Alma Mater Studiorum · Università di Bologna

SCUOLA DI SCIENZE

Corso di Laurea in Informatica Curriculum A - Tecniche del software

PROGETTAZIONE ED ANALISI DI TECNICHE UNSUPERVISED IN SUPPORTO ALLA HUMAN ACTIVITY RECOGNITION

Relatore:
Chiar.mo Prof.
Di Felice Marco
Correlatore:
Montori Federico

Presentata da:
Vallasciani Giacomo

Sessione IV
Anno Accademico 2021/2022

Abstract

Portare l'intelligenza artificiale nei dispositivi integrati è diventato un argomento di ricerca centrale in molti settori scientifici (ambiente, agricoltura, sociologia, salute...). Per il riconoscimento dell'attività umana, le reti neurali artificiali (ANN) hanno dimostrato la loro capacità di fornire prestazioni migliori rispetto ad altri importanti limiti. In primo luogo, spesso le ANN sono allenate utilizzando la tecnica del "supervised learning" che richiede dataset pre-etichettati. Inoltre, questi algoritmi sono genericamente molto costosi in termini di potenza di calcolo. Per questo motivo, la loro integrazione in microcontrollori a bassa potenza è stata finora valutata solo in misura limitata. In questo lavoro riprenderemo l'esperimento presentato in [18] verificandone prima i risultati, tentare di migliorare i risultati ottenuti con il dataset **UCI**, tentare di esportare i modelli unsupervised direttamente sul dispositivo M5Stack attraverso l'utilizzo di tecnologie come **Tensorflow-Lite**, esplorare nuove possibili soluzioni unsupervised che possono essere usate al posto della Self-Organizing-Map e cercando poi di ripetere l'esperimento acquisendo un nuovo dataset, seguendo la costruzione del dataset UCI, utilizzando i sensori **IMU** (accelerometro e giroscopio) di un M5Stack Gray, un micro-controllore con processore ESP-32.

Keywords: *Internet Of Things, Artificial Intelligence, Machine-Learning, Tensorflow-Lite, Human Activity Recognition, Microcontrollers*

Indice

Elenco delle figure

Listings

Capitolo 1

Introduzione

Il forte interesse verso l'Internet of Things e dispositivi indossabili porta alla proliferazione di casi d'uso che producono giorno dopo giorno grandi set di dati in vari ambiti come la salute, i trasporti o lo sport.
Smartphones e dispositivi indossabili permettono di catturare e processare un vasto range di dati come dati ambientali o fisiologici.
Questa vasta collezione di dati ha reso possibile lo sviluppo di varie applicazioni di assistenza sanitaria basate sul monitoraggio personale basato sui dati. "**Human Activity Recognition**" è un problema di classificazione che cerca di predire l'attività che un utente sta svolgendo tra quelle presenti in un insieme di attività conosciute (descritte dalle label).
La tecnica del **Machine Learning** (ML) è uno dei modi per cercare di imparare da un grande volume di dati con lo scopo di estrarre un modello predittivo generico. Uno dei modi attraverso il quale viene applicato il Machine Learning è l'utilizzo delle **ANN**, queste ultime hanno mostrato le performance migliori comparate ad altre tecniche di ML.

Questi casi d'uso formano un interessante campo di applicazioni per l'intelligenza artificiale integrata dove il flusso dei dati deve essere continuamente processato sotto forti vincoli energetici.
In questo contesto, i dati dei sensori vengono tipicamente rappresentati come "**time series**" che arrivano senza nessuna label. Inoltre, variazioni

non prevedibili dell'ambiente limitano l'utilizzo di dataset rappresentativi pre-costruiti.

Le tecniche più popolari come le **Support Vector Machines** (SVM) spesso necessitano di una creazione "manuale" delle features basata su campioni di con una finestra di acquisizione fissata per classificare "**time series**". Un problema con i microcontrollori è che la computazione delle features richiede un sostanziale ammontare di tempo per essere processate, impattando il consumo di energia.

Etichettare a mano il continuo flusso di dati che arriva dai sensori integrati non è sempre possibile, a meno che non si registrino sistematicamente i dati su un server remoto per etichettarli manualmente in seguito. Le tecniche di apprendimento supervisionate possono quindi risultare limitate. In aggiunta, quando la "Human Activity Recognition" viene performata da smartphones o dispositivi indossabili, il dispositivo viene spesso indossato da una persona soltanto durante il suo ciclo vitale. A causa di una grossa variazione tra diversi utenti, si può adattare il modello allenato al comportamento delle persone aggiungendo i dati acquisiti ogni volta a quelli precedenti e allenando nuovamente il modello.

In primo luogo questo lavoro presenterà un confronto tra alcune delle tecniche **supervised** presenti in letteratura sul dataset **UCI** [8], in particolare il confronto tra **Convolutional Neural Network** e **Long Short Term Memory Neural Network** (riprendendo [9]) e tecniche che uniscono le due già citate:

- **LSTM** - Long Short Term Memory: sono un tipo di Recurrent Neural Network capace di apprendere e ricordare attraverso lunghe sequenze di dati in input. Sono state create per usare quei dati che sono composti da lunghe sequenze di dati, da 200 a 400 time steps;

- **CNN** - Convolutional Neural Network: sono un tipo specializzato di neural networks che utilizzano un'operazione matematica chiamata **convoluzione** invece della classica moltiplicazione tra matrici in

almeno uno dei suoi livelli. Progettate specificatamente per elaborare pixel e in particolare elaborazione e riconoscimento di immagini;

- **CNN-LSTM**: CNN utilizzata per l'estrazione delle features dai dati in input LSTM per la sequence prediction. Create per predire sequenze temporali visuali e generare descrizioni testuali partendo da sequenze di immagini, più nello specifico, **activity**, **image** e **video** recognition.
- **CONVLSTM2D**: estensione della precedente, con la differenza che le convoluzioni sono eseguite come parte della LSTM.

Sempre sullo stesso dataset, poi, si è passati allo studio di tecniche **unsupervised** come **Self Organizing Map** [13] e **KMeans** [5].
Per lo studio di queste tecniche non è stato possibile utilizzare le sequenze temporali grezze come input, è stato necessario ricorrere all'utilizzo di altri dati in quanto le tecniche utilizzate, per cercare di mantenere i modelli "leggeri", non sono in grado di riconoscere sequenze temporali di dati.
Nel dataset UCI non sono presenti solo i dati temporali grezzi, ma ci sono:

1. Dati temporali grezzi;
2. Metriche statistiche calcolate sui dati temporali grezzi, queste statistiche sono da dividere in due parti:
   (a) statistiche calcolate sulle sequenze temporali;
   (b) statistiche calcolate sull'applicazione della Fast Fourier Transform (FFT).

Per l'analisi unsupervised è stato preso in considerazione il primo insieme di dati, essendo composto da molte features (più di 500 se si considerano anche le features derivate dall'applicazione della FFT dai dati grezzi, quelle prese in considerazione da noi sono 265) si è cercato di svolgere un lavoro di discriminazione delle features e di analisi del comportamento dei modelli testati al variare del numero delle features dovuto ai risultati dell'analisi statistica della varianza **ANOVA** per classe su ogni feature [15].

In seguito all'analisi delle performance ottenute sia processando le features con l'analisi ANOVA che non, si è cercato di implementare l'algoritmo che ha fornito le performance migliori all'interno di un **M5Stack Gray**, un dispositivo embedded (IoT) che monta un chip ESP-32 avente le seguenti caratteristiche:

- processore dual core 240MHz;
- memoria: 320Kb;
- modulo wi-fi integrato.

Il dispositivo è inoltre dotato di sensori **IMU** (accelerometro e giroscopio) che hanno permesso poi di procedere alla creazione di un dataset direttamente utilizzando il dispositivo stesso (lo stesso che poi verrà utilizzato per la fase ti testing).

Il resto del lavoro è organizzato nel modo seguente. Nel secondo capitolo sarà presentato lo stato dell'arte di quanto è stato studiato in questo lavoro, il capitolo tre mostrerà la mappa concettuale del "workflow" e i dettagli del dataset da noi costruito.
Nel quarto capitolo descriverà nel dettaglio il lavoro stand alone, l'analisi ANOVA e il lavoro svolto sul dispositivo embedded.
Il capitolo cinque è il capitolo dove verranno messi in mostra e discussi i risultati ed infine nel sesto ed ultimo capitolo verranno tirate le conclusioni e suggeriti sviluppi futuri per questo lavoro.

Capitolo 2

Stato dell'arte

In questo capitolo verrà descritto lo stato dell'arte attuale per le tecniche utilizzate e gli argomenti discussi all'interno dell'elaborato.

2.1 Supervised learning vs Unsupervised learning

Deep Neural Networks come le **Convolutional Neural Network** spesso si affidano al supervised learning sfruttando una coppia formata da input di allenamento (dati) e output desiderato (labels). È tipicamente richiesto di etichettare manualmente un grande volume di dati per costruire un modello accurato.
D'altro canto, le tecniche di unsupervised learning permettono di scoprire pattern ricorrenti in un dataset su cui non è stato eseguito nessun processo di etichettatura.
Di base, gli algoritmi usati all'interno delle tecniche unsupervised si basano su due approcci differenti: **probability-based self organization** e **cluster analysis**. Entrambi i metodi risolvono un problema di clustering raggruppando i dati che condividono pattern o attributi comuni.
Generalmente, i metodi unsupervised learning sono più flessibili di quelli supervised learning [11]. Infatti, i problemi di classificazione possono ancora

essere risolti anche se solo un sottoinsieme ridotto di etichette viene utilizzato per la fase finale di etichettatura: fino al 20% per il riconoscimento dell'attività [1] o anche solo all'1% per il riconoscimento di numeri scritti a mano [12].

2.1.1 Supervised learning con Convolutional Neural Networks

Una "Deep Neural Network" consiste in un **layer** di input e uno di output, connessi attraverso uno o più layers nascosti che possono essere a loro volta **convolutional**, **pooling** o **fully-connected** layers. **ANN** che fanno uso esclusivamente di **fully-connected** layers connessi in successione sono chiamate **multi-layer perceptrons**, mentre reti che fanno uso di **convolutional** layers sono dette **convolutional neural networks**. Un convolutional layer esegue una convoluzione tra l'output del layer precedente e un "kernel" (o filtro).
Convolutional layers sono spesso usati anche per l'estrazione automatica di features. I layer di Pooling tipicamente seguono quelli convolutional per ridurre la dimensione della mappa delle features. Il processo di pooling consiste nel calcolare o la media o il massimo di un insieme di valori derivati dall'output del layer precedente. In un layer di tipo fully-connected invece, ogni neurone è connesso a tutti i neuroni del livello precedente, eseguendo così un prodotto matrice-vettore tra i pesi delle sinapsi e l'output del livello precedente. Il layer finale è il layer di output che tipicamente è di tipo fully-connected, con i suoi output che sono le attività di ogni classe.

Durante la fase di apprendimento, il processo di backpropagation aggiusta i parametri allenabili (pesi e bias) di ogni layer successivamente. L'obiettivo del backpropagation è di ridurre la perdita (**loss**) calcolata tra la verità (es. labels) e la predizione.

2.1.2 Supervised learning con LSTM

Le **Recurrent Neural Networks**, di cui le LSTM (unità "long short-term memory") sono il sottoinsieme più potente e meglio conosciuto, sono un tipo di ANN progettate per riconoscere pattern in sequenze di dati, come dati **time series** ricavati da sensori, scrittura a mano, riconoscimento vocale, ecc. Ciò che differenzia RNN e LSTM dalle altre nerual networks è il fatto che tengono conto del tempo e della sequenza dei dati, in sostanza hanno una dimensione temporale.

Per comprendere meglio le RNN bisogna prima guardare alle fondamenta delle **reti feedforward**. Entrambe queste reti prendono il nome dal modo in cui incanalano le informazioni attraverso una serie di operazioni matematiche eseguite nei nodi della rete. La prima alimenta le informazioni direttamente (senza mai toccare un dato nodo due volte), mentre la seconda fa **scorrere** l'input attraverso un ciclo, e queste ultime vengono chiamate **recurrent** (ricorrenti).
In sostanza una feedforward network non ha nessuna nozione di ordinamento temporale, considera esclusivamente l'input corrente.

Le recurrent networks, d'altro canto, prendono in input non solo l'input corrente che stanno valutando, ma anche cosa hanno valutato precedentemente nel tempo.

Di seguito un semplice diagramma di esempio, dove l'input *BTSXPE* in basso rappresenta l'input corrente, e *CONTEXT UNITS* rappresenta l'output dell'istante precedente.

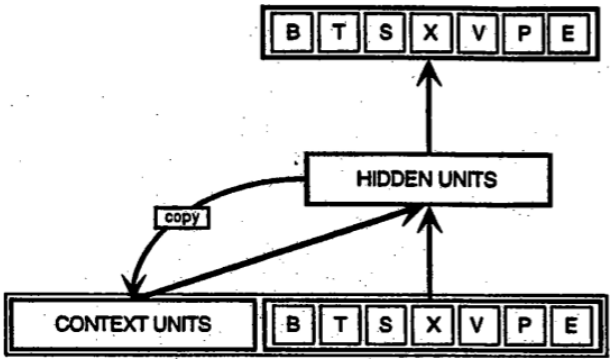


Figura 2.1: Esempio di input per RNN

La decisione che una rete recurrent ha preso all'istante $t-1$ influenza la decisione che prenderà all'istante t. Quindi le RNN hanno l'input composto da due parti:

1. il presente;
2. il passato recente;

queste due parti verranno poi combinate per determinare come rispondere a nuovi input.
Le RNN si distinguono dalle reti feedforward per quel ciclo di feedback connesso alle loro decisioni passate, "riciclando" i propri output momento dopo momento come input. L'aggiunta di memoria alle reti neurali ha uno scopo: ci sono informazioni nella sequenza stessa e le reti ricorrenti le usano per eseguire attività che le reti feedforward non possono.

A metà degli anni 90, una variante di RNN chiamata **Long Short-Term Memory units** o LSTM, è stata proposta dal ricercatore tedesco Sepp Hochreiter and Juergen Schmidhuber come soluzione al problema del gradiente evanescente [10]. Questa nuova tecnica aiuta a preservare l'errore che potrebbe essere propagato all'indietro attraverso tempo e layers. Riuscendo a mantenere un errore più costante, permetto alle RNN di continuare ad imparare anche con molti time steps (oltre 1000), aprendo così un canale per collegare cause ed effetti a distanza.

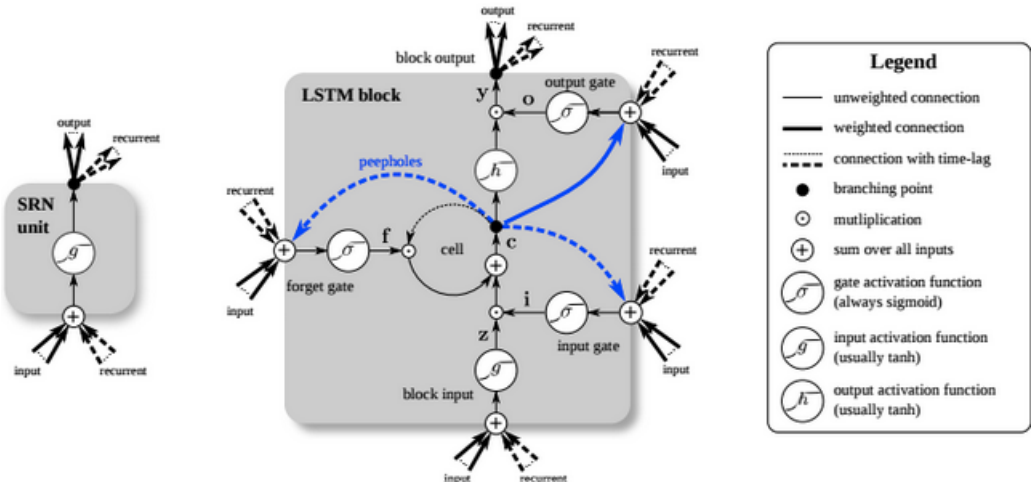


Figura 2.2: cella di una RNN (sinistra), cella di una LSTM (destra)

2.1.3 Unsupervised learning

Unsupervised learning è un metodo utile per scovare potenziali pattern nascosti in un dataset, permettendo così la classificazione di dati non etichettati. Permette inoltre analisi esplorativa dei dati e clustering per comprendere relazioni tra i pattern.
Ci sono differenti metodi che ottengono vantaggi sfruttando unsupervised learning insieme alle Artificial Neural Networks: autoencoders, modelli generativi, memorie associative e self-organizing maps. Questi metodi possono essere usati per varie applicazioni con classificazione di time series o predizione di dati [14]. Ovviamente questi metodi non possono avere performance equivalenti ai metodi supervised. Ma l'assunzione di avere sempre a disposizione le labels diventa incredibilmente difficile da soddisfare in applicazioni realistiche, è quindi rilevante valutare in che misura gli approcci non supervisionati possono essere utilizzati nel contesto dei dispositivi wearable.

Self-Organizing Maps

Le Self-Organizing Maps [13] consentono una riduzione di uno spazio multidimensionale in uno spazio bidimensionale (cioè una griglia).

La griglia consiste in diversi neuroni collegati tra loro da una relazione di vicinato relativo. I neuroni hanno un vettore dei pesi della stessa dimensione dello spazio di osservazione.
I pesi dei neuroni si evolvono progressivamente nel tempo secondo l'algoritmo di apprendimento. Quando l'apprendimento è stabilizzato, l'organizzazione dei neuroni riflette l'organizzazione vettoriale del spazio di input così come la densità di probabilità dei dati di input. In effetti, due dati simili nello spazio di osservazione corrispondono a regioni vicine sulla mappa.
Utilizzando le self-organizing maps, è possibile utilizzare campioni non etichettati per l'addestramento (al contrario dell'apprendimento supervisionato). Infine, è possibile eseguire una classificazione dopo aver identificato gruppi di neuroni che rappresentano lo stesso pattern.
Inizialmente, tutti i pesi dei neuroni vengono inizializzati in modo casuale. Un vettore di input viene presentato alla mappa. La distanza tra ciascun neurone e il vettore di input viene calcolata in base alla distanza euclidea tra entrambi i vettori. Il neurone più vicino al vettore di input è chiamato **Best Matching Unit** (**BMU**). Corrisponde al neurone più rappresentativo dei dati di input.
Il quartiere della BMU (Fig. 2.3) è definito dalla distanza di Manhattan sulla mappa tra ciascun neurone e l'unità di corrispondenza migliore.
Per una mappa 2D, la distanza di Manhattan tra due punti p_i e p_j, con rispettive coordinate (X_{p_i}, Y_{p_i}) e (X_{p_j}, Y_{p_j}) è definita da:

$$d_m(p_i, p_j) = |X_{p_j} - X_{p_i}| + |Y_{p_j} - Y_{p_i}| \tag{2.1}$$

Tutti i neuroni presenti presenti in questo vicinato aggiorneranno i loro pesi in modo da essere più vicini al vettore di input.
I pesi w_i di ogni neurone i vengono mossi attraverso il vettore di input v secondo la seguente regola di apprendimento [19]:

$$\Delta w_i = \epsilon(t)h(t,i,s)(v - w_i) \tag{2.2}$$

Dove s è la BMU e h è una funzione di vicinato della forma:

$$h(t,i,s) = e^{-\frac{d_m(p_i,p_s)^2}{2\theta(t)^2}} \tag{2.3}$$

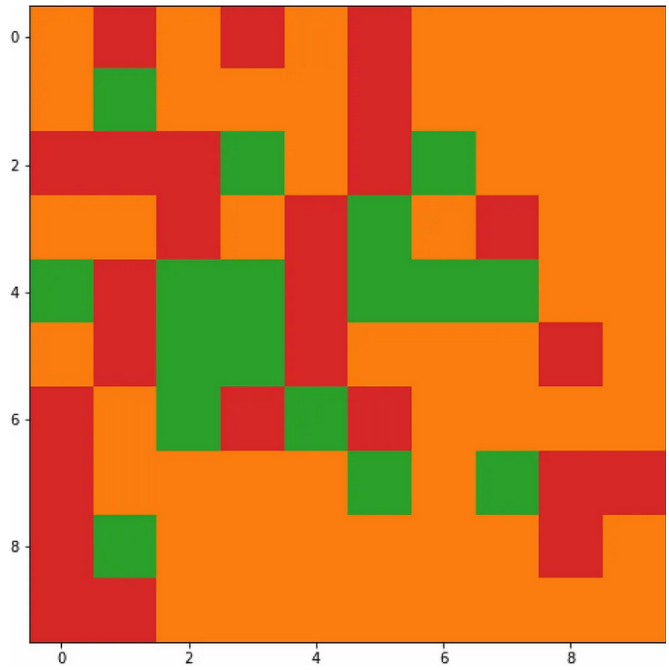
Con $\epsilon(t)$ il learning rate e $\theta(t)$ la dimensione del vicinato definiti come:

$$\theta(t) = \theta_i * \left(\frac{\theta_f}{\theta_i}\right)^{t/t_f} \tag{2.4}$$

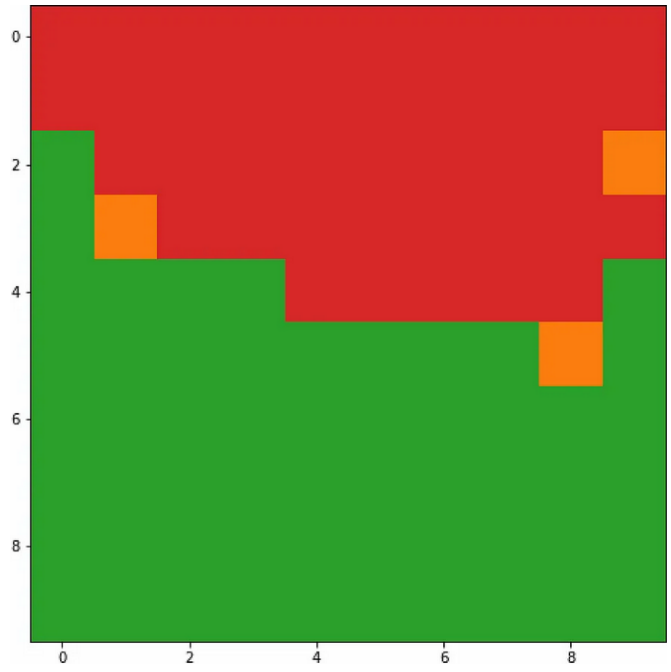
$$\epsilon(t) = \epsilon_i * \left(\frac{\epsilon_f}{\epsilon_i}\right)^{t/t_f} \tag{2.5}$$

Con θ_i e θ_f la dimensione del vicinato iniziale e finalem ϵ_i e ϵ_f il learning rate iniziale e finale. La funzione di apprendimento itera tra tempo $t = 0$ e $t = t_f$.

Le self-organizing maps sono in grado di rappresentare l'evoluzione temporale di un sistema. Per fare ciò, la mappa apprende sequenze temporali che sono insiemi di vettori a dimensione finita e ordinati nel tempo.
Ogni vettore rappresenta quindi un'evoluzione del sistema e la mappa descrive l'organizzazione dei diversi pattern che il sistema ha adottato.
Questi insiemi di serie temporali sono spesso la base di apprendimento per la maggior parte dei modelli di self-organizing maps. In [4], [16] e [6], gli autori propongono uno studio di diversi modelli e le loro applicazioni nell'ambito del riconoscimento di sequenze temporali.



(a) Neuroni della SOM alla prima iterazione



(b) Neuroni della SOM dopo l'iterazione finale

Figura 2.3: Situazione iniziale e finale della SOM

K-means

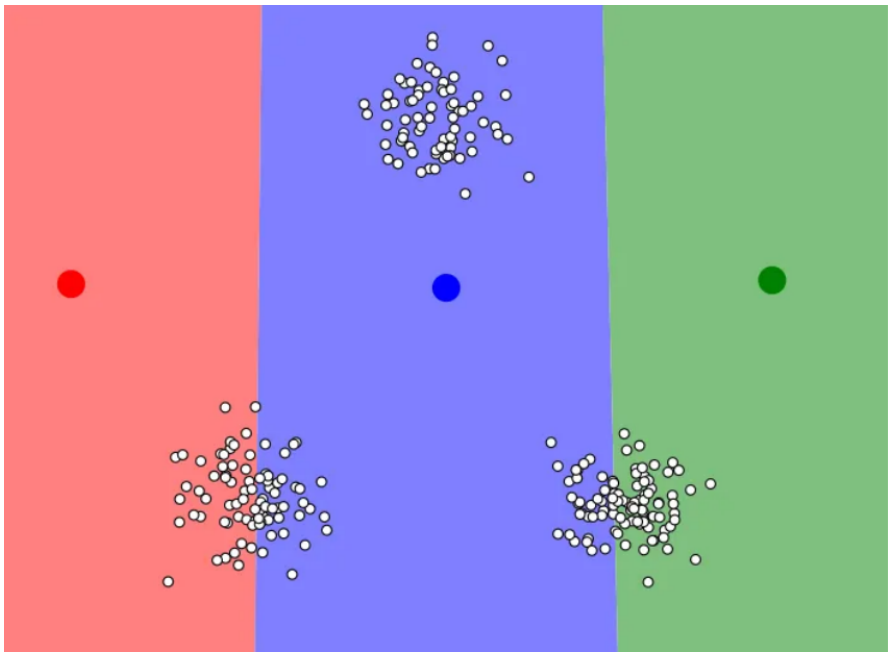
K-means è uno degli algoritmi unsupervised appartenenti al machine learning più semplici e popolari.
Generalmente, gli algoritmi unsupervised eseguono l'inferenza sui dati appartenenti al dataset di input senza fare riferimento a risultati noti o etichette. Questo algoritmo punta a risolvere la problematica di scovare e raggruppare dati simili e scoprirne modelli/pattern comuni sottostanti. Per raggiungere questo obiettivo, k-means cerca un numero prefissato (k) di cluster in un dataset.

Serve quindi definire un numero k, che fa riferimento al numero di **centroidi** necessari nel dataset. Un centroide è la posizione immaginaria o reale che rappresenta il centro del cluster.
L'assegnamento di ogni punto ad un cluster avviene calcolando la distanza di quel punto da ogni centroide, il cluster assegnato sarà determinato dal centroide più vicino. Solitamente per calcolare la distanza tra due punti viene usata la distanza **euclidea**.

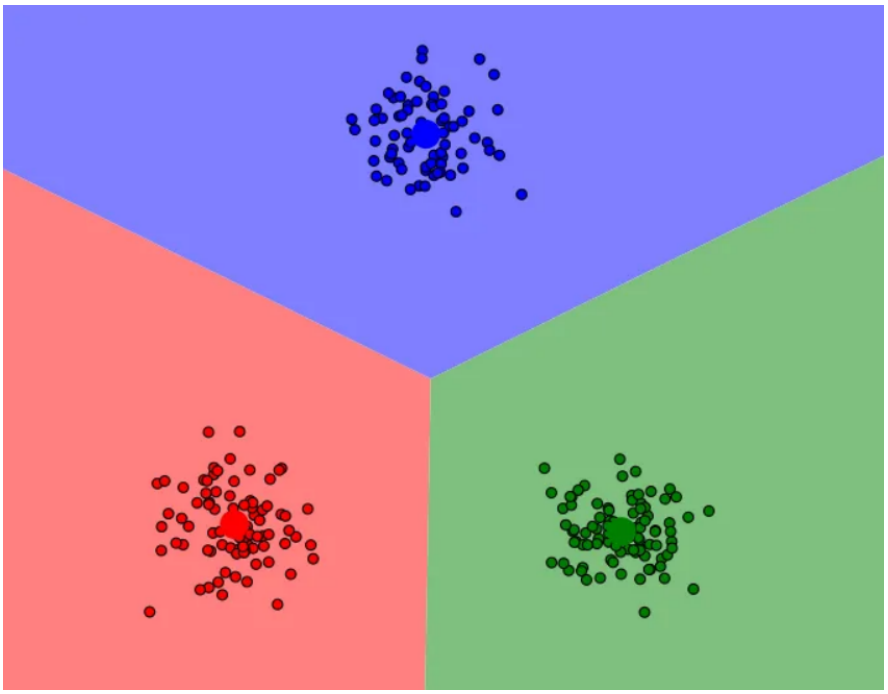
Il termine "mean" nel nome fa riferimento al calcolo delle medie dei dati, ovvero dei centroidi.

Per processare i dati di allenamento, k-means inizia con un primo gruppo di centroidi scelti in modo casuale (Fig. 2.4a), che vengono usati come punto di partenza per ogni cluster, e in seguito eseguire iterativamente calcoli per ottimizzare la posizione dei centroidi (Fig. 2.4b).
Il processo termina quando:

- i centroidi si sono stabilizzati: i valori dei centroidi alle iterazioni successive non cambiano più perchè il clustering ha avuto successo;
- il numero di iterazioni prefissato è stato raggiunto.



(a) Inizializzazione random dei centroidi di K-means



(b) Clustering finale di K-means

Figura 2.4: Situazione iniziale e finale di K-means

2.2 Human Activity Recognition

La **Human Activity Recognition** è un vasto campo di ricerca che si basa su informazioni e modelli biomedici realistici.
I dataset sono composti da acquisizioni provenienti da sensori o video-based per controlli gestuali, interazione uomo-robot o monitoraggio della salute.
Nella letteratura appartenente alla **Human Activity Recognition**, approcci unsupervised sono già stati valutati diverse volte [3], la maggior parte delle volte con metodi classici tra i quali K-means il più popolare.
Gli approcci bio-ispirati come la **Self-Organizing Map** sono rimasti inesplorati per quanto riguarda la **HAR** con sensori indossati, la maggior parte dei lavori precedenti riguardavano i dati video [17].

2.3 UCI HAR Dataset

La **University of California Irvine** (**UCI**) ospita un dataset per la human activity recognition [8] ampiamente utilizzato in numerose pubblicazioni.
Questo dataset contiene informazioni di movimento campionate da **accelerometro** e **giroscopio** di uno smartphone.
Multipli metodi supervised e unsupervised sono stati valutati usando il dataset UCI HAR con due tipi di dati in input:

1. dati grezzi;
2. features pre-calcolate.

La grande maggioranza dei metodi utilizza le features pre-calcolate che necessitano un pre-processing del segnale.
Nonostante forniscano una migliore accuratezza di classificazione, queste features pre-calcolate sono spesso generate utilizzando **Fast Fourier Transforms** (**FFT**) computazionalmente molto costose.

2.3.1 Analisi statistica ANOVA

Spesso quando si parla di analisi della varianza **ANOVA**, si pensa alla più comune delle tecniche appartenenti al "gruppo" delle analisi ANOVA, la **ANOVA One-way**.

L'analisi della varianza (ANOVA) one-way è un metodo statistico per testare le differenze tra le medie di tre o più gruppi. In genere viene usata quando si ha un'unica variabile indipendente, o fattore, e si vuole verificare se eventuali variazioni o diversi livelli di tale fattore abbiano un effetto misurabile su una variabile dipendente.

L'**ANOVA one-way** è applicabile solo in presenza di un singolo fattore e di una singola variabile dipendente. Nel confrontare le medie di tre o più gruppi, può dirci se almeno una coppia di medie presenta differenze significative, ma non è in grado di identificare di quale coppia si tratti. Inoltre, per funzionare bisogna che la variabile dipendente sia normalmente distribuita in ciascuno dei gruppi e che la variabilità all'interno del gruppo sia simile per tutti i gruppi.

In questo lavoro è stata applicata una variante che ci permette di far fronte ai limiti che l'ANOVA one-way ci pone davanti. Questa variante è la **ANOVA F** [15]: il valore ANOVA F di una feature è uguale alla varianza quadratica media dei valori della feature tra classi diverse divisa per la varianza quadratica media all'interno della stessa classe. Pertanto un valore F elevato indica che la rispettiva caratteristica può discriminare bene classi diverse.

2.4 HAR su micro-controllore

Come già descritto in [18], riuscire a far "sbarcare" l'intelligenza artificiale su dispositivi integrati (es. micro-controllori) è diventato un campo di ricerca centrale in vari domini scientifici.

Tecniche come Artificial Neural Networks si sono rivelate essere molto performanti in particolare per quanto riguarda la **Human Activity Recognition**, tuttavia queste tecniche in primo luogo spesso risultano essere molto costose in termini di consumo di risorse (elettricità, potenza di calcolo, memoria...) ed inoltre spesso vengono allenate su dati supervisionati, che come già detto in precedenza, in scenari realistici sono molto difficili da reperire.

Già alcune tecniche unsupervised per far fronte al problema del labelling sono state prese in analisi, come **KNN** [7]. Tuttavia, anche se risulta essere un ottima tecnica, non è stata testata direttamente su un dispositivo embedded e il pre-processing dei dati risulta essere complesso e costoso.

Per quanto riguarda il costo computazionale spesso si pensa che il dispositivo in questione si occupi semplicemente di raccogliere i dati ma non è così. In [9] viene mostrato che grazie a tecnologie come **Tensorflow Lite** è possibile ridurre notevolmente il peso e la dimensione dei modelli supervised rendendone così possibile l'esecuzione sui dispositivi embedded.

Capitolo 3

Mappa concettuale e dataset costruito

Nel seguente capitolo mostreremo il flusso di esecuzione di tutto l'esperimento e come è stato acquisito il dataset.

3.1 Mappa concettuale

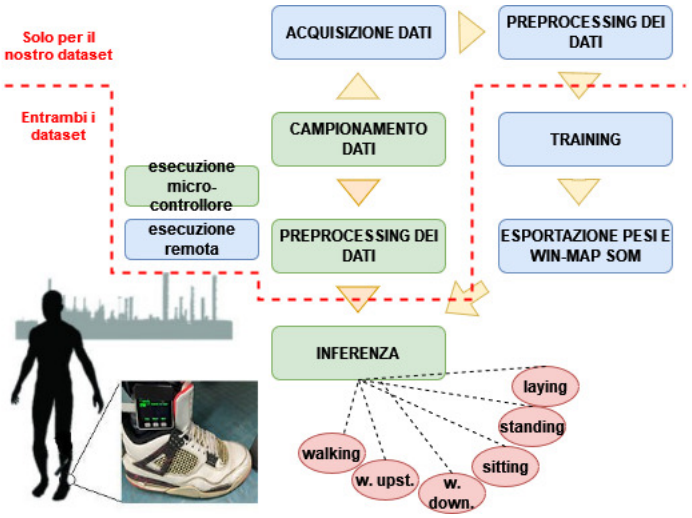


Figura 3.1: Flusso di esecuzione dell'esperimento

Nel diagramma in Fig. 3.1 viene mostrata la sequenza delle fasi che compongono l'esperimento. Per quanto riguarda la costruzione del dataset, il training dei modelli e l'esportazione della SOM (ramo blu Fig. 3.1):

1. campionamento dei segnali tramite i sensori IMU del dispositivo M5Stack Gray e salvataggio in formato .txt sulla microSD del dispositivo;
2. acquisizione dei dati su PC;
3. preprocessing [8]: i segnali grezzi raccolti vengono prima filtrati da eventuali rumori, in seguito viene filtrata l'accelerazione di gravità dai segnali dell'accelerometro, e poi vengono calcolate le 265 features concludendo così la costruzione del dataset;
4. training e testing dei modelli da testare;
5. esportazione dei pesi e della mappa neurone-label della SOM sulla microSD che andrà sul dispositivo.

Una volta conclusa questa fase di preparazione, sarà possibile eseguire l'inferenza della SOM completamente sul dispositivo M5Stack (compreso il calcolo delle features dai segnali grezzi) che comprende le seguenti fasi:

1. campionamento del segnale tramite i sensori IMU del dispositivo M5Stack Gray;
2. preprocessing del singolo segnale sul quale fare l'inferenza;
3. esecuzione dell'inferenza e output della classificazione.

3.2 Dataset costruito

L'esperimento di costruzione del dataset per il momento è stato eseguito da un singolo soggetto che ha eseguito sei azioni diverse:

1. camminare (walking);
2. salire le scale (walking upstairs);
3. scendere le scale (walking downstairs);
4. stare seduto (sitting);
5. stare in piedi (standing);
6. stare sdraiato (laying);

indossando alla caviglia un dispositivo embedded M5Stack Gray dotato di sensori IMU. La scelta della posizione di collocamento del sensore deriva da quanto mostrato in [2], nel quale viene descritto che le tre posizioni migliori nel piede per riconoscere il movimento sono:

1. primo cuneiforme (circa sul petto del piede);
2. tendine di Achille;
3. astragalo (circa caviglia).

per motivi anche legati all'ingombro del dispositivo di acquisizione la posizione scelta è stata la caviglia.
Utilizzando accelerometro e giroscopio integrati, sono stati catturati:

1. accelerazione lineare su 3 assi;
2. velocità angolare su 3 assi;

ad una frequenza costante di 41,29Hz. L'etichettatura dei dati è affidata al dispositivo stesso, nel quale attraverso un'interfaccia grafica, resa possibile grazie al display integrato, l'utente prima di iniziare il campionamento dei

segnali può scegliere per quale attività acquisire il segnale. Ogni record nel dataset sarà composto da 128 campionamenti in una finestra di 3,1 secondi. Al termine del processo di creazione delle features il dataset è stato diviso in maniera casuale in train e test set con rapporto 70/30.

I segnali provenienti dai sensori (accelerometro e giroscopio) sono stati pre-processati applicando filtri per eliminare il rumore. Il segnale del sensore di accelerazione, che ha due componenti: gravitazionale e di movimento del corpo, è stato sottoposto ad un processo di separazione di queste due componenti assumendo che la forza di gravità abbia solo componenti a bassa frequenza. Quindi per separare le due componenti è stato usato un **Butterworth low-pass filter** con una fequenza di taglio pari a 0,3Hz. Da ogni finestra di 128 campioni è stato ottenuto poi un vettore di features calcolando variabili dal dominio temporale.

Per ogni record viene fornito:

- accelerazione totale e stima dell'accelerazione del corpo tri-assiale proveniente dall'accelerometro;
- velocità angolare tri-assiale proveniente dal giroscopio;
- un vettore di 265 features con variabili appartenenti al dominio temporale, tra cui:
  - **jerk value**: valore che indica la frequenza con cui cambia l'accelerazione del segnale nel tempo;
  - **magnitude**: misura dell'ampiezza o forza complessiva del segnale;
  - **media**;
  - **deviazione standard**;
  - **median absolute deviation**;
  - **massimo**;
  - **minimo**;
  - **signal magnitude area**;

  - **energy**: l'energia di un segnale è una misura della potenza totale contenuta all'interno del segnale in un dato intervallo di tempo;
  - **inter-quartile range**: l'IQR è una misura della diffusione o della variabilità di un set di dati;
  - **entropy**: l'entropia è una misura dell'incertezza o casualità di un segnale;
  - **auto regression coefficients**: i coefficienti di auto-regressione sono comunemente usati nell'elaborazione dei segnali per modellare un segnale come una combinazione lineare dei suoi valori passati;
  - **correlation**: la correlazione è una misura della somiglianza o del grado di associazione tra due segnali;
- la rispettiva label.

Il dataset comprende i seguenti files:

- **README.txt**;
- **features_info.txt**: informazioni riguardo alle variabili usate nel vettore delle features;
- **features.txt**: lista di tutte le features;
- **activity_labels.txt**: mappa tra le etichette delle classi e il rispettivo nome;
- **train/X_train.txt**: training set;
- **train/y_train.txt**: training labels;
- **test/X_test.txt**: test set;
- **test/y_test.txt**: test labels;

I seguenti files sono disponibili sia per il training set che per il test set. Le loro descrizioni sono equivalenti:

- **train/Intertial Signals/total_acc_x_train.txt**: il segnale dell'accelerazione dall'accelerometro dell'M5Stack riguardante l'asse X in unità di gravità standard "g". Ogni riga mostra un vettore di 128 elementi. La stessa descrizione si applica a **total_acc_y_train.txt** e **total_acc_z_train.txt** rispettivamente per gli assi Y e Z;

- **train/Intertial Signals/body_acc_x_train.txt**: il segnale di accelerazione del corpo ottenuto sottraendo la gravità dall'accelerazione totale.

- **train/Intertial Signals/body_gyro_x_train.txt**: il vettore di velocità angolare misurato dal giroscopio per ogni finestra di campionamento. Le unità sono radianti/secondo.

Capitolo 4

Implementazione

Fino ad ora è stata illustrata prettamente la parte teorica dell'esperimento svolto, in questo capitolo inizieremo a discutere della parte tecnica, ovvero di come il tutto è stato implementato.
Il lavoro di implementazione può essere suddiviso in 3 parti:

1. **implementazione offline**: in questa sezione si discuterà del lavoro riguardante prettamente l'esecuzione su PC, ovvero:

   - implementazione dei modelli supervised;
   - implementazione dei modelli unsupervised;
   - confronto performance tra modelli supervised e unsupervised;
   - acquisizione dati campionati e creazione del dataset;
   - esportazione del modello SOM allenato.

   Il lavoro svolto per questa sezione è stato svolto in Python.

2. **lavoro statistico ANOVA (dimensionality reduction)**: in questa sezione si discuterà dell'analisi **ANOVA** effettuata per classe su ognuna delle 265 features calcolate. L'obiettivo di questa analisi è quello di determinare quali delle 265 features usare e se al variare del numero delle features in base al valore ANOVA scelto cambiano anche le performance delle tecniche testate.

Anche per quanto riguarda questa sezione il lavoro è stato svolto in Python.

3. **implementazione per micro-controllore**: in questa sezione verrà discusso il lavoro di implementazione svolto per il micro-controllore.
   L'implementazione consiste di due parti principali: la prima è la parte riguardante il campionamento dei dati e il salvataggio su microSD (dati che verranno poi usati per la creazione del dataset), la seconda è la parte riguardante l'implementazione dell'inferenza della **Self-Organizing Map**.
   Tutta la parte di lavoro riguardante l'implementazione per l'M5Stack è stata fatta in C++.

4.1 Implementazione offline

Per quanto riguarda questa fase, che coincide anche con la prima fase svolta del progetto, iniziamo descrivendo prima l'implementazione dei modelli supervised.
Iniziamo innanzitutto descrivendo la struttura del progetto:

- **UCI-HAR Dataset**: directory contenente il dataset UCI [8];
- **Dataset tesi**: directory contenente il dataset costruito usando il dispositivo M5Stack Gray;
- **plots**: directory che contiene tutti i grafici generati durante l'esecuzione dell'esperimento sul dataset da noi costruto;
- **plots UCI**: directory che contiene tutti i grafici generati durante l'esecuzione dell'esperimento sul dataset UCI;
- **np_arr**: directory contente alcune strutture dati salvante in formato **.npy** utili per l'esportazione dei modelli.
  Oltre i .npy è possibile trovare, in particolare per quanto riguarda la

self-organizing map, due files **.txt** che sono quelli che verranno poi esportati effettivamente sul micro-controllore e sono:

  - **map_lst... .txt**: contiene l'associazione neurone-label per i parametri descritti nel nome del file;
  - **weights_lst... .txt**: contiene i pesi del modello da esportare.

  Anche per questa directory ne esiste una versione per il dataset UCI;

- **som_models**: directory contenente modelli salvati già allenati per quanto riguarda le tecniche unsupervised.
  Anche per questa directory ne esiste una versione per il dataset UCI;

- **weights**: directory che contiene i pesi salvati dalle tecniche supervised;

- **analize_data.py**: script Python che si occupa di leggere i dati acquisiti dal micro-controllore e procede alla creazione del dataset;

- **main.py** script Python che consiste nel nucleo dell'esperimento. Al suo interno è contenuto il codice relativo al training e al testing di tutti i modelli e il codice riguardante l'espotazione dei file necessari all'esecuzione su dispositivo edge.
  Accetta i seguenti parametri in input (in ordine):

  1. "**s**" o "**u**" rispettivamente per eseguire l'analisi **supervised** o **unsupervised**.
  2. "**bal**" o "**no-bal**" rispettivamente per eseguire lo studio su dati **bilanciati** o **sbilanciati**;
  3. "**our**" o "**uci**" rispettivamente per eseguire l'analisi sul **nostro** dataset o sul dataset **uci**.
     IMPORTANTE: se si esegue l'analisi supervised questo va inserito come terzo parametro, altrimenti come quinto;
  4. "**kmeans**" o "**som**" rispettivamente per eseguire l'analisi unsupervised con **K-Means** o **SOM**;

5. "**avg**", "**min**" o "**avgmin**" rispettivamente, per l'analisi ANOVA, per fare solo l'analisi scegliendo la **media** delle varianze per classe, solo scegliendo il **minimo** delle varianze per classe, o **entrambe** le analisi;
6. "**our**" o "**uci**" da inserire ora se il primo parametro è "**u**";
7. "**y**" o "**n**" rispettivamente per dire al programma se deve o meno salvare, durante l'esecuzione, tutti i plots e le strutture dati utili da esportare.

4.1.1 Implementazione modelli supervised

I modelli supervised che sono stati usati sono stati implementati utilizzando **Keras**. Keras è una libreria open source per il machine learnine e le neural networks scritta in Python.
È progettata come un'interfaccia a un livello di astrazione superiore di altre librerie simili di più basso livello, e supporta come back-end le librerie **TensorFlow**, Microsoft Cognitive Toolkit (CNTK) e Theano.
Progettata per permettere una rapida prototipazione di deep neural networks, si concentra sulla facilità d'uso, la modularità e l'estensibilità. È stata sviluppata come parte del progetto di ricerca **ONEIROS**, e il suo autore principale è **François Chollet**, di Google.

```
# fit and evaluate lstm model
def evaluate_model(X_train, y_train, X_test, y_test):
    verbose, epochs, batch_size = 1, 50, 64
    n_timesteps, n_features, n_outputs = X_train.shape[1],
   ...
    model = Sequential()
    model.add(LSTM(100, input_shape=(n_timesteps, n_features)
   ))
    model.add(Dropout(0.1))
    model.add(Dense(100, activation='relu'))
    model.add(Dense(n_outputs, activation='softmax'))
    earlystop = EarlyStopping(...)
    checkpoint = ModelCheckpoint(...)
    model.compile(...)
    ...
    model.fit(...)
    ...
    # evaluate model
    _, accuracy = model.evaluate(X_test, y_test, batch_size=
   batch_size, verbose=1)

    return accuracy
```

Listing 4.1: Definizione modello LSTM

```
# fit and evaluate cnn model
def evaluate_cnn_model(X_train, y_train, X_test, y_test):
    verbose, epochs, batch_size = 1, 50, 64
    X_train = X_train.reshape((...))
    X_test = X_test.reshape((...))
    model = Sequential()
    model.add(Conv2D(16, (2, 2), activation='relu',
   input_shape=X_train[0].shape))
    model.add(Dropout(0.1))
    model.add(Conv2D(32, (2, 2), activation='relu'))
    model.add(Dropout(0.2))
    model.add(Flatten())
    model.add(Dense(64, activation='relu'))
    model.add(Dropout(0.5))
    model.add(Dense(6, activation='softmax'))
    earlystop = EarlyStopping(...)
    checkpoint = ModelCheckpoint(...)
    model.compile(...)
    # fit network
    ...
    model.fit(...)
    ...
    # evaluate model
    _, accuracy = model.evaluate(X_test, y_test, verbose=1)
    return accuracy
```

Listing 4.2: Definizione modello CNN

Il modello riportato di seguito, ovvero il modello CNN LSTM, leggerà le sotto-sequenze della sequenza principale come blocchi, estrarrà le caratteristiche da ciascun blocco, quindi consentirà a LSTM di interpretare le caratteristiche estratte da ciascun blocco.
Un approccio all'implementazione di questo modello consiste nel suddividere ogni finestra di 128 fasi temporali in sotto-sequenze che il modello CNN deve elaborare. Ad esempio, le 128 fasi temporali in ciascuna finestra possono essere suddivise in quattro sotto-sequenze di 32 fasi temporali.
Verrà quindi definito un modello CNN che prevede di leggere sequenze con

una lunghezza di 32 intervalli temporali e nove features. L'intero modello CNN può essere racchiuso in un livello TimeDistributed per consentire allo stesso modello CNN di leggere ciascuna delle quattro sotto-sequenze nella finestra. Le caratteristiche estratte vengono quindi appiattite e fornite al modello LSTM per la lettura, estraendo le proprie caratteristiche prima che venga effettuata una mappatura finale a un'attività.

```
# fit and evaluate cnn-lstm model
def evaluate_cnn_lstm_model(X_train, y_train, X_test, y_test):
    # define model
    verbose, epochs, batch_size = 1, 50, 64
    n_timesteps, n_features, n_outputs = X_train.shape[1], X_train.shape[2], y_train.shape[1]
    # reshape data into time steps of sub-sequences
    n_steps, n_length = 4, 32
    X_train = X_train.reshape((...))
    X_test = X_test.reshape((...))
    # define model
    model = Sequential()
    model.add(
        TimeDistributed(Conv1D(filters=64, kernel_size=3, activation='relu'), input_shape=(None, n_length, n_features)))
    model.add(TimeDistributed(Conv1D(filters=64, kernel_size=3, activation='relu')))
    model.add(TimeDistributed(Dropout(0.5)))
    model.add(TimeDistributed(MaxPooling1D(pool_size=2)))
    model.add(TimeDistributed(Flatten()))
    model.add(LSTM(100))
    model.add(Dropout(0.5))
    model.add(Dense(100, activation='relu'))
    model.add(Dense(n_outputs, activation='softmax'))
    earlystop = EarlyStopping(...)
    checkpoint = ModelCheckpoint(...)
    model.compile(...)
    ...
```

```
    # fit network
    model.fit(...)
    ...
    # evaluate model
    _, accuracy = model.evaluate(...)
    return accuracy
```

Listing 4.3: Definizione modello CNN-LSTM

Infine verrà mostrato l'ultimo dei 4 modelli, il **Convolutional LSTM** che a differenza di una LSTM che legge direttamente i dati per calcolare lo stato interno e le transizioni di stato, e diversamente dalla CNN-LSTM che interpreta l'output dai modelli CNN, la ConvLSTM utilizza le convoluzioni direttamente come parte della lettura dell'input nelle unità LSTM stesse.
La libreria Keras fornisce la classe ConvLSTM2D che supporta il modello ConvLSTM per i dati 2D. Può essere configurato per la classificazione di serie temporali multivariate 1D.
La classe ConvLSTM2D, per impostazione predefinita, prevede che i dati di input abbiano la forma: (**samples**, **time**, **rows**, **cols**, **channels**).
Dove ogni fase temporale dei dati è definita come un'immagine di (righe * colonne) punti dati.

Nella sezione precedente, abbiamo suddiviso una data finestra di dati (128 fasi temporali) in quattro sotto-sequenze di 32 fasi temporali. Possiamo utilizzare lo stesso approccio di sotto-sequenza per definire l'input ConvLSTM2D in cui il numero di passaggi temporali è il numero di sotto-sequenze nella finestra, il numero di righe è 1 poiché stiamo lavorando con dati unidimensionali e il numero di colonne rappresenta il numero di passaggi temporali nella sotto-sequenza, in questo caso 32.
Per questo inquadramento scelto del problema, l'input per ConvLSTM2D sarebbe quindi:

- **Samples**: n, per il numero di campioni nel dataset;
- **Time**: 4, per le sotto-sequenze in cui abbiamo diviso una finestra di 128 steps;

- **Rows**: 1, per la mono-dimensionalità di ogni sotto-sequenza;
- **Columns**: 32, per i 32 time steps in una sotto-sequenza di input;
- **Channels**: 9, per le 9 features in input (X-Y-Z dei 3 segnali).

```
# fit and evaluate convlstm2d model
def evaluate_convlstm2d_model(X_train, y_train, X_test, y_test):
    # define model
    verbose, epochs, batch_size = 1, 50, 64
    n_timesteps, n_features, n_outputs = X_train.shape[1], X_train.shape[2], y_train.shape[1]
    # reshape into subsequences (samples, time steps, rows, cols, channels)
    n_steps, n_length = 4, 32
    X_train = X_train.reshape((...))
    X_test = X_test.reshape((...))
    # define model
    model = Sequential()
    model.add(
        ConvLSTM2D(filters=64, kernel_size=(1, 3), activation='relu', input_shape=(n_steps, 1, n_length, n_features)))
    model.add(Dropout(0.5))
    model.add(Flatten())
    model.add(Dense(100, activation='relu'))
    model.add(Dense(n_outputs, activation='softmax'))
    earlystop = EarlyStopping(...)
    checkpoint = ModelCheckpoint(...)
    model.compile(...)
    ...
    # fit network
    model.fit(...)
    ...
    # evaluate model
    _, accuracy = model.evaluate(...)
    return accuracy
```

Listing 4.4: Definizione modello CONVLSTM2D

4.1.2 Implementazione modelli unsupervised

Come già anticipato per i modelli unsupervised sono stati implementati **K-Means** e una **Self-Organizing Map**.
Per quanto riguarda K-Means è stata usata l'implementazione fornita nella libreria **scikit-learn**, i dati presi in input sono stati prima sottoposti al processo di riduzione della dimensionalità con l'analisi **ANOVA** per classe sulle features, ed in seguito processate con **PCA** per ridurre tutto a due dimensioni.

```
# init pca and kmeans
pca = PCA(n_components=2)
kmeans = KMeans(init="k-means++", n_clusters=6)
...
# perform pca and fit k-means returning a list containing the
    cluster for each input
X_pca = pca.fit_transform(X_la)
X_pca = normalize(X_pca)
km_clusters = kmeans.fit_predict(X_pca)
```

Listing 4.5: Implementazione K-Means

Dopodiché, è stato necessario calcolare manualmente un punteggio di "accuracy" creando una associazione cluster-label e poi assegnare ad ogni cluster una label in base alla quale classe ha il maggior numero di punti in quel cluster, il tutto calcolato dalla seguente:

```
def cluster_assoc_calc(km_clusters, y):
    clusters_dict = {0: {0:0, 1:0, 2:0, 3:0, 4:0, 5:0},
                     1: {0:0, 1:0, 2:0, 3:0, 4:0, 5:0},
                     2: {0:0, 1:0, 2:0, 3:0, 4:0, 5:0},
                     3: {0:0, 1:0, 2:0, 3:0, 4:0, 5:0},
                     4: {0:0, 1:0, 2:0, 3:0, 4:0, 5:0},
                     5: {0:0, 1:0, 2:0, 3:0, 4:0, 5:0}}
    cluster_assoc_dict = {}
    ...
    while len(dup_lst) > 0:
        print("\nLabels presenti piu' di una volta\n")
        print(dup_lst)
```

```
13          cl_to_change = 0
14          for dup in dup_lst:
15              ...
16          ...
17      ...
18      for idx, val in enumerate(km_clusters):
19          if cluster_assoc_dict[val] == np.argmax(y[idx]):
20              count+=1
21      print("accuracy: " + str(count/X_pca.shape[0]))
```

Listing 4.6: Calcolo accuracy per k-means

Per quanto riguarda **K-Means** è stato detto tutto, passiamo ora alla descrizione dell'implementazione della **Self-Organizing Map**.
Per la SOM è stata utilizzata la libreria Python **minisom** [20].
MiniSom è un'implementazione minimalistica e basata su **NumPy** della **SOM**. È stata progettata per permettere a ricercatori di costruirci sopra e per dare agli studenti di coglierne rapidamente i dettagli.
Nonostante sia definita come "minimalistica", questa libreria tuttavia offre diverse configurazioni possibili per la som il che ha reso MiniSom la libreria adatta al nostro scopo.

Innanzitutto, per replicare l'esperimento in [18] è stata impostata la seguente configurazione:

- **Dimensione**: 10 x 10;
- **N° features**: da 4 a 265, dipende dal valore anova che si sta considerando;
- **Sigma**: 5;
- **Learning rate**: 0,1;
- **Neighborhood function**: gaussian;
- **Activation distance**: manhattan;
- **Train iterations**: 100.

```
n_neurons = m_neurons = neurons # = 10
som = MiniSom(n_neurons, m_neurons, X_la.shape[1], sigma=5,
                learning_rate=0.1,
                neighborhood_function='gaussian',
                activation_distance='manhattan')

som.random_weights_init(X_la)
# random training
som.train_random(X_la, train_iter, verbose=True)
```

Listing 4.7: Definizione SOM

Quanto descritto nel codice sopra riguarda solo la parte di clustering (training) della SOM, per poter eseguire la classificazione si necessita di una funzione ausiliaria, che per prima cosa calcola il label mapping sui neuroni allenati, ovvero attribuire ad ogni neurone della SOM una label (in base a quanto ottenuto dal training), ed in seguito calcola la **BMU** per l'input dato e in base alle coordinate della BMU si risale alla label grazie al mapping descritto in precedenza.

```
def classify(som, data, X_train, y_train, neurons, typ, a_val
    , train_iter):
    winmap = som.labels_map(X_train, y)
    default_class = np.sum(list(winmap.values()))
                        .most_common()[0][0]
    result = []
    for d in data:
        win_position = som.winner(d)
        if win_position in winmap:
            result.append(winmap[win_position]
                        .most_common()[0][0])
        else:
            result.append(default_class)
    return result
```

Listing 4.8: Classificazione per SOM

Per quanto riguarda il calcolo delle performance anche qui è stata considerata l'accuracy e il calcolo delle metriche è stato affidato al metodo "**classification_report**" di scikit-learn metrics.

```
class_repo = classification_report(y_test, classify(som,
   X_test[:, less_t_anova_vals], X_la, y_train, ...), ...)
```

Listing 4.9: Report di classificazione

4.1.3 Costruzione del dataset

Per costruire il dataset bisogna innanzitutto caricare i dati grezzi acquisiti con l'M5Stack.
Una volta caricato il dataset inizia la fase di pre-processing dei dati, in primo luogo il **noise filtering**:

```
# NOISE FILTERING
filtered_signals = np.empty((1, 128,))
for idx, val in enumerate(trainX):
    filt_ax_val = np.empty((128,))
    for i in range(6):
        ...
        filtered_signal = thirdord_butt_filt(
                        median_filter(signal, window_size))
        ...
    filt_ax_val = np.expand_dims(filt_ax_val, axis=0)
    ...
```

Listing 4.10: Noise filtering

Una volta filtrati i dati in input si procede a separare le due componenti del segnale dell'accelerometro:

```
# OBTAINING BODY AND GRAVITY ACC
body_acc = np.empty((1, 128,))
for idx, val in enumerate(filtered_signals):
    body_acc_val = np.empty((128,))
    for i in range(3):
        ...
```

```
7           filtered_signal = butterworth_filter(signal,
8                                                corner_frequency, order)
9           ...
10      body_acc_val = np.expand_dims(body_acc_val, axis=0)
11      ...
```

Listing 4.11: Separazione body e gravity acc.

A questo punto vengono salvati i segnali filtrati di:

- accelerazione totale (body + gravity);
- accelerazione del corpo (total - gravity);
- oscillazione del giroscopio.

Concludendo così la prima parte del preprocessing dei dati (i dati creati ora sono quelli che andranno in input ai modelli supervised). Ora si passa al calcolo delle 265 features, per prima cosa identifichiamo i 3 segnali da cui si parte per il calcolo delle features:

- accelerazione del corpo (X-Y-Z) che chiameremo **body_acc**;
- accelerazione gravitazionale (X-Y-Z) che chiameremo **grav_acc**;
- oscillazione del giroscopio (X-Y-Z) che chiameremo **body_gyro**.

Per body_acc e body_gyro è stato calcolato il **Jerk value**.

```
1 def calculate_jerk(sig, f_s):
2     jerk = np.gradient(sig, 1/f_s, axis=1)
3     return jerk
```

Listing 4.12: Calcolo jerk value

Per i 5 segnali ottenuti fino ad ora (senza considerare la divisione in x, y, z), è stata poi calcolata la **Magnitude** sui 3 assi. Nel contesto di un segnale tridimensionale con componenti lungo gli assi x, y e z, la magnitudine del segnale è la radice quadrata della somma dei quadrati delle sue componenti x, y e z.

```
def calculate_magnitude(x, y, z):
    x2 = np.power(x, 2)
    y2 = np.power(y, 2)
    z2 = np.power(z, 2)
    tmp_mag = np.add(x2, y2)
    tmp_mag = np.add(tmp_mag, z2)
    return np.sqrt(tmp_mag)
```

Listing 4.13: Calcolo magnitude

Ottenendo così un totale di 20 segnali da processare.
Per tutti questi segnali si procede ora a calcolare le vere features che sono:

1. **media**;
2. **deviazione standard**;
3. **median absolute deviation**;
4. **massimo**;
5. **minimo**;
6. **signal magnitude area**: si calcola partendo dalla magnitudine calcolata precedentemente, quindi, viene calcolata l'area sotto la curva di queste magnitudini nel tempo e il risultato viene diviso per la durata totale del segnale per ottenere la magnitudine media per unità di tempo.

```
def calculate_sma(mag):
    n = len(mag)
    tmp_sma = np.sum(mag, axis=1)
    return np.divide(tmp_sma, n)
```

Listing 4.14: Calcolo SMA

7. **energy**: l'energia di un segnale viene calcolata elevando al quadrato l'ampiezza del segnale in ciascun punto temporale e quindi integrando i valori risultanti per l'intera durata del segnale.

```
def calculate_energy(sig):
    tmp_en = np.sum(np.power(sig, 2), axis=1)
    return np.divide(tmp_en, len(sig))
```

Listing 4.15: Calcolo energy

8. **inter-quartile range**: l'inter-quartile range (IQR) è definito come la differenza tra i quartili superiore e inferiore dei dati. I quartili sono i tre punti che dividono i dati in quattro parti uguali, dove il quartile inferiore (Q1) è il valore che separa il 25% più basso dei dati, la mediana (Q2) è il valore che separa il 50% più basso da il 50% più alto e il quartile superiore (Q3) è il valore che separa il 25% più alto dei dati. Matematicamente: IQR = Q3 - Q1.

```
def calculate_iqr(sig):
    return np.percentile(sig, 75, axis=1)
               - np.percentile(sig, 25, axis=1)
```

Listing 4.16: Calcolo IQR

9. **entropy**: l'entropia viene calcolata in base alla distribuzione di probabilità dei valori del segnale, dove valori di entropia più alti indicano maggiore incertezza o casualità e valori di entropia più bassi indicano una maggiore prevedibilità.

```
# entropy
def calculate_entropy(sig):
    signal_hist, bin_edges = np.histogram(sig,
                                bins=np.unique(sig))
    signal_hist = signal_hist / np.sum(signal_hist)
    return entropy(signal_hist)

# calculate entropy for each signal in the set
def entropy_per_signal(sig):
    output = []
    for index, value in enumerate(sig):
        en = calculate_entropy(value)
        output.append(en)
```

```
    return np.array(output)
```

Listing 4.17: Calcolo Entropia

10. **auto regression coefficients**: 4 coefficienti per ogni feature sono stati calcolati.

```
# ar coeff
def signal_autoregression_coefficients_burg(sig, ordr =
    4):
     N = len(sig)
     a = np.zeros(ordr + 1)
     e = np.zeros(N)
     a[0] = 1
     for m in range(1, ordr + 1):
         num = 0
         den = 0
         for n in range(m, N):
             num += (sig[n] * sig[n - m])
             den += (sig[n] ** 2 + sig[n - m] ** 2)
         k = -2 * num / den
         a_temp = a.copy()
         a[m] = k * a[m - 1]
         for i in range(1, m):
             a[i] = a_temp[i] + k * a_temp[m - i]
     return a

# calculate ar coeff for each signal in the set
def calculate_arcoeff(sig):
     output = []
     for ind, v in enumerate(sig):
         ar_c = signal_autoregression_coefficients_burg(v)
         output.append(ar_c[1:])
     return np.array(output)
```

Listing 4.18: Calcolo Coefficienti AR

11. **correlation**: la correlazione può essere utilizzata per quantificare la relazione tra due segnali in una varietà di applicazioni, come l'analisi del segnale, il riconoscimento di modelli e l'estrazione di caratteristiche.

```
# correlation coeff
def correlation_coefficient(signal1, signal2):
    n = len(signal1)
    mean1 = np.mean(signal1)
    mean2 = np.mean(signal2)
    stddev1 = np.std(signal1)
    stddev2 = np.std(signal2)
    correlation = np.sum((signal1 - mean1) * (signal2 -
   mean2)) / (n * stddev1 * stddev2)
    return correlation

# calculate correlation coeff for each signal in the set
def calculate_correlation(sig1, sig2):
    output = []
    for ind, v in enumerate(sig1):
        corr = correlation_coefficient(v, sig2[ind])
        output.append(corr)
    return np.array(output)
```

Listing 4.19: Calcolo correlazione

Una volta calcolate e ordinate le 265 features per ogni record del dataset, il dataset viene diviso in train e test set e salvato in files .txt.

4.2 Lavoro statistico ANOVA (dimensionality reduction)

Questa parte di lavoro è una parte molto importante in quanto ci aiuta a determinare quali sono le features che caratterizzano più o meno i nostri dati e al variare della quantità di features usate possiamo vedere l'andamento delle performance dei nostri algoritmi.

Per eseguire questa analisi bisogna prima calcolare la varianza per classe di ogni feature;

```
X = np.concatenate((X_train, X_test))
y = np.concatenate((y_train, y_test))
df = pd.DataFrame(X)
df = df.T
...
# mean vals calc.
for j in df.iterrows():
    ...
    for i in range(len(j[1])):
        class_medio[np.argmax(y[i])] += 1
        x_medio += j[1][i]
        x_medio_tmp += j[1][i]
        elem_count += 1
        elem_count_tmp += 1
    for i in range(len(class_medio)):
        class_medio[i] /= elem_count_tmp
    x_medio_per_class.append(class_medio)
    x_medio_lst.append(x_medio_tmp / elem_count_tmp)
x_medio = x_medio / elem_count
...
# anova core calc.
for j in df.iterrows():
    ...
    for i in range(len(j[1])):
        class_dict[np.argmax(y[i])] += pow(j[1][i] -
                    x_medio_per_class[row_count]
                            [np.argmax(y[i])], 2)
```

```
28          varianza_tot += pow(j[1][i] - x_medio, 2)
29          varianza_par += pow(j[1][i] - x_medio, 2)
30          elem_count_tmp += 1
31      for i in range(len(class_dict)):
32          class_dict[i] /= elem_count_tmp
33      varianza_per_classe.append(class_dict)
34      varianza_par_lst.append(varianza_par / elem_count_tmp)
35  varianza_tot /= elem_count
```

Listing 4.20: Calcolo varianza per classe

Dopodiché vengono create due liste, una che contiene, per ogni feature, la media tra le varianze (valori anova) e l'altra che contiene il minimo tra le varianze. In base ai valori presenti in queste liste, in posizione corrispondente alla feature per il quale ne è stato calcolato il valore anova, verranno scelte le features che andranno a comporre l'input del nostro modello.

```
1  ...
2  for i in range(len(varianza_per_classe)):
3      ...
4      for j in range(len(varianza_per_classe[i])):
5          classi = len(varianza_per_classe[i])
6          accumulatore += varianza_per_classe[i][j]
7          if varianza_per_classe[i][j] < min_var_class:
8              min_var_class = varianza_per_classe[i][j]
9              classe_min = j
10     varianza_media_classi.append(accumulatore/classi)
11     varianza_min_classi.append(min_var_class)
12     classe_varianza_min.append(classe_min)
13 for i in range(len(varianza_par_lst)):
14     varianza_media_classi[i] /= varianza_par_lst[i]
15     varianza_min_classi[i] /= varianza_par_lst[i]
16     for j in range(len(varianza_per_classe[i])):
17         varianza_per_classe[i][j] /= varianza_par_lst[i]
18 anova_f_lst = []
19 for idx, val in enumerate(varianza_par_lst):
20     anova_f_lst.append(val / varianza_tot)
21
22 if sys.argv[4] == 'avg' or sys.argv[4] == 'avgmin':
```

```
    # CALCOLO RISULTATI CONSIDERANDO MEDIA ANOVA PER CLASSE
    ...
    for a_val in range_lst:
        ...
        for idx, val in enumerate(varianza_media_classi):
            if val > a_val/divider:
                greater_t_anova_vals.append(idx)
            else:
                less_t_anova_vals.append(idx)
        X_ga = X[:, greater_t_anova_vals]
        X_la = X[:, less_t_anova_vals]
        ...
    ...
...
```

Listing 4.21: Selezione features

Il codice è speculare anche per l'if che non è stato riportato, ovvero *if sys.argv[4] == 'min':*.

L'importanza di riuscire a ridurre il numero di features grazie a questa analisi, oltre al fatto che potrebbe migliorare le performance del modello, sta nel fatto che ci aiuta a risparmiare memoria per l'esecuzione della SOM sul dispositivo M5Stack.

4.3 Implementazione per micro-controllore

In questa parte si discuterà del lavoro svolto per permettere il funzionamento dell'esperimento sul dispositivo M5Stack, verranno anche descritte le strade percorse che però non hanno portato a risultati.
Il lavoro descritto qui è stato svolto programmando in C++, inoltre bisogna fare una importante introduzione, quando si programmano dispositivi embedded ci si trova davanti inizialmente a due funzioni principali da "riempire":

- **setup()**: all'interno di questa funzione va inserito il codice relativo alla **fase di boot**, ovvero codice che verrà eseguito solo all'avvio del dispositivo;
- **loop()**: all'interno di questa funzione va inserito il codice da eseguire dopo la fase di boot. La particolarità di questa funzione è che esegue un loop infinito.

Iniziamo mostrando in primo luogo cosa avviene nella fase di boot del microcontrollore.

```
// global variables
const int N_WEIGHTS = 100;
const int N_FEATURES = 118;
float w_mtx[N_WEIGHTS][N_FEATURES];
int lbls[N_WEIGHTS];
// MAIN PROGRAM
void setup() {
    ...
    File file = SD.open("/weights_lst_avg_bal_10.txt");
    ...
    file.close();
    file = SD.open("/map_lst_bal_10.txt");
    ...
    file.close();
}
```

Listing 4.22: Fase di boot

Nella fase di boot del dispositivo in sostanza vengono inizializzati i pesi della SOM e la mappa neurone-label.
Proseguiamo poi distinguendo i tre trask che vengono svolti dal dispositivo embedded:

1. **Campionamento sensori IMU**;
2. **Calcolo delle features** (per singolo record);
3. **Inferenza**.

4.3.1 Campionamento sensori IMU

Il campionamento dei dati dal sensore è piuttosto semplice, grazie alla libreria "M5Stack.h" vengono acquisiti i valori di accelerometro e giroscopio, i valori vengono convertiti in stringa e concatenati fino al raggiungimento della finestra di 128 campionamenti. Raggiunti i 128 campionamenti i record vengono concatenati ai rispettivi files .txt nella microSD.

```
// WRITE FILE TO SD
void appendFile(fs::FS &fs, const char * path, const char *
    message){
    File file = fs.open(path, FILE_APPEND);
    ...
    file.close();
}

// get sensor data
M5.IMU.getGyroData(&imu_data[0], &imu_data[1], &imu_data[2]);
M5.IMU.getAccelData(&imu_data[3], &imu_data[4],&imu_data[5]);
list_count +=1;
if (list_count <= 129)
{
    if (list_count == 129){

        appendFile(SD, path1.c_str(), (gyroX +"\n").c_str());
        ...
        list_count = 0;
        records += 1;
        gyroX = "";
        ...
    }
    else
    {
        gyroX += to_string(imu_data[0]) + "  ";
        ...
    }
}
```

Listing 4.23: Campionamento sensori IMU

4.3.2 Calcolo delle features

Per poter permettere l'esecuzione dell'inferenza in modo completamente autonomo, le stesse features viste in precedenza sono state calcolate anche in C++.

```
// median filter
vector<float> medianFilter(vector<float> input, int
    windowSize){
     ...
}

//third order butterworth filter
vector<float> thirdordButtFilt(vector<float> signal, float dt
    , float cutoffFrequency) {
     ...
}

// Function to calculate the Butterworth filter coefficients
vector<float> butterworthCoeffs(float cutoff, int order) {
     ...
}

// Function to apply the Butterworth filter
vector<float> butterworthFilter(vector<float> data, float
    cutoff, int order) {
     ...
}


// CALCULATE VELOCITY
vector<float> calculate_velocity( vector<float> acceleration,
      vector<float> time) {
     ...
}

// CALCULATE JERK
```

```
28 vector<float> calculate_jerk( vector<float> velocity,  vector
       <float> time) {
29     ...
30 }
31
32 // CALCULATE MEAN
33 float average(vector<float>  v){
34     ...
35 }
36
37 // CALCULATE STD DEV
38 float std_dev(vector<float>  v, float  mean){
39     ...
40 }
41
42 // CALCULATE MEDIAN ABSOLUTE DEVIATION
43 float mad(vector<float> data) {
44     ...
45 }
46
47 // calculate max
48 float max_v(vector<float> v){
49     return *max_element(v.begin(), v.end());
50 }
51
52 // calculate min
53 float min_v(vector<float> v){
54     return *min_element(v.begin(), v.end());
55 }
56
57 // SIGNAL MAGNITUDE AREA
58 float signalMagnitudeArea(vector<float> x, vector<float> y,
       vector<float> z, int start, int end) {
59     ...
60 }
61
62 // SIGNAL ENERGY
```

```
63 float signalEnergy(vector<float> signal, int start, int end)
       {
64     ...
65 }
66
67 // SIGNAL INTERQUARTILE RANGE
68 float signalInterquartileRange(vector<float> signal_o) {
69     ...
70 }
71
72 // SIGNAL ENTROPY
73 float signalEntropy(vector<float> signal, int start, int end)
       {
74     ...
75 }
76
77 // SIGNAL AR COEFFICIENT with order = 4
78 vector<float> signalAutoregressionCoefficientsBurg(vector<
       float> signal, int order) {
79     ...
80 }
81
82 // CORRELATION BETWEEN 2 SIGNALS
83 float correlationCoefficient(vector<float> signal1, vector<
       float> signal2) {
84     ...
85 }
86
87 // MAG CALCULATION
88 float signalMagnitude(float x, float y, float z) {
89     // Calculate the signal magnitude
90     return sqrt(x * x + y * y + z * z);
91 }
92
93 // MAG FOR EACH RECORD
94 vector<float> getMagVector(vector<float> x_v, vector<float>
       y_v, vector<float> z_v){
95     ...
```

```
96 }
97
98 // SIGNAL MAGNITUDE AREA FROM MAGNITUDE
99 float getSMAfromMag(vector<float> mag){
100     ...
101 }
```

Listing 4.24: Calcolo delle features

Le features andranno poi inserite in un "vector<float>" che sarà passato in input alla funzione che calcolerà l'inferenza.

4.3.3 Inferenza

Per quanto riguarda la parte di implementazione dell'inferenza sono stati provati diversi approcci, in ordine:

1. minificazione della libreria **MiniSom**;
2. implementazione tramite custom layer con tensorflow per poi esportare il modello quantizzato con **tensorflow-lite**;
3. implementazione in C++.

Per quanto riguarda il primo approccio, data la possibilità attraverso M5Burner e UIFlow (software forniti da M5 stessa) di programmare il dispositivo in Python, si è cercato di minificare la libreria MiniSom lasciando solo il minimo indispensabile per l'inferenza.

```
1 from numpy import (unravel_index, linalg, subtract)
2
3 class MiniMiniSom(object):
4     def __init__(self, w, winmap, act_map, def_class):
5         self._weights = w
6         self._activation_map = act_map
7         self._activation_distance = self._manhattan_distance
8         self.winmap = winmap
9         self.default_class = def_class
10
```

```
    def _activate(self, x):
        self._activation_map = self._activation_distance(x,
                                                self._weights)

    def _manhattan_distance(self, x, w):
        tmp = subtract(x, w)
        return linalg.norm(tmp, ord=1, axis=-1)

    def winner(self, x):
        self._activate(x)
        return unravel_index(self._activation_map.argmin(),
                             self._activation_map.shape)

    def predict(self, data):
        result = []
        for d in data:
            win_position = self.winner(d)
            if win_position in self.winmap:
                result.append(self.winmap[win_position]
                                    .most_common()[0][0])
            else:
                result.append(self.default_class)
        return result
```

Listing 4.25: Versione minificata di MiniSom

Tuttavia a causa dei limiti di memoria del dispositivo non è stato possibile usufruire di questa opzione.

Per il secondo approccio si è cercato di inserire all'interno di un layer Keras il contenuto della versione minificata della libreria MiniSom.

```
class SOMLayer(keras.layers.Layer):
    def __init__(self,X_train,y_train,w,n,input_shape):
        ...
    @tf.function
    def call(self, inputs):
        ...
    # @tf.function
    def _activate(self, x):
```

```
        ...
    def _activate_lblmap(self, x):
        ...
    @tf.function
    def _manhattan_distance(self, x, w):
        return tf.linalg.norm(tf.subtract(x, w), ord=1,
                                                 axis=-1)

    def _manhattan_distance_lblmap(self, x, w):
        return linalg.norm(subtract(x, w), ord=1, axis=-1)

    def _check_input_len(self, data):
        ...
    @tf.function
    def winner(self, x):
        ...
    def winner_lblmap(self, x):
        ...
    def labels_map(self, data, labels):
        ...
```

Listing 4.26: Versione Keras layer di MiniSom

Tuttavia anche questo approccio si è rivelato essere fallimentare in quanto con tensorflow e keras, l'input del layer di input del modello viene convertito automaticamente in tensore e l'esecuzione viene automaticamente impostata in **graph mode**, a causa di questo non è risultato possibile riuscire a leggere i valori contenuti in questi tensori risultando cosi impossibile calcolare la **BMU** e di conseguenza l'output del modello.

Il terzo ed ultimo approccio invece si è rivelato essere quello corretto, partendo da una nostra piccola **implementazione della SOM in C++**, abbiamo estratto solo la funzione che calcola l'indice della **BMU** e inserita nel dispositivo:

```
int winner_index(vector<float>& sample) {
    int winner = 0;
    float min_dist = 3.402823466e+38;
    for (int i = 0; i < N_WEIGHTS; i++) {
```

```
5           float dist = 0;
6           for (int j = 0; j < N_FEATURES; j++) {
7               dist += fabs(sample[j] - w_mtx[i][j]);
8           }
9           if (dist < min_dist) {
10              min_dist = dist;
11              winner = i;
12          }
13      }
14      return winner;
15  }
```

Listing 4.27: Calcolo Best Matching Unit

In questa funzione viene calcolata la distanza **Manhattan** (o **TaxiCab**) del campione da ogni neurone nella matrice dei pesi "w_mtx" (caricata da file in fase di boot nella funzione **setup()**) e viene restituito l'indice del vincitore (BMU). Da questo indice si risale alla label corrispondente grazie alla mappa neurone-label anch'essa caricata in fase di boot.

```
1 int win_idx = winner_index(features);
2 int win_lbl = lbls[win_idx];
3 string win_act = menu_items[win_lbl];
```

Listing 4.28: Ricerca della label

In "win_lbl" si avrà quindi un valore intero (da 0 a 5) che identificà la classe dell'attività corrispondente. L'attività sarà contenuta in "win_act".

Capitolo 5

Risultati ottenuti

In questa sezione discuteremo e confronteremo i risultati ottenuti durante il corso di tutto l'esperimento.
Iniziamo mostrando i risultati ottenuti con le tecniche supervised studiate sui due dataset presi in considerazione.

5.1 Risultati modelli supervised

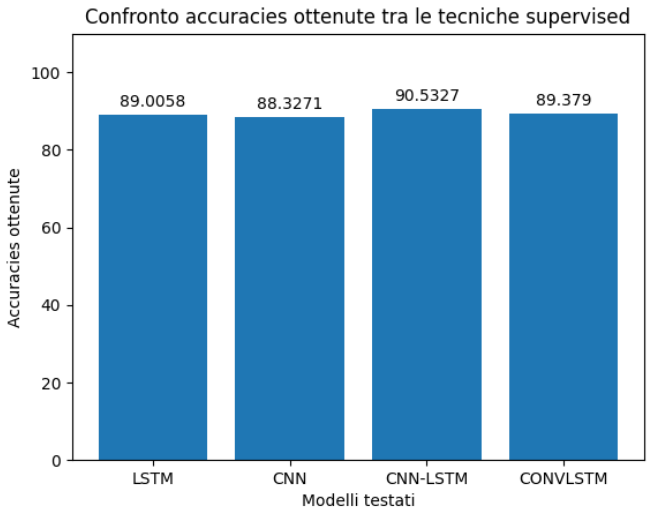
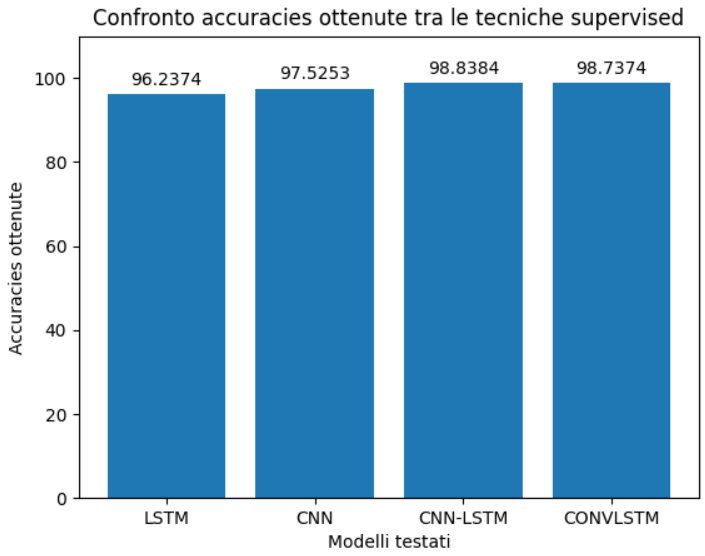
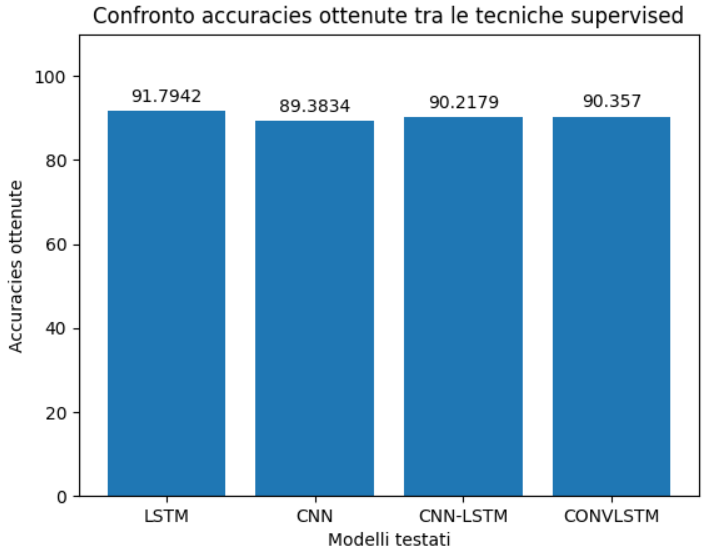


Figura 5.1: Modelli supervised su dataset uci con dati non bilanciati



(a) Modelli supervised sul nostro dataset con dati non bilanciati



(b) Modelli supervised su dataset uci con dati bilanciati

Figura 5.2

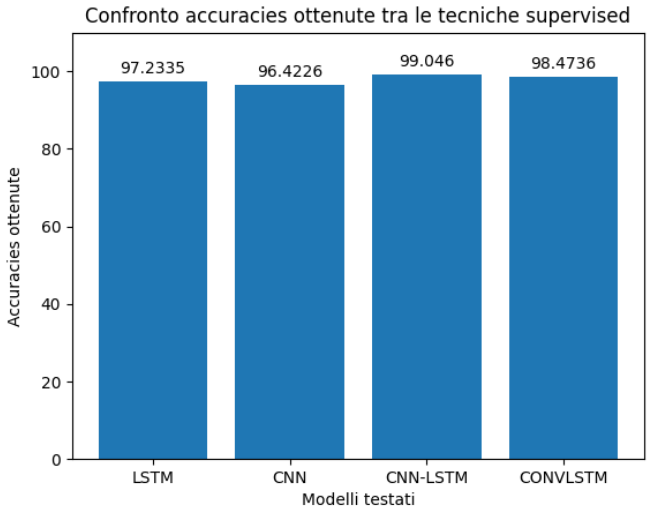


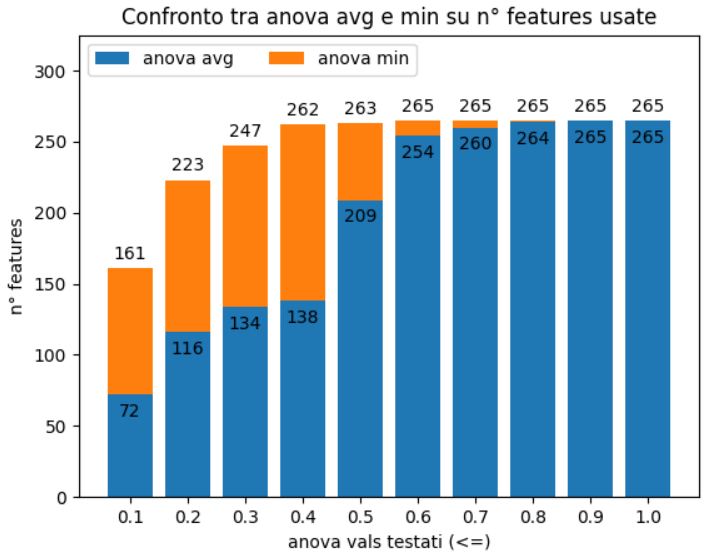
Figura 5.3: Modelli supervised sul nostro dataset con dati bilanciati

Dai grafici presentati si possono notare due cose in particolare:

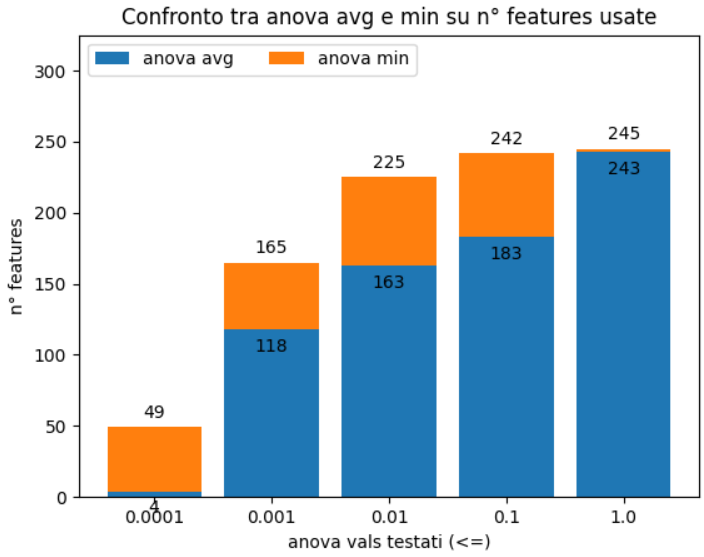
1. su dati bilanciati, prevedibilmente, i risultati tendono ad essere migliori;
2. sul nostro dataset le performance migliorano notevolmente, questo potrebbe essere dovuto a due fattori in particolare:
   - differente collocazione del sensore;
   - dataset raccolto campionando le attività di un singolo individuo piuttosto che molti.

5.2 Risultati modelli unsupervised

In questa sezione invece procederemo a mostrare i risultati che realmente sono interessanti ai fini dell'esperimento. Dato che le tecniche supervised solitamente offrono le performance migliori in quanto a classificazione, i risultati precedentemente mostrati servono proprio come metodo di comparazione tra le due tecniche.
Introduciamo mostrando innanzitutto come ha agito l'analisi ANOVA sulle features.



(a) Numero di features sul dataset uci al variare del val. anova



(b) Numero di features sul nostro dataset al variare del val. anova

Figura 5.4: Confronto numero di features usate con diversi valori ANOVA tra il dataset uci e il nostro

In Fig. 5.4 si può notare che per i due dataset sono stati usati due differenti range di valori anova.
Questa scelta è dovuta al fatto che nel dataset da noi acquisito già considerando solo le features con valore anova ≤ 0.1, se ne considerano già un numero importante (quasi il numero massimo considerando il minimo tra le varianze per classe) quindi abbiamo scelto un range che permettesse una migliore curva di crescita del numero delle features.

Il fatto che siano necessari valori molto piccoli può essere dovuto sempre al fatto che il dataset sia stato acquisito campionando le attività di un singolo individuo.

5.2.1 K-Means

La scelta di testare k-means è ricaduta sul fatto che è un algoritmo abbastanza leggero da poter essere implementato su un dispositivo embedded, oltre al fatto di essere uno dei più comuni algoritmi unsupervised presenti in letteratura (come già anticipato in precedenza).

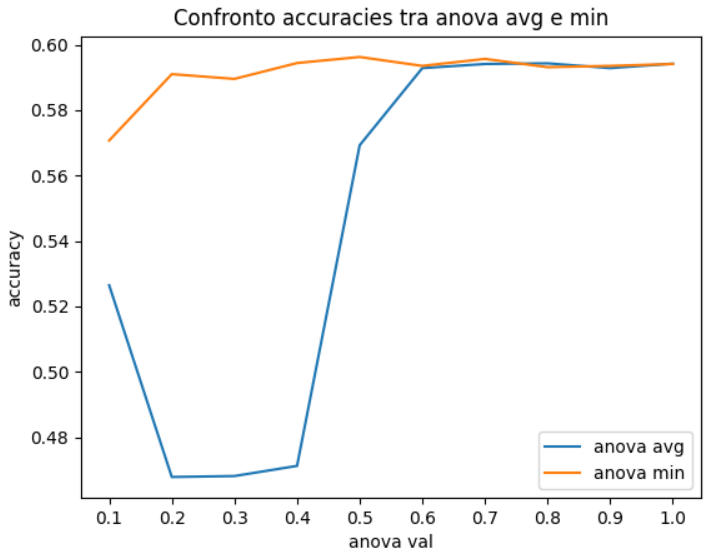


Figura 5.5: Confronto accuracies ottenute con K-Means su dataset uci

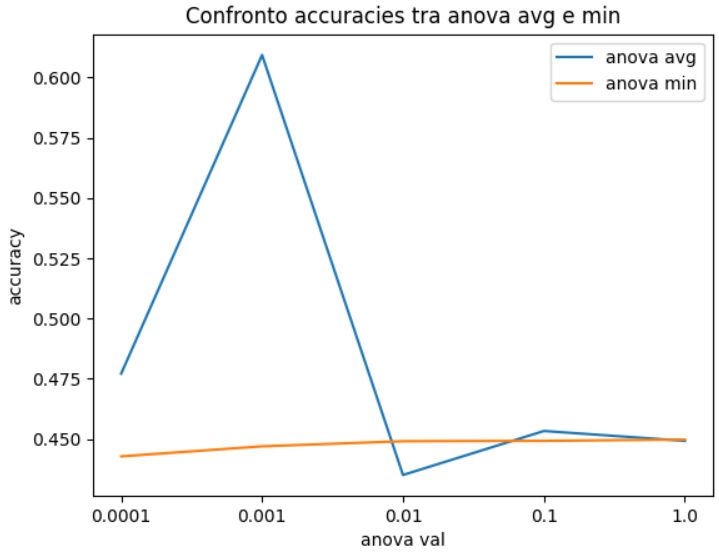
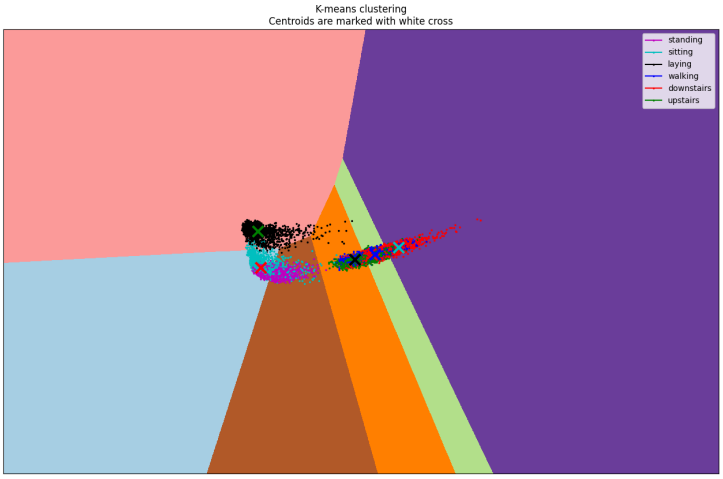


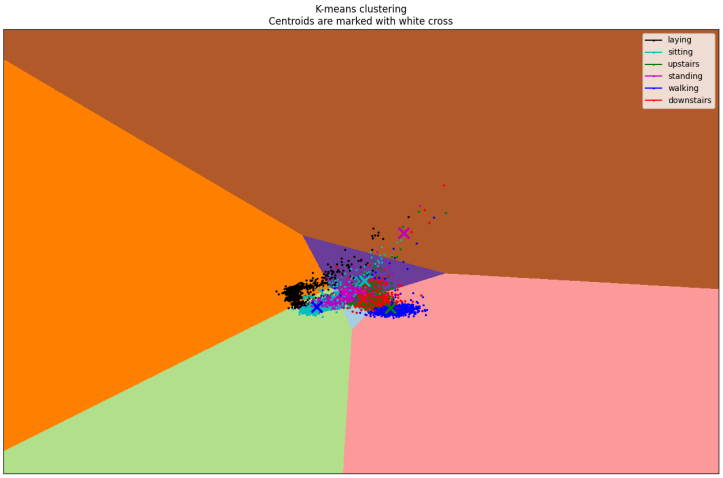
Figura 5.6: Confronto accuracies ottenute con K-Means sul nostro dataset

Si noti come per il dataset UCI k-means sembra comportarsi meglio all'aumentare del numero di features a disposizione, mentre con il nostro dataset k-means raggiunge il picco di performance con un numero medio-basso di features (Fig. 5.4) per poi calare di performance.
Le performance ottenute da k-means sono nella media, ma se paragonate a quelle ottenute con le tecniche supervised sono altamente insoddisfacenti.

Mostreremo di seguito il clustering risultante per entrambi i dataset scegliendo il valore anova che offre le performance migliori.



(a) Clustering k-means su dataset uci



(b) Clustering k-means sul nostro dataset

Figura 5.7: Confronto clustering tra i due dataset

5.2.2 Self-Organizing Map

Per quanto riguarda la SOM, oltre ai diversi valori anova, sono state testate diverse configurazioni anche in termini di dimensione, per la precisione sono state testate 5 dimensioni di SOM:

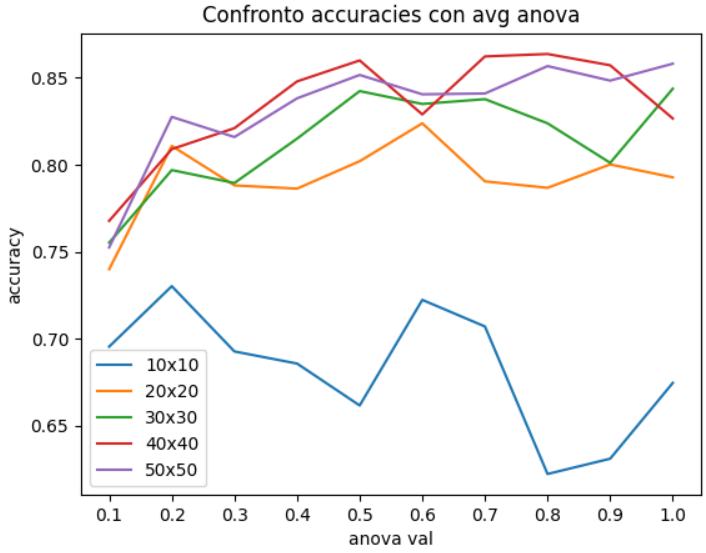
- 10×10 (usata anche in [18]);
- 20×20;
- 30×30;
- 40×40;
- 50×50.

Questo "giocare" con le dimensioni della SOM è stato effettuato per verificare se con un'area maggiore dove distribuire i dati (numero di neuroni maggiore) ne avrebbe potuto beneficiare il training (clustering) effettuato dall'algoritmo.

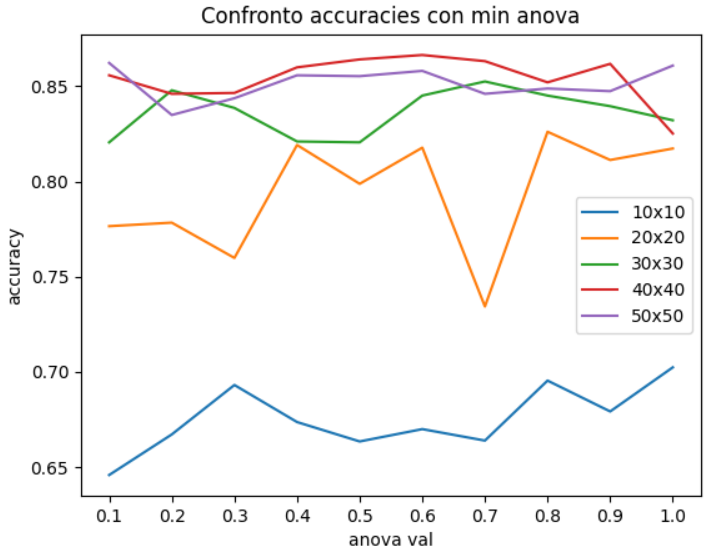
In questa sezione verranno mostrati separatamente i grafici riguardanti anova min e anova avg in quanto all'interno dei grafici è già presente un confronto ma tra le diverse dimensioni della SOM testate.
Inoltre non verranno mostrati i risultati ottenuti su dati non bilanciati dato che seguono lo stesso andamento di quelli su dati bilanciati ma con circa 2 punti percentuali in meno in quanto ad accuracies ottenute.

I risultati verranno illustrati secondo il seguente schema:

- dataset uci:
  - anova avg;
  - anova min;
- dataset nostro:
  - anova avg;
  - anova min;

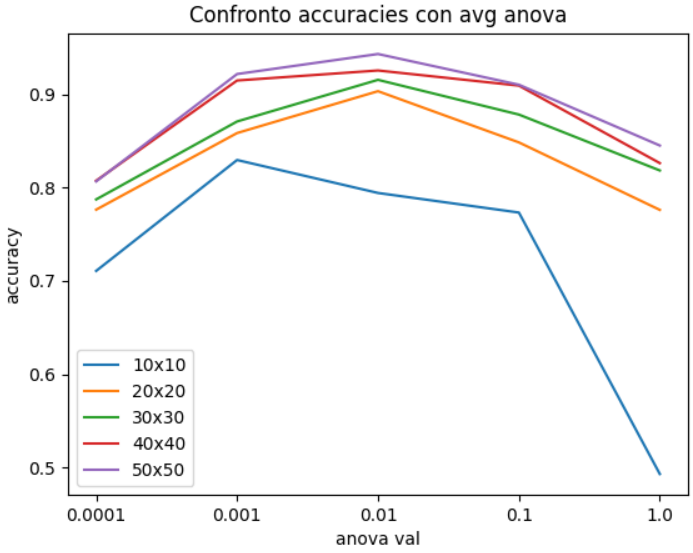


(a) Confronto selezionando le features con anova avg

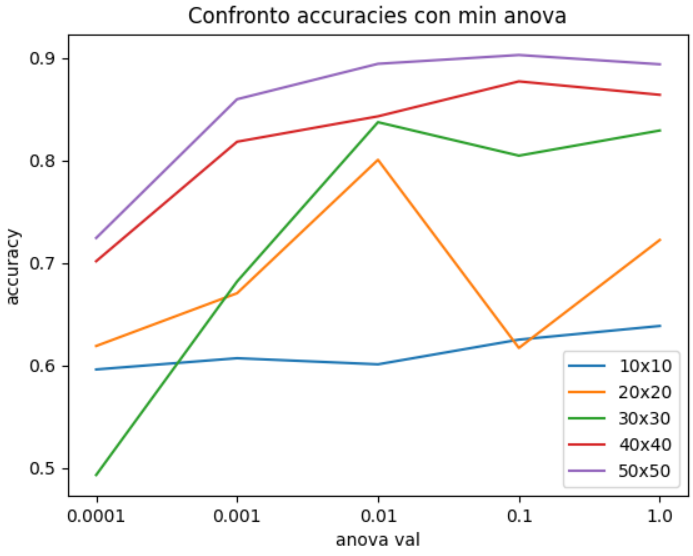


(b) Confronto selezionando le features con anova min

Figura 5.8: Confronto anova avg e anova min



(a) Confronto selezionando le features con anova avg



(b) Confronto selezionando le features con anova min

Figura 5.9: Confronto anova avg e anova min

Si può subito notare il diverso comportamento sui due dataset.
Sul dataset uci, all'aumentare del numero delle features passate in input sembrerebbero aumentare anche le performance dell'algoritmo, invece sul dataset da noi costruito l'andamento tendenzialmente è a campana.

Per quanto riguarda le dimensioni testate della SOM, si nota che le performance tendono a migliorare all'aumentare delle dimensioni della SOM.
Di seguito mostreremo il risultato del training di una SOM 50×50.

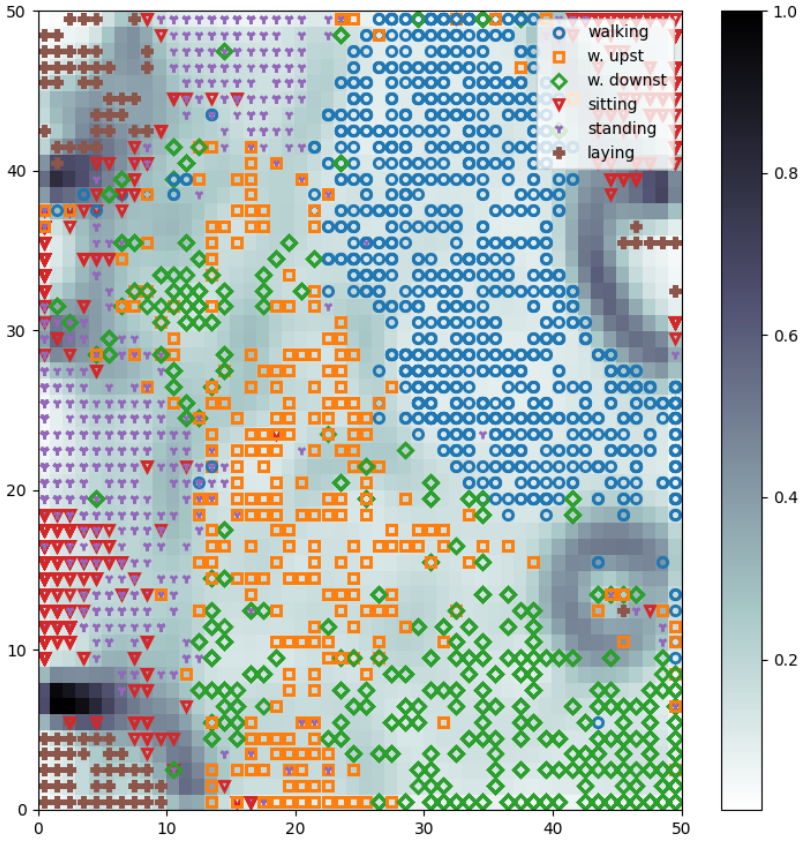


Figura 5.10: Risultato della fase di training della SOM

In Fig. 5.10 è possibile vedere la rappresentazione grafica della **SOM** alla fine della fase di training. Quella che viene mostrata è una SOM 50×50 (quindi ogni quadrato presente nell'immagine rappresenta un neurone della SOM).
I simboli presenti sui neuroni rappresentano la classe che ogni neurone "rappresenta" e si può notare che la fase di training ha portato alla formazione di "regioni" di neuroni vicini che rappresentano la stessa classe (migliore è questo raggruppamento e migliori saranno le performance che la SOM offrirà).

Si può notare che nell'immagine viene rappresentata anche una seconda informazione, infatti ogni quadrato (neurone) ha anche un colore di sfondo (rappresentato dalla barra a sinistra della SOM in figura) che rappresenta la distanza di ogni neurone dal proprio "vicinato", e più è scuro il colore più questo neurone è distante dai neuroni ad esso adiacenti (distanza euclidea). Questa distanza è calcolata sommando le distanze del neurone in questione da quelli ad esso adiacenti.

Conclusioni

Dopo aver visto i risultati ottenuti dalle tecniche unsupervsed si può notare che la tecnica che più si avvicina ai risultati delle tecniche supervised (e di molto) è la **Self-Organizing Map**.
Alla luce di questo risultato si è proceduto all'implementazione in C++ della SOM per poter permettere al micro-controllore di poter eseguire l'inferenza in autonomia.

Per motivi legati alla memoria del dispositivo in questione al suo interno è stata inserita una som 10×10 con 118 features, che garantisce comunque un punteggio di accuracy intorno all'80%.

Questo lavoro comunque non è arrivato alla sua forma definitiva e ci sono ancora molte cose da testare ed eventualmente migliorare o aggiungere, ad esempio tra gli possibili sviluppi futuri possiamo inserire:

- espandere il dataset acquisendo dati da più soggetti;
- ottimizzare il codice C++ del micro-controllore: si potrebbe infatti cercare di ottimizzare e ridurre il consumo di risorse dell'attuale codice caricato sul dispositivo al fine di cercare di aumentare le dimensioni della SOM caricata sul dispositivo. Si ipotizza una dimensione massima di 16×16 o 17×17.

Bibliografia

[1] Hande Alemdar, Tim LM van Kasteren, and Cem Ersoy. Using active learning to allow activity recognition on a large scale. In *Ambient Intelligence: Second International Joint Conference on AmI 2011, Amsterdam, The Netherlands, November 16-18, 2011. Proceedings 2*, pages 105–114. Springer, 2011.

[2] Arif Reza Anwary, Hongnian Yu, and Michael Vassallo. Optimal foot location for placing wearable imu sensors and automatic feature extraction for gait analysis. *IEEE Sensors Journal*, 18(6):2555–2567, 2018.

[3] Paola Ariza Colpas, Enrico Vicario, Emiro De-La-Hoz-Franco, Marlon Pineres-Melo, Ana Oviedo-Carrascal, and Fulvio Patara. Unsupervised human activity recognition using the clustering approach: A review. *Sensors*, 20(9):2702, 2020.

[4] Guilherme de Alencar Barreto and Aluizio Fausto Ribeiro Araújo. Time in self-organizing maps: An overview of models. *International Journal of Computer Research*, 10(2):139–179, 2001.

[5] John Burkardt. K-means clustering. *Virginia Tech, Advanced Research Computing, Interdisciplinary Center for Applied Mathematics*, 2009.

[6] Simon Dablemont, Geoffroy Simon, Amaury Lendasse, Alain Ruttiens, François Blayo, and Michel Verleysen. Time series forecasting with som and local non-linear models-application to the dax30 index prediction. In *WSOM 2003, Workshop on Self-Organizing Maps*, 2003.

[7] Giuseppe De Leonardis, Samanta Rosati, Gabriella Balestra, Valentina Agostini, Elisa Panero, Laura Gastaldi, and Marco Knaflitz. Human activity recognition by wearable sensors: Comparison of different classifiers for real-time applications. In *2018 IEEE international symposium on medical measurements and applications (memea)*, pages 1–6. IEEE, 2018.

[8] Dheeru Dua and Casey Graff. UCI machine learning repository, 2017.

[9] Alessandro Ghibellini, Luciano Bononi, and Marco Di Felice. Intelligence at the iot edge: activity recognition with low-power microcontrollers and convolutional neural networks. In *2022 IEEE 19th Annual Consumer Communications & Networking Conference (CCNC)*, pages 707–710. IEEE, 2022.

[10] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[11] Lyes Khacef, Bernard Girau, Nicolas Rougier, Andres Upegui, and Benoit Miramond. Neuromorphic hardware as a self-organizing computing system. *arXiv preprint arXiv:1810.12640*, 2018.

[12] Lyes Khacef, Benoît Miramond, Diego Barrientos, and Andres Upegui. Self-organizing neurons: toward brain-inspired unsupervised learning. In *2019 International Joint Conference on Neural Networks (IJCNN)*, pages 1–9. IEEE, 2019.

[13] Teuvo Kohonen. The self-organizing map. *Proceedings of the IEEE*, 78(9):1464–1480, 1990.

[14] Roland Kromes, Adrien Russo, Benoît Miramond, and François Verdier. Energy consumption minimization on lorawan sensor network by using an artificial neural network based application. In *2019 IEEE Sensors Applications Symposium (SAS)*, pages 1–6. IEEE, 2019.

[15] Xiaosheng Li and Jessica Lin. Linear time complexity time series classification with bag-of-pattern-features. In *2017 IEEE International Conference on Data Mining (ICDM)*, pages 277–286. IEEE, 2017.

[16] Manuel Martín-Merino and Jesus Román. A new som algorithm for electricity load forecasting. In *Neural Information Processing: 13th International Conference, ICONIP 2006, Hong Kong, China, October 3-6, 2006. Proceedings, Part I 13*, pages 995–1003. Springer, 2006.

[17] Rashmika Nawaratne, Damminda Alahakoon, Daswin De Silva, and Xinghuo Yu. Ht-gsom: Dynamic self-organizing map with transience for human activity recognition. In *2019 IEEE 17th international conference on industrial informatics (INDIN)*, volume 1, pages 270–273. IEEE, 2019.

[18] Pierre-Emmanuel Novac, Andrea Castagnetti, Adrien Russo, Benoît Miramond, Alain Pegatoquet, and Francois Verdier. Toward unsupervised human activity recognition on microcontroller units. In *2020 23rd Euromicro Conference on Digital System Design (DSD)*, pages 542–550. IEEE, 2020.

[19] Nicolas Rougier and Yann Boniface. Dynamic self-organising map. *Neurocomputing*, 74(11):1840–1847, 2011.

[20] Giuseppe Vettigli. Minisom: minimalistic and numpy-based implementation of the self organizing map, 2018.