DAVIDE ORENGO - S4322441

GIORGIO DEMARZI - S4227621

Descrizione modelli di machine learning per prevedere il Freezing of Gait (FOG)

Lo scopo di questo elaborato è quello di illustrare la realizzazione di modelli di machine learning utilizzando il Daphnet Freezing of Gait Dataset [1] per riconoscere il Freezing of Gait (FOG)[2]. Questo set di dati contiene letture di tre sensori di accelerazione all'anca, alla gamba e al tronco dei pazienti affetti da malattia di Parkinson che subiscono una breve ed imprevedibile impossibilità di camminare (FOG) durante le attività di deambulazione.

Il progetto è stato sviluppato utilizzando i moduli *sklearn* della libreria *scikit-learn*, *tensorflow* e *keras* di Python per eseguire algoritmi di machine learning. Le simulazioni sono state effettuate utilizzando l'ambiente Google Colab.

Descrizione dataset

Il dataset comprende un insieme di file relativi ad un totale di dieci pazienti (definiti come S01R01, S01R02, S02R01,..ecc) alcuni di questi (l'1, il 2 ed il 5) presentano due trials. Ogni file comprende i dati in un formato matriciale: una riga rappresenta un campione e le colonne le letture dei sensori. Ogni paziente viene fatto camminare lungo un percorso generico durante il quale vengono registrati tutti i suoi movimenti attraverso i sensori che indossa. Le osservazioni sono state fatte con una frequenza di 66Hz

I valori sono i seguenti:

- Accelerazione della caviglia (stinco) - accelerazione orizzontale in avanti [mg]
- Accelerazione della caviglia (stinco) - verticale [mg]
- Accelerazione della caviglia (stinco) - orizzontale [mg]
- Accelerazione della parte superiore della gamba (coscia) - accelerazione orizzontale in avanti [mg]
- Accelerazione della gamba (coscia) - verticale [mg]
- Accelerazione della gamba (coscia) - laterale orizzontale [mg]
- Accelerazione del tronco - accelerazione orizzontale in avanti [mg]
- Accelerazione del tronco - verticale [mg]
- Accelerazione del tronco - laterale orizzontale [mg]
- Etichetta [0, 1 o 2]

Il significato delle etichette è per specificare in quale fase del test è stato rilevato il campione:

- 0: Il campione non fa parte dell'esperimento. I sensori sono installati sull'utente ma sta eseguendo attività non correlate al protocollo sperimentale
- 1: Il campione fa parte dell'esperimento. L'utente non presenta FOG e può essere in qualsiasi posizione
- 2: Il campione rappresenta un evento di FOG

Stato dell’arte

Il problema del riconoscimento degli eventi FOG è stato già studiato in due articoli che abbiamo preso come riferimento e punto di partenza per l’applicazione dei nostri modelli. Il primo articolo nella letteratura risale al 2010 [2] e spiega il modo in cui il dataset è stato generato e studiato per la prima volta impiegando metodi statistici e di analisi frequenziale (FFT) sia online che a posteriori. L’articolo [2] evidenzia che gli eventi di FOG durano ai 0.5 a 40.5 secondi con una media di 7.3 secondi e una maggioranza (93.2%) sotto i 20 secondi. I ricercatori riportano inoltre che il sistema utilizzato non funziona bene allo stesso modo con tutti i pazienti a causa dei diversi modi di camminare, ma globalmente hanno ottenuto una sensitività del 73.1% e specificità del 81.6%. Questi salgono globalmente a 78.1% e 86.9% quando i parametri del sistema vengono ottimizzati per ogni singolo paziente.

In letteratura è presente un secondo articolo risalente al 2020 [3] in cui ricercatori applicano per la prima volta algoritmi di machine learning standard sul medesimo dataset. Per mezzo di metodi *ensemble* e PCA il migliore rate di successo è stato del 86.6% per tutti i pazienti con una media di 81.52% (77% in letteratura). I ricercatori evidenziano che l’utilizzo di PCA e feature extraction non danno significativi miglioramenti. I migliori risultati raggiunti in termini di sensitività e specificità media sono rispettivamente di 91.9% e 71.14%. Inoltre si evidenzia anche qui che si ottengono dei miglioramenti se si selezionano singoli pazienti.

Metodologia adottata

Per poter cominciare ad utilizzare metodi di machine learning è necessario fare delle considerazioni iniziali riguardo il dataset utilizzato.

Al fine di ottenere un migliore riconoscimento degli eventi di FOG, infatti è essenziale evitare di avere un numero eccessivo di casi negativi perché altrimenti il modello memorizza caratteristiche specifiche dei dati di training, ma non è sufficientemente flessibile per eseguire stime sui dati di testing e quindi eventualmente incapace di rilevare dei casi di FOG.

I dataset S03R03 S04R01 S06R02 S10R01 non contengono casi di FOG e quindi per questi motivi non sono stati considerati. Analogamente S03R02 e S07R02 vengono esclusi perché buona parte dei sample (>70%) sono tutti negativi.

Prima di inoltrarci ulteriormente sull'argomento è necessario definire alcuni parametri che verranno utilizzati per verificare la bontà dei risultati ottenuti. Fra i più importanti vi sono i concetti di vero positivo, falso positivo, vero negativo, falso negativo:

- Vero positivo (*true positive - TP*): è il caso di una classificazione corretta. Ad esempio quando si diagnostica una malattia ad un paziente che è realmente affetto da quella malattia;
- Vero negativo (*true negative - TN*): è il caso di una classificazione corretta dell’evento negativo. Ad esempio nella diagnosi per un paziente una certa malattia viene esclusa e realmente il paziente non soffre di quella malattia;
- Falso positivo (*false positive - FP*): è il caso di una classificazione non corretta dell’evento positivo. Ad esempio quando si diagnostica ad un paziente una certa malattia, ma il paziente non è realmente affetto da quella malattia;

- Falso negativo (*false negative - FN*): è il caso di una classificazione non corretta. Ad esempio quando non si diagnostica ad un paziente una certa malattia, ma il paziente è realmente affetto da quella malattia;

Sensibility, specificity e accuracy sono descritte in termini di TP, TN, FN e FP.

- Sensibility = TP/(TP + FN) = (Numero di vere valutazioni positive)/(Numero di tutte le valutazioni positive)

  La sensibility è la misura di quanto un classificatore si sia dimostrato valido nel trovare gli attuali casi positivi.

- Specificity = TN/(TN + FP) = (Numero di valutazioni negative reali)/(Numero di tutte le valutazioni negative)

  La specificity è la misura di quanto il nostro classificatore si sia dimostrato abile nel classificare i casi negativi.

- Accuracy = (TN + TP)/(TN + TP + FN + FP) = (Numero di valutazioni corrette)/(Numero di tutte le valutazioni)

  L'accuratezza è la somma delle classificazioni corrette diviso per il numero totale dei casi

Questi parametri permettono di costruire un ulteriore strumento utile per l'analisi: La matrice di confusione.

Una matrice di confusione è una tabella che viene utilizzata per descrivere le prestazioni di un modello di classificazione su un insieme di dati di prova per i quali sono noti i valori reali. La parola "confusione" nel nome deriva semplicemente dal fatto che un modello "confonde" o etichetta gli esempi in modo certo.

Lungo la prima riga della tabella troviamo il totale degli individui negativi: i veri negativi (TN) ed i falsi positivi (FP). Nella seconda riga troviamo il totale degli individui positivi: la somma dei casi falsi negativi (FN) e veri positivi (TP).

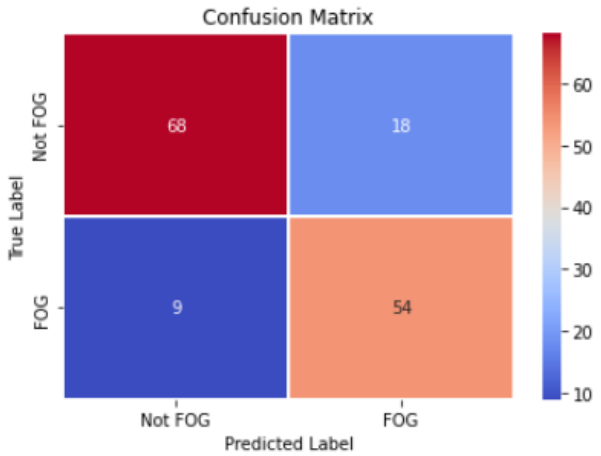

Figura 1: Esempio matrice confusione

La matrice di confusione di un modello valido avrà la maggior parte dei campioni lungo la diagonale.

K-NN

K-NN (K-Nearest Neighbors) è un algoritmo di classificazione in Machine Learning. Appartiene al dominio dei metodi di apprendimento supervisionato e trova un'intensa applicazione nel riconoscimento di pattern, data mining e rilevamento delle intrusioni. È ampiamente applicabile negli scenari di vita reale poiché non è parametrico, cioè non fa alcuna ipotesi di base sulla distribuzione dei dati. L'idea alla base dell'algoritmo è la seguente: se un oggetto A ha caratteristiche simili a un oggetto B, probabilmente A appartiene alla stessa classe di B.

Inizialmente lo spazio dei punti viene partizionato in regioni in base alle posizioni e alle caratteristiche degli oggetti di apprendimento. Ai fini del calcolo della distanza degli elementi sono rappresentati attraverso vettori di posizione in uno spazio multidimensionale. Di solito viene usata la distanza euclidea, ma anche altri tipi di distanza sono ugualmente utilizzabili, ad esempio la distanza Manhattan. In questo caso specifico è stata utilizzata una distanza euclidea:

$$d(x, x') = \sqrt{(x_1 - x_1')^2 + (x_2 - x_2')^2 + \cdots + (x_n - x_n')^2}$$

Un punto è quindi assegnato ad una certa classe se questa è la più frequente fra i k esempi più vicini all'oggetto sotto esame, la vicinanza si misura in base alla distanza fra punti. I vicini sono presi da un insieme di oggetti per cui è nota la classificazione corretta.

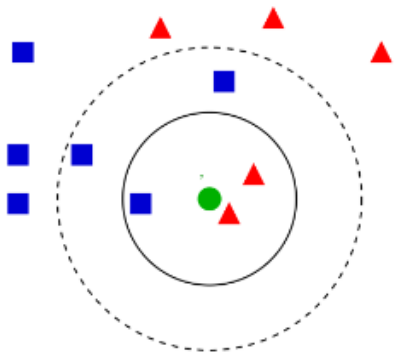

Figura 2: k-esima divisione per la classificazione k-nn di un spazio contenente elementi

La scelta di k è importante e dipende dalle caratteristiche dei dati. La scelta di k dipende dalle caratteristiche dei dati. Generalmente all'aumentare di k si riduce il rumore che compromette la classificazione, ma il criterio di scelta per la classe diventa più labile. Per questo motivo è preferibile utilizzare delle varianti della K-NN tradizionale.

Un altro problema è l'approccio che permette di combinare le etichette delle classi. Il metodo più semplice è considerare il voto di maggioranza, ma questo può essere un problema se la distanza dei nearest neighbors varia significativamente e i neighbors più vicini indicano in modo più affidabile la classe dell'oggetto.

Per superare questi svantaggi, si utilizza la Weighted K-NN. In questo caso ai punti k vicini viene dato un peso utilizzando una funzione chiamata funzione kernel. L'intuizione alla base di questo metodo è quella di dare più peso ai punti vicini e meno peso ai punti lontani. Qualsiasi funzione può essere usata come funzione kernel per il classificatore di Weighted K-NN il cui valore deve diminuire con l'aumentare della distanza. La funzione utilizzata è la reverse square.

Perché K-NN

I motivi che ci hanno spinto a selezionare questo metodo di machine learning per la realizzazione del modello sono legati alle sue caratteristiche. Il K-NN è infatti un algoritmo di apprendimento automatico efficace, intuitivo, relativamente semplice da implementare e che si adatta immediatamente ai nuovi dati di training. Si tratta di un metodo particolarmente adatto alla tipologia di dataset supervised considerato senza nessuna necessità di formulare ipotesi aggiuntive.

Un'altra caratteristica importante riguarda la varietà di metriche di distanza utilizzabili. La flessibilità da parte degli utenti per utilizzare una metrica di distanza più adatta alla loro applicazione. Come già evidenziato in questo caso è stata utilizzata la distanza euclidea che ha permesso i risultati migliori.

Implementazione K-NN

Per implementare l'algoritmo di K-NN è stata utilizzata una libreria Python chiamata pyts, che permette di implementare l'algoritmo in poche righe con le caratteristiche richieste.

```
def train_knn(X_train, Y_train, n):
  clf = KNeighborsClassifier(metric='minkowski', p=2,  n_neighbors = n, weights=reverseSquare)
  clf.fit(X_train.values, Y_train.values)
  return clf
```

Figura 3

In particolare i parametri all'interno della funzione KNeighborsClassifier contengono informazioni sulla distanza (metric e p, relativi alla distanza euclidea), sui neighbors e sulla tipologia della funzione utilizzata per i pesi (weights).

Un passo successivo importante riguarda la divisione dei dati divisi in due set: uno contenente il 70% e un altro il 30% dei dati di origine, rispettivamente per il training e il testing del modello.

Questi due insiemi di dati saranno processati in due cicli for analoghi per permettere al nostro modello di imparare le relazioni tra le nostre variabili di input e la nostra etichetta che vogliamo predire (FOG/Not FOG) ed infine poterlo testare su un altro set di dati sconosciuto.

```
for file_name in file_list:
  df = pd.read_csv('gdrive/My Drive/dataset/'+file_name+'.txt')
  s = df['N'] != 0
  tmp = df.loc[s]

  n_row = tmp.shape[0]
  up = round(n_row * 0.7)
  down = n_row - up

  train_tmp = tmp.head(up)
  #test_tmp = tmp.tail(down)

  li.append(train_tmp)


training_dataframe = pd.concat(li, axis=0, ignore_index=True)

X_train = training_dataframe[["A0", "A1", "A2", "U0", "U1", "U2", "T1", "T2", "T3"]]
Y_train = training_dataframe[["N"]]

#Parametri
N = 4 #neighbors

#Training
model = train_knn(X_train, Y_train, int(N))
```

Figura 4: Blocco di codice del training del K-NN

```
for file_name in file_list:
  df = pd.read_csv('gdrive/My Drive/dataset/'+file_name+'.txt')
  s = df['N'] != 0
  tmp = df.loc[s]

  n_row = tmp.shape[0]
  up = round(n_row * 0.7)
  down = n_row - up

  test_tmp = tmp.tail(down)

  X_test =  test_tmp[["A0", "A1", "A2","U0", "U1", "U2", "T1", "T2", "T3"]]
  Y_test = test_tmp["N"]

  #Prediction
  Y_pred = model.predict(X_test.values)

  sp = calcSpecificity(Y_pred, Y_test.values)
  se = calcSensitivity(Y_pred, Y_test.values)

  sens.append(se)
  spec.append(sp)

  print("Patient: " + file_name + "   " + "Accuracy: " + str(calcAccuracy(Y_pred, Y_test)))
  print("Patient: " + file_name + "   " + "Specificity: " + str(sp))
  print("Patient: " + file_name + "   " + "Sensitivity: " + str(se))
```

Figura 5: Blocco di codice del testing del K-NN

Valutazione modello

Possiamo osservare i risultati ottenuti attraverso questo grafico sensitivity/specificity. Ogni punto blu rappresenta le performance della KNN testata su ogni singolo paziente.

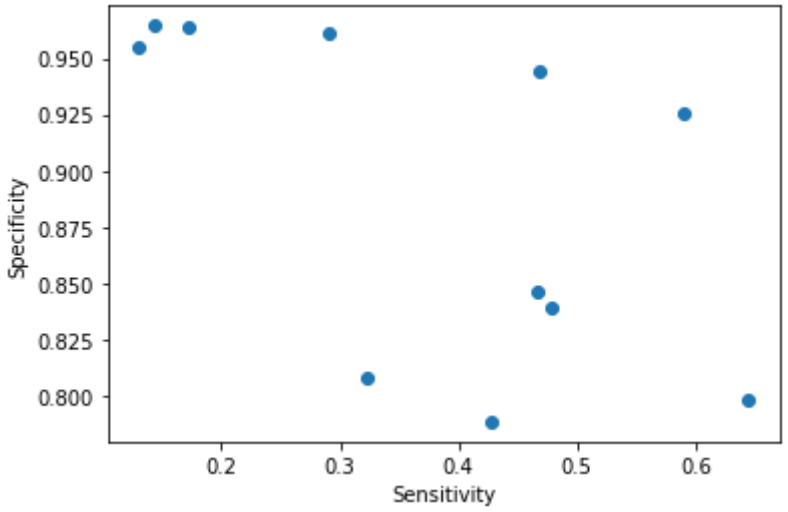

Figura 6: Grafico sensitivity/specificity

Come si può osservare, i risultati evidenziano che in base al paziente i risultati possono variare di molto, inoltre quanto abbiamo ottenuto non è molto aderente con i risultati riportati dalla letteratura. I valori ottenuti con questo dataset hanno purtroppo evidenziato i limiti di questo approccio di machine learning. Inoltre l'algoritmo diventa estremamente lento a causa delle grandi dimensioni del dataset, questo perché è necessario memorizzare tutti i dati di training e vi è una richiesta di memoria elevata.

Un altro elemento che viene evidenziato e che verrà ripreso successivamente riguarda lo sbilanciamento del dataset: la maggior parte dei dati su cui viene eseguito il training del modello è rappresentato dalla etichetta Not FOG, tale etichetta avrà quindi un'alta probabilità di essere prevista. Per questo motivo devono essere presi in considerazione dei metodi per equilibrare il dataset come la decimazione (ovvero selezionare un campione ogni certo numero di righe) oppure il bilanciamento del dataset.

Queste problematiche ci hanno costretto ad utilizzare un approccio alternativo: la Convolutional Neural Network (CNN).

Le Convolutional Neural Network - CNN

In questo contributo applichiamo al problema del riconoscimento degli eventi di FOG una rete neurale convoluzionale. Una CNN, Convolutional Neural Network è un tipo di rete neurale che fa uso di un filtro detto *kernel* che viene fatto "scorrere" su una matrice di dati effettuando una moltiplicazione scalare ottenendo una nuova matrice più piccola detta *feature map*. Questa operazione detta convoluzione può anche essere eseguita più volte nella stessa rete impiegando filtri diversi con lo scopo di estrarre feature differenti. Un filtro differisce da un'altro in base ai valori numerici che contiene e in base al filtro cambia la feature map che si ottiene.

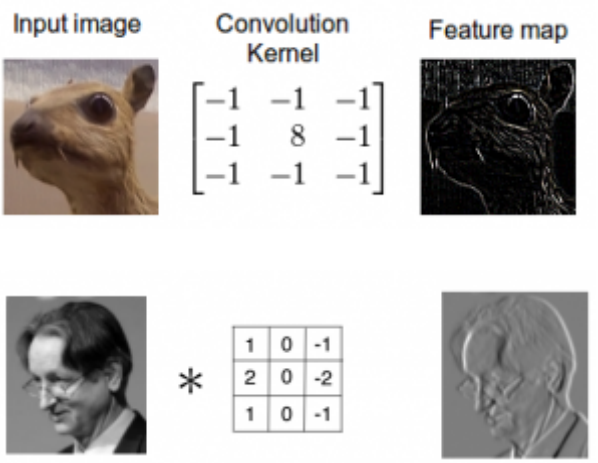

Figura 7

L'immagine sottostante riporta l'architettura classica di una CNN in cui è presente una parte di feature learning e una parte di classificazione. I layer di *pooling* seguono l'operazione di convoluzione e servono a diminuire la dimensione della matrice originale eliminando i dati non più necessari aumentando così il livello di astrazione. Il risultato del processo di feature learning diventa l'input di una rete fully connected la quale sarà in grado di stimare la classe di appartenenza dell'input dato.

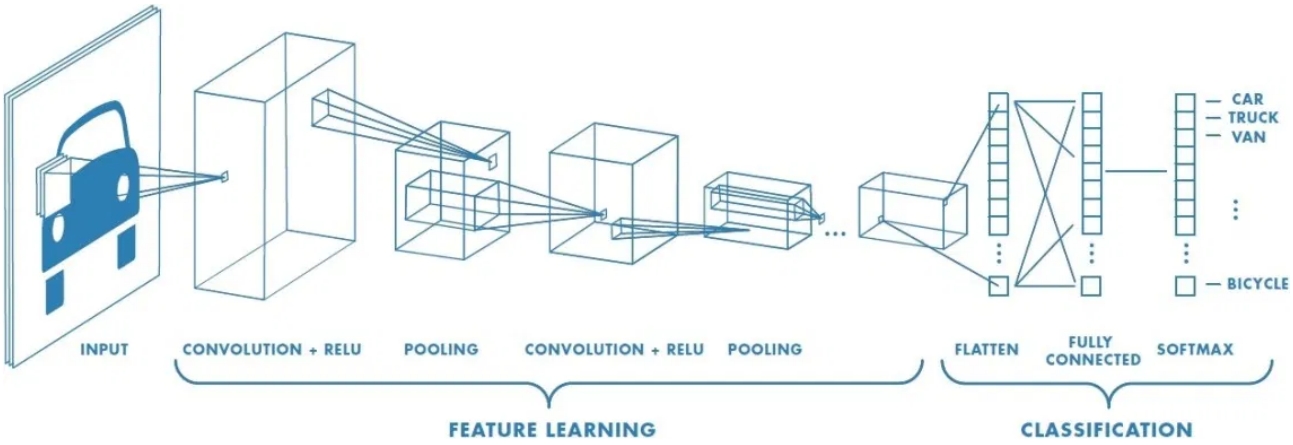

Figura 8

I parametri principali di un layer convoluzionale sono la dimensione della matrice di kernel, il numero di filtri utilizzati, la dimensione di input e il tipo di attivazione che possiede il layer.

Per costruire una rete CNN vanno definiti il numero di layer convoluzionali, il tipo di pooling e le caratteristiche della rete fully connected (numero di layer, dimensione di input, output e tipo di attivazione)

Perchè CNN

Per costruire la rete adatta al nostro problema ci siamo affidati ad un articolo di Jason Brownlee[4] aggiornato al 2020 pubblicato sul suo blog in cui per risolvere il problema di *Human Activity Recognition* applica una CNN ad un dataset[5] di misure prese da 3 sensori di uno smartphone.

La scelta dell'uso di una CNN viene dal fatto che una una rete classica fully connected porta un grandissimo numero di parametri da imparare che fa crescere a dismisura il tempo di addestramento della rete. Inoltre una rete convoluzionale è adatta a riconoscere *pattern* all'interno dei dati grezzi e per questo motivo vengono usate con le immagini perchè la loro rappresentazione in pixel si presta naturalmente alla convoluzione.

Come abbiamo visto nel primo paragrafo il nostro dataset presenta 9 feature che devono essere studiate contemporaneamente con lo scopo di estrarre dei pattern che ci consentano di riconoscere gli stati di FOG. I dati in forma di serie temporale suggeriscono di fare uso di layer convoluzione a una dimensione (1D) piuttosto che i classici 2D usati per le immagini poiché più adatti a scovare features legate temporalmente. La differenza fra i due tipi di convoluzione è evidenziata nelle due immagini sottostanti prese da [6]

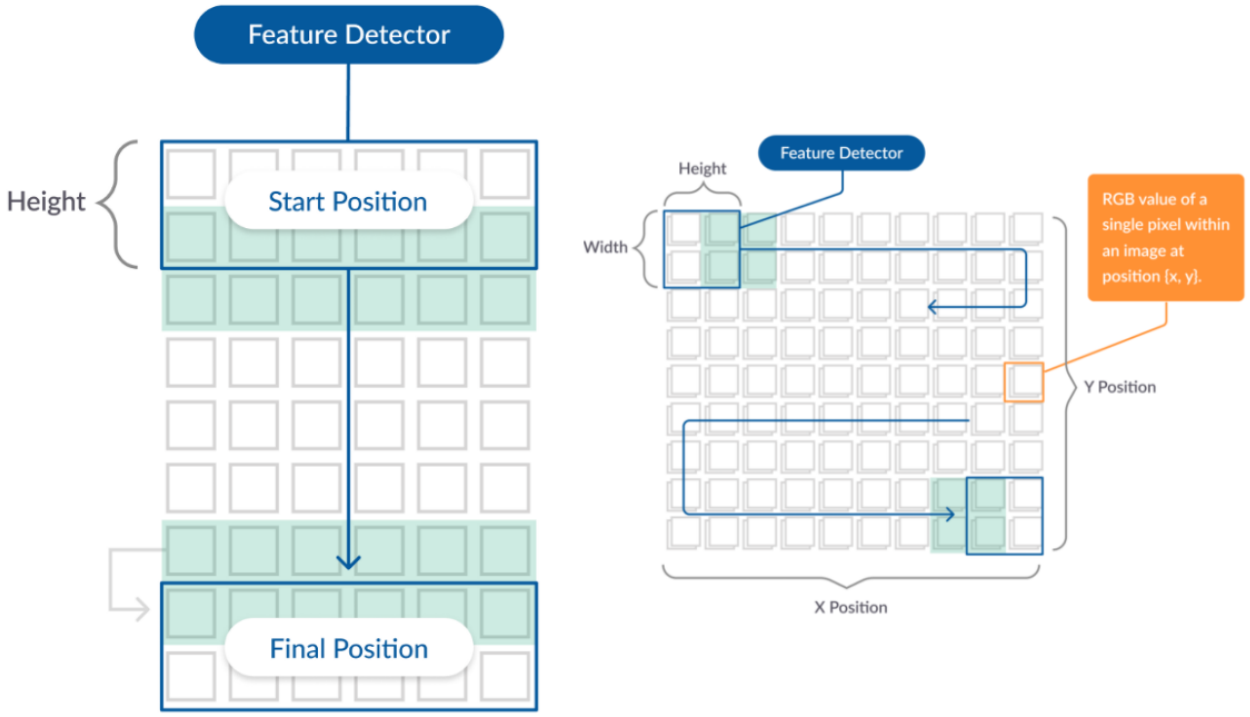

Figura 9

L'immagine a destra mostra un classico esempio di convoluzione 2D come avverrebbe con una immagine, mentre a sinistra un tipo 1D in cui il filtro viene traslato solo verticalmente passando da un istante temporale a quello successivo. L'altezza del filtro identifica il numero di istanti temporali che vengono presi in considerazione per l'estrazione delle feature map, mentre le colonne rappresentano le feature, nel nostro caso le misure dei sensori (9 colonne).

Preprocessing

Il primo passo è caricare il dataset e lo dividiamo in train e test come abbiamo già effettuato nel capitolo sulla K-NN. In seguito dividiamo test e train in finestre temporali poiché è computazionalmente impossibile dare come input alla rete tutte le righe del dataset contemporaneamente. Nel codice sottostante i dati del training vengono suddivisi in finestre non sovrapposte che contengono un numero di campioni pari a *timesteps.*

```
trainX = list()

n_segment = round(len(pre_trainX) / timesteps) - 1
i = 0
for s in range(n_segment):
  segment = pre_trainX.iloc[i:i+timesteps].values
  i += sample_in_segment
  trainX.append(segment)

trainy = list()

i = 0
for s in range(n_segment):
  segment = pre_trainy.iloc[i+timesteps].values
  i += sample_in_segment
  trainy.append(segment)
```

Figura 10

Ad esempio, nel caso in cui si usasse una finestra di 100 campioni, avremo che ogni riga del nuovo dataset corrisponde ad una finestra con 100 elementi ognuno contenente 9 feature. Per forza ogni riga del dataset deve corrispondere ad una sola label per questo la seconda parte del codice si occupa di ridurre le label di uscita.

Implementazione della CNN

Le librerie tensorflow e keras ci forniscono tutti gli strumenti per generare una rete CNN con l'architettura che preferiamo. Lo schema della nostra rete è rappresentato nell'immagine sottostante.

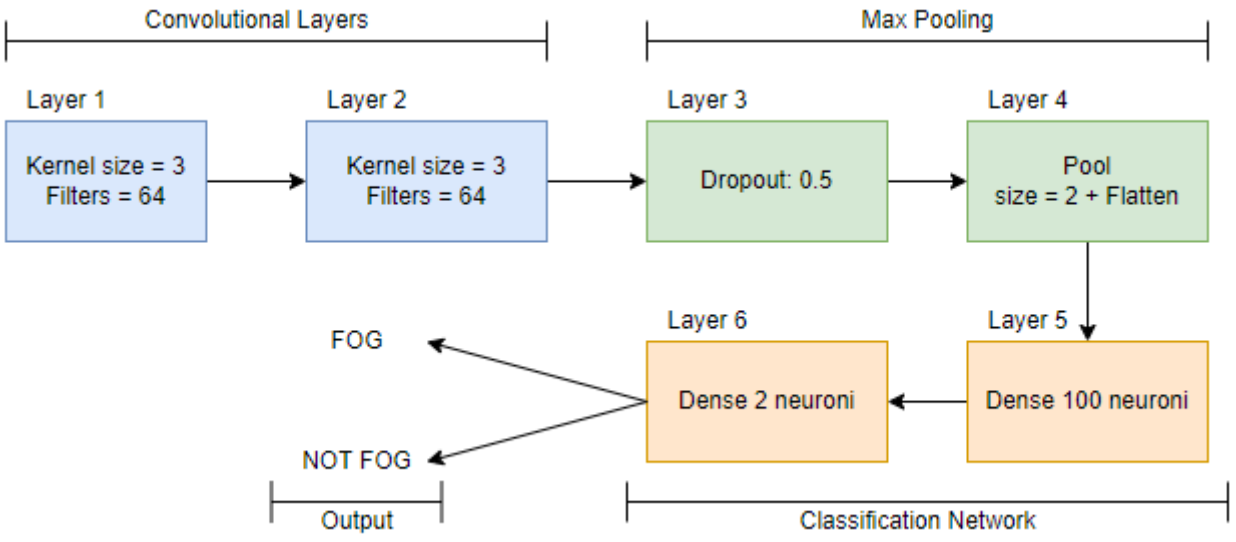

Figura 11

Di seguito il codice che descrive la rete CNN usata.

```
model = Sequential()
model.add(Conv1D(filters=64, kernel_size=3, activation='relu',
                 input_shape=(timesteps,features), name='Layer_1'))
model.add(Conv1D(filters=64, kernel_size=3, activation='relu', name = 'Layer_2'))
model.add(Dropout(0.5))
model.add(MaxPooling1D(pool_size=2))
model.add(Flatten())
model.add(Dense(100, activation='relu'))
model.add(Dense(outputs, activation='softmax'))
model.compile(loss='categorical_crossentropy', optimizer='adam',metrics=['accuracy'])
model.fit(trainX, trainy, epochs=epochs, batch_size=batch_size, verbose=verbose)
```

Figura 12

Inizialmente l'architettura della rete è stata realizzata seguendo l'articolo [4] data la similarità del problema.

Valutazione del modello

Il training della rete viene effettuato mediante la funzione *fit* che prende in ingresso il dataset di training, il numero di epoche e la dimensione dei batch da impiegare. Successivamente si effettua la valutazione del modello mediante la funzione *evaluate* che in ingresso riceve il dataset di test, la dimensione dei batch e ritorna l'accuratezza del modello. Essendoci in gioco meccanismi stocastici nel processo di training della rete la valutazione del modello verrà effettuata facendo 10 cicli fit/evaluate valutando poi l'accuratezza media e la deviazione standard.

Facciamo un primo esperimento con 64 filtri e dimensione variabile del kernel da 3 a 20 creando modelli singoli per ogni paziente. I training e testing set vengono generati dividendo i dataset di ogni paziente in 70% e 30%. L'accuratezza ottenuta dai modelli viene rappresentata nel grafico sottostante.

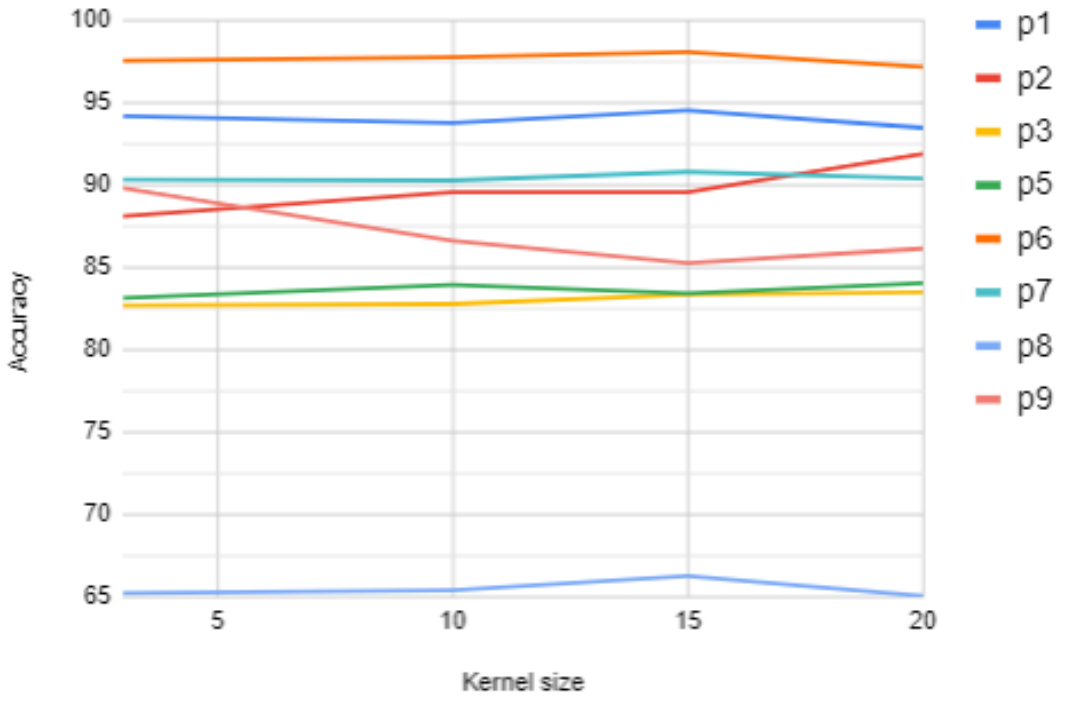

Figura 13

Come si vede dal grafico si può notare che praticamente tutti i modelli hanno una accuratezza pressoché costante al variare della dimensione del kernel, ma con una certa varietà. In particolare il modello del paziente p8 che in [3] viene definito come quello più affetto da parkinson ha una accuratezza di circa 65% contro quella degli altri pazienti che varia fra 82% e 97%. Ad eccezione di quest'ultimo osservando questi risultati potremmo ritenerci soddisfatti dell'accuratezza ottenuta, purtroppo però la situazione cambia se osserviamo i valori di specificità e sensitività.

Osservando le matrici confusione derivanti dai modelli ci accorgiamo immediatamente che nel dataset esiste un forte sbilanciamento fra le label "FOG"e "Not FOG". Notiamo infatti che nonostante l'alta accuratezza i pochi casi di "FOG" rilevati vengono erroneamente scambiati per "Not FOG". Un esempio è riportato nella figura che segue che riporta una matrice confusione per il paziente 3 a sinistra e per il paziente 8 a destra.

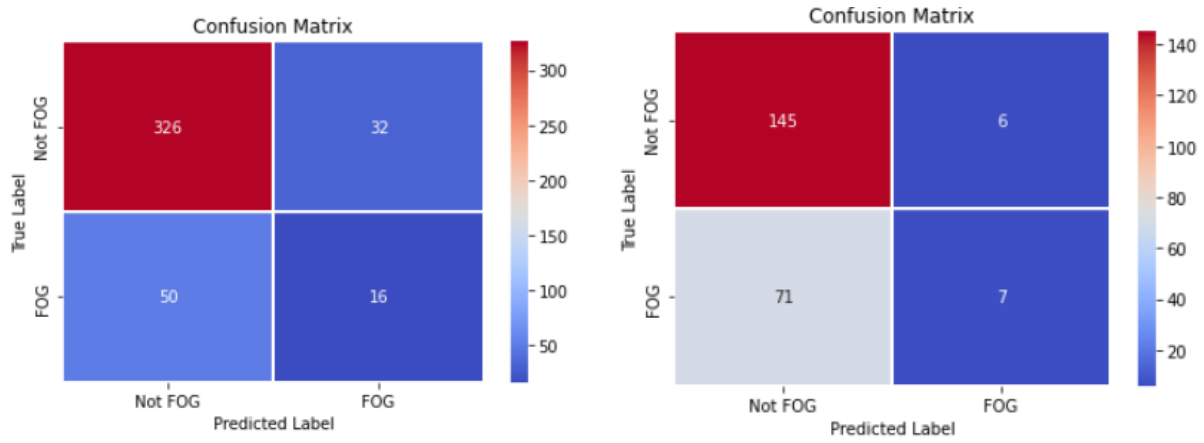

Figura 14: Due matrici confusione estratte durante il training del paziente 3 (a sinistra) e paziente 8 (a destra)

C'è un altissimo numero di predizioni corrette di "Not FOG" il che significa che il dataset presenta un numero di label molto sbilanciato. Di conseguenza la sensitività ha valori mediamente bassi e altalenanti con una varianza davvero alta come riassunto nella tabella seguente.

Sensitivity		
Paziente	%	std
P1	4	6.799
P2	50	20.156
P3	29.444	21.089
P5	37.708	22.194
P6	5	15
P7	0	0
P8	27.949	20.939
P9	66.111	26.926

Il fatto che i casi “FOG” rispetto ai “Not FOG” sono molto minori è una problema importante nella fase di training della CNN. Servirebbe una distribuzione più equilibrata dei casi per permettere all’algoritmo di riconoscere dettagliatamente i due casi.

Bilanciamento del dataset

A questo punto è necessario effettuare un’operazione di bilanciamento del dataset per cercare di equilibrare il numero di casi di “FOG” con quelli di “Not FOG”. L’algoritmo di bilanciamento che abbiamo usato consiste nell'eliminare randomicamente un certo numero di sample “Not FOG” affinchè le due label si eguagliano in numero. Il codice utilizzato è riportato di seguito.

```
FOG = df[df.N == 1].shape[0]
NOT_FOG = df[df.N == 0].shape[0]


if NOT_FOG > FOG:
  da_togliere = NOT_FOG - FOG

  new_df = df.drop(df[df['N'].eq(0)].sample(da_togliere).index)


FOG = new_df[new_df.N == 1].shape[0]
NOT_FOG = new_df[new_df.N == 0].shape[0]
```

Figura 15: Codice utilizzato per il bilanciamento del dataset

Adesso rifacciamo nuovamente i training dei modelli per ogni paziente per ottenere dei risultati più affidabili. Otteniamo le accuratezze al variare del kernel mostrate nel grafico seguente con numero di filtri fissato a 64.

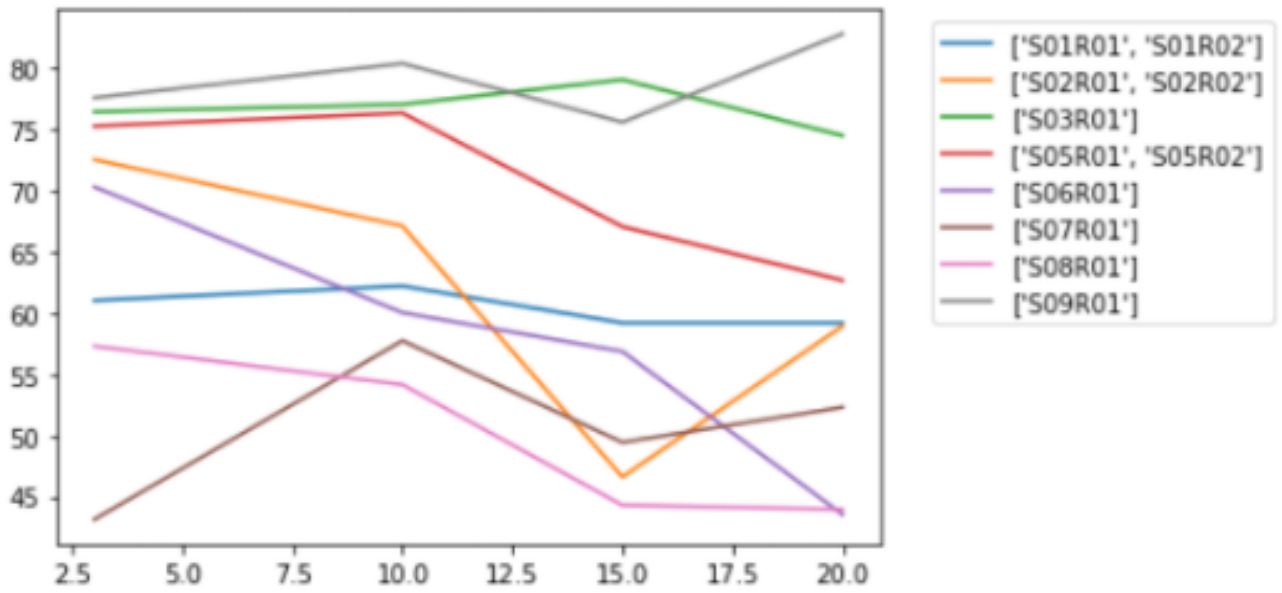

Figura 16: Accuratezza dei modelli per ogni paziente al variare della dimensione kernel da 3 a 20

Notiamo che i valori di accuratezza per i vari modelli vanno dal 44% al 85% in base al paziente, però a differenza dei modelli precedenti al variare del kernel notiamo delle differenze evidenti. Possiamo affermare che con il kernel di dimensione 10 otteniamo i risultati mediamente migliori. Per ottimizzare i modelli si potrebbe scegliere un valore di kernel diverso per ogni caso.

Fissando la dimensione del kernel a 10 possiamo fare nuovi esperimenti modificando il numero di filtri per ogni layer convoluzionale. Di seguito si riportano i risultati ottenuti variando il numero di filtri a step da 4 a 192. E' evidente che le migliori performance si ottengono con un numero di filtri pari a 32. Entrambi gli esperimenti sono stati condotti più volte per verificare che i risultati siano stabili.

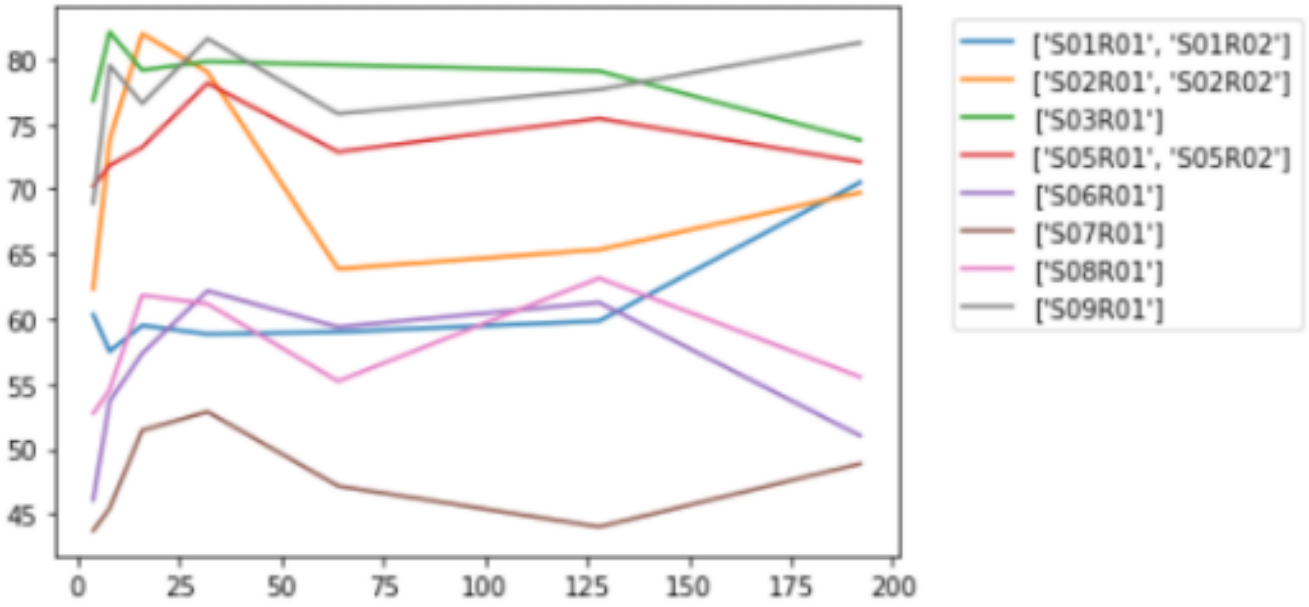

Figura 17: Accuratezza dei modelli al variare del numero di filtri da 4 a 192 con dimensione kernel fissata a 3

Grazie al dataset bilanciato questa volta è possibile osservare delle matrici confusioni più equilibrate.

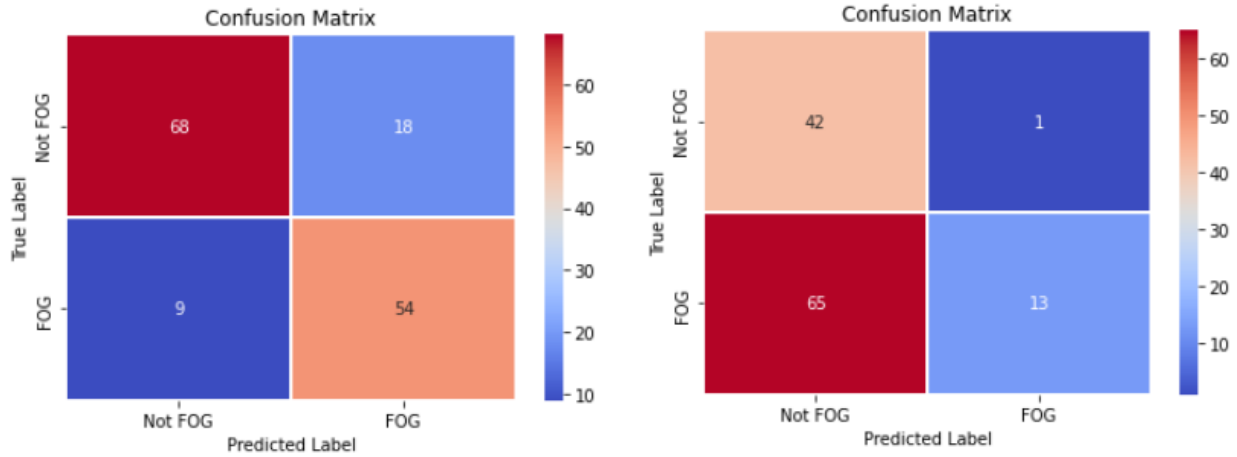

Figura 18: A sinistra la matrice confusione del paziente 3 mentre a destra quella per il paziente 8

I nuovi risultati che abbiamo ottenuto rispecchiano quanto evidenziato nell'articolo [3]. Se prendiamo come esempio il modello del paziente 8, che sappiamo essere quello più affetto da Parkinson, abbiamo un numero di Falsi Negativi alto che porta ad un basso valore di sensitività circa 27% (28.7% in letteratura). Molto spesso il modello non riesce a distinguere quando il paziente rimane fermo volontariamente e quando presenta casi di FOG. Poiché come afferma [3] gli eventi di Fog sono caratterizzati da piccoli movimenti ad alta frequenza. Questo fenomeno si presenta proprio nei pazienti più affetti dalla malattia come il paziente 8.

Class weights

Considerando i risultati ottenuti con il metodo precedente, abbiamo deciso di provare diversi metodi di bilanciamento del dataset per confermare la validità dell'approccio utilizzato. Il primo di questi è il Class weights.

L'obiettivo rimane quello di identificare i casi di FOG, ma poiché non si dispone di molti campioni positivi è necessario permettere al classificatore di considerare maggiormente questi pochi esempi disponibili. Per farlo è possibile modificare i pesi di ogni classe attraverso un parametro Keras. In questo modo il modello "presterà più attenzione" agli esempi della classe FOG sottorappresentata.

Implementazione

Per l'implementazione del metodo Class Weights utilizziamo quella fornita dal sito TensorFlow applicata alla classificazione su dati sbilanciati[9]. Il codice relativo al caricamento del dataset è simile a quello già utilizzato nei capitoli precedenti.

```
# Scaling by total/2 helps keep the loss to a similar magnitude.
# The sum of the weights of all examples stays the same.
weight_for_0 = (1 / neg) * (total / 2.0)
weight_for_1 = (1 / pos) * (total / 2.0)

class_weight = {0: weight_for_0, 1: weight_for_1}

print('Weight for class 0: {:.2f}'.format(weight_for_0))
print('Weight for class 1: {:.2f}'.format(weight_for_1))
```

Figura 19: Il peso viene calcolato a seconda della percentuale del numero dei casi positivi

```
weighted_model = make_model()
weighted_model.load_weights(initial_weights)

weighted_history = weighted_model.fit(
    train_features,
    train_labels,
    batch_size=BATCH_SIZE,
    epochs=EPOCHS,
    callbacks=[early_stopping],
    validation_data=(val_features, val_labels),
    # The class weights go here
    class_weight=class_weight)
```

Figura 20: Codice per training e valutazione del modello con le classi pesate

É importante notare nella Figura 20 che in questa implementazione del modello venga utilizzata la tecnica dell' early-stopping per selezionare un numero corretto di epochs.

Un numero elevato di epochs può portare ad un overfitting del dataset di training, al contrario troppo poche possono portare ad un modello underfit. L'early-stopping è un metodo che consente di specificare un numero arbitrario di epochs di training e di conseguenza interrompere il ciclo iterativo una volta che le prestazioni del modello cessano di migliorare sul validation set.

Valutazione del modello

Per ottenere un confronto diretto rispetto ai risultati precedenti possiamo andare ad osservare le matrici confusione del paziente 3 e del paziente 8.

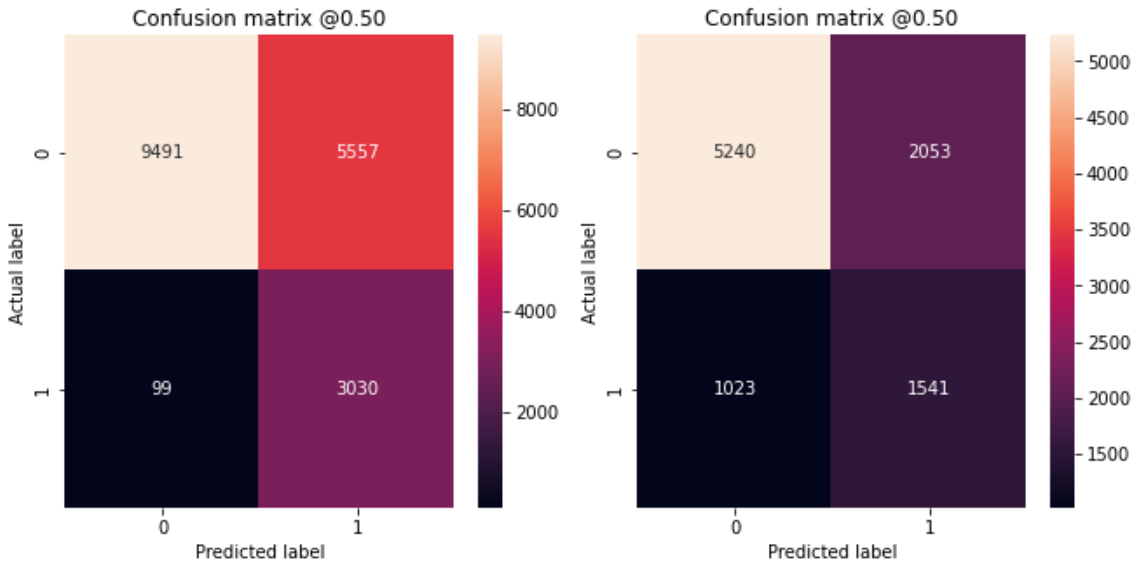

Figura 21: A sinistra la matrice confusione del paziente 3 mentre a destra quella per il paziente 8

Si può notare che con le classi pesate l'accuratezza e la precisione sono inferiori perché ci sono più falsi positivi, ma al contrario del dataset sbilanciato il modello è riuscito a predire più veri positivi. Rispetto al modello precedente vengono considerate ogni singola riga del dataset non perdendo alcuna informazione originale sul training.

Naturalmente c'è un costo per entrambi i tipi di errore. Per ottenere una situazione ottimale è necessario considerare attentamente i vari compromessi tra questi diversi equivoci nei modelli.

Oversampling

Un approccio simile al Class weight è l'oversampling. L'idea alla base di questo metodo è ricampionare il dataset di training andando a sovracampionare la classe meno rappresentata.

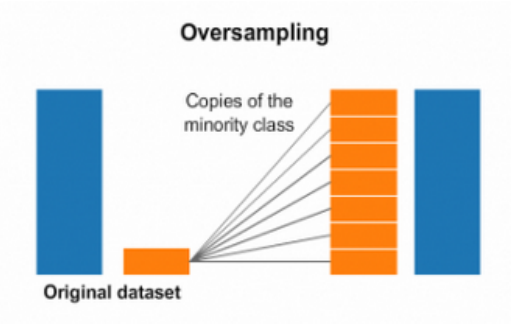

Figura 22: Concetto di oversampling

Naturalmente conviene non eccedere con il sovracampionamento di dati FOG perché si rischia di inserire nel modello dei dati fittizi. La principale problematica dell'oversampling infatti è il rischio di far allontanare il modello ottenuto dalla realtà osservata.

Implementazione

Per l'implementazione del metodo Oversampling utilizziamo quella fornita dal sito TensorFlow applicata alla classificazione su dati sbilanciati[9]. Il codice relativo al caricamento del dataset è simile a quello già utilizzato nei capitoli precedenti.

Per produrre un dataset bilanciato è quello di iniziare con un insieme di dati positivi e negativi equlibrati random ed unirli.

```
BUFFER_SIZE = 100000

def make_ds(features, labels):
  ds = tf.data.Dataset.from_tensor_slices((features, labels))#.cache()
  ds = ds.shuffle(BUFFER_SIZE).repeat()
  return ds

pos_ds = make_ds(pos_features, pos_labels)
neg_ds = make_ds(neg_features, neg_labels)
```

Figura 23

Poiché il training è più semplice su dati bilanciati, è possibile implementare due cicli di training soprattutto per risolvere la tendenza all'overfitting del primo ciclo di apprendimento. In questo modo inoltre vi è un controllo più accurato sull'early-stopping.

```
resampled_model = make_model()
resampled_model.load_weights(initial_weights)

# Reset the bias to zero, since this dataset is balanced.
output_layer = resampled_model.layers[-1]
output_layer.bias.assign([0])

val_ds = tf.data.Dataset.from_tensor_slices((val_features, val_labels)).cache()
val_ds = val_ds.batch(BATCH_SIZE).prefetch(2)

resampled_history = resampled_model.fit(
    resampled_ds,
    epochs=EPOCHS,
    steps_per_epoch=resampled_steps_per_epoch,
    callbacks=[early_stopping],
    validation_data=val_ds)
```

Figura 24: Primo ciclo di training

```
resampled_model = make_model()
resampled_model.load_weights(initial_weights)

# Reset the bias to zero, since this dataset is balanced.
output_layer = resampled_model.layers[-1]
output_layer.bias.assign([0])

resampled_history = resampled_model.fit(
    resampled_ds,
    # These are not real epochs
    steps_per_epoch=20,
    epochs=10*EPOCHS,
    callbacks=[early_stopping],
    validation_data=(val_ds))
```

Figura 25: Secondo ciclo di training, è possibile notare un elevato numero di epochs per garantire un early-stopping migliore

Valutazione del modello

Andiamo ad osservare i risultati delle matrici confusione del paziente 3 e del paziente 8.

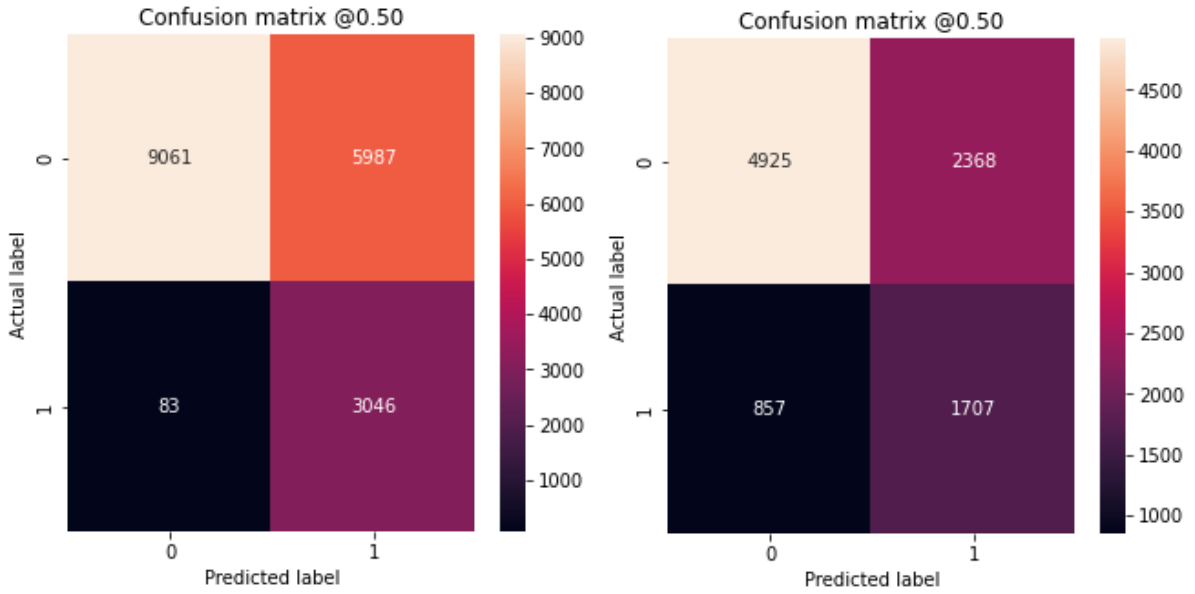

Figura 26: A sinistra la matrice confusione del paziente 3 mentre a destra quella per il paziente 8

Si può notare che utilizzando dataset identici nel processo di training, questo metodo di oversampling permette di ottenere risultati sostanzialmente identici alla Class weights.

Questa considerazione ci permette di avere la libertà di poter selezionare due metodi con risultati comparabili ma partendo da approcci differenti. A seconda del dataset utilizzato o dalle performance del dispositivo classificatore potrebbe essere preferibile un metodo rispetto all'altro.

Transformer Architecture

Applichiamo adesso al nostro problema un modello allo stato dell'arte basato sull'Architettura Transformer. Questo modello è stato creato da dei ricercatori di Google nel 2017 originariamente per compiere traduzioni di testi Inglese-Fracese, ma poi si è rivelato efficace anche nella risoluzione di altri problemi come riconoscimento di immagini e analisi di serie temporali. L'architettura del modello *Transformer* è basata su una struttura encoder-decoder. L'encoder mappa una sequenza di simboli in input in una nuova sequenza di rappresentazioni continue. Successivamente il decoder usa questi simboli per generare l'uscita. Il modello è auto-regressivo ovvero usa i simboli generati precedentemente come input per generare quelli successivi [7].

Implementazione

Utilizziamo l'implementazione del modello Transformer fornita da *Theodoros Ntakouris* applicata alle serie temporali[8]. Manteniamo il caricamento del dataset come già usato nei capitoli precedenti aggiungendo però il codice relativo all'implementazione del nuovo modello.

La funzione di seguito prende in input gli iperparametri della rete e la dimensione dell'input, ritorna il modello Transformer.

```
def build_model(input_shape,head_size,num_heads,ff_dim,num_transformer_blocks,mlp_units,dropout=0, mlp_dropout=0,):
    inputs = keras.Input(shape=input_shape)
    x = inputs
    for _ in range(num_transformer_blocks):
        x = transformer_encoder(x, head_size, num_heads, ff_dim, dropout)

    x = layers.GlobalAveragePooling1D(data_format="channels_first")(x)
    for dim in mlp_units:
        x = layers.Dense(dim, activation="relu")(x)
        x = layers.Dropout(mlp_dropout)(x)
    outputs = layers.Dense(n_classes, activation="softmax")(x)
    return keras.Model(inputs, outputs)
```

Figura 27: Generazione del modello transformer

La funzione *transformer_encoder* è mostrata di seguito e crea un singolo encoder il quale viene aggiunto ripetutamente al modello tramite un ciclo for.

```
def transformer_encoder(inputs, head_size, num_heads, ff_dim, dropout=0):
    # Normalization and Attention
    x = layers.LayerNormalization(epsilon=1e-6)(inputs)
    x = layers.MultiHeadAttention(
        key_dim=head_size, num_heads=num_heads, dropout=dropout
    )(x, x)
    x = layers.Dropout(dropout)(x)
    res = x + inputs

    # Feed Forward Part
    x = layers.LayerNormalization(epsilon=1e-6)(res)
    x = layers.Conv1D(filters=ff_dim, kernel_size=1, activation="relu")(x)
    x = layers.Dropout(dropout)(x)
    x = layers.Conv1D(filters=inputs.shape[-1], kernel_size=1)(x)
    return x + res
```

Figura 28: Funzione che costruisce l'encoder

Il modello completo ha alla fine 42 layer per un totale di 30,282 parametri addestrabili. Il training viene fatto effettuando 200 epoche con dimensione di batch di 64 e un *Early Stopping* con 10 step di attesa. La tecnica di early stopping è fondamentale per evitare di far proseguire il training quando l'accuratezza del modello non migliora più.

Valutazione del modello

I primi training vengono effettuati sui pazienti 1 e 8 ottenendo rispettivamente un accuratezza sul test molto stabile rispettivamente di 0.94 e 0.68. mentre la loss ottenuta è di 0.21 e 0.65. I risultati piuttosto bassi per il paziente 8 erano attesi per via della natura del paziente come già visto nei capitoli precedenti. Le epoche sono state di 55 e 61 e la sessione di training in entrambi casi è durata circa 3h, un tempo piuttosto lungo se paragonato ai modelli già utilizzati. Data la lunghezza delle sessioni di training questa volta i modelli vengono salvati per essere riutilizzati successivamente dopo averli ricaricati. Il codice usato per salvare e ricaricare il modello è molto semplice ed è il seguente il seguente

```
model.save(path+'/Modelli/Modello_paziente_1')

model = keras.models.load_model('/Modelli/Modello_paziente_1')
```

FIgura 29: Codice che salva e ricarica un modello

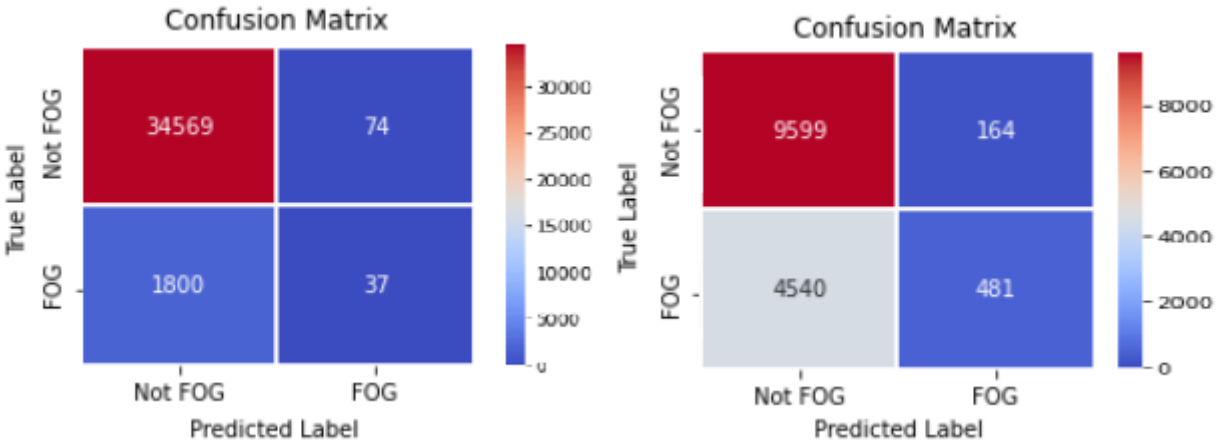

Figura 30: Matrici confusioni per i pazienti 1 (a sinistra) e paziente 8 (a destra) usando i dataset invariato

Analogamente a quanto ottenuto con quando abbiamo utilizzato la CNN sul dataset sbilanciato possiamo osservare una forte preponderanza dei casi di predizione corretta di Not fog. Inoltre si nota ancora nel paziente 8, il più affetto dalla malattia che è molto alto il numero di falsi negativi.

A questo punto aggiungiamo al preprocessing del dataset il bilanciamento dei campioni con lo stesso algoritmo randomico utilizzato precedentemente. Otteniamo per il paziente 1 e 8 rispettivamente una accuracy di 0.64 e 0.59, mentre abbiamo una loss pari a 0.86 e 0.74 ottenendo un certo miglioramento dall'uso delle CNN, probabilmente con una accurata ottimizzazioni degli iperparametri è possibile ottenere risultati migliori. Di seguito le matrici confusioni associate ai nuovi due modelli.

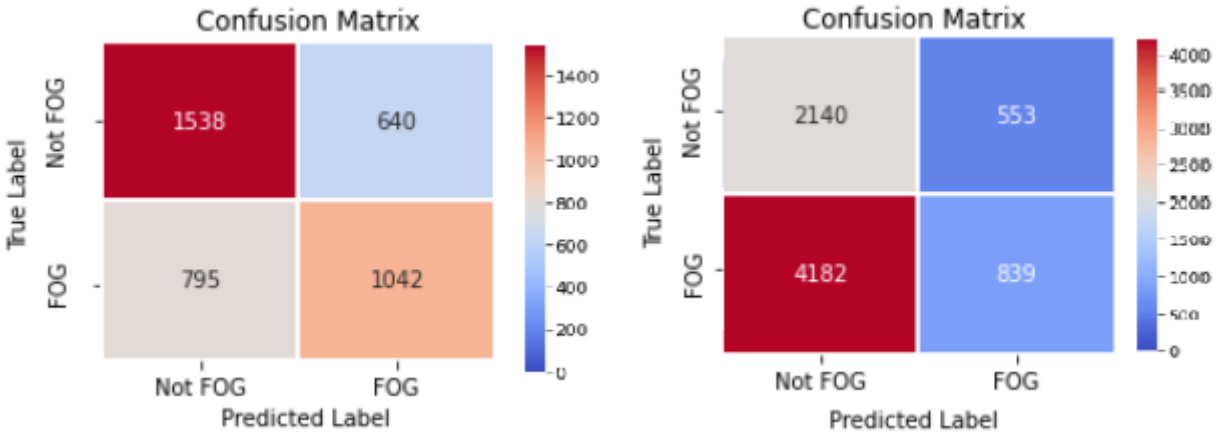

Figura 31: Matrici confusione per i pazienti 1 e 8 dopo aver bilanciato il dataset

Conclusioni e confronto dei metodi

Lo scopo di questa relazione era quello di illustrare la realizzazione di modelli di machine learning utilizzando il Daphnet Freezing of Gait Dataset per riconoscere gli eventi di FOG. Per permettere un approccio corretto all'argomento abbiamo affrontato lo studio della letteratura disponibile sul problema e successivamente abbiamo provato ad applicare al dataset i modelli di apprendimento che abbiamo ritenuto migliori.

Il metodo K-NN è stato istruttivo perché ci ha permesso di rilevare le principali problematiche che si possono incontrare affrontando un esercizio di modellazione di supervised learning. Infatti i risultati relativamente negativi ottenuti ci hanno permesso di approcciare i metodi Neural Network con maggiore efficacia, andando a superare i limiti della K-NN.

I differenti metodi trattati relativi alle Neural Network sono rivolti esclusivamente all'ottimizzazione del processo di training per permettere al modello di predire il maggior numero di casi FOG possibili.

Andiamo a riassumere le metodologie utilizzate:

- **Segmentazione in finestre temporali:** Il metodo della segmentazione in finestre temporali è stato utilizzato nella CNN. I primi modelli generati con l'intero dataset hanno evidenziato una eccessiva lentezza nella generazione dei risultati ed inoltre le predizioni dei casi di FOG erano prossime allo zero. L'utilizzo dei timesteps deriva proprio dalla volontà di risolvere queste problematiche: a causa degli scarsi risultati era necessario disporre di un processo di generazione più agile e reattivo alle modifiche. La segmentazione ha permesso quindi di correggere alcune problematiche del codice, osservare le prime predizioni significative e inconsapevolmente anche un primo tentativo di bilanciamento del dataset.
- **Bilanciamento casuale:** Il bilanciamento casuale deriva dalla necessità di effettuare operazioni di bilanciamento del dataset per cercare di equilibrare il numero di casi di "FOG" con quelli di "Not FOG". Utilizzato nella CNN e nella Transformer Architecture, consiste nell'eliminare randomicamente un certo numero di sample "Not FOG" per permettere alle due label di eguagliarsi in numero. Questo processo ha permesso di ottenere risultati soddisfacenti contenendo il numero di previsioni errate del modello e di conseguenza sia nel caso della CNN, sia nella Transformer Architecture sono state ottenute delle buone matrici confusione. Questo tipo di approccio è molto simile a quello della decimazione dove vengono rimossi dal dataset dati di rumore inutili a ciò che si vuole osservare. La principale problematica di questo metodo è il mancato utilizzo del dataset completo
- **Bilanciamento Class Weights e bilanciamento Oversampling:** Analogamente al bilanciamento casuale, questi metodi sono necessari per equilibrare le etichette. Nel caso della Class weight vengono assegnati pesi diversi alle label a seconda dello sbilanciamento, mentre nell'Oversampling ad ogni insieme di dati positivi vengono uniti in modo equilibrato dati random negativi. Questi approcci sono decisamente più pragmatici, permettono di utilizzare completamente il dataset e restituiscono risultati molto simili ma al contrario del metodo precedente vi sono in uscita un numero considerevole di falsi positivi. Il training dei due metodi è differente in quanto l'oversampling ha bisogno di due cicli di training e un preprocessing che prevede una variabile random: entrambi i fattori sono assenti nella class weight. Come già affermato in precedenza potrebbe essere preferibile un metodo rispetto all'altro a seconda del dataset utilizzato o dalle performance del classificatore.

La Transformer Architecture rappresenta lo stato dell'arte e sono state ottenute delle ottime performance. In questo caso abbiamo preferito utilizzare il bilanciamento casuale principalmente per l'equilibrio delle previsioni errate del modello. La Class weights e l'Oversampling invece restituiscono un numero eccessivamente elevato di falsi positivi rispetto ai falsi negativi.

DI seguito si riporta una tabella riassuntiva dei risultati ottenuto con i vari modelli implementati.

Metodo	Accuracy	Precision	Recall	F1-score	Tempo Training	Tempo Inferenza
KNN	0,82	0,32	0,37	0,34	ND	10 sec
CNN un-balanced	0,64	0,04	0,02	0,03	7 min 53 sec	0,71 sec
CNN balanced	0,79	0,75	0,32	0,45	138 sec	0,14 sec
CNN Class Weight	0,68	0,42	0,6	0,49	8 min 30 sec	0,4 sec
CNN Class Oversampled	0,67	0,41	0,66	0,51	7 min 22 sec	0,3 sec
Transformer un-bal	0,53	0,11	0,13	0,12	1h 21 min	18 sec
Transformer bal	0,81	0,78	0,34	0,47	35 min	12.4 sec

Questo progetto ci ha permesso di capire come affrontare un problema di Machine Learning con dati sperimentali appartenenti ad un problema attuale. In futuro potrebbe essere possibile effettuare una implementazione di questi metodi su un sistema embedded con la possibilità di effettuare predizioni online degli eventi di FOG.

Riferimenti

[1] *Daphnet Freezing of Gait Data Set, UCI Learning Repository, Center for Machine Learning and Intelligent Systems*

[2] *Wearable Assistant for Parkinson's Disease Patients With the Freezing of Gait Symptom, Marc B¨achlin, Meir Plotnik, Daniel Roggen, Inbal Maidan, Jeffrey M. Hausdorff, Nir Giladi, and Gerhard Tr¨oster, Senior Member, IEEE*

[3] *Daphnet Freezing Recognition with Gait data by Using Machine Learning Algorithms, Selda Guney & Busra Boluk Electrical-Electronics Engineering faculty of Engineering Baskent University Ankara Turkiue*

[4] *1D Convolutional Neural Network Models for Human Activity Recognition, by Jason Brownlee on September 21, 2018*

[5] *Human Activity Recognition Using Smartphones Data Set, UCI Machine Learning Repository,* DITEN - Università degli Studi di Genova, Genoa (I-16145) Italy, Universitat Politècnica de Catalunya Spain

[6] *How do AI-Based fitness trackers work? Ft. 1D CNNs, Start it up, Nemath Ahmed*

[7] *Attention Is All You Need, Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, Illia Polosukhin*

[8] *Timeseries classification with a Transformer model, Theodoros Ntakouris*

[9] https://www.tensorflow.org/tutorials/structured_data/imbalanced_data#oversampling