Metodi di Intelligenza Artificiale e Machine Learning per la Fisica
Progetto 2

Gruppo 27: Marco Guarriello, Simone Fioretto

5 Giugno 2024

1 Abstract

L'obiettivo principale del secondo progetto è la realizzazione di un modello AutoEncoder (AE), che riesca a svolgere:

- la ricostruzione dei dati di input;
- la separazione delle classi nello spazio latente;
- la ricostruzione dei dati e l'anomaly detection su un nuovo dataset di test

2 Introduzione

Il seguente progetto verrà **diviso in due parti principali**, a seconda del dataset che si sta implementando e applicando al modello[1].

Parte 1: Nella prima parte verrà implementato, analizzato, ottimizzato e ridimensionato il dataset di train (2) per l'allenamento del modello. Verrà inoltre definita la classe per l'autoencoder, istanziato il modello e allenato sul dataset di train precedentemente perfezionato, andando ad osservare successivamente le performance di ricostruzione delle tracce audio del modello.

Parte 2: Nella seconda parte del progetto verrà implementato il dataset di test (2), dato ciò verrano eseguiti i medesimi passaggi di analisi e ottimizzazione applicati al dataset di train. Con il modello già precedentemente allenato si vanno a valutare le sue performance generali, verificandone le prestazioni di ricostruzione su dati mai visti (test). Come ultimo task verrà eseguita un anomaly detection con classificazione tramite clustering sul dataset di test.

Il **dataset di train** fornito (*gruppo_0.zip*) per l'allenamento del modello di AutoEncoder (che verrà trattato successivamente), contiene 450 file audio (.wav) di durata 4s campionati a 16kHz. Ogni file, ha come nome per essere distinto, una stringa di caratteri casuali. In base al nome di ogni file è associato un label in un ulteriore file csv nel formato: /content/data/N_CLASSE/caratteri.wav , randomstring.wav.
Il dataset fornito contiene 3 tipologie differenti di audio, divise nel seguente ordine:

1. **label 0 = Bird**: file audio contenente principalmente suoni di uccelli e della natura. La seguente categoria risultà però essere particolarmente problematica data la poca accuratezza nella selezione dei suoni, essendo molto variegati e talvolta con quantità di rumore non indifferente;
2. **label 1 = Korean songs**: file audio contenente canzoni coreane per bambini;
3. **label 2 = Arabic**: file audio contenente dialoghi o monologhi in lingua araba.

Per il **dataset di test** fornito, esso contiene 70 file (4s campionati a 16kHz) contenente le stesse tipologie di audio. Ciò che differisce dal dataset di train è l'assenza di label associate e la presenza di file audio anomali all'interno del dataset, sulla quale si andrà a fare una classificazione non supervisionata con l'aggiunta di un anomaly detection.

[1] L'intero codice è implementato in modo tale da essere utilizzabile sia con risorsa CPU, sia con GPU.

Parte 1 - *gruppo_0*

3 Analisi dei dati e scelta della strategia di implementazione

All'inizio della prima parte, ci si soffermerà, per entrambi i dataset a disposizione [2], sull'analisi e l'ottimizzazione di quest'ultimi, attraverso le seguenti azioni:

1. **Estrazione e preparazione dei dati dalla cartella *gruppo_0* e dal file *labels_0***: Estrazione delle tracce e dei labels e preparazione dei dati audio associando ogni label alla rispettiva traccia;
2. **Normalizzazione e rimozione tracce fuori shape**: Normalizzazione degli ordini di grandezza per ogni audio e rimozioni delle tracce con lunghezza diversa da 64000 punti per non incorrere successivamente a problematiche dovute dallo shape;
3. **Taglio delle tracce su frammenti di audio**: Analisi su frammenti di lunghezza arbitraria;
4. **Preparazione dataset e dataloader per training e validation del modello**: preparazione e trasformazione dei dati in tensori Torch, utilizzati successivamente per la creazione del dataset e dataloader per il training e validation del modello.

3.1 Estrazione e preparazione dei dati dalla cartella *gruppo_0* e dal file *labels_0*

Utilizzando l'attributo di Pandas per la lettura di file *.csv* si sono estratti dal file *labels.csv* i nomi dei file associati alle rispettive labels.
Attraverso una funzione *wav_labels_file_reading*, che prende in ingresso la cartella di origine dei file e la cartella di estrazione dei file, si sono estratti i file audio da *gruppo_0.zip* facendoli scorrere attraverso il dataset delle label ricavato in precedenza. La funzione restituisce i nomi, i dati audio e la frequenza di campionamento di tutti i 450 file. A ciascun file audio corrisponde un array di dati audio di (circa) 64000 punti[3].

3.2 Normalizzazione e rimozione tracce fuori shape

Si utilizza una funzione *Normalize*, che ricerca i valori massimo e minimo dell'intero dataset, per poter poi normalizzare tra 0 e 1 i dati, mantenendo così le "dimensioni" (ampiezza della traccia) originarie.
Per fare in modo, inoltre, che tutti gli array audio abbiano la stessa dimensione si sono ricercati gli array che non avessero lunghezza pari a 64000. Ne sono stati trovati 5, che sono stati successivamente eliminati per l'allenamento del modello, per non incorrere così ad eventuali problemi di dimensione degli audio.

3.3 Taglio delle tracce su frammenti di audio

L'allenamento del modello non viene effettuato sui dati audio nella loro interezza, ma su frammenti di lunghezza personalizzabile, in maniera tale da acellerare la convergenza dell'allenamento.
La funzione *cutter* implementata a tal proposito taglia un frammento di ciascun file audio della dimensione ricercata. Si può scegliere inoltre, se si vuole che gli audio vengano tagliati in frammenti "localizzati" nello stesso punto dell'audio o se questi vengano scelti in maniera casuale per ciascun file.

3.4 Preparazione dataset e dataloader per training e validation del modello

Prima di poter passare per un AutoEncoder convoluzionale, si convertono gli array di audio e labels in tensori Torch che vanno poi a formare un unico Tensor Dataset *dataset_audio*. Si utilizzano le varie librerie e funzioni di Torch. Il dataset viene separato in due campioni:

- *dataset_train*: campione di train, corrispondente all'80%
- *dataset_val*: campione di validation, corrispondente al 20%

Da questi dataset vengono creati i corrispettivi DataLoader di train e validation, da cui vengono estratti i primi batch e di cui si esegue un reshape per avere una struttura del tipo:
torch.Size([n_batch,1,lunghezza_frammento_audio]).
Avendo alleggerito il dataset tramite il taglio delle tracce, è stato possibile impostare il valore della *batch_size* = 1 (essendo comunque un operazione che appesantisce il carico di lavoro del modello durante il training), in

[2] Anche se nella prima parte della relazione, questi sono passaggi di analisi dati che verranno effettuati su entrambi i dataset, di conseguenza non verranno citati/spiegati nuovamente nella parte 2 della relazione.
[3] Per il secondo dataset (test), dato che non vi sono label viene utilizzata un'altra funzione *wav_file_reading*.

maniera tale che venga preso un campione per volta per l'aggiornamento dei pesi, ottenendo così prestazioni nettamente migliori rispetto ad una *batch_size* più alta.

4 Implementazione dell'AutoEncoder

Successivamente alla preparazione, analisi e ottimizzazione dei dati, è stata implementata la classe per la definizione dell'AutoEncoder. La scelta sull'implementazione dell'architettura da utilizzare è ricaduta sugli strati convoluzionali 1D, creando così un AutoEncoder convoluzionale. Esse, oltre ad essere più leggere degli strati ricorrenti, si adattano particolarmente bene ai file audio.

4.1 Metodologia di implementazione e utilizzo della classe

Come primo passo, è stata definita la classe per poter successivamente istanziare il modello AE. La metodologia usata per creare la sudétta classe, è stata quella di comporla nella maniera più generalizzata possibile.
Questa scelta è stata dettata dall'esigenza di dover allenare svariate volte il modello, per poter comprendere al meglio il pattern di valori ottimali per gli iperparametri, senza dover andare ad alterare la struttura interna della classe per ogni minima modifica.
Essa è stata realizzata in maniera tale, da prendere 7 parametri in input, nel momento in cui il modello viene istanziato (quindi all'interno del costruttore **__init__**). I seguenti parametri sono:

- **Encoder_size**: Esso prende in input una lista di numeri, che andranno a definire il numero di strati con relativo numero di filtri in input e output per la convoluzione (es: encoder_size = [10,8,6] definisce tre strati con rispettivamente 10, 8 e 6 filtri applicati agli ouput di ogni layer);
- **Decoder_size**: Medesimo utilizzo del Encoder_size ma applicato agli strati del decoder;
- **Latent_dim**: Definisce il numero di filtri da applicare all'output dell'ultimo strato del encoder. A discapito del nome, esso non rappresenta la dimensione effettiva dello spazio latente, essendo applicata ad uno strato convolutivo e non ad uno denso, non ottenendo cosi la dimensione che si inserisce;
- **Kernel_size, Stride e Padding**: tutti e tre i parametri prendono in input una lista di numeri, che definisce il valore del kernel, stride e padding per ogni strato del modello;
- **Dropout**: Prende in input un numero reale $\in (0, 1)$ aggiungendo tra uno strato e l'altro il dropout. Inserendo Dropout=False esso non verrà applicato;

Riguardo ai **metodi della classe**, oltre al costruttore __init__ e al metodo forward, essa comprende altri 4 metodi di istanza, implementati per migliorare l'ordine e la rapidità dei passaggi una volta istanziato il modello. La classe comprende dunque i seguenti metodi aggiuntivi:

- **Metodo compile**: definisce la loss, l'ottimizzatore, il learning rate, lo scheduler (ulteriori iperparametri) e stampa il sommario del modello;
- **Metodo save**: salva gli audio ricostruti (ad una lunghezza arbitraria) all'interno della cartella Reconstructed_Audios_Folder;
- **Metodo fit**: allena il modello e calcola le loss sul training e validation set, ad ogni epoca salva i parametri e i pesi del modello all'interno della cartella Training_Checkpointers_Folder. Concluso il training, restituisce le liste delle loss su training e validation set per ogni epoca.

4.2 Scelta della funzione di Loss e degli iperparametri del modello

La **funzione di Loss** scelta per l'allenamento del modello è ricaduta sulla **SmoothL1loss**, è una funzione di perdita che combina le caratteristiche delle funzioni di perdita L1 (Mean Absolute Error) e L2 (Mean Squared Error), cercando di mitigare i problemi che ciascuna di queste funzioni può causare individualmente:

$$\text{SmoothL1Loss}(x, x_{rec}) = \begin{cases} \frac{1}{2}(x - x_{rec})^2 & \text{se } |x - x_{rec}| < 1 \\ |x - x_{rec}| - \frac{1}{2} & \text{altrimenti} \end{cases} \tag{1}$$

Si passa successivamente alla **scelta degli iperparametri del modello**: non essendoci un vero e proprio metodo teorico per la scelta ottimale di tutte le combinazioni degli iperparametri presenti all'interno della definizione del modello, sono state effttuate e analizzate svariate prove empiriche (seguendo ovviamente delle linee teoriche generali) tramite i risultati ottenuti dal training. Dunque, successivamente a quest'ultime, la scelta per la combinazione migliore degli iperparametri del modello, è ricaduta sui seguenti valori:

- *Encoder_size = [16,8], Decoder_size = [8,16]*: Sono stati applicati solamente 2 strati aggiuntivi con 16 e 8 filtri applicati agli output e input dei rispettivi layer. Contando inoltre l'input iniziale (encoder) e l'output finale (decoder) che è pari a 1 (numero di canali), e l'ouput dell'encoder (rispettivo input del decoder), il modello avrà un totale di 6 strati; la seuguente scelta del numero di strati aggiuntivi e filtri dell'autoencoder è risultata la più performante tra tutte. Il modello, grazie alla seguente configurazione di strati e filtri, non risulta essere oltremisura pesante, evitando dunque una eccessiva complessità (scanzando il problema dell'overfitting) e lavorando nella maniera più ottimale possibile sul dataset proposto;
- *Latent_dim = 5*: Porta semplicemente risultati migliori sulle loss del training e validation set, su tutte le possibili combinazioni degli iperparametri;
- *Kernel_size = 4, Stride = 2, Padding = 1*: per tutti e tre gli iperparametri, i seguenti valori sono stati applicati per ogni strato (es: 3 strati Encoder: kernel_size = [4,4,4]). La seguente scelta è stata dettata dall'esigenza di dover avere la dimensione dell'output finale dell'AutoEncoder, uguale a quella dei dati in input, che è stata possibile grazie alla scelta dei sopra citati valori;
- *Dropout = False*: La presenza del Dropout non aiutava a mitigare il rischio di overfitting, dato anche dalla sua bassa presenza per la semplicità del modello proposto, rischiando dunque inutilmente di abbassarne le prestazioni durante l'allenamento; [4]
- all'interno del **metodo compile**: sono stati scelti un learning rate = 0.003 , ottimizzatore = Adam, scheduler di tipo = ReduceLROnPlateau.

Oltre alla scelta ottimale degli iperparametri, per aumentare ulteriormente le prestazioni del modello, è stato deciso di applicare l'inizializzazione intelligente dei pesi, utilizzando la **Glorot uniform initializer**, essa inizializza i pesi in maniera tale che la varianza delle attivazioni dei neuroni sia la stessa in tutti i layer della rete, mantenendo la varianza costante attraverso gli strati, definita come:

$$W^i \sim \mathcal{U}\left(-\sqrt{\frac{6}{n^i + n^{i+1}}}, \sqrt{\frac{6}{n^i + n^{i+1}}}\right) \tag{2}$$

dove:

- W^i sono i pesi dell' i-esimo layer.
- $\mathcal{U}(a, b)$ indica una distribuzione uniforme tra a e b.
- n^i è il numero di unità in input (neuroni nel layer precedente) dell' i-esimo layer.
- n^{i+1} è il numero di unità in output (neuroni nel layer corrente) dell' i-esimo layer.

4.3 Allenamento del modello

Data la scelta degli iperparametri, si passa all'allenamento del modello sul training set e andandone a valutare le prestazioni generali sul validation set. Per il training del modello sono bastate **100 epoche**, ottenendo il seguente andamento della smooth loss in funzione delle epoche:

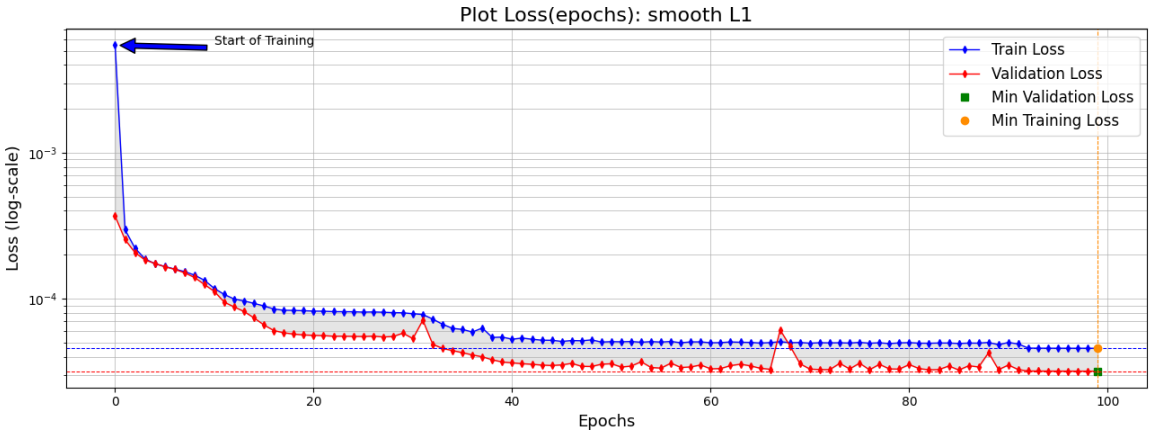


Figura 1: Smooth Loss in funzione delle epoche

[4] è stato ugualmente deciso di lasciarlo all'interno della classe di definizione del modello nel caso si volesse aggiungere per eventuali prove.

Si tenga conto del fatto che, ogni volta che si istanzia e allena il modello da zero, si ottengono sempre valori delle loss diferenti, in media **Train/Validation Loss**=0.00008 ÷ 0.00002.
Si vanno dunque a calcolare le incertezze su quest'ultime, utilizzando il metodo della ripetizione degli esperimenti **Bootstrap**. Il seguente metodo prevede una ripetizione dell'allenamento del modello per un certo numero di volte arbitrario, su frammenti o insiemi di dati sempre differenti (ciò sarà facilmente possibile farlo grazie alla funzione cutter che prende per ogni traccia audio, un frammento casuale della grandezza desiderata). Una volta che sono state ottenute n-Loss, si calcola la media e la deviazione standard su quest'ultime, ottenendone la stima finale. [5]
Si effettuano dunque 10 allenamenti differenti del modello, ottenendo i seguenti valori delle loss per train e validation con relative stime finali:

Train Loss	Val loss
0.00003823	0.00003149
0.00004660	0.00004593
0.00004944	0.00004665
0.00006080	0.00004161
0.00003895	0.00004012
0.00006668	0.00005441
0.00005441	0.00002761
0.00004947	0.00005175
0.00006849	0.00003454
0.00003363	0.00004847

Tabella 1: Valori finali della train e validation loss per ogni allenamento del modello

Train Loss finale:	0.000051 ± 0.000011
Validation Loss finale:	0.0000423 ± 0.0000083

Tabella 2: Stime finali per Train e Validation Loss

4.4 Ricostruzione delle tracce audio

Per ottenere una visualizzazione delle performance del modello, si procede ricavando una ricostruzione dei frammenti di file audio del training.
Si utilizza la funzione *Reconstruction* (spiegata nel notebook colab). Di seguito sono riportati i plot di un frammento estratto casualmente dal dataset di training (delle tracce tagliate), con la sua ricostruzione, visualizzandolo in due punti zommati della traccia, rispettivamente in assenza e presenza di rumore, in maniera tale da poter osservare più efficientemente le performance del modello:

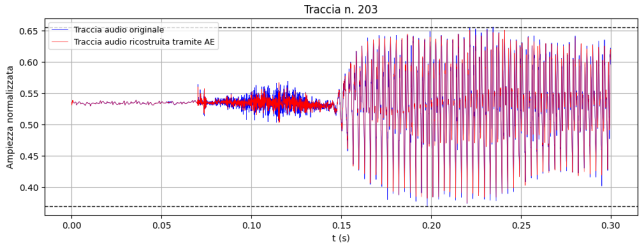


Figura 2: Grafico traccia originale con decoded su frammenti di 4800 punti

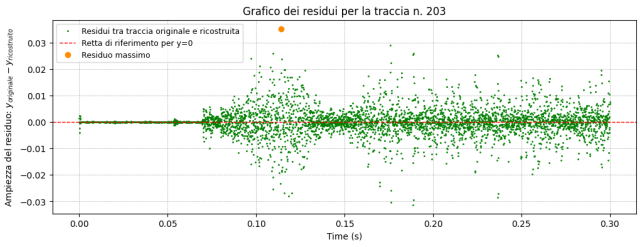


Figura 3: Grafico dei residui tra frammenti di audio originale e ricostruito della traccia in esame

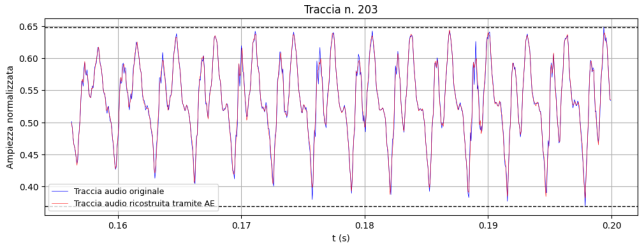


Figura 4: Zoom del grafico precedente nella zona (in assenza di rumore) [2500,3200]

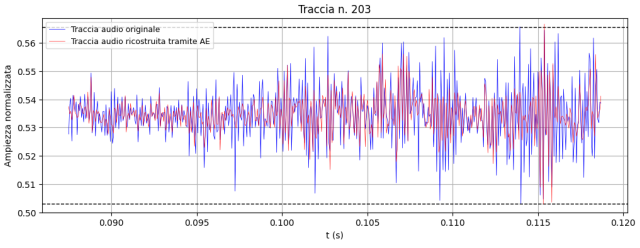


Figura 5: Zoom del grafico precedente nella zona (in presenza di rumore) [1400,1900]

[5] L'analisi per il calcolo delle stime finali non è presente nel notebook colab, essendo stato effettuato manualmente per poi salvare i risultati desiderati e calcolare le stime finali.

Come si puo notare dai grafici di ricostruzione ottenuti, l'AutoEncoder realizzato lavora in maniera ottimale sia nella zone con poco (o assente) rumore, effettuando una ricostruzione fedele della traccia, sia nelle zone con alta presenza di rumore, diminuendone dunque l'ampiezza dell'audio ricostruito.
Dopo aver verificato che la ricostruzione sia fedele all'originale, si procede tentando di ricostruire interamente gli audio del dataset in analisi, richiamando la funzione *Reconstruction*, includendo il salvataggio degli audio ricostruiti in una cartella *Reconstructed_ Audios_ Gruppo0* creata nell'archivio file di Colab. Di seguito è riportata la ricostruzione di un'intera traccia scelta causalmente.

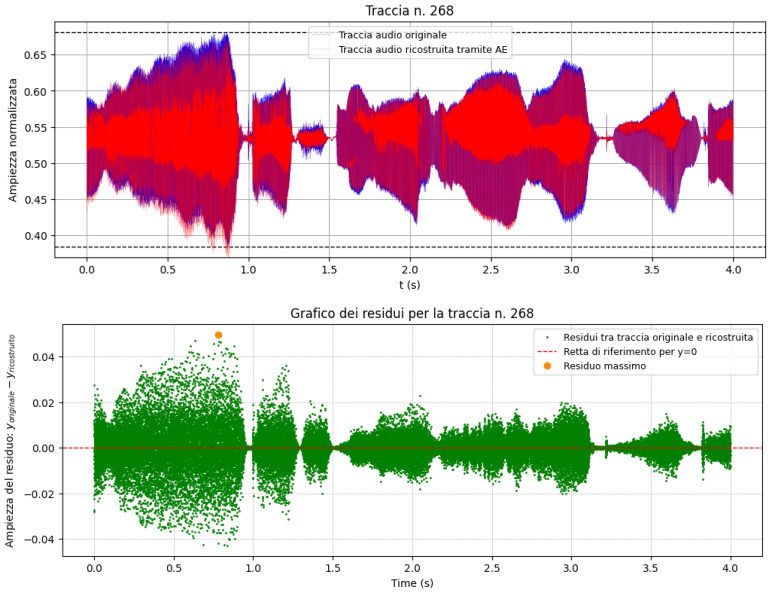
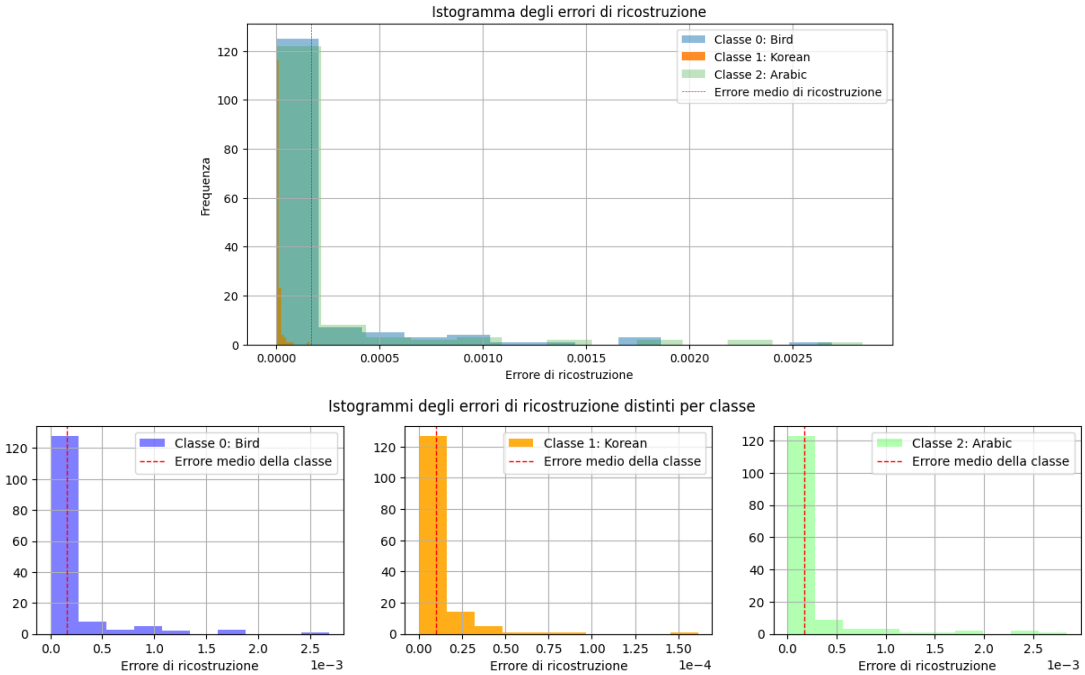


Figura 6: Esempio di ricostruzione di una traccia intera con grafico dei residui. La ricostruzione ha un margine di errore del 7% circa. Ovviamente la performance della ricostruzione è dovuta anche alla struttura dell'audio e alla sua forma d'onda: più l'audio ha "rumore", più la ricostruzione avrà errori maggiori.

4.5 Istogramma degli errori

Si vuole visualizzare quanto varia l'errore di ricostruzione per ciascuna delle tre classi del dataset attraverso un istogramma degli errori di ricostruzione.
Viene chiamato il modello in modalità di valutazione, per poi estrapolarne le rappresentazioni latenti di ciascun file audio, calcolando l'errore di ricostruzione. Si suddividono successivamente per classe gli errori ottenuti, si definisce un dizionario degli errori di ricostruzione per ogni classe, che verranno visualizzati in un istogramma.



Si può notare come, tra tutte le classi, la categoria '*Korean*' è quella che ha errori di ricostruzione più piccoli, di conseguenza quella con ricostruzione tendenzialmente più fedele all'originale. Le classi '*Arabic*' e '*Bird*' hanno invece errori più grandi di due ordini. I suoni della natura hanno comunque un range di errore più ampio data la loro poca accuratezza.

4.6 Separazione delle clasi nello spazio latente

Viene installa e importata *UMAP* per ottenere una visualizzazione dello spazio latente.
I dati audio originali vengono convertiti in un tensore PyTorch e preparati per essere inseriti in un DataLoader, che gestisce la suddivisione dei dati in batch.

Il codice procede all'estrazione delle rappresentazioni latenti, iterando sui batch di dati attraverso il modello preaddestrato, e raccoglie gli output dell'encoded. Queste rappresentazioni vengono quindi concatenate e convertite in un array.
Dato che le rappresentazioni latenti del seguente modello hanno una dimensionalità maggiore a 2 (dimensione desiderata per poter realizzare uno scatter plot 2d sullo spazio latente), viene utilizzato UMAP così da poter ridurre le dimensioni delle rappresentazioni a due, rendendone più facile la visualizzazione e l'analisi. Nella figura accanto è riportato il plot dello spazio latente classificato con le labels.

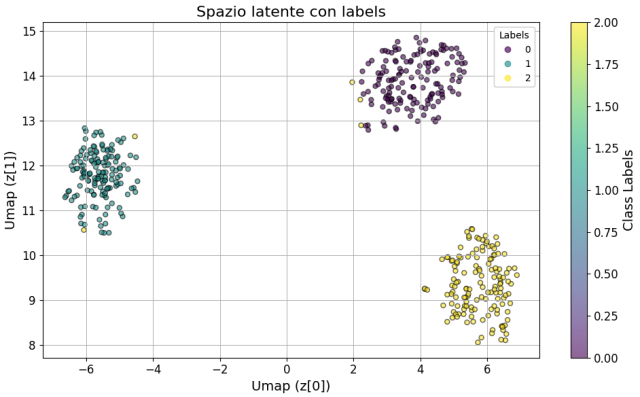


Figura 7: Plot dello spazio latente

Si può notare come, l'encoder del modello proposto riesca a separare nitidamente le classi del dataset, che è stato possibile visualizzare grazie alla riduzione dimensionale effettuata con UMAP.

Parte 2 - *gruppo_27*

5 Applicazione del modello al dataset di test

Si inizia ora ad analizzare il dataset di test *gruppo_27* (senza label).
Vengono eseguite le operazioni preliminari di trattamento dei dati riportate nelle sezioni 3.1 e 3.2.
Si passa dunque alla **ricostruzione delle tracce di test**. Il tensore ricavato dopo normalizzazione e "taglio" viene incapsulato in un TensorDataset, che viene poi utilizzato per creare un DataLoader con batch di dimensione 1, senza mescolamento dei dati.
Il modello viene quindi impostato in modalità di valutazione e viene inizializzato il calcolo della loss con la funzione SmoothL1Loss.
La funzione *Reconstruction* viene chiamata per ricostruire l'audio, calcolare la loss e salvare i file ricostruiti nella cartella *Reconstructed_Audios_Gruppo27*, utilizzando una frequenza di campionamento di 16kHz.
Si riporta di seguito il grafico di una traccia casuale ricostruita e i suoi residui.

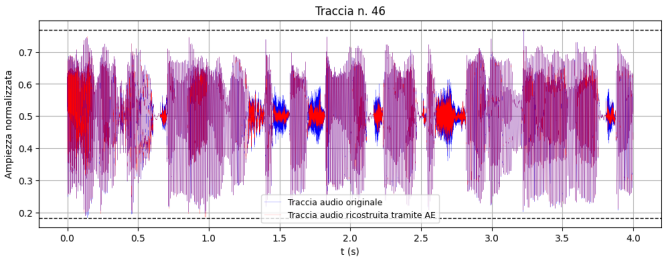


Figura 8: Grafico intera traccia originale (blu) con decoded (rosso).

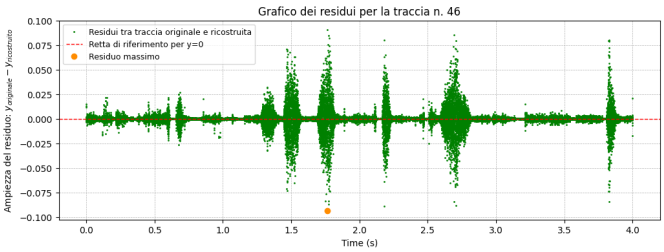


Figura 9: Grafico dei residui tra audio originale e ricostruito della traccia in esame

6 Classificazione delle tracce e Anomaly Detection

Dopo aver verificato che la ricostruzione si presta ad una buona approssimazione delle tracce originali, si procede all'estrazione di rappresentazioni latenti, del clustering e della rilevazione delle anomalie nei dati audio.
Viene importato il metodo di clustering KMeans da scikit-learn.
Le rappresentazioni latenti degli audio vengono estratte passando i dati attraverso il modello pre-addestrato, e contemporaneamente vengono calcolati gli errori di ricostruzione. Esse vengono poi ridotte a due dimensioni utilizzando UMAP, se necessario, per facilitare il clustering. Successivamente, viene eseguito il **clustering** delle rappresentazioni latenti bidimensionali utilizzando l'algoritmo KMeans. Un modello KMeans viene addestrato con tre cluster, e le rappresentazioni latenti vengono classificate nei cluster corrispondenti.

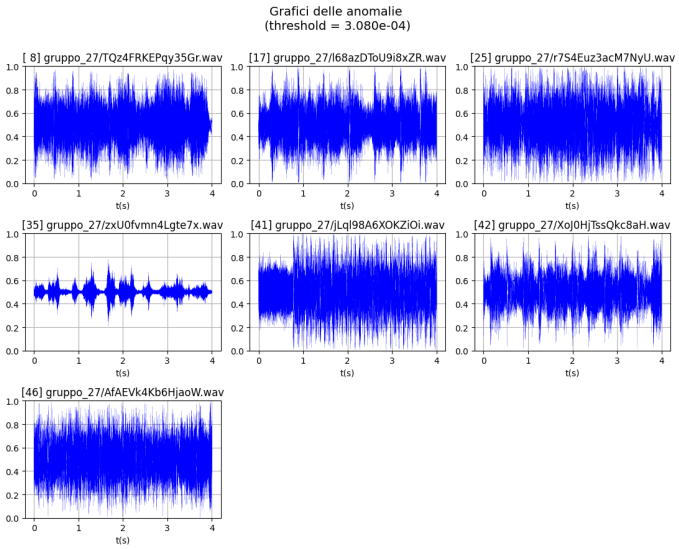


Figura 10: Risultati A-D con soglia $3.08 \cdot 10^{-4}$.

Per la rilevazione delle anomalie, è stata calcolata una soglia basata sul 90° percentile degli errori di ricostruzione.
Ogni punto di dati test viene classificato come normale o anomalo in base a questa soglia e alla sua appartenenza ai cluster. Se l'errore di ricostruzione supera la soglia, il punto è considerato un'anomalia e gli viene assegnata l'etichetta *-1*. Altrimenti, il punto viene assegnato al cluster predetto dal modello KMeans.
Infine, il codice stampa i risultati della rilevazione delle anomalie e della classificazione, elencando i file che sono stati identificati come anomali e i loro rispettivi cluster.
Se non vengono rilevate anomalie, viene indicato chiaramente nell'output.
Nela figura accanto sono riportati i risultati dell'anomaly detection [a].

[a] NB. L'indice riportato sul grafico corrisponde al numero del file, non al suo effettivo indice nel file, l'indice di ciascun file sul documento csv è *indice-1*.

Conclusioni e osservazioni

Prestazioni del modello

Vengono riportate di seguito i valori della loss per training, validation, test ottenuti e l'errore quadratico medio di ricostruzione per ogni classe:

Train Loss	Validation Loss	Test Loss
0.000051 ± 0.000011	0.0000423 ± 0.0000083	0.000287

Tabella 3: Valori finali delle loss di ricostruzione delle tracce (il risultato della test loss è molto variabile)

MSE generale	MSE classe Bird	MSE classe Korean	MSE classe Arabic
$1.1 \cdot 10^{-04}$	$1.8 \cdot 10^{-4}$	$1.11 \cdot 10^{-5}$	$2.5 \cdot 10^{-4}$

Tabella 4: Valori finali dell'errore quadratico medio di ricostruzione delle tracce per classe (mostrate negli istogrammi introdotti nella prima parte).

Dai risultati ottenuti per le loss, ricostruzione delle tracce audio, separazione delle classi nello spazio latente e dall'istogramma degli errori, sì evince che, il modello di AutoEncoder convoluzionale definito, si è adattato in maniera ottimale sul dataset di train, ottenendo successivamente ottime prestazioni sul dataset di test.

Classificazione e Anomaly Detection

Nella seguente sotto sezione, andiamo a discutere dei risultati ottenuti dalla classificazione non supervisionata applicata allo spazio latente ridotto del modello e dell'anomaly detection.

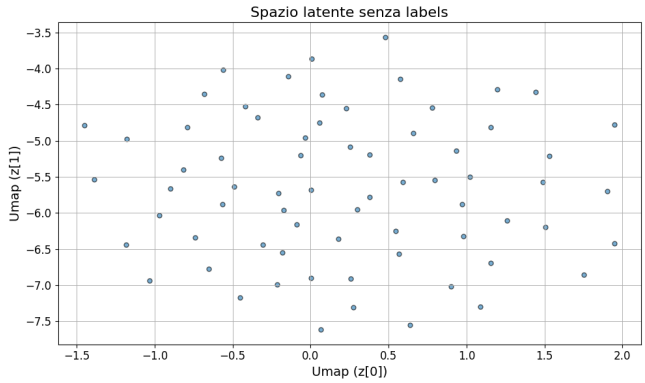


Figura 11: Scatter plot dello spazio latente dell'autoencoder (senza labels) ridotto tramite UMAP.

La **classificazione** sul dataset di test, ottenuta grazie al clustering sullo spazio latente (ridotto a 2D) dell'AutoEncoder, risulta essere parzialmente imprecisa. I risultati ottenuti possono essere spiegati dalla mancata capacità di riuscire a separare nitidamente le classi del dataset di test, data la rappresentazione latente del modello (a discapito di quanto ottenuto col dataset di train). Nella figura di fianco è riportato lo scatter plot 2D dello spazio latente del modello (senza labels), sul dataset di test proposto. Come è possibile notare, non avviene una separazione netta delle classi (facilmente osservabile anche in assenza della gamma di colori a suddividere le classi presenti).

Ciò può spiegare la bassa precisione nella classificazione non supervisionata, essendo stata applicata proprio sui valori dello spazio latente ridotto.

L' **anomaly detection** che si ottiene tramite il metodo spiegato nella sezione (6), funziona (per lo più) in maniera corretta. Per ogni esecuzione diversa del codice, il modello tramite i passaggi spiegati precedentemente, in media, riesce quasi sempre a selezionare 6 anomalie giuste su 7.
Purtroppo, il dataset di test proposto, non avendo un elevato numero di tracce e quindi una alta quantità di anomalie, non basterà per avere una statistica affidabile e completa sul funzionamento esatto dell'anomaly detection. Ma per quanto proposto e mostrato dai risultati ottenuti, il modello sembra distinguere in maniera abbastanza efficiente la maggior parte delle anomalie presenti nel dataset.