Realizzazione di un Filtro Convolutivo con Programmazione Genetica per l'Analisi di ECG

RELAZIONE PROGETTO MACHINE LEARNING

Giulia Oddi

Ottobre 2023

1 Introduzione

Il progetto ha avuto come obiettivo la realizzazione di un **filtro convolutivo** con **programmazione genetica**. La programmazione genetica (GP) è una tecnica di calcolo evolutivo che risolve automaticamente i problemi senza richiedere all'utente di conoscere o specificare la struttura della soluzione. Generazione dopo generazione, GP trasforma popolazioni di programmi in nuove popolazioni, migliori, valutando la qualità del funzionamento del programma e poi confrontandone il comportamento con quello ideale. Questo confronto viene quantificato tramite la fitness.
Per quanto riguarda il progetto, l'obiettivo è, tramite programmazione genetica, trovare, se esiste, un filtro convolutivo in grado di produrre una uscita (segnale) che possa essere classificata in modo migliore rispetto al segnale originale classificato da un classificatore prefissato. Per effettuare questo confronto, è stato scelto di utilizzare il classificatore Random Forest.
Il segnale utilizzato per il progetto è un segnale temporale che riguarda gli elettrocardiogrammi o ECG. In particolare, un ECG è un esame diagnostico volto a registrare l'attività elettrica del cuore, al fine di valutarne lo stato di salute ed individuare diverse anomalie cardiache, patologie oppure aritmie.
Per il progetto è stato necessario trovare, attraverso la programmazione genetica, un **filtro** che trasformi il segnale in input in un altro segnale tramite **convoluzione**. L'obiettivo è quindi trovare un kernel in grado di trasformare i segnali nel dataset, in modo che il risultato della successiva classificazione col Random Forest sia migliore rispetto al risultato ottenuto sui dati non trasformati. La metrica di valutazione utilizzata è la score F1.

In generale, il progetto è stato realizzato in Python, con il supporto di varie librerie. Per l'utilizzo del classificatore Random Forest, e per il calcolo di tutte le metriche necessarie per effettuare il confronto, è stata utilizzata la libreria **sklearn**. Invece, per realizzare l'algoritmo di programmazione genetica è stata utilzzata la libreria **DEAP**.

2 Dataset

Come detto, il dataset utilizzato riguarda dei segnali temporali di **ECG**. Il dataset contiene all'interno segnali relativi a singoli battiti cardiaci. In precedenza, per mantenere una lunghezza fissa predefinita per tutte le porzioni selezionate, queste sono state riempite con zeri. Il dataset è disponibile in un file CSV, in cui nella prima colonna è stata inserita la **label** della classe e il resto delle colonne rappresenta la lunghezza del **segnale**, riempita con zeri per ottenere una lunghezza fissa.

Il dataset è presente in due versioni: una versione più corta, contenente solamente 3000 elementi che è stato utilizzato per rendere più rapide le esecuzioni e quindi la scelta della migliore dimensione del kernel, ed una versione completa. Il dataset completo è caratterizzato da **6000 elementi**, inizialmente etichettati in due **classi**:

- **Normali**: label 0.0. Questi segnali si riferiscono ad elettrocardiogrammi normali, quindi senza nessuna anomalia cardiaca, patologia oppure aritmia.

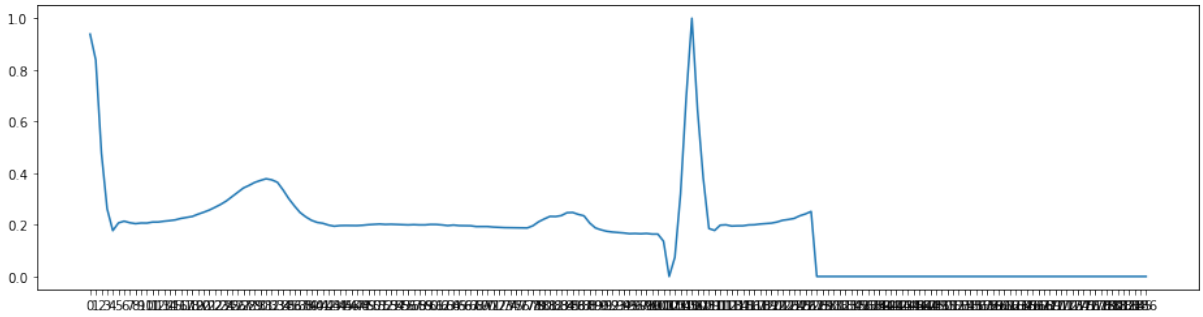

Figura 1: Esempio segnale ECG normale.

- **Anomali**: label 1.0. Questi segnali, invece, riportano elettrocardiogrammi anomali, caratterizzati quindi dalla presenza di possibili anomalie cardiache, patologie o aritmie.

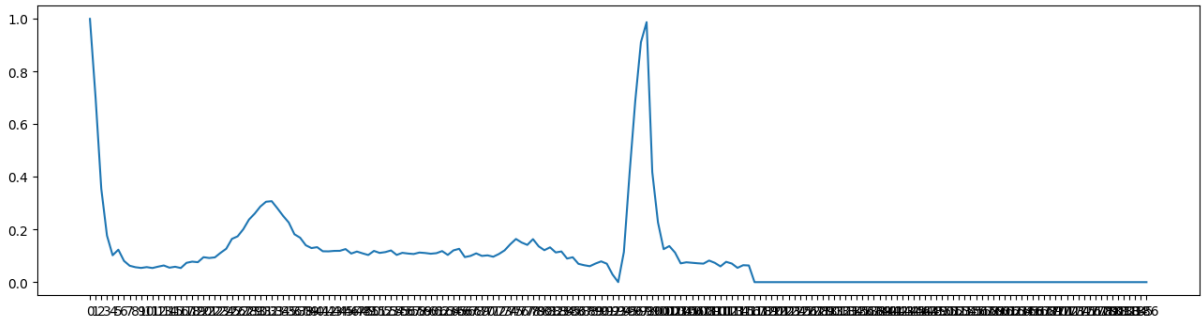

Figura 2: Esempio segnale ECG anomalo.

3 Classificatore Random Forest

Per poter valutare la bontà dell'algoritmo di programmazione genetica è stato necessario un **confronto** con un classificatore **Random Forest**. Per eliminare la componente di randomicità di questo classificatore è stato scelto all'inizio un seed_state = 1200, che poi è stato utilizzato in ogni Random Forest del progetto.
Quindi, il dataset è stato suddiviso in **train-set** e **test-set** tramite la funzione:

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
    random_state=seed_state)
```

Successivamente, il classificatore Random Forest è stato addestrato sul train-set e le performance sono state valutate sui risultati ottenuti sul test-set:

```
clf = RandomForestClassifier(n_estimators=100, random_state=seed_state)
clf.fit(X_train, y_train)
predictions = clf.predict(X_test)
```

3.1 Risultati Ottenuti Random Forest

Nel dettaglio, questo classificatore ha raggiunto, sul dataset da 6000 elementi, la **score F1** pari a 0.9534. Le performance del Random Forest possono essere visualizzate anche tramite le seguenti *matrici di confusione*:

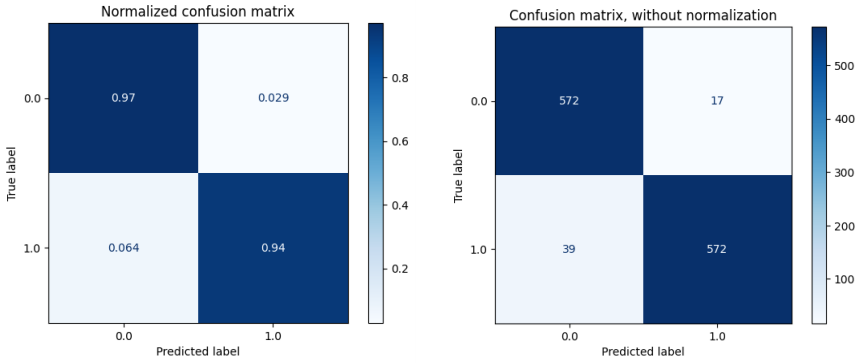

Figura 3: Matrici di confusione test-set Random Forest.

Inoltre, è stato anche valutato il *valore di AUC*, che è risultato essere pari a 0.9537, associato alla seguente *curva ROC:*

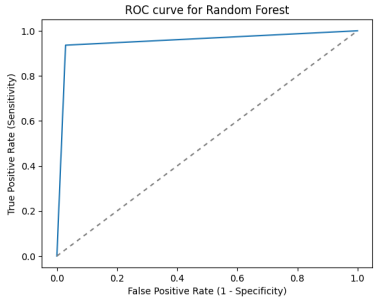

Figura 4: Curva ROC test-set Random Forest.

4 Algoritmo di Programmazione Genetica

Per trovare il miglior filtro convolutivo e verificare se quindi fosse possibile ottenere performance migliori rispetto al classificatore Random Forest, è stata utilizzata la **programmazione genetica**.
Nella programmazione genetica ogni individuo risultante dall'evoluzione può essere visto come una funzione che prende un certo numero di parametri in input e produce un output. In questo caso, l'obiettivo era avere un programma genetico in grado di generare individui con un numero di parametri di input corrispondente alla dimensione del kernel e che potesse generare un singolo valore come output. In generale, quindi, il punto di partenza è stato trovare la dimensione ottimale del kernel, caratterizzata dalla variabile KERNEL_SIZE.

Per capire quale fosse la **dimensione ottimale del kernel**, l'algoritmo di programmazione genetica, che verrà descritto successivamene, è stato addestrato utilizzando il dataset ridotto, ovvero quello caratterizzato da solamente 3000 elementi, per rendere le operazioni più rapide. Anche in questo caso il confronto è stato fatto in base alla metrica score F1 ottenuta da un classificatore Random Forest, addestrato sullo stesso train-set. La score F1 ottenuta dal classificatore Random Forest sul dataset ridotto è pari a 0.9389.
Al seguito di alcuni tentativi, riportati nella tabella, è risultato che la dimensione del kernel migliore per apportare una convoluzione su questo segnale fosse una **dimensione = 6**, seguito da quello con **dimensione = 7**.

Dimensione Kernel	Score F1
4	0.9294
5	0.9299
6	0.9458
7	0.9449
8	0.9424
9	0.9341
12	0.9308

Tabella 1: F1 finale in base a dimensione kernel

Per quanto riguarda la creazione dell'algoritmo di programmazione genetica, il primo passo cruciale consiste nell'inizializzare il set di primitive. Ogni **primitiva** nel set è caratterizzata da una funzione associata, che viene richiamata quando il nodo corrispondente viene valutato, e da una specifica cardinalità, cioè il numero di parametri che accetta come input. Inoltre, le funzioni generate dal set di primitive richiedono un numero di argomenti pari a KERNEL_SIZE (dimensione del kernel) per essere valutate correttamente.
Il set di primitive è stato quindi definito nel seguente modo:

```
pset = gp.PrimitiveSet("MAIN", KERNEL_SIZE)
pset.addPrimitive(operator.add, 2)
pset.addPrimitive(operator.sub, 2)
pset.addPrimitive(operator.neg, 1)
pset.addPrimitive(operator.mul, 2)
pset.addPrimitive(protectedDiv, 2)
```

Dopo aver specificato il set di primitive, è stato quindi necessario definire il **toolbox**, che fornisce un insieme di metodi per registrare e definire le operazioni fondamentali dell'algoritmo di programmazione genetica. Tra le altre cose, è stato necessario specificare la funzione utilizzata per valutare gli individui:

```
def evaluate_fitness(individual):
    fitness, rf = fitness_function(individual)
    return fitness,
toolbox.register("evaluate", evaluate_fitness)
```

La funzione *fitness_function* è responsabile di valutare e assegnare a ciascun individuo della popolazione un punteggio, chiamato **fitness**. Per valutare la bontà dell'algoritmo, è stato necessario dividere il train-set, utilizzato anche per il classificatore Random Forest, in **train-set** e **validation-set**. Nella funzione di valutazione, il processo inizia prendendo come input l'individuo da valutare, che rappresenta una possibile soluzione al problema. Questo individuo viene compilato per ottenerne la funzione e poi utilizzato per generare, tramite convoluzione, un nuovo train-set usato per addestrare il modello di classificazione Random Forest, e un nuovo validation-set su cui viene poi valutato poi il modello addestrato.

4.1 Convoluzione

La principale operazione effettuata dall'algoritmo riguarda, quindi, la **convoluzione**, necessaria per ottenere i nuovi set di segnali, *conv_train* e *conv_val.* Infatti, all'interno della funzione *fitness_function*, l'addestramento del classificatore Random Forest avviene solamente dopo aver applicato la convoluzione al train-set e al validation-set.
La convoluzione viene applicata **ad ogni riga** dei set, utilizzando la funzione generata compilando l'individuo (*func*). In particolare, ad ogni riga del train-set e del validation-set viene applicata la funzione *apply_convolution_to_row*:

```
def apply_convolution_to_row(row, func):
    num_windows = len(row) - KERNEL_SIZE + 1
    result = []
    for i in range(num_windows):
        window = row[i:i + KERNEL_SIZE]
        valore = func(*window)
        result.append(valore)
    return result
```

Nel dettaglio, per ogni riga del dataset vengono considerate tutte le finestre di dimensione pari a KERNEL_SIZE. Il numero di finestre possibili in una riga è quindi pari a len(riga) - KERNEL_SIZE + 1. Per ogni finestra, i KERNEL_SIZE valori della riga considerati saranno i parametri della funzione generata compilando l'individuo. Il segnale restituito da questa funzione farà parte del nuovo train-set o del nuovo validation-set.
Una volta trasformati i segnali, il classificatore Random Forest viene addestato sul nuovo train-set e valutato usando il nuovo validation-set:

```
rf = RandomForestClassifier(n_estimators=100, random_state=seed_state)
rf.fit(conv_train, y_train)
gp_predictions = rf.predict(conv_val)
```

Per ogni individuo viene restituito il suo valore di **fitness** (score F1), insieme al classificatore.

Successivamente, il training sul dataset completo è stato effettuato utilizzando le migliori dimensioni trovate per il kernel, ossia dimensione 6 e dimensione 7, come descritto in precedenza. Quindi, l'algoritmo è stato addestrato con il dataset completo provando ad utilizzare entrambe le dimensioni. É risultato **migliore**, in base alla score F1, il **kernel** con **dimensione 7**. Le statistiche riscontrate durante l'addestramento, con 30 generazioni e KERNEL_SIZE = 7 sono state le seguenti:

gen	nevals	avg	std	min	max
0	50	0.875426	0.221695	0	0.9488
1	30	0.917923	0.131727	0	0.952649
2	34	0.934178	0.0164277	0.88523	0.953664
4	28	0.943636	0.0113274	0.906395	0.958458
5	23	0.947509	0.00923054	0.917554	0.958458
6	32	0.944594	0.0144717	0.873174	0.958458
7	33	0.944759	0.0137463	0.900798	0.960596
8	26	0.94494	0.0146221	0.894957	0.960596
9	36	0.949022	0.0102455	0.914495	0.963177
10	34	0.944871	0.0163191	0.874628	0.963177
11	34	0.947093	0.0131162	0.90618	0.963177
12	32	0.949812	0.0128095	0.882511	0.965099
13	25	0.951422	0.0135386	0.878613	0.965099
14	31	0.950622	0.0122981	0.896503	0.965099
15	33	0.952564	0.0117151	0.910163	0.966161
16	32	0.948942	0.0202464	0.856037	0.965099
17	33	0.952542	0.0132219	0.912287	0.965353
18	38	0.953177	0.0130451	0.902799	0.967639
19	34	0.954653	0.0137541	0.905907	0.967639
20	35	0.956145	0.0104001	0.90958	0.966318
21	35	0.952009	0.0164417	0.88417	0.966318
22	27	0.955233	0.0120333	0.918286	0.966318
23	19	0.958544	0.00780572	0.931896	0.966307
24	33	0.953826	0.0176506	0.860224	0.967173
25	38	0.953738	0.0117457	0.915649	0.966307
26	33	0.952571	0.0156954	0.878306	0.965611
27	39	0.953225	0.00857354	0.926859	0.965611
28	26	0.953232	0.0118469	0.904863	0.965611
29	32	0.951907	0.0123881	0.902665	0.965611
30	32	0.954407	0.00858491	0.928048	0.965611

Figura 5: Statistiche fitness 30 generazioni

Quindi, al seguito della fase di training, è avvenuta la fase di testing. La fase di **testing** è stata effettuata sul test-set, per valutare la performance dell'algoritmo, ed in questo caso del miglior kernel trovato, su dati mai visti prima. Infatti, dopo aver selezionato l'individuo migliore, è stato necessario applicare la convoluzione anche sul test-set. Per accedere all'individuo e al kernel migliori trovati dall'algoritmo di programmazione genetica, è stata utilizzata la **HallOfFame** di DEAP, che tiene traccia delle migliori soluzioni trovate durante l'evoluzione.

```
hof = tools.HallOfFame(1)
best_individual = hof[0]
best_fitness, best_rf = fitness_function(best_individual)
best_kernel_function = gp.compile(gp.PrimitiveTree(best_individual), pset)
```

La convoluzione sul **test-set** è stata applicata utilizzando, come in fase di addestramento, la funzione *apply_convolution_to_row* applicata ad ogni riga, ed utilizzando come funzione quella generata compilando l'individuo migliore (*best_individual*).

4.2 Risultati Ottenuti

Quindi, i risultati migliori in base alla score F1 e quindi alla fitness del miglior individuo sono stati riscontrati con il kernel di **dimensione 7**. Infatti, con il kernel di dimensione 6 la score F1 finale è risultata pari a 0.9668, mentre con il kernel di dimensione 7 è stata pari a 0.9696. Si può notare in entrambi i casi un miglioramento rispetto al classificatore Random Forest originale.
Le performance sul test-set, con un filtro convolutivo di dimensione 7, possono essere visualizzate tramite le seguenti *matrici di confusione*:

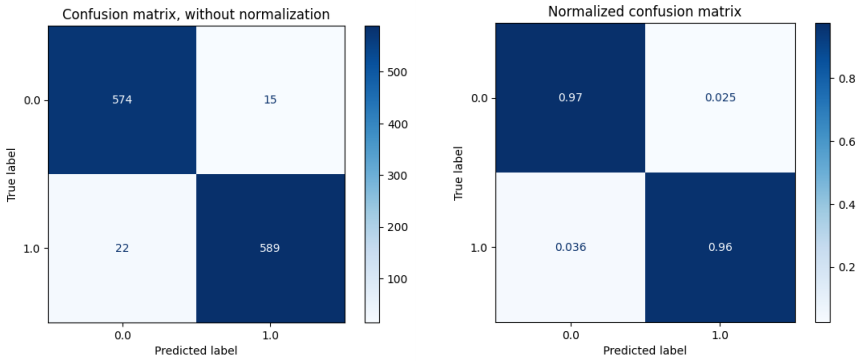

Figura 6: Matrici di confusione test-set dopo convoluzione.

Inoltre, è stato anche valutato il *valore di AUC*, che è risultato essere pari a 0.9693, maggiore anch'esso rispetto al classificatore Random Forest, associato alla seguente *curva ROC:*

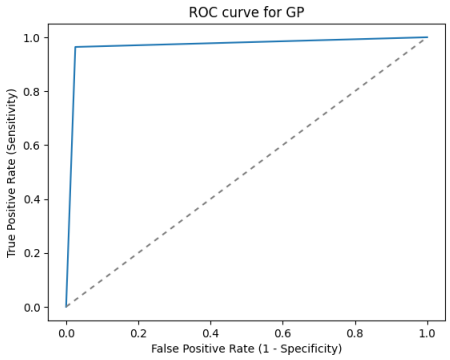

Figura 7: Curva ROC test-set dopo convoluzione.

Nel complesso, il **miglior individuo** trovato tramite programmazione genetica applica le seguenti operazioni al segnale in ingresso:

```
add(sub(sub(ARG5, ARG1), neg(mul(sub(sub(sub(ARG5, ARG1), neg(sub(sub(ARG5
    , ARG1), neg(ARG5)))), ARG6), sub(ARG0, ARG3)))), sub(add(mul(mul(ARG6,
     ARG6), mul(neg(ARG5), mul(ARG3, ARG0))), mul(mul(ARG1, ARG3), mul(add(
    ARG3, ARG2), ARG5))), mul(ARG0, ARG2)))
```

Vengono riportati nelle seguenti figure due esempi di **segnale** di ECG originale dopo la convoluzione con il miglior kernel trovato:

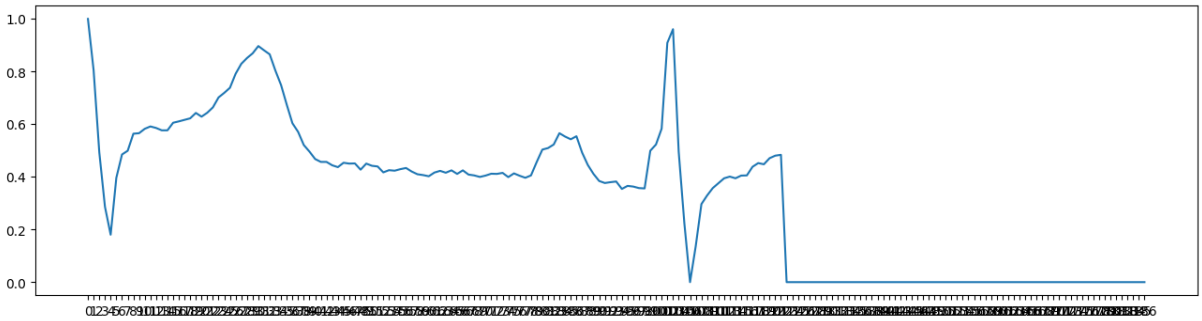

Figura 8: Esempio segnale ECG anomalo prima della convoluzione

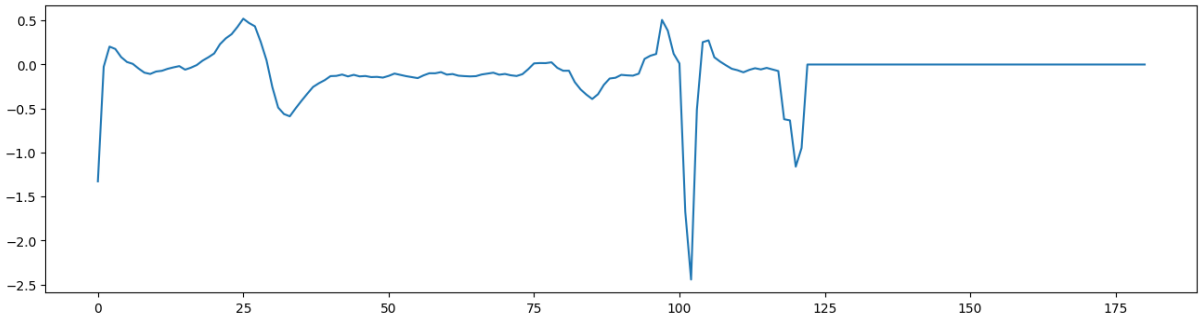

Figura 9: Esempio segnale ECG anomalo dopo la convoluzione

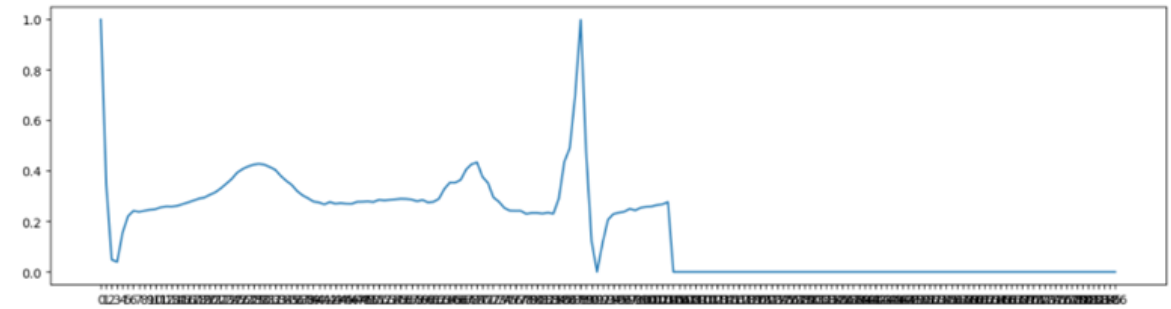

Figura 10: Esempio segnale ECG normale prima della convoluzione

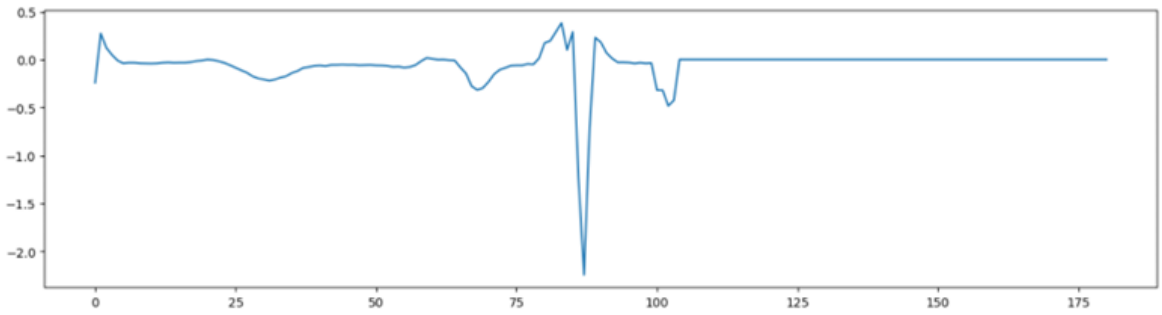

Figura 11: Esempio segnale ECG normale dopo la convoluzione

5 Conclusioni

In conclusione, quindi, sembra che effettivamente il miglior kernel con dimensione = 7 trovato dall'algoritmo di programmazione genetica apporti dei **miglioramenti** rispetto al classificatore Random Forest di base. Infatti, l'utilizzo della programmazione genetica, e successivamente l'applicazione del filtro convolutivo al segnale, ha permesso di passare da un valore di **score F1** pari a 0.9534 ad uno pari a **0.9696**.

In generale, il kernel trovato dall'algoritmo genetico permette di applicare una convoluzione sul segnale ECG in input che permette ad un classificatore Random Forest di avere **prestazioni migliori sulla classificazione** e in particolare, come si può notare dalle matrici di confusione riportate in precedenza, permette ad esso di avere una precisione maggiore nel riconoscimento dei segnali che riguardano elettrocardiogrammi anomali.