Relazione di fine tirocinio

Implementazione su sistema embedded di una rete neurale per la classificazione di patologie neurodegenerative tramite analisi di segnali EEG

Indice

Introduzione

Tema

La malattia di Alzheimer (Alzheimer's Disease, AD), è la forma di demenza più frequente nella popolazione anziana. Attualmente, il morbo di Alzheimer non è curabile, tuttavia i farmaci attualmente disponibili sono in grado di migliorare i sintomi della malattia e di rallentarne temporaneamente la progressione, soprattutto se il trattamento avviene durante le prime fasi della malattia. Per questo motivo è molto utile trovare un modo per riuscire ad individuare, in una fase precoce, i soggetti affetti da AD. Uno strumento utilizzato per la diagnosi è l'elettroencefalogramma (EEG), che, misurando l'attività elettrica cerebrale, può identificare anomalie nelle onde cerebrali legate a determinati disturbi.

Questi segnali possono essere analizzati e classificati utilizzando le reti neurali multistrato che hanno la capacità di estrarre informazioni rilevanti dal segnale EEG chiamate "features", e grazie a queste caratteristiche sono in grado di distinguere automaticamente i pazienti sani da quelli affetti da AD, dopo essere state adeguatamente implementate e addestrate.

Obiettivo

L'attività principale del tirocinio è stata quella di analizzare l'architettura di una rete neurale già implementata, per aggiungere un ulteriore preprocessing dei dati al fine di migliorare i risultati del modello. La rete è stata addestrata a distinguere un paziente sano da uno malato utilizzando un dataset ottenuto dalle analisi EEG di 35 pazienti, 20 affetti da Alzheimer e 15 sani. Questi dati provengono dal Dipartimento di Medicina Sperimentale e Clinica di Ancona.

Prima di essere dati in input alla rete, tutti i segnali EEG sono stati convertiti in un formato adeguato per poi essere pre-elaborati tramite diversi filtraggi e tecniche di riduzione del rumore, con l'obiettivo di rimuovere gli artefatti.

Gli artefatti sono parti indesiderate del segnale EEG registrato che non sono state generate dall'attività neuronale del cervello e quindi possono introdurre cambiamenti significativi nei segnali. Gli artefatti possono provenire da fonti *non fisiologiche*, come rumore della linea elettrica o variazioni delle impedenze degli elettrodi, oppure da fonti *fisiologiche*, come le distorsioni causate da movimenti oculari o movimenti muscolari.

Preprocessamento

Strumenti utilizzati

EDFbrowser

EDFbrowser è un software che permette di visualizzare graficamente ed analizzare i file di archiviazione di serie temporali come EEG, EMG, ECG, ecc. In questo caso l'ho utilizzato per visualizzare i segnali EEG in formato EDF. Una volta caricato il segnale EEG, l'applicazione offre informazioni sul soggetto, la data di registrazione e la durata, e consente di selezionare, aggiungere o rimuovere segnali.

Google Collaboratory

Google Colab è un IDE che consente a qualsiasi utente di scrivere codice sorgente nel suo editor ed eseguirlo dal browser. Nello specifico, supporta il linguaggio di programmazione Python ed è orientato a compiti di machine learning.

Librerie di Python

Python dispone di una collezione di moduli pronti all'uso, contenuti nella Standard Library di Python. Ogni modulo è come una libreria (un insieme) di funzioni già scritte, che forniscono soluzioni standardizzate ai problemi che si verificano più spesso nella programmazione.
I moduli principalmente usati in questo progetto sono: **Numpy** per eseguire calcoli numerici e manipolare array multidimensionali e matrici; **Matplotlib** per la creazione di grafici; **SciPy** per il calcolo scientifico; **Scikit-learn** per la realizzazione di modelli predittivi di machine learning.
A queste si aggiunge anche **TensorFlow**: una libreria software open source che oggi è utilizzata in molti ambiti scientifici e aziendali per svolgere funzioni in deep learning. Viene usata ad esempio da Google negli algoritmi di riconoscimento delle immagini, oppure per la lettura e la digitalizzazione delle frasi scritte a mano, per il riconoscimento dei numeri, degli oggetti e dei simboli in una foto.

EEGLAB

Uno strumento per il preprocessamento dei segnali è EEGLAB: il toolbox di Matlab utilizzato per la visualizzazione e l'elaborazione di segnali EEG. EEGLAB fornisce un'interfaccia utente grafica interattiva (GUI) che consente agli utenti di elaborare in modo flessibile e interattivo i segnali EEG. Il pacchetto che utilizza EEGLAB è **BioSignal Toolbox**, una libreria software open source per l'elaborazione di segnali biomedici, come ad esempio l'elettroencefalogramma (EEG), con Matlab, Octave, C/C++ e Python.

Sono supportati circa 50 diversi formati di dati. Nelle figure sono riportate le tabelle con l'elenco dei formati supportati, e sono evidenziati quelli utilizzati in questo progetto.

(L'elenco dei formati supportati è disponibile qui: pub.ista.ac.at/~schloegl/biosig/TESTED)

```
LEGEND:
-------
r       read mode is supported
rw      read and write mode is supported
z       on-the-fly (de-)compression
?       not tested, probably yes
-       does not work, or not yet implemented
```

			Matlab	Matlab	Matlab	Octave	Octave	C/C++	Python	Java	SigViewer
EDF		R	rw	rw	rw	rwz	rwz	rwz	rz	.	rz
EDF+	+	R	r	r	r	r	rz	rz	rz		rz
EEG(Neurscan)	-	R	r	r	r	r	r				
EEG1100		R	rz (9)	rz (9)	rz (9)	rz (9)	rz (9)	rz	rz		rz
EEG(COHERENCE)											
EEGLAB		R	r	r	r	r	r				
EEProbe		R	r (a)	r (a)	r (a)						
EGI/RAW	+	R	r	r	r	r	rz	rz			rz
EMBLA		R	r	r	r		r				
EMSA			rz (9)	rz (9)	rz (9)	rz (9)	rz (9)	rz			rz
EPL	-	R	r	r	r	r	r				
ePrime (h)		R	r	r	r	r	r				
ESPS											
ET-MEG	+	R	r	r	r	r	r				
ETG4000	+		r	r	r	r	r	rz			rz
EZ3(Cadwell)											
FAMOS (2)			r (9)	r (9)	r (9)	-	r (9)	r	r		?
FDA-XML			rwz (9)	rwz (9)	rwz (9)	-	rwz (9)	rwz	rz	.	rz
FEF-Vital											

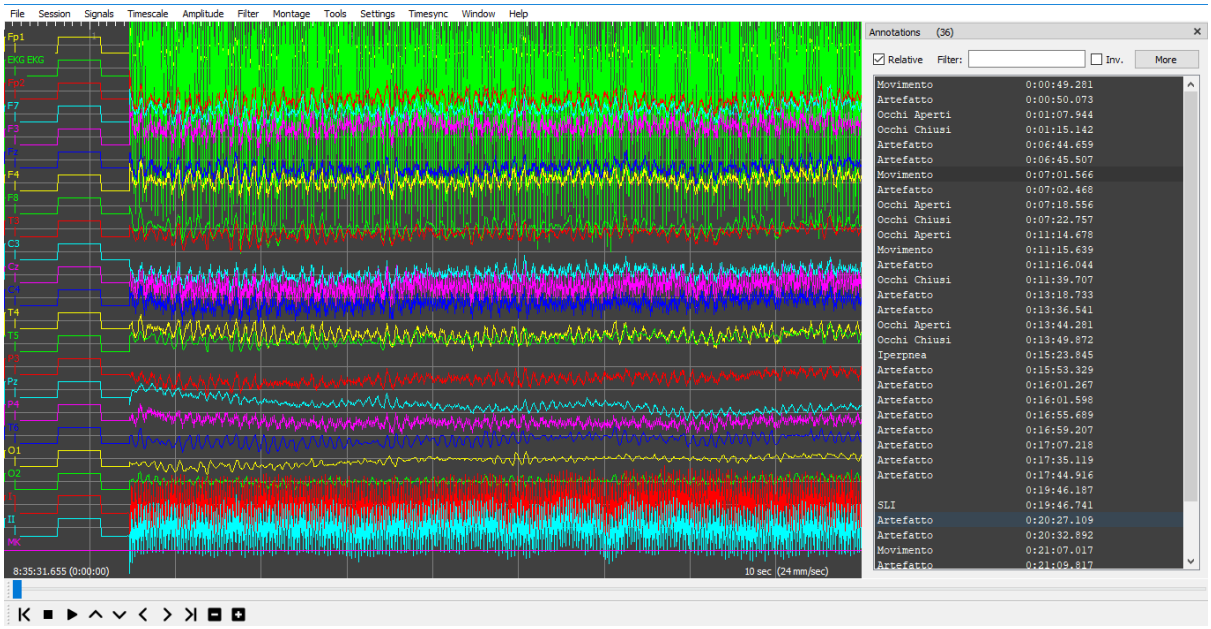
...

ITX		R	rz (9)	rz (9)	rz (9)	rz (9)	rz (9)	rz	rz		rz
JPEG											
KWIK											
LABVIEW											
Lexicor											
MAT (e)		R	r	r	r						
MM	M		r	r	r	r	r				
MFER (5)		R	r	r	r	rz	rz	rz	r		rz
MOV/QTFF											
MPEG											
MRI											
NeuroShare											
NetCDF											
NEURON		R	rz (9)	rz (9)	rz (9)	rz (9)	rz (9)	rz	rz		rz

Formato del dataset

Il dataset a disposizione è costituito dai segnali EEG di 35 pazienti, 20 affetti da Alzheimer (AD) e 15 sani (N). I dati sono stati forniti in formato **EDF** (European Data Format), un formato utilizzato in ambito medico per lo scambio e l'archiviazione di segnali biologici e fisici, che permette di memorizzare serie temporali e dati multicanale. Ogni segnale EEG acquisito, infatti, ha più canali (da 21 a 23 canali) poiché ad ogni elettrodo corrisponde una regione ben precisa del cervello, quindi l'EEG di ogni paziente sarà caratterizzato da più segnali provenienti da aree cerebrali diverse. Alcuni esempi: i segnali Fp (cioè fronto-parietali), F (cioè provenienti dal lobo frontale), C (centrali), T (lobo temporale), P (lobo parietale), O (lobo occipitale).

Un esempio è mostrato in figura: è un segnale EEG con 22 canali (i nomi dei 22 segnali visibili a sinistra) e presenza di artefatti, visualizzato in formato EDF con il software EDFbrowser.



A partire dal formato EDF iniziale, è stato fatto un preprocessing iniziale per avere un set di dati in input alla rete coerente e ordinato: sono stati riordinati e rimossi alcuni canali per far sì che tutti i segnali in input abbiano lo stesso numero di canali (16) e nello stesso ordine.

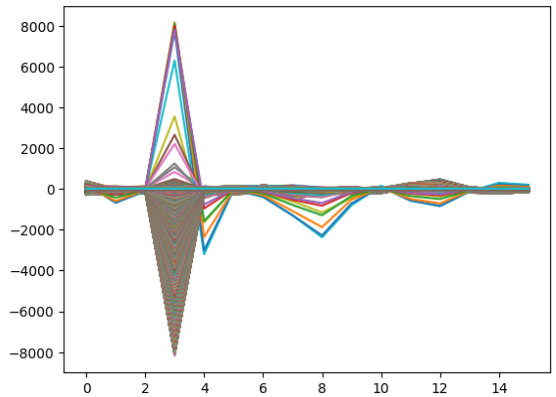
In questo processo i dati sono stati salvati in formato **NPY**. Un file NPY è un file di array NumPy creato con la libreria NumPy di Python. Il file NPY memorizza tutte le informazioni necessarie per ricostruire correttamente l'array. Per ogni segnale, inoltre, è stato creato un file in formato **NPZ** (*NumPy Zipped Data*), cioè un file zip contenente più file NPY, uno per ogni array.

Per processare i segnali con le funzioni dell'EEGLAB, però, è necessario convertire il formato NPZ in formati supportati dall'EEGLAB, per questo progetto sono stati scelti i formati **MAT** e **CSV**. Quindi, dopo aver caricato il set di segnali in formato NPZ, ho creato una funzione per convertire rapidamente tutti i segnali (20 AD + 15 N), in entrambi i formati. I codici di conversione sono scritti interamente in Python.

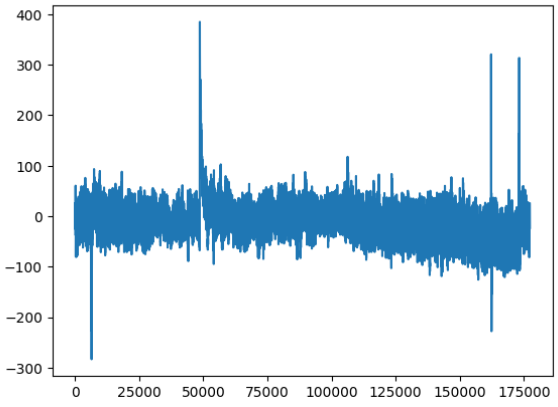
```python
def npz_to_CSV (filename):
  # carico il file .npz dalla cartella 'eeg2'
  data = np.load('/content/drive/MyDrive/eeg_DNN-Conti/eeg2/' + filename + '.npz')
  newfile= filename+'.csv'

  #salvo il file .csv nella cartella 'eeg2_csv'
  working_dir = '/content/drive/MyDrive/eeg_DNN-Conti/eeg2_csv/'+ newfile
  arr=data.files[0]
  csv=np.savetxt(working_dir, data[arr], delimiter=",")
  return data, csv
```

Per verificare che la conversione sia stata eseguita correttamente ho confrontato i segnali di formati diversi, visualizzandoli graficamente, con l'apposita libreria di Python Matplotlib, verificando che abbiano lo stesso andamento nel tempo. Ogni segnale EEG è una matrice di dimensione nx16, con n numero di istanti temporali, e 16 numero di canali. Dal grafico di sinistra è visibile la sovrapposizione dei 16 segnali. Utilizzando gli indici da 0 a 15, invece, è possibile visualizzare ogni canale singolarmente.

```python
import matplotlib.pyplot as plt
plt.plot(data['eeg'])
plt.show()
```




```python
plt.plot(data['eeg'][0])
plt.show()
```

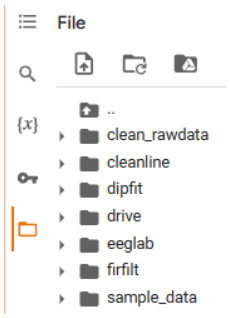


EEGLAB in Python

Dopo aver portato i segnali in un formato adeguato, il passo successivo è quello di applicare le funzioni dell'EEGLAB ai segnali del dataset per eliminarne le distorsioni.

L'EEGLAB però non funziona nativamente in Python, ma lavora in MATLAB e sulla piattaforma open source Octave. Inoltre, lavora principalmente con il formato **SET**, che è un formato proprietario del MATLAB, quindi non integrabile in codici open source come Python. Oltre a questo, dato che il preprocessing dei dati deve essere effettuato in un'architettura di rete già implementata in Python, per automatizzare la procedura si è deciso di integrare la parte EEGLAB direttamente in Python. Per questi motivi è stato necessario cercare un collegamento tra EEGLAB e Python.

Per chiamare e quindi eseguire le funzioni dell'EEGLAB in Python, la soluzione trovata è installare Octave (*!apt-get install octave*) e usare il pacchetto Python *Oct2py* (*!pip install oct2py*). Questa libreria Python converte le strutture dati Python in strutture dati MATLAB/Octave e viceversa. Tutte le funzioni di elaborazione del segnale EEGLAB, infatti, funzionano anche su Octave.



Oltre al pacchetto Oct2py sono stati aggiunti altri pacchetti non presenti in Octave, necessari per l'utilizzo delle funzioni: *eeglab, dipfit, firfilt, clean_rawdata* e *cleanline*. Sono state importate le funzioni desiderate ed è stato caricato un segnale di prova.

Su questo è stato applicato un filtro passa alto alla frequenza di 0.2 Hz utilizzando la funzione *pop_eegfiltnew* del pacchetto *firfilt*, che contiene tutte le funzioni necessarie ad implementare diversi tipi di filtri.
Un'altra funzione utilizzata per il filtraggio è *pop_cleanline*, dal pacchetto *CleanLine* ("linea pulita"), che rimuove gli artefatti dai componenti ICA, in particolare il rumore di linea, che può derivare da fluttuazioni della linea di alimentazione (ad es. rumore di linea con ripple a 50/60 Hz). La funzione per implementare l'ICA con l'EEGLAB, invece, è *pop_runica*, contenuta nel pacchetto *popfunc*, tra le '*functions*' di EEGLAB.

L'obiettivo era quello di implementare tutto il preprocessing con l'EEGLAB, ma alcune funzioni EEGLAB non sono ancora supportate in Python.

Un'altra soluzione trovata per poter lavorare in Python è **MNE**, un pacchetto Python per l'esplorazione, la visualizzazione e l'analisi dei dati neurofisiologici, tra cui l'EEG. Questo software open source fornisce algoritmi implementati in Python per il preprocessing, il filtraggio e l'ICA. Lo svantaggio di questo metodo è che tra i formati supportati non ci sono quelli di nostro interesse, cioè NPZ o CSV, ma solo l'EDF o il SET.

Quindi il preprocessing è stato elaborato sempre in Python ma utilizzando un altro pacchetto: *scikit-learn*, che permette di utilizzare i dati in formato NPZ, già pre-elaborati.

Scikit-learn: FastICA

Scikit-learn è una libreria di machine learning per Python che supporta apprendimento supervisionato e non supervisionato. Fornisce inoltre vari strumenti per adattamento del modello, pre-elaborazione dei dati, selezione del modello, valutazione del modello, e molte altre utilità.

La funzione principale presa da questa libreria, utilizzata nella pre-elaborazione, è **FastICA**: un algoritmo veloce per eseguire l'**analisi indipendente dei componenti** (ICA, Independent Component Analysis).

L'**ICA** è un metodo di elaborazione del segnale che permette di rimuovere/sottrarre artefatti incorporati nei dati *senza* rimuovere le parti di dati interessate, poiché riesce a separare i segnali misti. Il segnale EEG, infatti, è un segnale oscillatorio con caratteristiche stocastiche, ed è una miscela di segnali utili (quelli provenienti dal cervello) e segnali provenienti da sorgenti di disturbo. L'ICA quindi è una tecnica di analisi statistica in grado di individuare e separare le diverse sorgenti che generano i rumori sovrapposti alle misurazioni che si stanno analizzando, poiché di solito le sorgenti sono indipendenti da un punto di vista statistico. (Problema del cocktail-party)

La funzione FastICA, dato che fa parte di una libreria Python, riceve in input i segnali direttamente in formato npz, quindi a differenza di quello che è stato fatto per poter utilizzare le funzioni dell''EEGLAB, in questo caso non c'è bisogno di convertire il formato dei dati a disposizione.

La funzione è applicata nella parte iniziale di codice, in cui viene elaborato il dataset per migliorare il più possibile i dati in input alla rete.

L'implementazione è la seguente:

```
transformer= sklearn.decomposition.FastICA(n_components=None,  algorithm='parallel', whiten='unit-variance',
                                           fun='logcosh', fun_args=None, max_iter=200, tol=0.0001, w_init=None,
                                           whiten_solver='svd', random_state=None)
eeg= transformer.fit_transform(eeg)
```

I parametri principali che caratterizzano la funzione sono:

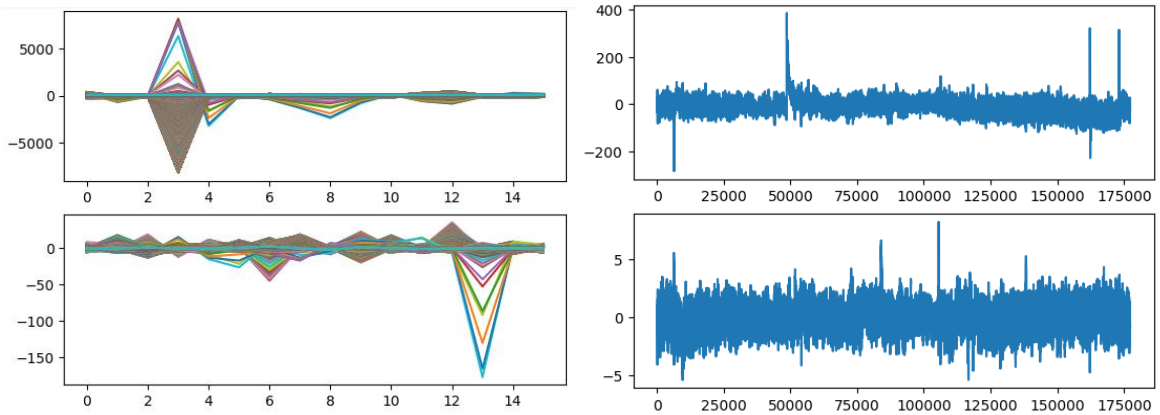
- ***n_components***: Specifica il numero di componenti da utilizzare. Di default vengono utilizzati tutti. (default=*'None'*)
- ***algorithm***: Specifica l'algoritmo da utilizzare per FastICA, *'parallel' o 'deflation'* (default=*'parallel'*)
- ***whiten***: Specifica la strategia di sbiancamento da utilizzare. (*)
  '*arbitrary-variance*': sbiancamento con varianza arbitraria.
  '*unit-variance*': varianza unitaria. (default=*unit-variance*)
  'False': i dati sono già considerati sbiancati e non viene eseguito lo sbiancamento.

- ***fun:*** La funzione utilizzata nella stima della matrice di demiscelazione (**). Potrebbe essere *'logcosh', 'exp'* o *'cubo*'. (default='logcosh')
- ***max_iter:*** Numero massimo di iterazioni durante l'adattamento. (default=200)
- ***tol***: Uno scalare positivo che fornisce la tolleranza alla quale si considera che la matrice di demiscelazione sia convergente. (default=0,0001)
- ***whiten_solver:*** Il risolutore da utilizzare per lo sbiancamento. "*eigh*" o "*svd*". (default="svd")

(*) sbiancamento (whitening): è un'importante fase di pre-elaborazione prima di eseguire l'ICA, e serve per ridurre la ridondanza nei dati di input. Il termine sbiancamento, infatti, deriva dal rumore bianco, caratterizzato da campioni indipendenti tra loro (non correlati). Lo sbiancamento trasforma quindi un vettore casuale in un vettore con componenti non correlate. Oltre a questo algoritmo, un altro metodo utilizzato più avanti nel codice per ottenere lo sbiancamento è la PCA.

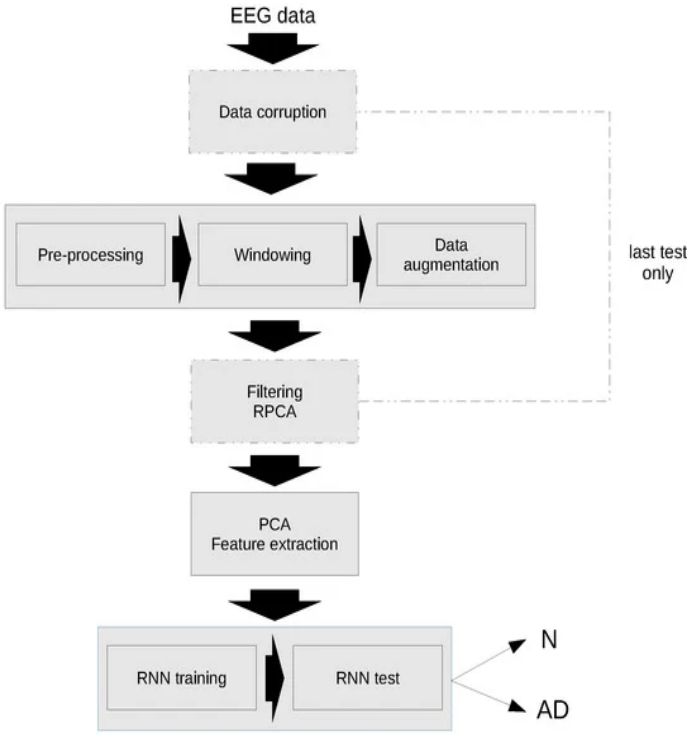
(**) matrice di demiscelazione: matrice (A) che lega le misurazioni (x) ai segnali sorgente originali (s): s=Ax

Per verificare che la funzione abbia agito correttamente, ho applicato l'ICA su un solo segnale e ho confrontato il plot iniziale del segnale, con quello dopo l'ICA. Prima dell'ICA nel segnale sono visibili dei picchi molto ampi causati dagli artefatti, che sono sovrapposti al segnale utile. Nei grafici in basso ottenuti dopo l'ICA, invece, si può osservare come l'ampiezza dei picchi sia notevolmente ridotta. Questo è evidente sia nel plot del segnale con 16 canali (sin), sia in quello con singolo canale (dx).



Implementazione del modello

Algoritmo di classificazione



In figura è rappresentato un diagramma schematico dell'algoritmo di classificazione.

I segnali EEG acquisiti sono rumorosi e sporchi, quindi prima di essere dati in input alla rete neurale sono necessari diversi step per renderli più puliti e ottenere una migliore accuratezza del modello.

Il primo passaggio è il **pre-processing**, cioè la pulizia dei dati tramite diversi filtraggi e rimozione di artefatti, con le tecniche di cui si è precedentemente discusso.

Il passo successivo è il **windowing**: i dati di input vengono suddivisi lungo l'asse temporale in N finestre di dimensioni fisse, che si sovrappongono di una determinata quantità. La lunghezza della finestra (w) e la sovrapposizione (overlap) sono iper-parametri stabiliti sperimentalmente al fine di ottimizzare l'accuracy. (w=256, overlap=50%)

Poi c'è bisogno di un **aumento dei dati**: poiché il numero di ingressi appartenenti alle due diverse classi non è equamente rappresentato, la rete potrebbe essere sbilanciata verso una classe specifica. Una soluzione a questo problema è il **sovracampionamento**, che permette di aumentare i dati duplicando quelli delle classi con meno occorrenze, così

i dati utilizzati per l'addestramento sono distribuiti in modo più uniforme tra le diverse classi. Ai dati duplicati è stato aggiunto un rumore casuale gaussiano, in modo da non avere dati identici nel set di addestramento. Il sovracampionamento è stato applicato prima di ogni fase di addestramento al sottoinsieme di soggetti utilizzati per l'addestramento, lasciando inalterati i soggetti del test.

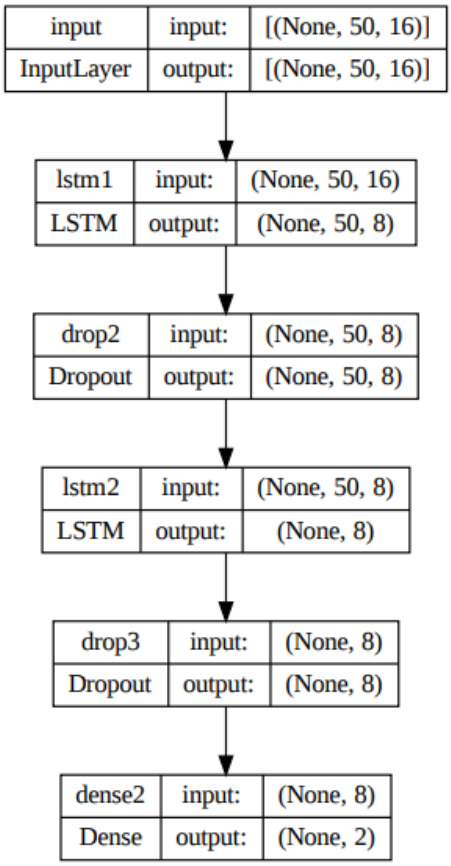
Successivamente viene eseguita l'**analisi delle componenti principali** (**PCA**). L'obiettivo principale della PCA è quello di ridurre la dimensione del dataset: seleziona un certo numero di 'componenti principali', cioè solo quelle che contengono le caratteristiche (feature) più importanti, scartando le variabili che danno uno scarso contributo all'informazione totale che i dati contengono. La riduzione della dimensionalità non solo semplifica il calcolo, ma migliora il rapporto segnale / rumore, e anche l'accuratezza finale del modello. Dal punto di vista matematico la PCA permette di proiettare un dataset n-dimensionale in un piano dimensionale più basso, conservando il maggior numero di informazioni rilevanti (cioè selezionando le componenti che raccolgono al meglio la varianza dei dati originali). Nel nostro caso il dataset originale ha 3 dimensioni (è un tensore): $N \times w \times 16$, dove N è il numero di finestre, w la lunghezza della finestra e 16 il numero di canali. Con la PCA il tensore 3D viene ridotto in una matrice bidimensionale. Il numero di componenti principali selezionate (p) è un'iperparametro scelto sperimentalmente (p=50).

Prima della PCA invece, viene applicata la **MSPCA**, un'altra tecnica di filtraggio che oltre ad avere la capacità di estrarre le features principali riducendo la dimensionalità come la PCA, è anche un metodo efficace per la rimozione del rumore.

Inoltre, per testare la robustezza dell'RNN rispetto alla corruzione dei segnali originali, i dati sono stati corrotti artificialmente creando dei "**buchi**" nei segnali originali, con delle sequenze di zeri, applicati simultaneamente a tutte le 16 tracce, in momenti casuali. Dopodiché, un altro algoritmo utilizzato è la **RPCA** (PCA robusta), un miglioramento della PCA, che può essere applicata quando la matrice contiene osservazioni corrotte.

Ora i segnali sono pronti per essere dati in input alla rete neurale RNN. L'architettura di questa rete è illustrata nel capitolo successivo.

Architettura della rete RNN



In Fig. viene mostrata una rappresentazione grafica della struttura della rete neurale utilizzata.

La rete è di tipo **RNN**: si differenzia da una normale rete neurale perché tiene conto della sequenza dei dati, infatti questo algoritmo di apprendimento viene sfruttato soprattutto quando si utilizzano dati di serie temporali per fare delle previsioni, come in questo caso. Una RNN crea una memoria interna alla rete che raccoglie e memorizza informazioni su ciò che il sistema ha calcolato precedentemente, quindi permette di passare l'output di un'esecuzione come input alla successiva, creando un loop di feedback negli strati nascosti.

La rete ha 6 strati: 1 di input, 4 nascosti e 1 di output.

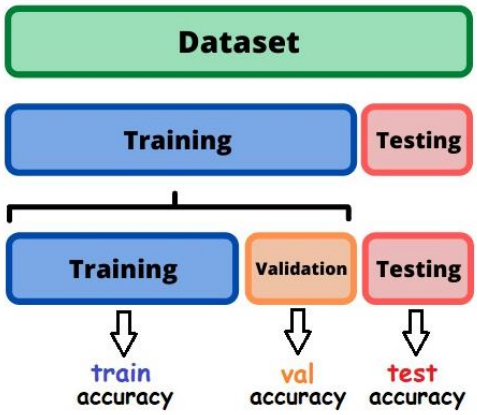
Il nucleo della rete neurale ricorrente è costituito da due strati **LSTM** in cascata.
La rete LSTM (Long Short Term Memory, memoria a lungo-breve termine) è una variante della RNN classica. Uno dei suoi vantaggi principali è che è in grado, anche a lungo termine, di elaborare intere sequenze di dati ma conservare solo le informazioni utili sui dati precedenti nella sequenza, e scartare il resto.

Ogni livello LSTM è seguito da un livello **Dropout**.
Il dropout è una tecnica di regolarizzazione utile per ridurre il problema di overfitting, che consiste nel rimuovere un certo numero di nodi della rete in input ad ogni iterazione della fase di addestramento. Questo metodo viene implementato dopo lo strato sul quale viene applicato, come se fosse un nuovo strato della rete, e i nodi da scartare vengono selezionati in modo casuale.
Tutti gli strati intermedi hanno dimensione 8, questo iper-parametro è stato scelto sperimentalmente, ottimizzando l'accuracy.

Infine, l'ultimo è uno strato **Denso**, cioè completamente connesso, di dimensione 2.

Questo è un modello di classificazione binaria. In output alla rete, infatti, ci saranno due possibili esiti: AD (soggetto malato) o N (soggetto sano).

Solitamente, per validare la qualità di una rete neurale, questa va testata su un set di dati non presentato durante l'addestramento. Per far ciò è stato diviso il dataset iniziale in due subset, uno per l'addestramento ed uno per il test (**test set**). Il set di addestramento va suddiviso a sua volta in altri due insiemi: uno per l'addestramento effettivo (**train set**), e uno per la convalida (**validation set**). Nel nostro caso sono stati utilizzati 30 soggetti per l'addestramento e la validazione e 5 per i test.



I parametri principali che forniscono una valutazione dell'adattamento del modello al dataset, sono i valori delle accuracy sui rispettivi subset di dati a cui si riferiscono: la train accuracy (sul train set), la val accuracy (sul validation set) e la test accuracy (sul test set).

Risultati finali

Confronto dei risultati ottenuti

Per confrontare le prestazioni della rete prima e dopo l'applicazione dell'algoritmo FastICA, ho analizzato i valori di train val e test accuracy e le loro variazioni in seguito a modifica di alcuni parametri della rete.

Rete di partenza senza ICA:

Parametri	train accuracy	val accuracy	test accuracy
PCA=true epochs=20	0.9831	0.9716	0.9657
PCA=true epochs=100	0.9940	0.9897	0.8925
PCA=false epochs=20	0.9259	0.9277	0.4282 ↓

Tab. 1

Rete con algoritmo FastICA:

Ho modificato i parametri della funzione FastICA lasciando invariato il resto del codice, per valutare le prestazioni della rete. Nella prima riga sono presenti i valori di default.

Parametri ICA	train accuracy	val accuracy	test accuracy
n_components= None max_iter=200 tol=0.0001	0.9876	0.9857	0.4404 ↓
n_components= 16 max_iter=200 tol=0.0001	0.9898	0.9884	0.5775
n_components= None max_iter=200 **tol=0.001**	0.9899	0.9889	0.5355
n_components= None **max_iter=1000** tol=0.0001	0.9913	0.9887	0.5882 ↑
n_components= None **max_iter=2000** tol=0.0001	0.9892	0.9844	0.4140 ↓

n_components= 16 **max_iter=1000** tol=0.0001	0.9886	0.9863	0.4224
n_components= 16 **max_iter=1000** **tol=0.001**	0.9879	0.9846	0.5678
whiten='arbitrary-variance'	0.9893	0.9872	0.4882
whiten=False	0.9998	0.9992	0.3231 ↓
whiten_solver='eigh'	0.9897	0.9867	0.4407
fun='cube'	0.9900	0.9862	0.4730
fun='exp'	0.9922	0.9799	0.2010 ↓
whiten='arbitrary-variance' **fun='cube'** **whiten_solver='eigh'**	0.9919	0.9892	0.5142
whiten='arbitrary-variance' **fun='cube'** **whiten_solver='eigh'** **max_iter=1000**	0.9915	0.9882	0.5497

Tab. 2

In seguito ho modificato i parametri della rete (PCA e numero di epoche) lasciando nell'algoritmo FastICA i parametri di default, e ho ottenuto:

Parametri	train accuracy	val_accuracy	test accuracy
FastICA=True PCA=True epochs=50	0.9945	0.9905	0.5150
FastICA=True PCA=True epochs=100	0.9969	0.9912	0.5434
FastICA=True PCA=False epochs=20	0.7518 ↓	0.7568 ↓	0.4625
FastICA=True PCA=False epochs=100	0.9936	0.9919	0.3384 ↓

Tab. 3

Analisi dei risultati ottenuti

Nella Tab. 1 sono riportati in verde i valori ottimali, raggiunti con la rete di partenza senza ICA, con l'algoritmo della PCA e con un adeguato numero di epoche. L'obiettivo è quello di raggiungere risultati migliori aggiungendo un ulteriore preprocessing sui dati, cioè l'ICA.

Come si vede dalla Tab. 2, però, l'ICA porta a un crollo del valore della test accuracy, mentre la train e val accuracy restano invariate. Ho eseguito diverse prove modificando i parametri della funzione FastICA, ma la test accuracy non migliora. Con i valori di default l'ICA non converge. Con tol=0,001 scompare il warning sulla non convergenza dell'ICA, ma si abbassa ulteriormente la val accuracy. Il valore migliore con l'ICA si ottiene portando il numero di iterazioni a 1000.

Nella Tab. 3 invece ho modificato i parametri della rete, provando a non eseguire la PCA e aumentando il numero di epoche.

Un'epoca consiste in un ciclo completo di allenamento sul train set. Una volta che tutti i campioni del set sono stati visti, la rete ricomincia, segnando l'inizio della 2° epoca. Quindi all'aumentare del numero di epoche, il modello sarà più allenato nella predizione dei risultati, ma il rischio è quello di un overfitting: una scarsa capacità di adattamento a nuovi dati mai visti prima. Questo potrebbe essere il motivo per cui, aumentando il numero di epoche il modello non diventa più preciso ma, come si vede dalla Tab. 3, la test accuracy diminuisce ancora.

Conclusioni e futuri sviluppi

Dopo aver aggiunto l'algoritmo FastICA al preprocessing dei dati, i valori di train e val accuracy si mantengono alti, invece quello della test accuracy è molto minore.

Una prima ipotesi è che il modello sia soggetto a **overfitting**. L'overfitting è un fenomeno che si verifica quando un modello si adatta perfettamente ai dati di addestramento e, di conseguenza, quando viene sottoposto al test set, cioè a dati mai osservati in precedenza, non generalizza bene, quindi non funziona correttamente. Infatti, quando si tenta di valutare il modello sul test set, dai risultati sperimentali si nota che le prestazioni sono molto peggiori di quelle che ci si aspetta. Una causa di questo fenomeno potrebbe essere che il dataset acquisito presso Torrette sia insufficiente. La soluzione per un futuro sviluppo del progetto è quella di ampliare il dataset per avere un train set più esteso su cui far addestrare nuovamente la rete.

La seconda ipotesi potrebbe riguardare l'applicazione dell'ICA.
L'ICA, come visto in precedenza, riesce a separare il rumore dal segnale utile: dopo aver rilevato le componenti che rappresentano gli artefatti, qualsiasi segnale sorgente che ha una caratteristica di artefatto viene rimosso (moltiplicando per 0) e trasformato di nuovo in EEG.
Questo metodo però potrebbe avere alcune limitazioni:

- L'**aggressività** dell'algoritmo: un componente sorgente che si presume sia un artefatto viene rimosso, in realtà non sappiamo se l'algoritmo sta davvero rimuovendo una componente rumorosa o se insieme alla componente presunta ci sono anche importanti informazioni neurali che potrebbero essere utili per la modellazione predittiva. L'aggressività di qualsiasi algoritmo di rimozione degli artefatti nell'EEG è distruttiva per la modellazione predittiva. Quindi sarebbe necessario un controllo visivo per determinare quali componenti rappresentino gli artefatti, per poterli eliminare correttamente.
- La **convergenza** effettiva di ICA: dovuta al fatto che l'ICA per stimare la matrice di demiscelazione, lavora sulla minimizzazione delle informazioni reciproche tra i componenti indipendenti dei segnali sorgente, c'è sempre un problema con la convergenza della matrice. Potrebbero non esserci segnali sorgente indipendenti.
- Durante la rimozione di qualsiasi componente, introduce componenti di frequenza extra, che possono essere osservate nello spettro (grafico del periodogramma). Nel dominio del tempo, queste componenti corrispondono a dei **picchi,** cioè brusche variazioni del segnale.