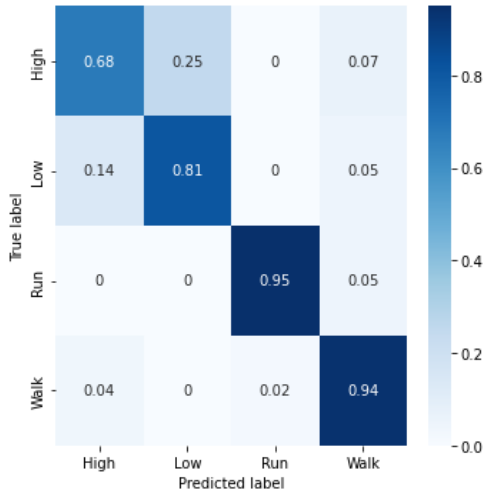

Figura 3.7 *Matrice di Confusione Ottima*

3.3.1 *Prestazioni della RNN al variare del Pre-Processing*

Per valutare l'efficacia della selezione dei segnali disponibili per ogni attività e del pre-processing realizzato, si fa un confronto tra i risultati ottenuti con e senza decomposizione PCA, utilizzando come ingresso alla rete alternativamente il solo segnale PPG, e i soli segnali di accelerazione. In ***Figura 3.9*** vengono riportati anche i tempi di inferenza della rete: si nota che l'utilizzo della PCA comporta una consistente riduzione dei tempi (circa il 95%) rispetto al segnale completo.

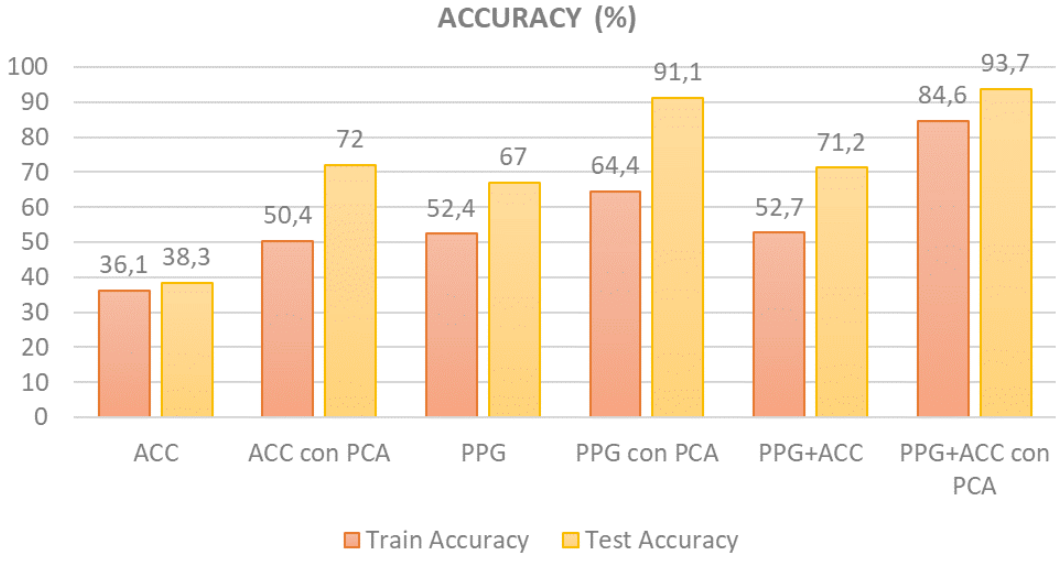

Figura 3.8 *Performance al variare del pre-processing*

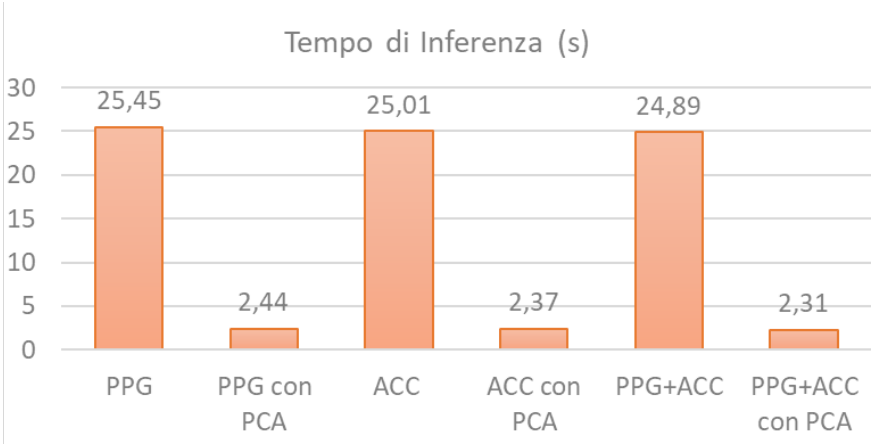

Figura 3.9 *Tempi di calcolo al variare del pre-processing*

Utilizzando esclusivamente il segnale PPG si ottengono performance mediocri: viene correttamente identificata soltanto l'attività di Walking, ovvero quella meno impegnativa e più regolare in termini di sforzo fisico. Con segnale semplificato PCA si ottiene un generale miglioramento.

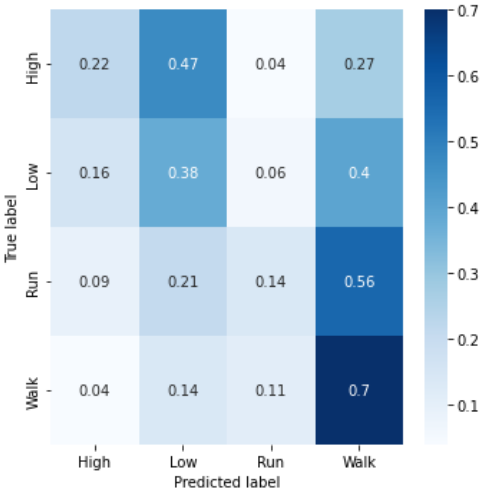

Figura 3.11 - Matrice di confusione con input alla rete il segnale PPG

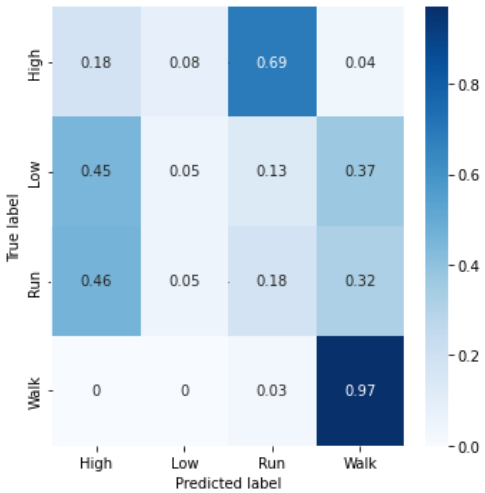

Figura 3.10 - Matrice di confusione con input alla rete il segnale PPG con decomposizione PCA

Utilizzando i tre segnali dell'accelerometro, i rate di riconoscimento migliorano per tutti gli esercizi, ad eccezione di quelli in cui il movimento previsto è simile, ovvero le attività di Low Bike e High Bike. Anche in questo caso la decomposizione PCA semplifica il segnale e restituisce risultati migliori della controparte con segnali non semplificati.

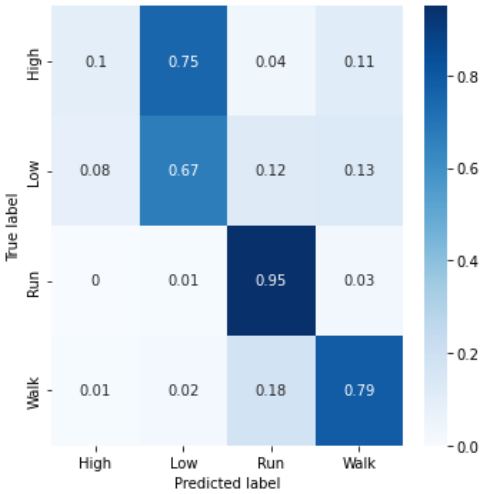

Figura 3.13 Matrice di confusione con input alla rete il segnale ACC

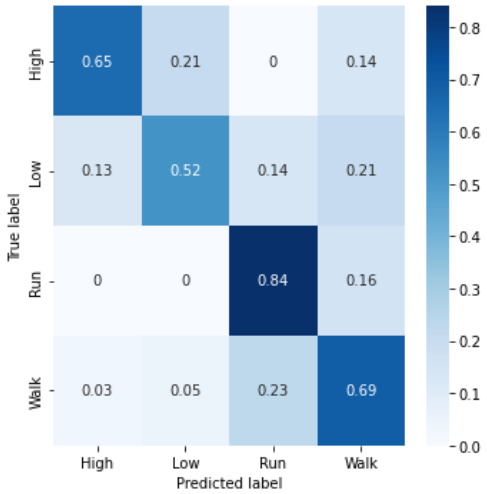

Figura 3.12 - Matrice di confusione con input alla rete il segnale ACC con decomposizione PCA

La configurazione che garantisce le migliori performance è quella costituita dalla concatenazione dei 3 segnali dell'accelerometro con il segnale PPG, e in cui ciascun segnale è stato decomposto tramite PCA.

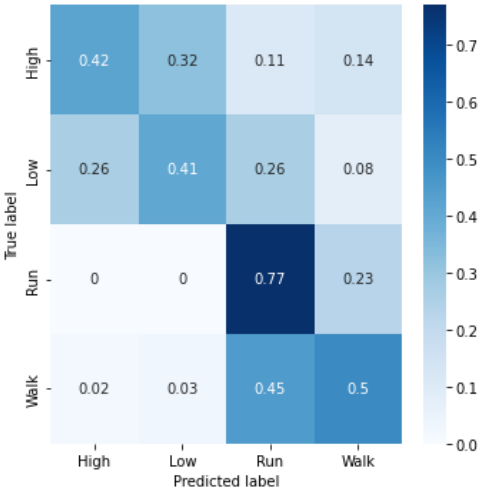

Figura 3.15 - Matrice di confusione con input alla rete il segnale PPG + ACC

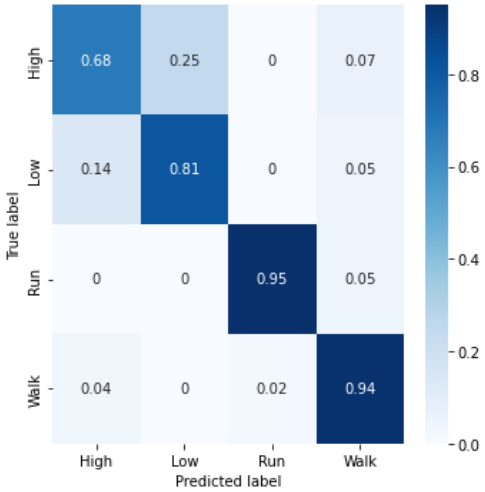

Figura 3.14 - Matrice di confusione con input alla rete il segnale PPG + ACC con decomposizione PCA

Dal confronto dei risultati si deduce che:

- Le reti che utilizzano segnali senza decomposizione PCA hanno prestazioni basse, con una accuracy di poco superiore al 50% e una classificazione delle attività incerta, in particolar modo per quelle con tipologia di movimento simile.
- Le reti che utilizzano solo il segnale PPG non hanno informazioni sufficienti per classificare correttamente le attività, in particolare gli esercizi di sforzo, come la corsa e la bicicletta ad alta intensità, vengono confusi tra loro, mentre i segnali dell'accelerometro, da soli, non consentono il riconoscimento delle attività con stessa movimentazione, ovvero la bicicletta a diversa intensità.

Si conclude che la decomposizione PCA, garantendo le migliori prestazioni in termini di accuracy e di corretta previsione delle classi, e consentendo una notevole riduzione del tempo di elaborazione, è una operazione indispensabile da applicare ai segnali in ingresso alla rete. La combinazione dei segnali ACC e PPG è anch'essa necessaria: ci sono attività diverse che coinvolgono essenzialmente gli stessi movimenti, differendo solo per la quantità di sforzo esercitato, come pedalare su una bicicletta a diversi livelli di resistenza. Queste attività sono difficilmente identificabili utilizzando esclusivamente i segnali di accelerazione; con le informazioni provenienti da un sensore PPG la precisione del riconoscimento può essere notevolmente migliorata, purché i due tipi di segnali siano opportunamente combinati. Nel nostro caso specifico, la corretta classificazione delle attività nei casi in cui l'entità dello sforzo è simile (Running/High Bike e Walking/Low Bike), è possibile sfruttando l'informazione contenuta nel segnale di accelerazione, mentre nei casi in cui ad essere simile è la tipologia dell'esercizio, ovvero il tipo di movimento svolto (Low Bike/High Bike), viene sfruttata principalmente l'informazione contenuta nel segnale PPG.

3.3.2 *Confronto con lo Studio di Riferimento*

Facciamo adesso un confronto con i risultati ottenuti nello studio di riferimento [8], in cui si ricorre alla decomposizione in valori singolari (SVD) per l'estrazione e la riduzione delle caratteristiche, e alla classificazione bayesiana per il riconoscimento delle classi.

Utilizzando i criteri di valutazione descritti in 1.8, si osserva che le prestazioni ottenute con la RNN sono in media migliori rispetto a quelle ottenute con il classificatore bayesiano: l'accuracy complessiva del classificatore bayesiano si ferma al 78%, mentre con la RNN si raggiunge l'84,6%, inoltre, analizzando nel dettaglio i tutti i punteggi ottenuti, la RNN garantisce risultati nettamente migliori per le attività di Walk e Run, mentre ha qualche incertezza solo per le attività di High e Low Bike, in cui si hanno risultati peggiori solo in termini di sensibilità, ovvero la metrica che, più di tutte, può dare risultati non precisi in caso di classi sbilanciate. Si deduce che le NN, oltre ad essere versatili ed efficienti su dataset di grandi dimensioni, sono efficaci anche su dataset ridotti. Ciò fa presupporre che, avendo a disposizione una maggiore quantità di dati, la RNN abbia notevoli margini di miglioramento, consentendo una classificazione più precisa delle attività svolte. La dimensione limitata del set di dati può compromettere la generalità dei risultati, producendo una forte distorsione, a seconda della partizione del soggetto. Un set di dati più ampio potrebbe risolvere questo tipo di problemi e fornire risultati più generali; questo può essere oggetto di futuri lavori in questo campo.

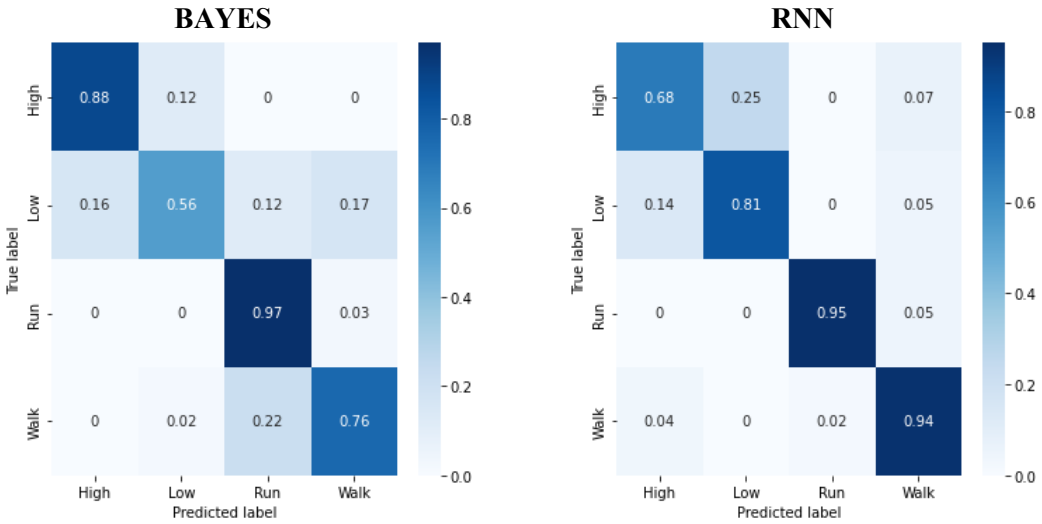

Figura 3.16 *Matrice di confusione del classificatore bayesiano, a sinistra, e della rete RNN a destra*

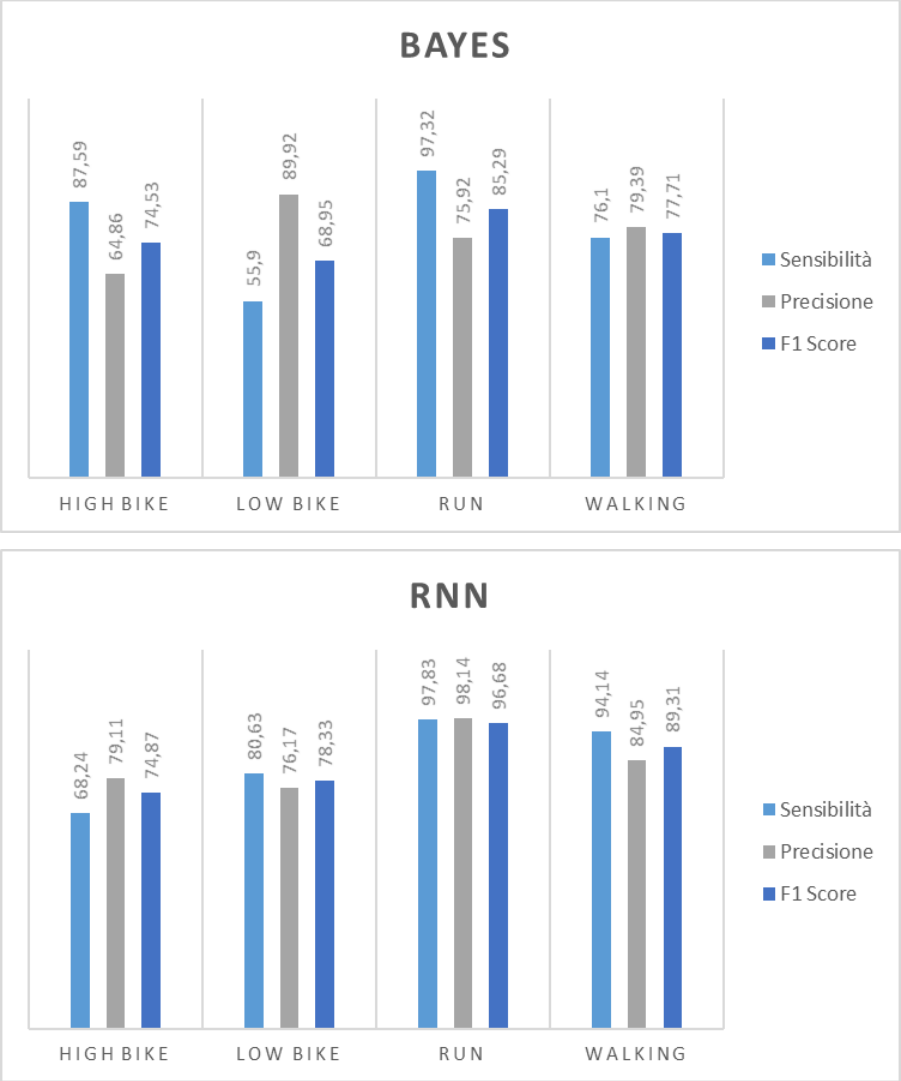

Figura 3.17 *Prestazioni modello di riferimento e della rete RNN a confronto*

Capitolo 4
PORTING SU DISPOSITIVO EMBEDDED

Gli algoritmi basati su metodi di Deep Learning permettono di raggiungere buone prestazioni, ma comportano un alto costo computazionale. La soluzione 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 si hanno una serie di problematiche [20] inerenti:

- **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.
- **Privacy**: i dati vengono trasferiti su server remoti, su cui potrebbero verificarsi violazioni della sicurezza durante la trasmissione o l'archiviazione nel cloud.

L'elaborazione locale su dispositivi edge risolve molti di questi problemi, è però necessario considerare che, nonostante i recenti progressi nell'apprendimento automatico e nella progettazione hardware, le risorse in termini di memoria, capacità computazionale e budget di potenza su queste piattaforme sono limitate e non possono supportare carichi di calcolo elevati.

Per implementare reti neurali su microcontrollori è quindi fondamentale valutare la struttura della rete ed eventualmente adottare tecniche che consentano 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 [21]. 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.

4.1 Ambiente di sviluppo

La piattaforma hardware scelta è la scheda Cloud-JAM L4 [22], dotata di un microcontrollore STM32F401RE con una CPU Arm® Cortex®-M4 a 32 bit, frequenza fino a 80 MHz, 512 KB di memoria flash, 128 KB di RAM e circa 3 mA di consumo di corrente della CPU a piena velocità.

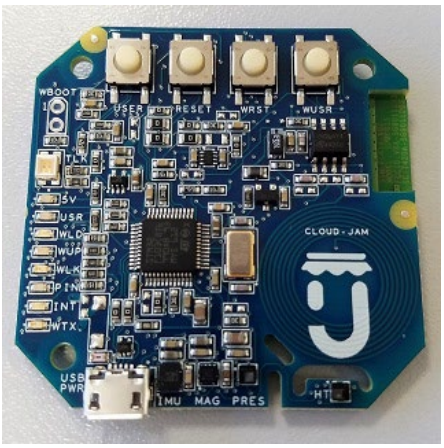

***Figura 18** – Board Cloud-JAM L4*

Cloud-JAM è l'acceleratore di prodotto del pacchetto funzionale ST P-NUCLEO-CLD1 [23], che combina:

- **NUCLEO F401RE**: Scheda di sviluppo STM32 Nucleo-64 con MCU STM32F401RE
- **NUCLEO SHIELD IKS01A2**: Scheda di espansione con sensori MEMS di movimento e ambientali
- **NUCLEO SHIELD NFC01A1**: Scheda di espansione per tag NFC dinamico basata su M24SR
- **NUCLEO SHIELD IDW01M1**: Scheda di espansione Wi-Fi basata sul modulo SWPF01S

La scelta è ricaduta su questo dispositivo poiché è una versione miniaturizzata della STM32 Nucleo ed essendo dotata di accelerometro, è sufficiente la connessione di un sensore PPG mediante interfaccia I^2C per ottenere l'applicazione completa.

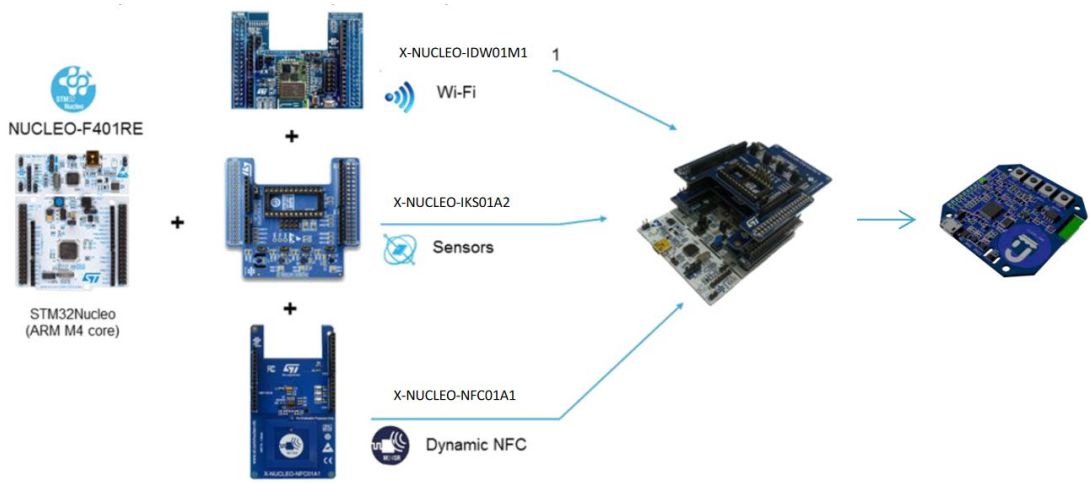

Figura 19 *– Board Cloud-JAM L4*

Per valutare le prestazioni della rete e trasferirla sul dispositivo, si è utilizzato l'STM32 Cube IDE, una piattaforma di sviluppo C/C++ basata sul framework Eclipse per la configurazione e programmazione di microcontrollori e microprocessori STM32. Per semplificare lo sviluppo di algoritmi AI sui microcontrollori STM32, STMicroelectronics ha sviluppato una soluzione denominata STM32Cube.AI, un pacchetto di espansione che:

- Genera codice STM32 da una rete neurale Python descritta nei framework AI supportati
- Supporta lo schema di quantizzazione e le ottimizzazioni per STM32, riducendo i requisiti di memoria fino a un fattore 4 e diminuendo la latenza e il consumo energetico fino a un fattore 3.
- Analizza la rete e verifica la compatibilità con le risorse del dispositivo target
- Convalida il codice C generato sul PC o sul target rispetto al modello originale

Il flusso delle operazioni necessarie per l'importazione ed esecuzione della rete neurale sulla piattaforma, come mostrato in Figura 4.20, è prima di tutto la creazione del modello di riferimento utilizzando uno dei frameworks di intelligenza artificiale supportati dalla piattaforma. Successivamente si utilizza l'estensione X-CUBE-AI dall'interfaccia STM32CubeMX per importare il modello, verificare il soddisfacimento dei requisiti di compatibilità e memoria, e generare il codice necessario per eseguire l'inferenza. 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 nel modello C ottimizzato per l'architettura di destinazione.

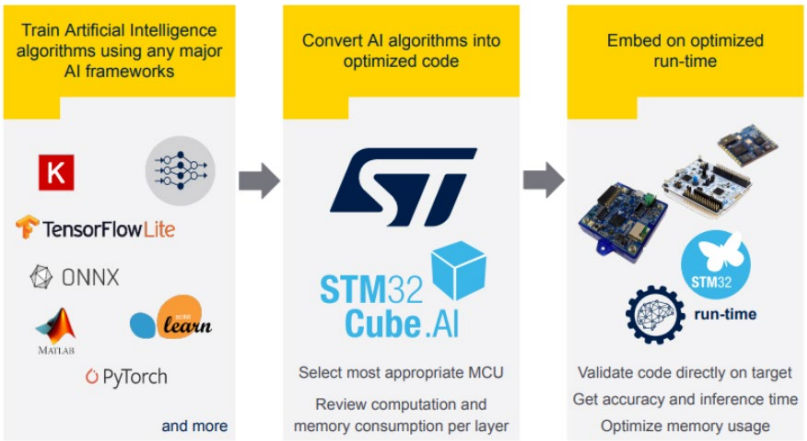

Figura 4.20 Flusso delle operazioni X-CUBE-AI

Il motore di ottimizzazione del generatore di codice C cerca di ottimizzare l'utilizzo della memoria (RAM e ROM) rispetto al tempo di calcolo dell'inferenza. Si basa su un approccio senza set di dati, il che significa che non è richiesto alcun set di dati valido o di test addestrato per applicare gli algoritmi di compressione e ottimizzazione (non è previsto alcun peso/stadio di bias riaddestrato/raffinato per preservare l'accuratezza del modello iniziale). Tale ottimizzazione si basa su:

- **Compressione dei pesi**: è applicabile solo per strati di tipo denso, consente un rapido processo di compressione, ma non è privo di perdite e l'accuratezza globale può risentirne.

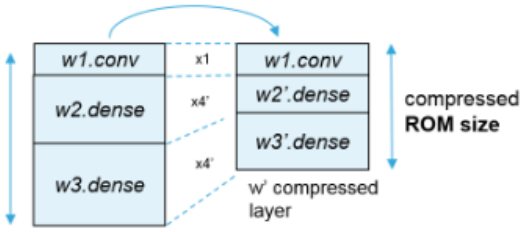

Figura 4.21 Compressione Pesi

- **Fusione:** Unisce alcuni livelli nello stato precedente per ottimizzare il tempo di elaborazione e ridurre il numero di livelli rispetto alla rete originale.

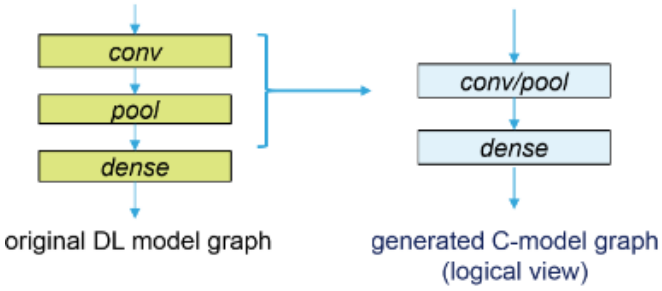

Figura 4.22 Operazione di fusione

4.2 Implementazione

Con il pacchetto di espansione X-Cube-AI, si è creata una applicazione di Validation tramite cui si è verificata l'occupazione della rete su board e si sono convalidati i risultati ottenuti dalla RNN [24].

Pack / Bundle / Component	Status	Version	Selection
STMicroelectronics.FP-SNS-MOTENV1		4.3.2	Install
STMicroelectronics.X-CUBE-AI		8.1.0	
Artificial Intelligence X-CUBE-AI		8.1.0	
Core		8.1.0	☑
Device Application		8.1.0	
Application		8.1.0	Validation

Figura 4.23 Validation Application

Come prevedibile, caricando il modello della nostra rete in formato keras .h5, e facendo una pre-analisi dal punto di vista del sistema di integrazione, pur completando correttamente la fase di analisi, si ottiene una rete non implementabile poiché vengono superati i limiti di memoria resi disponibili dalla scheda, in particolare la rete eccede i limiti relativi alla memoria flash, con una occupazione di 888.57 KB rispetto ai 512 KB disponibili.

Compression: High Optimization: Balanced

Validation inputs: Random numbers

Validation outputs: None

Show graph

Analyze

Validate on desktop

Validate on target

Complexity: 11155588 MACC
Used Flash: 888.57 KiB (888.57 KiB over 512.00 KiB Internal)
Used Ram: 42.92 KiB (42.92 KiB over 128.00 KiB Internal)
Achieved compression: 1.01
Analysis status: done

Figura 4.24 Pre-Analisi .h5 con compressione

Il porting dell'inferenza sul dispositivo comprende diverse strategie di ottimizzazione che vanno dalla riduzione delle dimensioni dei dati immessi scegliendo funzionalità più informative, all'abbassamento della frequenza di campionamento del segnale e alla riduzione del numero di bit utilizzati per rappresentare un numero per diminuire l'impronta della memoria [14]. Affronteremo ciascuno di questi approcci seguendo lo schema proposto in **Figura** 4.25.

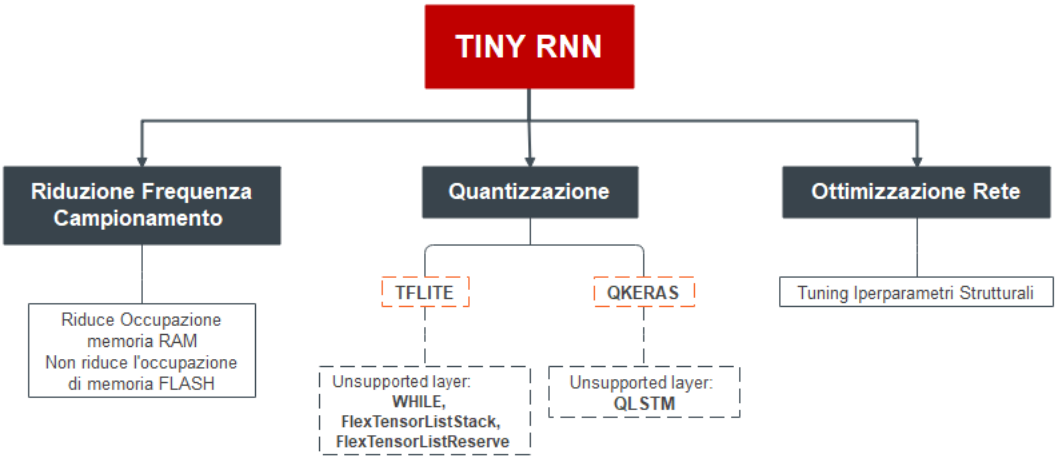

Figura 4.25 Strategie di compressione

Partiamo dalla considerazione che, mentre l'utilizzo della memoria RAM dipende fortemente dalla dimensione dei dati di input (e quindi dalla frequenza di campionamento), i requisiti di memoria FLASH dipendono solo dall'architettura di rete, ovvero dalla quantità di pesi e altri parametri che sono valori di sola lettura al termine dell'addestramento. Non è quindi possibile operare sui dati, è necessario intervenire sulla struttura della rete, per semplificarla e adattarla ad un'architettura vincolata.

4.3 Quantizzazione

Una famiglia di metodi di ottimizzazione per migliorare le prestazioni di una RNN è basata sulla quantizzazione della rete, ovvero nella riduzione del numero di bit utilizzati per rappresentare un valore. Ciò comporterà uno spazio di archiviazione più piccolo e un minor utilizzo della memoria in fase di esecuzione, migliorando la latenza della CPU a discapito di una leggera degradazione dell'accuratezza. Esistono due metodi di quantizzazione classici: Post-Training Quantization (PTQ) e Quantization Aware Training (QAT). Il primo è relativamente più semplice da usare. Consente la quantizzazione di un modello pre-addestrato con un set di dati limitato e rappresentativo (chiamato anche dataset di calibrazione). Nel QAT

l'addestramento basato sulla quantizzazione viene eseguito durante il processo di addestramento ed è spesso migliore per l'accuratezza del modello. X-CUBE-AI supporta l'implementazione di modelli con quantizzazione lineare a 8 bit, e al momento è in grado di fornire la quantizzazione solo per le reti in Keras e in Tensorflow Lite.

4.3.1 *TensorFlow Lite*

TensorFlow Lite (TFL) è una versione semplificata di TensorFlow, ottimizzata per l'esecuzione di modelli neurali su dispositivi mobili e con risorse limitate. La rete, addestrata utilizzando TensorFlow, viene salvata e quantizzata dal framework TensorFlow Lite, principalmente tramite l'utilità "Convertitore TFLite", generando un file in formato .tflite che viene convertito nell''implementazione C ottimizzata per la piattaforma di destinazione. L'obiettivo di questa tecnica è ridurre le dimensioni del modello migliorando allo stesso tempo la latenza della CPU e dell'acceleratore hardware (compresi gli aspetti relativi al consumo energetico) con un degrado minimo della precisione del modello.

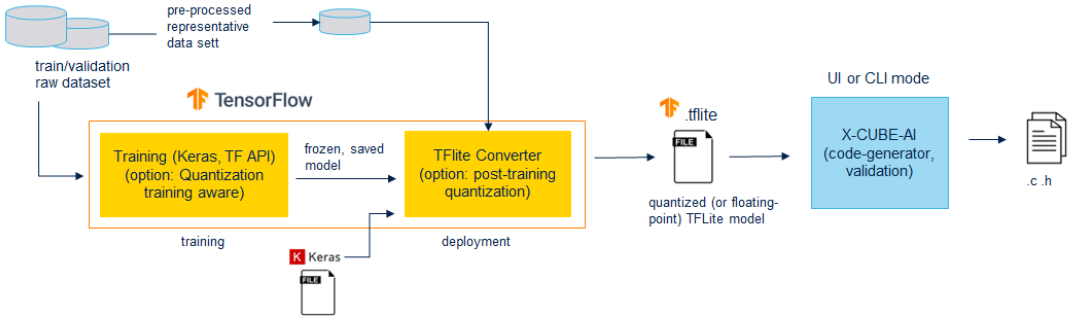

Figura 4.26 *Flusso di quantizzazione*

Tensorflow implementa le due forme di quantizzazione PTQ e QAT. La prima è più facile da usare sebbene la QAT sia spesso migliore per l'accuratezza del modello. Questo tipo di quantizzazione permette 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 4-1, che consente di convertire la scala dei numeri in virgola mobile nella scala dei numeri interi come rappresentato in Figura 4.27

$$real_value = (sint8_value - zero_point) \times scale \quad \textbf{(4-1)}$$

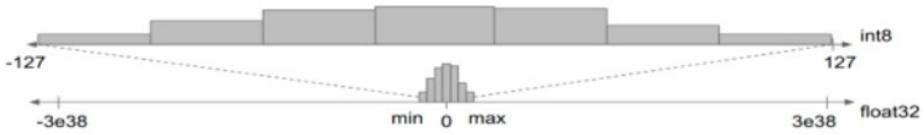

Figura 4.27 **-** Conversione scala valori virgola mobile nella scala dei valori interi

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 la QAT. Tuttavia, ciò richiede modifiche durante l'addestramento del modello per aggiungere nodi di quantizzazione fittizi.

Nel nostro caso abbiamo utilizzato la quantizzazione post training a gamma dinamica, in cui vengono quantizzati staticamente solo i pesi da virgola mobile a intero al momento della conversione, senza dover fornire un set di dati per la calibrazione. Abbiamo convertito il modello TensorFlow già addestrato nel formato TensorFlow Lite quantizzato utilizzando il TensorFlow Lite Converter:

```
converter = tf.lite.TFLiteConverter.from_keras_model(model)
converter.optimizations = [tf.lite.Optimize.DEFAULT]
converter.experimental_new_converter=True
converter.target_spec.supported_ops = [tf.lite.OpsSet.TFLITE_BUILTINS,
tf.lite.OpsSet.SELECT_TF_OPS]
tflite_model = converter.convert()
```

Listato 4.1

La quantizzazione ha comportato la riduzione delle dimensioni del modello, che da 2.683 KB si è ridotto a 253 KB, con un fattore di compressione superiore a 10.

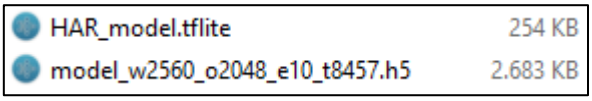

Figura 4.28 – Compressione Modello

Sfortunatamente, l'attuale versione di STM32Cube.AI, la 8.1.0, non supporta alcune operazioni specifiche generate per il nostro modello: con entrambi i runtime CubeAI e TFLite Micro, durante la fase di analisi, la conversione tflite introduce dei layer non supportati, nello specifico:

`NOT IMPLEMENTED: Unsupported layer types: WHILE, FlexTensorListStac, FlexTensorListReserve, stopping`.

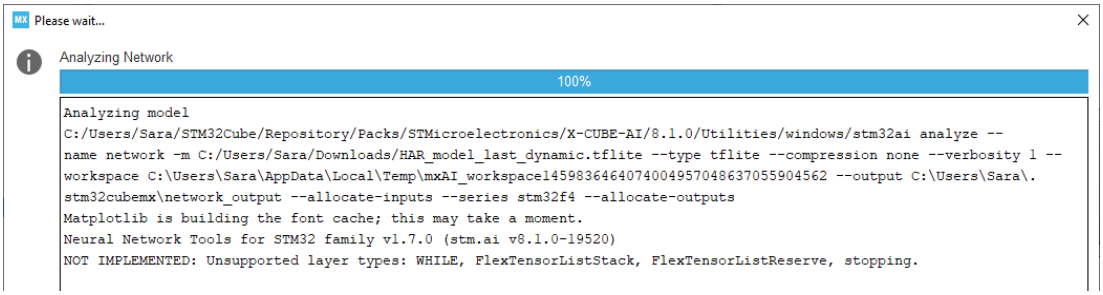

Figura 4.29 *Analisi Cube-AI*

Analizzando la rete con il visualizzatore Netron, si osserva che il TFLite Converter aggiunge i layer *WHILE, FlexTensorListStack, FlexTensorListReserve* in corrispondenza dei livelli LSTM. Abbiamo verificato che questi livelli vengono introdotti per qualsiasi tipo di quantizzazione (dinamica, int8 e float) e con diverse topologie di rete ricorrente, anche per reti con un singolo strato LSTM. L'incompatibilità è nota e in fase di analisi e risoluzione da parte del team di sviluppo ST [25].

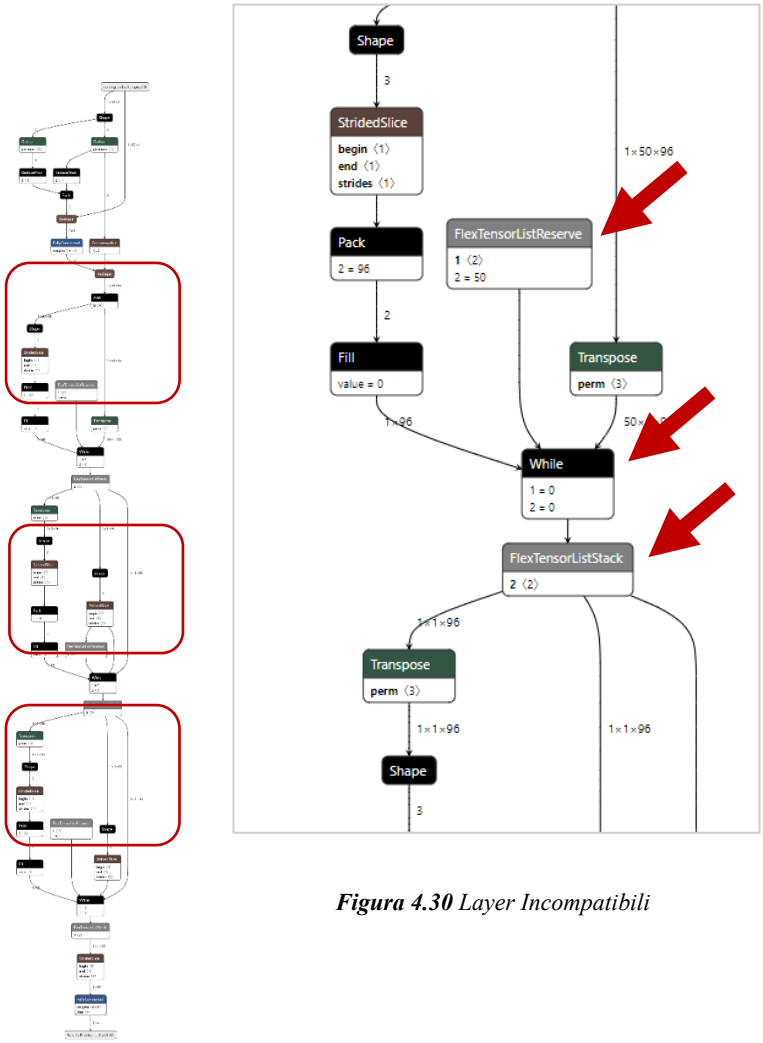

Figura 4.30 *Layer Incompatibili*

La conversione TFlite senza quantizzazione consente la generazione di un modello compatibile con il pacchetto di espansione Cube-AI, tuttavia, anche in questo caso, l'occupazione della rete eccede i limiti della memoria Flash disponibile:

HAR_model.tflite	881 KB
model_w2560_o2048_e10_t8457.h5	2.683 KB

Figura 4.31 *Compressione TFlite*

Complexity: 11155588 MACC
Used Flash: 892.21 KiB (892.21 KiB over 512.00 KiB Internal)

Figura 4.32 *Analisi Cube-AI*

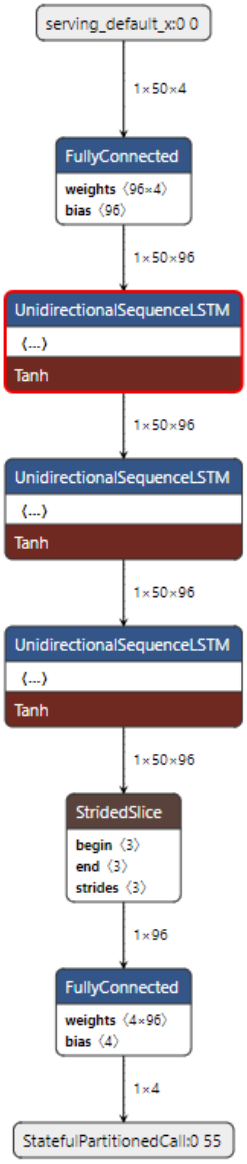

Figura 4.33 *Modello TFlite della RNN*

4.3.2 *QKeras*

Come visto, i modelli quantizzati codificano ciascun peso e attivazione utilizzando un formato dati intero con o senza segno a 8 bit. L'ottimizzazione e quantizzazione di una rete in virgola mobile consente quindi l'implementazione di una rete con aritmetica che utilizza numeri interi più piccoli, e che è quindi più efficiente in termini di risorse computazionali. X-CUBE-AI però fornisce supporto anche per le Deep Quantized Neural Network (DQNN), ovvero modelli che utilizzano una dimensione inferiore a 8 bit per la codifica degli operatori statici (pesi, attivazione) [26]. È inoltre supportata una codifica mista per ciascun operatore, con ad esempio una codifica binaria per il peso, e una codifica con segno a 8 bit per l'attivazione, consentendo un compromesso in termini di accuratezza e precisione rispetto all'utilizzo di memoria.

Le istruzioni Arm Cortex-M e le manipolazioni dei dati (come le operazioni di compressione e decompressione dei dati durante i trasferimenti dalla memoria) non consentono un'implementazione efficiente per tutte le combinazioni di tipi di dati. X-CUBE-AI si concentra principalmente sulle implementazioni per migliorare l'utilizzo della memoria (flash, RAM o entrambe) e ridurre la latenza, non offre quindi ottimizzazione per le dimensioni dei dati. Pertanto, solo i tipi di dati float a 32 bit, con segno a 8 bit e con segno binario (1 bit) vengono considerati dal generatore di codice per distribuire i kernel ottimizzati.

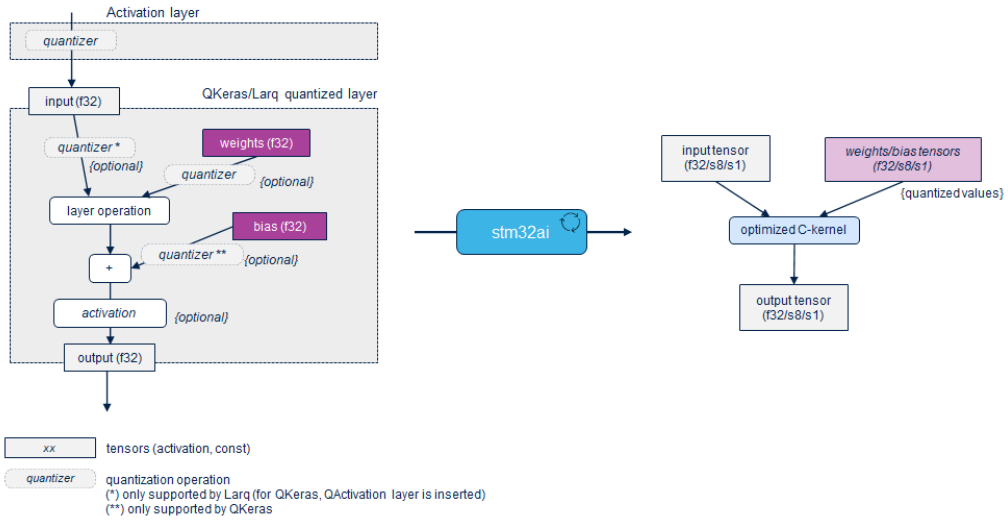

Figura 4.34 - *Generazione codice da livello quantizzato*

Il tipo di dati dei tensori di input è dedotto dal tipo di dati degli operatori in input. Nel caso in cui i tensori di input, output e peso siano quantizzati con un classico schema intero a 8 bit (come per i modelli quantizzati TensorFlow Lite), vengono utilizzate le rispettive

implementazioni ottimizzate int8 del kernel C. Analogamente per gli operatori in virgola mobile. Nello specifico X-CUBE-AI consente l'implementazione di modelli DQNN che siano stati progettati e addestrati con la libreria QKeras.

La libreria QKeras è progettata come un'estensione dell'API Keras di alto livello e fornisce un modo semplice per creare rapidamente una versione quantizzata della rete Keras originale descrivendo come quantizzare le attivazioni e i pesi in entrata. È stata progettata secondo i principi di semplicità di utilizzo, modularità ed estensibilità, oltre al essere "minimamente invasiva" delle funzionalità native di Keras. QKeras consente una sostituzione immediata per alcuni livelli di Keras, in particolare quelli che creano parametri e livelli di attivazione ed eseguono operazioni aritmetiche, in modo da poter creare rapidamente una versione quantizzata della rete Keras.

I livelli quantizzati sono completamente compatibili con l'API Keras, quindi possono essere utilizzati in modo intercambiabile e combinato con i Layers Keras, in modo da creare modelli misti con livelli quantizzati e non quantizzati. Il risultato è riportato nel Listato 4.2, da cui si può notare che le modifiche necessarie consistono nell'aggiunta di una Q prima del nome del livello originale Keras e nella definizione del tipo di quantizzazione, ad esempio con i parametri kernel_quantizer e bias_quantizer. Come detto, queste modifiche sono state applicate solo ai livelli che effettuano dei calcoli sui dati in ingresso, mentre restano invariati i livelli che effettuano solo delle manipolazioni.

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.

```
def create_qmodel(sample_window, dense1, lstm1, lstm2, lstm3 = 0, dropout = 0.2):
  qmodel = keras.Sequential()
  qmodel.add(keras.Input(shape = sample_window.shape, name = 'input'))
  if dense1: qmodel.add(QDense(dense1, kernel_quantizer='quantized_bits(6,0,1)',
bias_quantizer='quantized_bits(6)', name = 'dense5'))
   qmodel.add(QLSTM(lstm1, activation='quantized_tanh(6)',
recurrent_activation='quantized_relu(4,0,1)',
                   kernel_quantizer='stochastic_ternary("auto")',
recurrent_quantizer='quantized_bits(2,1,1,alpha=1.0)' ,
                   bias_quantizer='quantized_bits(4,0,1)',  return_sequences =
bool(lstm2), name = 'lstm1'))
  if lstm2:
    qmodel.add(Dropout(dropout, name = 'drop2'))
    qmodel.add(QLSTM(lstm2, activation='quantized_tanh(4)',
recurrent_activation='quantized_relu(4,0,1)',
                     kernel_quantizer='stochastic_ternary("auto")',
recurrent_quantizer='quantized_bits(2,1,1,alpha=1.0)' ,
                     bias_quantizer='quantized_bits(4,0,1)', return_sequences =
bool(lstm3), name = 'lstm2'))
  if lstm3:
    qmodel.add(Dropout(dropout, name = 'drop3'))
    qmodel.add(QLSTM(lstm3, activation='quantized_tanh(4)',
recurrent_activation='quantized_relu(4,0,1)',
```

```
                    kernel_quantizer='stochastic_ternary("auto")',
recurrent_quantizer='quantized_bits(2,1,1,alpha=1.0)' ,
                    bias_quantizer='quantized_bits(4,0,1)', name = 'lstm3'))
  qmodel.add(Dropout(dropout, name = 'drop4'))
  qmodel.add(QDense(num_classes, kernel_quantizer='quantized_bits(4,0,1)',
bias_quantizer='quantized_bits(4)', name = 'dense2'))
  qmodel.compile(loss = keras.losses.SparseCategoricalCrossentropy(from_logits = True),
    optimizer = 'adam', metrics = ['accuracy'])
  qmodel.summary()
  return qmodel
```

Listato 4.2

Anche in questo caso, l'attuale versione di STM32Cube.AI, la 8.1.0, non supporta alcune operazioni specifiche generate per il modello quantizzato: durante la fase di analisi, è stata segnalata l'incompatibilità con un layer:

`QLSTM: UNSUPPORTED LAYER TYPE.`

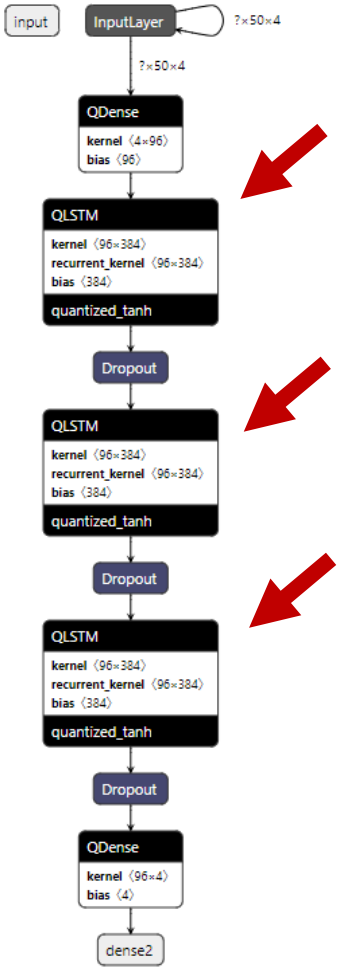

Figura 4.35 *Architettura RNN con quantizzazione QKERAS*

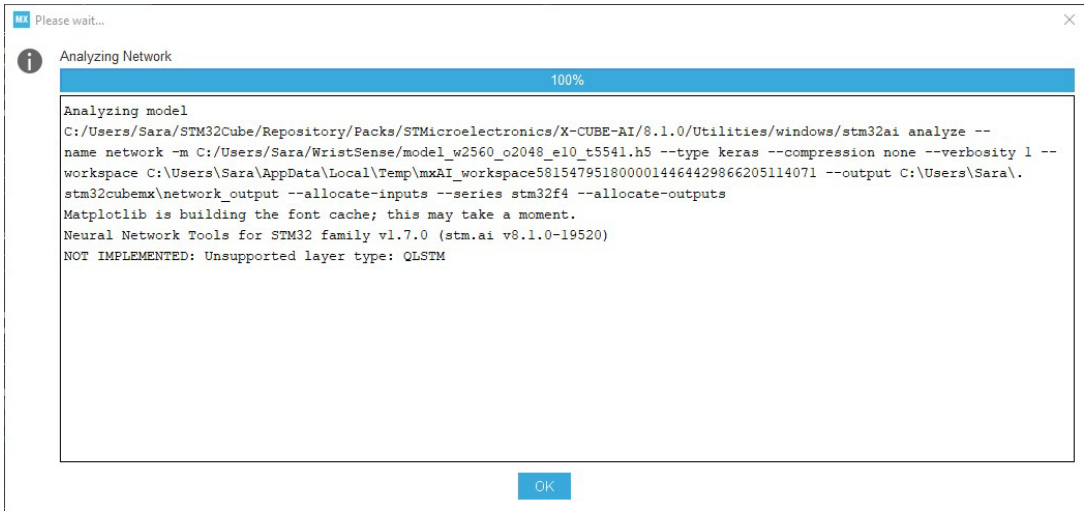

Figura 4.36 *– Analisi per modello QKeras*

4.4 Tiny RNN

Le diverse strategie di quantizzazione sono risultate non compatibili con il core STM32-CUBE-AI, si è quindi seguita una terza strategia, che consiste nella semplificazione della struttura della rete: si riduce il numero di parametri e si identificano parti della rete che non siano essenziali per il processo decisionale, in modo da trovare un compromesso tra la complessità del modello e l'accuratezza finale, e poter quindi importare il modello su piattaforma embedded.

La messa a punto di una rete neurale implica la taratura dei suoi parametri per ottimizzare le sue prestazioni e ottenere la massima efficacia in termini di accuratezza. In questa fase vogliamo invece ottimizzare la complessità della rete per ridurre le esigenze di calcolo del modello (cicli della CPU, utilizzo della memoria e consumo energetico) per soddisfare i vincoli di risorse del dispositivo embedded e, allo stesso tempo, per ridurre al minimo la perdita di prestazioni del modello. Ci concentreremo sulla messa a punto degli iperparametri strutturali e del numero di parametri addestrabili per ottenere un buon compromesso tra accuratezza dell'inferenza e utilizzo delle risorse.

Le prestazioni della rete sono state studiate modificando: (i) la struttura e le dimensioni della rete, (ii) la dimensione dell'elaborazione della finestra e (iii) il numero di funzionalità di input. Si è visto che l'impronta di memoria del modello non è influenzata dalla dimensione della finestra di elaborazione poiché il numero di neuroni sullo strato di input dipende solo dal numero di caratteristiche calcolate. Per mantenere un buon compromesso tra prestazioni di classificazione e utilizzo delle risorse, per tutti gli esperimenti condotti è stata mantenuta la dimensione della finestra originaria.

Andando ad operare sugli iperparametri strutturali, si è trovato un compromesso con la seguente architettura, che consente la riduzione del 70% della complessità della rete rispetto a quella originale:

Livello	Valore
DROP1	0,2
LSTM1	50
DROP1	0,2
LSTM2	50
DROP2	0,2
LSTM3	50
DROP3	0,2
Batch	20
Epoch	10
Window	10s
Overlap	8s
Oversampling	Sì
PCA	sì

Tabella 4-1

	Tiny RNN		RNN	
LAYER	**OUTPUT SHAPE**	**PARAM #**	**OUTPUT SHAPE**	**PARAM #**
Dense	50x50	250	50x96	480
LSTM_1	50x50	20200	50x96	74112
Dropout_1	50x50	0	50x96	0
LSTM_2	50x50	20200	50x96	74112
Dropout_2	50x50	0	50x96	0
LSTM_3	50	20200	96	74112
Dropout_3	50	0	96	0
Dense	4	204	4	388

Tabella 4-2

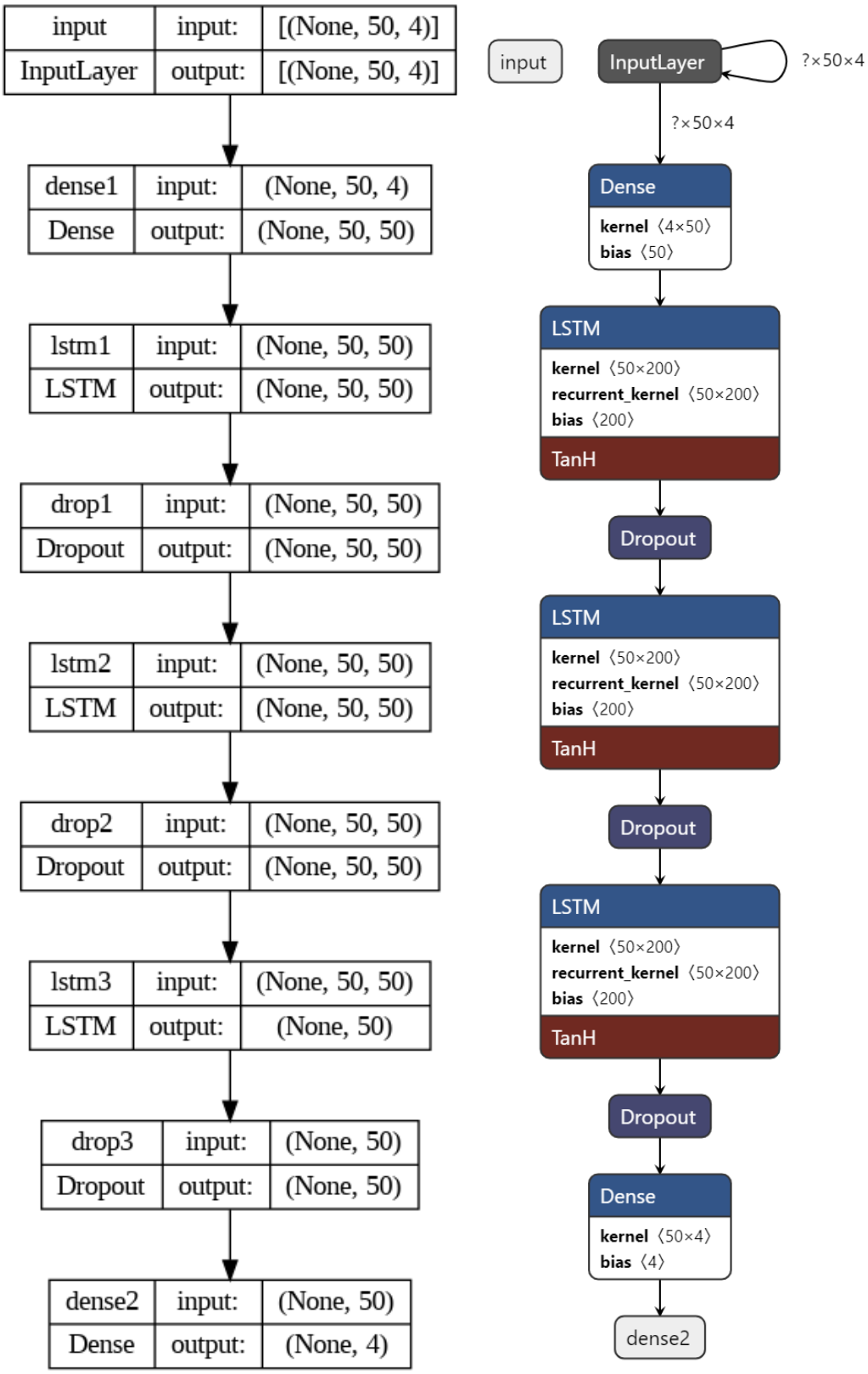

Figura 4.37 *Architettura Tiny RNN*

Con questa configurazione i parametri della rete si riducono ad un terzo rispetto alla rete originale (Tabella 4-2), ne consegue che la rete così semplificata ha un'occupazione di memoria dimezzata: l'analisi di X-Cube-AI conferma la compatibilità della rete con le risorse disponibili sul dispositivo.

```
elapsed time (analyze): 9.909s
Model file:      model_w2560_o2048_e30_t7198.h5
Total Flash:     436760 B (426.52 KiB)
    Weights:     418536 B (408.73 KiB)
    Library:     18224 B (17.80 KiB)
Total Ram:       43484 B (42.46 KiB)
    Activations: 41088 B (40.12 KiB)
    Library:     2396 B (2.34 KiB)
    Input:       800 B (included in Activations)
    Output:      16 B (included in Activations)
Done
Analyze complete on AI model
```

```
Complexity:  5207104 MACC
Used Flash: 426.52 KiB (426.52 KiB over 512.00 KiB Internal)
Used Ram: 42.46 KiB (42.46 KiB over 96.00 KiB Internal)
Achieved compression: -
Analysis status: done
```

Figura 4.38 *Analisi Cube-AI*

La semplificazione della rete neurale comporta un miglioramento delle prestazioni in termini di tempi di inferenza, con una riduzione del 20% rispetto alla rete originaria.

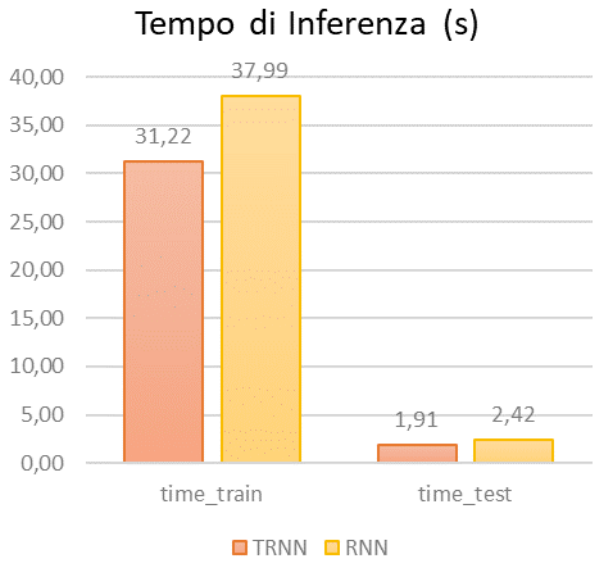

Figura 4.39 *Tempi di calcolo*

Analizziamo di seguito le prestazioni della Tiny RNN, mettendole a confronto con le prestazioni della rete iniziali.

In ***Figura 4.40*** osserviamo un peggioramento di circa 5% su tutte le metriche.

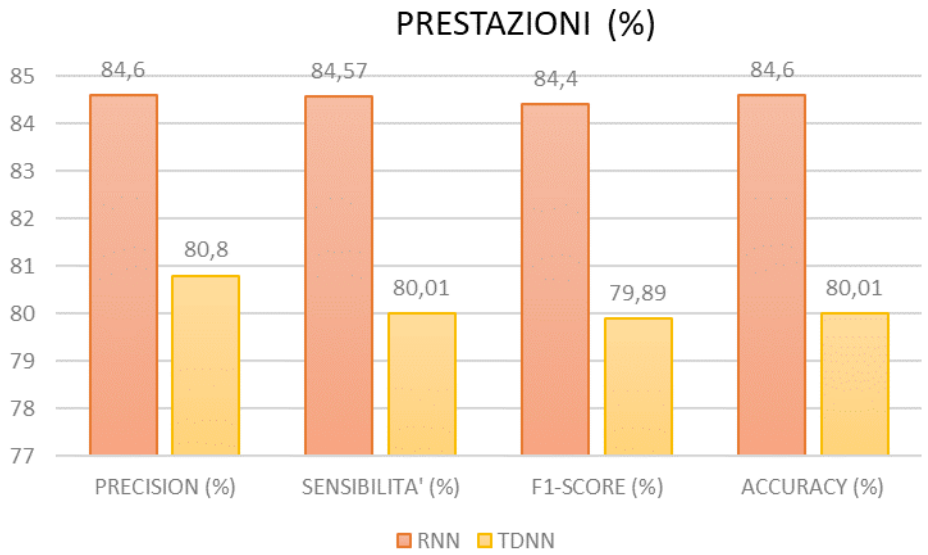

Figura 4.40 *Confronto prestazioni tra RNN e Tiny RNN*

Nel dettaglio, le matrici di confusione confermano un leggero peggioramento delle prestazioni, soprattutto nell'identificazione della classe Low Bike. Ancora una volta, ciò può essere spiegato dal fatto che il set di dati è di dimensioni limitate e quindi un singolo soggetto potrebbe non essere sufficientemente rappresentativo per essere utilizzato nella fase di test. Si ha quindi conferma che la dimensione limitata del set di dati può limitare la generalità dei risultati, producendo una distorsione, a seconda della partizione dei soggetti del dataset.

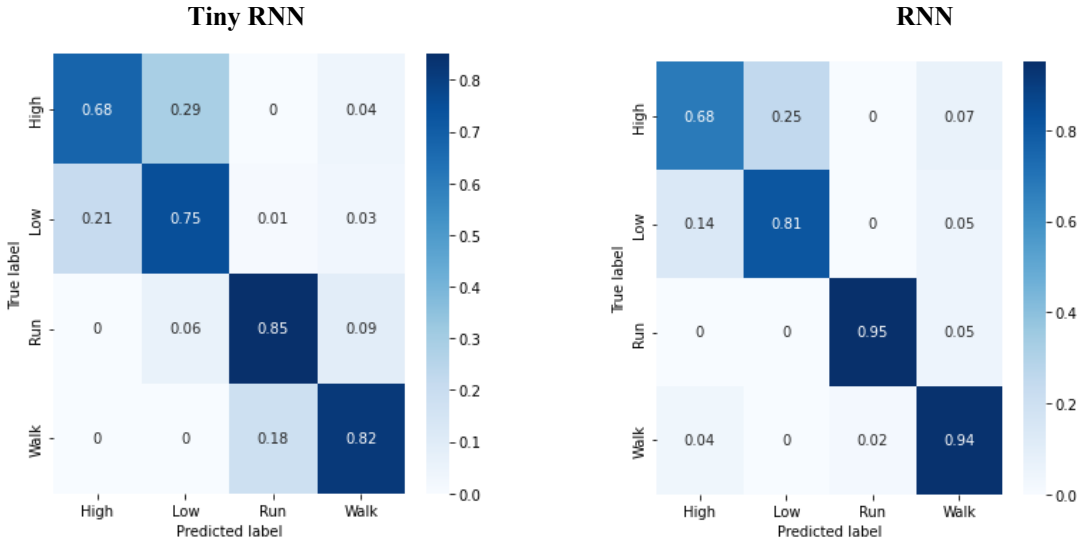

Figura 4.41 *Matrice di Confusione Tiny RNN (a sinistra) e RNN (a destra)*

Al fine di ottimizzare ulteriormente i risultati, valutiamo le performance del modello TFlite ottenuto dalla conversione della rete TinyRNN.

Come prevedibile, si ottiene una compressione delle dimensioni della rete, con una occupazione di memoria che si riduce del 70% rispetto alla occupazione della rete originaria (***Figura 4.42***). Analogamente il tempo di inferenza si riduce del 50% rispetto alla TinyRNN, e del 61% rispetto alla RNN originaria ***Figura 4.43***.

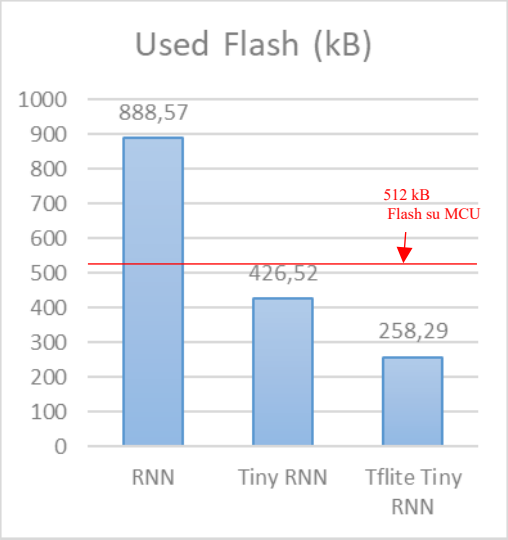

Figura 4.42 *Occupazione memoria Flash delle tre reti*

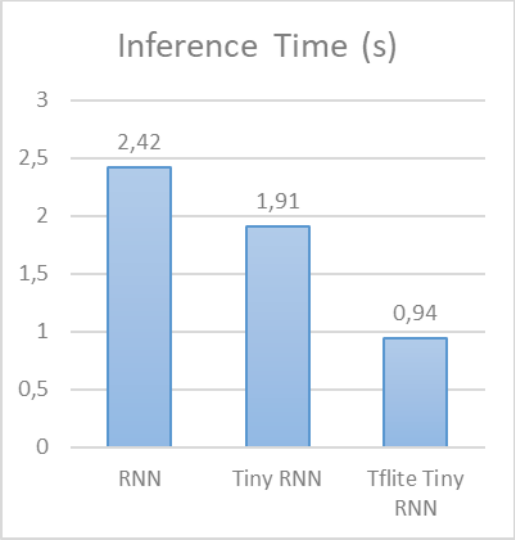

Figura 4.43 *Tempo di inferenza delle tre reti*

Il miglioramento dei tempi di risposta e dell'occupazione di memoria avviene però a discapito dell'accuratezza di previsione: mentre con la TinyRNN le prestazioni si mantengono al di sopra dell'80%, per la rete TFlite ci si ferma al 77,8%, con un peggioramento complessivo dell'8% rispetto alla rete originaria.

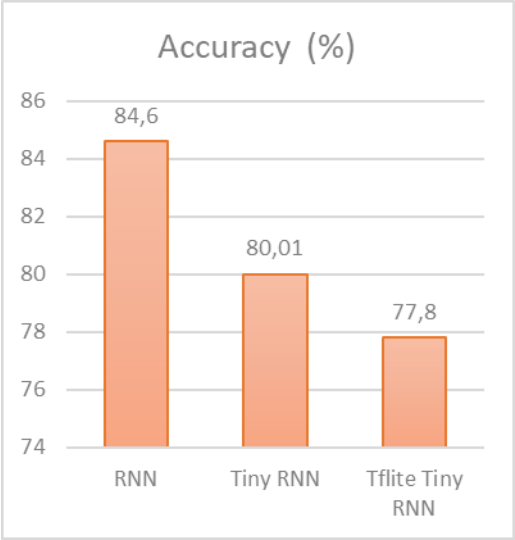

Figura 4.44 *Accuratezza delle tre reti*

Nonostante diversi tentativi di ottimizzazione, non si è riusciti a trovare una configurazione della rete che consenta un miglior compromesso tra risorse impiegate e accuratezza di previsione. Considerando che l'occupazione di memoria flash dipende principalmente dall'architettura della rete e da una serie di altri parametri che sono di sola lettura dopo la fase di training, si ritiene che nella maggior parte delle applicazioni sia conveniente adottare la soluzione che garantisce migliore accuratezza, ovvero la rete TinyRNN senza conversione TFlite. Per applicazioni a bassa latenza o su dispositivi con risorse ridotte, è invece indicato l'utilizzo della configurazione che sfrutta al meglio le prestazioni microcontrollore, ovvero la rete TinyRNN con conversione TFlite.

CONCLUSIONI

Il continuo sviluppo della tecnologia ha reso possibile la produzione di sistemi elettronici indossabili dotati di capacità di rilevamento, calcolo e comunicazione con consumi energetici, dimensioni e costi ridotti. La possibilità di eseguire inferenze non banali direttamente su tali dispositivi è di notevole importanza, ci siamo quindi posti l'obiettivo di progettare una Rete Neurale Ricorrente per il riconoscimento di attività fisiche che fosse integrabile su dispositivi di edge computing e che al tempo stesso garantisse buone prestazioni.

Partendo da un set di dati pubblico, abbiamo verificato che la classificazione delle attività risulta più accurata se i segnali forniti in ingresso vengono opportunamente processati e combinati, in particolare si sono rivelati decisivi da un lato l'estrazione delle caratteristiche essenziali del segnale tramite analisi PCA, dall'altro la combinazione di segnali PPG, che forniscono informazioni relative allo sforzo fisico, e segnali di accelerazione, che riducono il rumore introdotto dagli artefatti da movimento e danno informazioni sul tipo di movimento svolto. La fase di progettazione e ottimizzazione della rete ha confermato quanto affermato in letteratura: lunghe sequenze di dati prediligono l'utilizzo di strati LSTM, che consentono l'identificazione di dipendenze a lungo termine, e strati di dropout che, rimuovendo casualmente alcune connessioni durante la fase di training, consentono la riduzione dell'overfitting.

Dal confronto della rete con un classificatore bayesiano, sviluppato nello studio di riferimento [8], abbiamo verificato che le tecniche classiche di Machine Learning, pur rappresentando un approccio efficace, richiedono un'attenta ingegnerizzazione e garantiscono buoni risultati con dataset limitati e poco complessi. Le reti neurali, partendo direttamente dai dati grezzi dei sensori, forniscono prestazioni migliori in termini di efficienza e accuratezza della classificazione, ciò nonostante la limitatezza dei dati disponibili nel dataset.

La fase più critica del progetto è stata l'esportazione del modello su piattaforma embedded: avendo generato una architettura piuttosto complessa, le dimensioni della rete hanno superato il limite fisico di memoria flash installata sul dispositivo embedded, è stato quindi necessario

adottare delle strategie di compressione della rete che non compromettessero i valori di accuratezza ottenuti.

Le due strategie volte alla quantizzazione della rete si sono rivelate inefficaci: in entrambi i casi si è riscontrato un problema di compatibilità tra i modelli quantizzati e il pacchetto di espansione X-Cube-AI. Da un successivo e approfondito studio della rete, si è riusciti ad operare una semplificazione, risultato di un compromesso tra occupazione di memoria, tempo di risposta, consumo energetico e precisione del modello della rete. Tale semplificazione ha causato un modesto peggioramento delle prestazioni, ma ha dimezzato le risorse richieste e ridotto i tempi di risposta della rete. Ciò ha consentito alla rete di essere implementata sul dispositivo embedded, aprendo la strada a un classificatore di attività completamente autonomo, efficiente ed economico, realizzabile come dispositivo incorporato indossabile.

In sintesi, in questo lavoro viene confermata la natura multiforme del problema di riconoscimento delle attività fisiche a bordo di dispositivi caratterizzati da risorse limitate e si è mostrato come un'attenta indagine della interazione tra componenti hardware e software possa fornire soluzioni per la progettazione di modelli compatibili con severi requisiti energetici senza influenzare in modo significativo i livelli di accuratezza. Con la rapida diffusione dell'Internet of Things (IoT) e la crescita esponenziale degli oggetti connessi alla rete, l'edge computing, e quindi il trasferimento della potenza di calcolo e dello spazio di archiviazione all'interno del perimetro del dispositivo, si stia affermando come la scelta migliore per colmare il divario tra IoT e cloud, è quindi fondamentale adottare tecniche di ottimizzazione e compressione che consentano ai sistemi di diventare più flessibili, economicamente vantaggiosi, sicuri e scalabili.

BIBLIOGRAFIA

[1] F. FOKUS, «https://www.fokus.fraunhofer.de/en/fame/workingareas/ai,» [Online].

[2] G. Cybenko, «Approximation by superpositions of a sigmoidal function,» *Mathematics of Control, Signals and Systems,* vol. 2, pp. 303-314, 1989.

[3] M. Alessandrini, G. Biagetti, P. Crippa, L. Falschetti e C. Turchetti, «Recurrent Neural Network for Human Activity Recognition in Embedded Systems Using PPG and Accelerometer Data,» *Electronics,* vol. 10, 2021.

[4] C. Olah, «Understanding LSTM Networks,» 27 08 2015. [Online]. Available: https://colah.github.io/posts/2015-08-Understanding-LSTMs/.

[5] S. Ruder, An overview of gradient descent optimization, 2016: CoRR, abs/1609.04747.

[6] E. Khan, F. Hossain, S. Uddin, S. Alam e M. Hasan, «A Robust Heart Rate Monitoring Scheme Using Photoplethysmographic Signals Corrupted by Intense Motion Artifacts,» *IEEE Trans. Biomed. Eng.,* vol. 63, pp. 550-562, 2016.

[7] B. G., C. P., F. L., O. S. e T. C., «An efficient technique for real-time human activity classification using accelerometer data,» *Intelligent Decision Technologies ,* p. 425–434, 2016.

[8] G. Biagetti, P. Crippa, L. Falaschetti, S. Orcioni e C. Turchetti, «Human Activity Recognition Using Accelerometer and Photoplethysmographic Signals,» *Intelligent Decision Technologies,* vol. 73, pp. 57-58, 2018.

[9] A. Casson, «Wrist PPG During Exercise,» 20 10 2017. [Online]. Available: https://www.physionet.org/content/wrist/1.0.0/.

[10] A. Goldberger, L. Amaral e L. Glass, «hysioBank, PhysioToolkit, and PhysioNet: Components of a New Research Resource for Complex Physiologic Signals,» *Circulation,* vol. 101, pp. 215-220, 2000.

[11] D. Jarchi e A. Casson, «Description of a Database Containing Wrist PPG Signals Recorded during Physical Exercise with Both Accelerometer and Gyroscope Measures of Motion,» *Data,* 2017.

[12] S. Eddins, «Classify ECG Signals Using LSTM Networks,» [Online]. Available: https://blogs.mathworks.com/deep-learning/2018/08/06/classify-ecg-signals-using-lstm-networks/.

[13] M. Alessandrini, G. Biagetti, P. Crippa, L. Falaschetti, S. Luzzi e C. Turchetti, «EEG-Based Alzheimer's Disease Recognition Using Robust-PCA and LSTM Recurrent Neural Network,» *Sensors,* n. 22, 2022.

[14] M. D. V. F. E. Lattanzi, «Exploring Artificial Neural Networks Efficiency in Tiny Wearable Devices for Human Activity Recognition,» *Sensors,* n. 7, 2022.

[15] A. Charu C, Neural Networks and Deep Learning, Springer, 2018.

[16] M. Musci, D. De Martini, N. Blago, T. Facchinetti e M. Piastra, «Online Fall Detection using Recurrent Neural Networks on Smart Wearable Devices,» *IEEE Trans. Emerg. Top. Comput,* 2020.

[17] Tensorflow. [Online]. Available: https://www.tensorflow.org/api_docs/.

[18] B. Schmeiser., «Batch size effects in the analysis of simulation output,» *Operations Research,* n. 30, p. 556–568, 1982.

[19] R. Rawat, J. Patel e M. Manry, «Minimizing validation error with respect to network size and number of training epochs,» *The 2013 international joint conference on neural networks (IJCNN,* pp. 1-7, 2013.

[20] P. S. e. S. Sarangi, «Internet of Things: Architectures, Protocols,and Applications,» *Journal of Electrical and Computer Engineering ,* pp. 1-25, 2017.

[21] Tensorflow.org, «TensorFlow model optimization.,» [Online]. Available: https://www.tensorfow.org/model_optimization/guide.

[22] «Cloud-JAM,» [Online]. Available: https://github.com/rushup/Cloud-JAM.

[23] STM32, «STM32Nucleo pack for IoT node,» [Online]. Available: https://www.st.com/en/evaluation-tools/p-nucleo-cld1.html.

[24] STMicroelectronics, Getting started with X-CUBE-AI Expansion Package for Artificial Intelligence (AI), 2022.

[25] ST, «STMicroelectronics Community,» 26 05 2023. [Online]. Available: https://community.st.com/t5/stm32-machine-learning-ai-mcu/keras-2-12-0-support/m-p/53351.

[26] «QKeras: a quantization deep learning library for Tensorflow Keras,» [Online]. Available: https://github.com/google/qkeras.

[27] Y. B. G. Hinton Y. LeCun, Deep learning, Nature, 2015.

[28] K. Xu, X. Jiang, H. Ren, X. Liu e W. Chen, «Deep Recurrent Neural Network for Extracting Pulse Rate Variability from Photoplethysmography During Strenuous Physical Exercise.,» *In Proceedings of the 2019 IEEE Biomedical Circuits and Systems Conference (BioCAS),* pp. 1-4, 2019.

[29] S. Badillo, B. Banfai, F. Birzele, I. Davydov e L. Hutchinson, «An Introduction to Machine Learning,» *CLINICAL PHARMACOLOGY & THERAPEUTICS* , vol. 107, n. 4, April 2020.

[30] G. Biagetti, P. Crippa, A. Curzi, O. S. e C. Turchetti, «A multi-class ECG beat classifier based on the truncated KLT representation,» *UKSim-AMSS 8th EMS on Computer Modelling and Simulation,* pp. 93-98, 2014.

[31] Bac`a, C. Biagetti, Crippa, Falaschetti, Orcioni, Rossini, Tonelli e Turchetti, «A robust motion artifact reduction algorithm for heart rate monitoring from PPG signals,» *23rd European Signal Processing Conference,* p. 2696–2700, 2015.