# classificatore per il MOVIMENTO DEL BRACCIO

## INTRODUZIONE

Questo progetto ha come obiettivo la classificazione di due azioni distinte: braccio in movimento e braccio fermo. Per addestrare il modello viene usato l’algoritmo k-fold cross validation, che consiste in una divisione del dataset in k parti (fold). Per ogni iterazione, k-1 fold vengono usati per fare il training e 1 fold viene usato per fare il testing. Il processo viene ripetuto k volte e ogni fold viene usato una volta come test.

Il sensore utilizzato è il Nordic Thingy 52, che al suo interno presenta una IMU. È possibile quindi rilevare dati inerziali relativi ad accelerometro, giroscopio e magnetometro.

## RACCOLTA DEI DATI

Il primo passaggio è la raccolta dei dati, eseguita grazie al file python chiamato “data\_collector.py”. Viene innanzitutto avviata la scansione dei dispositivi BLE disponibili nelle vicinanze e viene cercato tra questi l’indirizzo MAC del Thingy specifico. Nel codice viene utilizzata la classe “Thingy52Client” per connettersi al dispositivo e ricevere dati inerziali, che vengono salvati su un file CSV.

--------------------------------------------------------------------------------------------------------------------------------------

#### LA CLASSE THINGY52CLIENT

La classe Thingy52Client eredita tutte le funzionalità di base di BleakClient, necessaria per gestire la comunicazione con dispositivi BLE. All’interno della classe sono definiti parametri come l’indirizzo MAC del Thingy e il buffer per la raccolta dei dati.

Vengono poi definiti i diversi metodi, che permettono di interfacciarsi con il sensore. Il metodo “connect” permette di connettersi con il dispositivo, cambiare il suo stato e di conseguenza cambiare il colore del suo led (verde = “connected”, rosso = “recording”).

Il metodo “disconnect” disconnette il dispositivo e chiude il file di registrazione dei dati. Il “cuore” della classe è il metodo che consente di ricevere i dati inerziali e salvarli all’interno di un file CSV, che viene definito nel metodo “save\_to”. Ogni qual volta vengono ricevuti nuovi dati, viene chiamato il metodo “raw\_data\_callback”, che consente di fare la decodifica dei dati per ottenere i valori di accelerometro, giroscopio e magnetometro sui tre assi.

All’interno della classe è stato integrato anche un modello ONNX per la predizione in tempo reale e quindi in grado di effettuare inferenza sui dati raccolti.

--------------------------------------------------------------------------------------------------------------------------------------

La raccolta dei dati è avvenuta in condizioni statiche, quindi il corpo è rimasto fermo, e il Thingy52 è stato tenuto in mano. Sono quindi stati creati due file CSV: uno relativo al braccio fermo e uno relativo al braccio in movimento. Inoltre, i dati per entrambe le azioni sono stati registrati per una durata di 1 minuto e 30 secondi.

## CREAZIONE DEL DATASET

Il dataset viene creato all’interno di file python “dataset.py”, all’interno del quale viene definita la classe CSVDataModule e viene utilizzata la libreria PyTorch Lightning.

Innanzitutto, i file CSV vengono letti da una directory specificata (root\_dir) e i dati vengono suddivisi in finestre di lunghezza pari a 1 secondo. Le finestre possono sovrapporsi tra di loro e questo parametro è definito “overlap”, che in questo caso corrisponde al 100% (overlap = 1). Ad ogni finestra è associata un’etichetta, che viene estratta dal nome del file CSV (“still” o “moving”).

Viene usato il k-fold cross validation, perciò i dati vengono divisi in k fold (k = 3) e ad ogni iterazione, un gruppo viene usato per la validazione e gli altri per il training. Infine, vengono creati i dataloader per training e validation.

## MODELLO DI CLASSIFICAZIONE

Il modello è definito dalla classe CNN, che contiene il necessario per allenarlo, testarlo e valutarlo. Al termine di ogni fase di training o validazione, viene calcolata la previsione, che viene poi confrontata con il target. Grazie a ciò vengono registrati indicatori come perdita, precisione, f1 score, recall e matrice di confusione.

Una matrice di confusione è organizzata nel seguente modo:

|  |  |  |
| --- | --- | --- |
|  | **Actually positive** | **Actually negative** |
| **Predicted positive** | True positives | False positives |
| **Predicted negative** | False negatives | True negatives |

La precisione è un parametro che indica quale percentuale è veramente positiva tra tutti i dati positivi previsti.

La recall indica quante delle vere positività sono state effettivamente rilevate dal modello. Se la recall è bassa, il modello è sbilanciato.

L’F1-score indica quanto il modello stia imparando bene, mentre la perdita indica quanto il modello predice in modo errato rispetto all’obiettivo.

All’interno della classe CNN è specificato anche il learning\_rate, che in una prima prova è stato impostato a Dopo il training, la matrice di confusione di uno dei tre fold mostrava che in alcuni casi il modello predicesse erroneamente i dati relativi al label “moving”, assegnandogli il label “still”. Impostando un learing rate più lento (), le matrici di confusione sono risultate ottime e di conseguenza la precisione è migliorata.

## TRAINING E VALIDAZIONE

Le classi definite in precedenza vengono utilizzate all’interno del file python “train.py” e il processo viene eseguito su una GPU NVIDIA.

All’interno del main vengono inizialmente preparati i dati, definendo i parametri principali del modello: il numero di fold, la cartella all’interno della quale si trovano i file CSV, la frequenza di campionamento (in Hertz) e la durata delle finestre dei dati da analizzare (in secondi). Viene poi richiamata la classe CSVDataModule, che prepara i dati.

Inizia così il ciclo che implementa l’algoritmo k-fold cross-validation, che scorre i 3 fold definiti. Per ogni fold viene inizializzato il modello, grazie all’utilizzo della classe CNN, che restituisce la probabilità che ogni finestra appartenga a una delle classi “moving” o “still”.

Viene usata una classe di PyTorch Lightning, ModelCheckpoint, per monitorare la perdita sui dati di validazione (val\_loss). Più la perdita è piccola, più il modello è accurato. Questa classe inoltre salva il modello migliore, in termini di prestazioni, per ogni fold.

Immagine che contiene testo, schermata, diagramma, Carattere

Descrizione generata automaticamenteGrazie all’utilizzo della classe Trainer di PyTorch Lightning, il modello viene addestrato utilizzando rispettivamente il set di training e di validazione del fold corrente. Infine, il modello viene valutato sui dati di validazione. Tutto questo procedimento viene ripetuto k (3) volte.

Dopo aver avviato il training, le matrici di confusione di tutti e tre i fold sono risultate uguali. Accanto viene illustrata la matrice di confusione del terzo fold. Si evince che il classificatore sia in grado di predire correttamente i dati relativi alle classi “moving” e “still”, senza errori.

## TEST IN TEMPO REALE

Il prossimo passaggio è testare il classificatore in tempo reale. Per fare ciò è necessario salvare il modello addestrato in formato ONNX. Questo viene fatto con lo script python “convert\_model.py”, che esegue tre operazioni fondamentali: carica il modello addestrato salvato come checkpoint, lo converte nel formato ONNX e salva questa nuova versione.

Per provare effettivamente il classificatore è stato realizzato un nuovo script, con una struttura molto simile al “data\_collector.py”, chiamato “inference.py”. Tramite quest’ultimo avviene la connessione al Thingy e inizia l’inferenza in tempo reale. Tutti i dati vengono salvati all’interno di un nuovo file CSV chiamato “movement\_data”. La classe “Thingy52Client” ha un ruolo fondamentale in quanto utilizza la libreria “onnxruntime” per caricare il modello ONNX precedentemente creato. Il metodo “raw\_data\_callback” svolge un ruolo fondamentale nell’inferenza in tempo reale. I dati vengono salvati su un buffer di dimensione 60 Hz e quando il buffer si riempie, i dati vengono mandati al modello ONNX per l’inferenza. Quest’ultimo restituirà sul terminale una predizione relativa al tipo di movimento (“moving” o “still”).

## LIMITI DEL MODELLO

Provando il modello in tempo reale è emerso che funziona molto bene quando la persona è ferma. Tuttavia, ci sono delle limitazioni legate al movimento del corpo. Il modello è stato addestrato esclusivamente su dati acquisiti in condizioni statiche e quindi presenta delle discrepanze che riguardano il sistema di riferimento rispetto al quale si muove il braccio.

Quando il soggetto di muove mantenendo il braccio fermo rispetto al corpo, il classificatore alterna predizioni tra “moving” e “still”. Quando invece il soggetto si muove mantenendo il braccio fermo rispetto al “mondo”, il classificatore è in grado di fornire la predizione corretta.

Quando il corpo si muove, nonostante il braccio sia fermo, la IMU registra un’accelerazione, prevalentemente su una sola direzione (corpo che si muove in avanti o indietro). Il classificatore non è addestrato su questo pattern di movimento; perciò, non riesce a distinguere se l’accelerazione sia dovuta al movimento del braccio o al movimento del corpo.

Un caso particolare lo si ha quando il soggetto si muove molto lentamente con il braccio teso e fermo. In questa situazione il classificatore predice correttamente “still”. Forse i dati di accelerazione relativi al corpo lento sono molto diversi dai pattern di accelerazione relativi al braccio in movimento.

In questo caso quindi il modello ha imparato a riconoscere il movimento del braccio rispetto a un sistema di riferimento fisso, cioè il soggetto fermo. Non riesce a riconoscere quando il sistema di riferimento stesso (il corpo) è in movimento.

In alternativa si potrebbe implementare un sistema in grado di riconoscere quando il corpo è in movimento e che quindi si concentri esclusivamente sul movimento del braccio, in qualsiasi scenario.