

ALMA MATER STUDIORUM · UNIVERSITÀ DI BOLOGNA

---

SCUOLA DI INGEGNERIA E ARCHITETTURA

DIPARTIMENTO DI INFORMATICA - SCIENZA E INGEGNERIA -

CORSO DI LAUREA MAGISTRALE IN INGEGNERIA INFORMATICA

ANNO ACCADEMICO 2020/2021

**- RELAZIONE FINALE -  
TRASCRIZIONE AUTOMATICA  
DI TABLATURE PER CHITARRA  
TRAMITE L'USO DI CNN**

CORSO DI SISTEMI DIGITALI M

GRUPPO:  
**DE NARDI-TORNATORE**



# Indice

<b>Introduzione</b>	<b>1</b>
<b>1 Chitarra</b>	<b>3</b>
1.0.1 Corde . . . . .	4
1.0.2 Tasti . . . . .	4
1.0.3 Tab . . . . .	5
<b>2 Trasformata a Q Costante</b>	<b>7</b>
<b>3 Architettura e addestramento del modello</b>	<b>9</b>
3.1 Machine learning . . . . .	9
3.2 Librerie utilizzate . . . . .	10
3.3 Set dati GuitarSet . . . . .	10
3.3.1 Ricavare le tab dai file .jams . . . . .	11
3.3.2 Ricavare le immagini dai file audio . . . . .	15
3.3.3 Pre-elaborare i dati . . . . .	17
3.4 Modello della rete . . . . .	20
3.4.1 Uso di Keras . . . . .	21
3.4.2 Addestramento del modello . . . . .	26
3.5 Valutazione del modello . . . . .	28
<b>4 Implementazione su dispositivo Embedded</b>	<b>31</b>
4.1 Conversione del modello da Keras a Tensorflow Lite . . . . .	31
4.2 Sviluppo applicazione Android . . . . .	32
4.2.1 Sviluppo applicazione con Java 1.8 e Android Studio 4.1.2 .	32
4.2.2 Pre-elaborazione dell'audio . . . . .	34
4.2.3 Implementazione di Tensorflow Lite . . . . .	36
4.3 Sviluppo applicazione iOS . . . . .	37
4.3.1 Conversione del modello da Keras a CoreML . . . . .	37
4.3.2 Sviluppo applicazione con Swift 5 e Xcode 12 . . . . .	41
4.3.3 Uso di un server per l'uso della libreria librosa . . . . .	43

4.3.4	Implementazione di Tensorflow Lite . . . . .	44
4.3.5	Installazione su dispositivo fisico . . . . .	46
<b>5</b>	<b>Conclusioni</b>	<b>47</b>

# Introduzione

La musica ha assunto un ruolo di primo piano nella storia dell'uomo. Forse è la voglia di esprimere i propri sentimenti e le proprie emozioni che hanno obbligato l'essere umano a comporre sempre nuovi brani. Senza alcun dubbio, la chitarra è uno degli strumenti più usati. Per esempio, quando si va in campeggio e alla sera ci si siede davanti al falò, sono proprio le sue note a riempire l'aria.

Il tema libero lasciato dai professori ha permesso di approfondire ma soprattutto conoscere meglio il vastissimo campo della musica.

Il progetto ha l'obiettivo di trascrivere in modo automatico le tablature per chitarra tramite l'utilizzo di reti neurali convoluzionali (*CNN*) in modo che anche chi non è in grado di suonare questo strumento lo possa fare. Il tutto funziona tramite l'ausilio di un semplice *smartphone*.

La relazione è articolata come segue:

- nel **capitolo 1** viene descritta brevemente la chitarra;
- nel **capitolo 2** viene introdotto il dominio in cui lavoreremo;
- nel **capitolo 3** viene descritto il modello della nostra rete, l'addestramento che è stato eseguito e la sua accuratezza;
- nel **capitolo 4** viene realizzata l'implementazione sul dispositivo *embedded*, nel nostro caso sono gli *smartphone*;
- il **capitolo 5** conclude la relazione presentando i risultati ottenuti, gli obiettivi raggiunti ed eventuali problematiche che potrebbero essere risolte in futuro.

La relazione è stata scritta come un *diario* mettendo in risalto tutti i passaggi, i tentativi e i problemi che si sono verificati.

Il codice, il modello e i *log* di tutto il progetto possono essere reperiti nella seguente repository git: <https://github.com/it9tst/tab-writer>



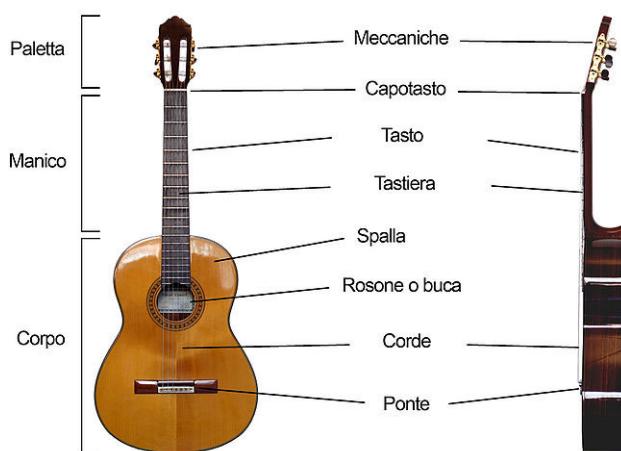
# Capitolo 1

## Chitarra

La chitarra ha una lunga tradizione che affonda le sue radici addirittura al tempo degli arabi. I primi esemplari risalgono al tredicesimo secolo. Inizialmente dotata di quattro corde, si è accresciuta nel Rinascimento di un'altra corda, arrivando poi nel periodo Barocco all'attuale numero di sei.

La chitarra è composta da due parti principali:

- il **manico**, su cui si trova la tastiera e che termina con la paletta la quale ospita le meccaniche per l'accordatura;
- la **cassa di risonanza** o tavola armonica con una cavità centrale, che serve ad amplificare il suono prodotto dalle corde.

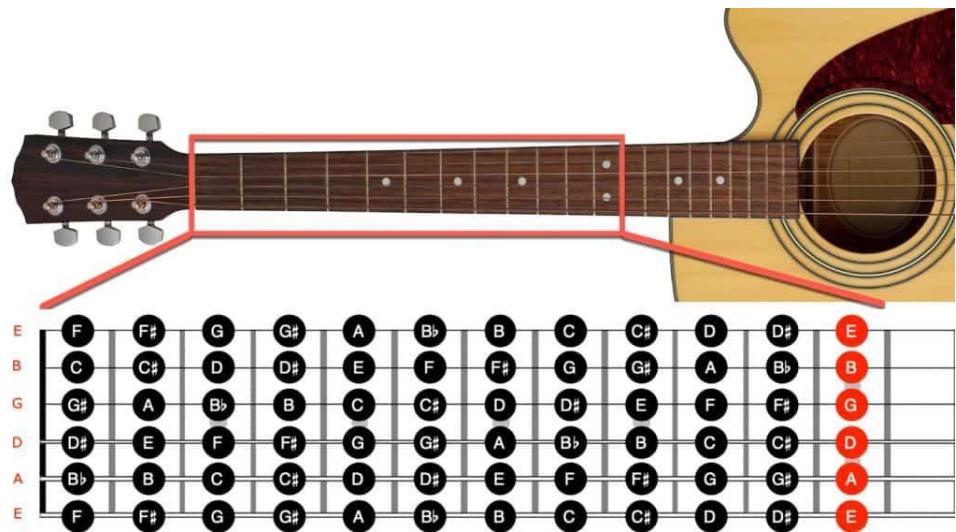


### 1.0.1 Corde

Le corde delle chitarre moderne sono sei e sono ordinate dall'alto verso il basso nel seguente modo:

- la prima corda corrisponde alla nota *Mi cantino* (e);
- la seconda corrisponde alla nota *Si* (B);
- la terza corrisponde alla nota *Sol* (G);
- la quarta corrisponde alla nota *Re* (D);
- la quinta corrisponde alla nota *La* (A);
- la sesta corrisponde alla nota *Mi basso* (E).

L'ultima corda dell'elenco è quella più spessa, mentre la prima è la più sottile.



### 1.0.2 Tasti

Sul manico della chitarra c'è la tastiera. Si chiama tastiera proprio perchè ci sono i tasti. Quest'ultimi sono delimitati da delle barrette di metallo e ognuno di essi corrisponde a una nota. Dunque, se abbiamo una chitarra a diciannove tasti, possiamo fare diciannove note diverse per ogni corda.

La distanza tra due tasti della stessa corda prende il nome di **semitono**. Ad esempio, se premiamo la sesta corda in corrispondenza del *La*, poi premendo la corda al tasto adiacente più vicino alla cassa di risonanza (un semitono più alto)

ascolteremo un *La*#. Se non premiamo nessun tasto la corda si dice che è suonata a vuoto. Le sei corde suonate a vuoto devono emettere dei suoni ben precisi. Dunque, la chitarra deve essere accordata. L'accordatura classica delle sei corde, ovvero la nota che devono suonare le corde a vuoto (dal basso all'alto), è la seguente: *Mi, La, Re, Sol, Si, Mi*.

Conoscendo il suono prodotto dalle sei corde suonate a vuoto e sapendo che ogni nota suonata ad un tasto dista di un semitono dalla nota suonata al tasto adiacente possiamo mappare tutta la tastiera della chitarra.

MI	FA	FA#	SOL	SOL#	LA	LA#	SI	DO	DO#	RE	RE#	MI
SI	DO	DO#	RE	RE#	MI	FA	FA#	SOL	SOL#	LA	LA#	SI
SOL	SOL#	LA	LA#	SI	DO	DO#	RE	RE#	MI	FA	FA#	SOL
RE	RE#	MI	FA	FA#	SOL	SOL#	LA	LA#	SI	DO	DO#	RE
LA	LA#	SI	DO	DO#	RE	RE#	MI	FA	FA#	SOL	SOL#	LA
MI	FA	FA#	SOL	SOL#	LA	LA#	SI	DO	DO#	RE	RE#	MI

### 1.0.3 Tab

La *tab* è una rappresentazione delle corde della chitarra. Una tablatura è solitamente scritta usando sei linee orizzontali, ognuna corrispondente a una corda.

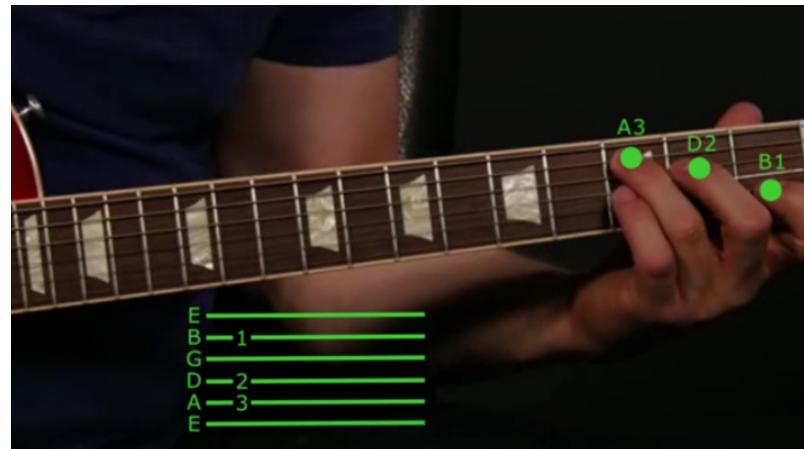
Al contrario dei normali spartiti, su una tablatura non ci sono le note da suonare ma si trovano le indicazioni su dove mettere le dita.

I numeri sulle linee corrispondono ai tasti della tastiera. Ad esempio, un "1" sulla prima corda, indica di suonare il *Mi cantino*, tenendo premuto il primo tasto.

Se il numero è maggiore o uguale a uno, bisogna premere il tasto corrispondente quando si suonerà quella corda. Se troviamo uno **zero** allora si suona la corda a vuoto, senza premere alcun tasto.



Spesso leggendo una tablatura si trovano dei numeri che sono allineati verticalmente. In questo caso si premono più tasti contemporaneamente. Le *tab* vanno lette come libri cioè da sinistra a destra.



# Capitolo 2

## Trasformata a Q Costante

Le note musicali si possono classificare nel seguente modo:

MUSICAL NOTE FREQUENCY CHART

NOTE	FREQ (Hz)																		
C0	16.35	C1	32.70	C2	65.41	C3	130.81	>C4<	>261.63<	C5	523.25	C6	1046.50	C7	2093.00	C8	4186.01		
C#0	17.32	C#1	34.65	C#2	69.30	C#3	138.59	C#4	277.18	C#5	554.37	C#6	1108.73	C#7	2217.46				
D0	18.35	D1	36.71	D2	73.42	D3	146.83	D4	293.67	D5	587.33	D6	1174.66	D7	2349.32				
D#0	19.45	D#1	38.89	D#2	77.78	D#3	155.56	D#4	311.13	D#5	622.25	D#6	1244.51	D#7	2489.02				
E0	20.60	E1	41.20	E2	82.41	E3	164.81	E4	329.63	E5	659.26	E6	1318.51	E7	2637.02				
F0	21.83	F1	43.65	F2	87.31	F3	174.61	F4	349.23	F5	698.46	F6	1396.91	F7	2793.83				
F#0	23.12	F#1	46.25	F#2	92.50	F#3	185.00	F#4	369.99	F#5	739.99	F#6	1479.98	F#7	2959.96				
G0	24.50	G1	49.00	G2	98.00	G3	196.00	G4	392.00	G5	783.99	G6	1567.98	G7	3135.96				
G#0	25.96	G#1	51.91	G#2	103.83	G#3	207.65	G#4	415.31	G#5	830.61	G#6	1661.22	G#7	3222.44				
A0	27.50	A1	55.00	A2	110.00	A3	220.00	A4	440.00	A5	880.00	A6	1760.00	A7	3520.00				
A#0	29.14	A#1	58.27	A#2	116.54	A#3	233.08	A#4	466.16	A#5	932.33	A#6	1864.66	A#7	3729.31				
B0	30.87	B1	61.74	B2	123.47	B3	246.94	B4	493.88	B5	987.77	B6	1975.53	B7	3951.07				

La lettera a sinistra identifica la nota musicale mentre il numero a destra rappresenta la sua frequenza.

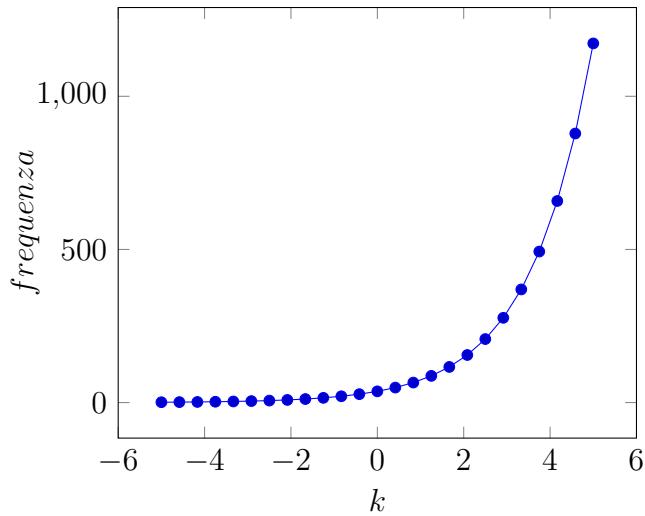
In musica, un'ottava è l'intervallo di otto note posizionate a frequenza diversa nella scala musicale. Le frequenze intermedie sono altre sei note. La frequenza tra una nota di un'ottava e la stessa nota di un'ottava successiva è doppia. Per esempio, il *La* centrale (A4) ha frequenza di 440 Hz, il *La* (A5) posto un'ottava sopra ha frequenza 880 Hz, quello un'ottava sotto (A3) ha frequenza 220 Hz.

Se si rappresentassero le prime sei ottave della nota *Do* (C) potremmo vedere che la sua frequenza raddoppia ad ogni ottava.

I tasti che intercorrono fra gli estremi della stessa ottava (esempio *Do* (C4) - *Do* (C5)) sono dodici semitonni per cui la frequenza deve raddoppiare ogni dodici semitonni. Si può rappresentare quanto detto dalla seguente formula:

$$F_k = 440 \text{ Hz} \cdot 2^{\frac{k}{12}}$$

Nel campo della musica viene utilizzata la trasformata a Q costante, a discapito della più nota trasformata di Fourier, proprio per la sua natura esponenziale. Inoltre, l'accuratezza della trasformata a Q costante è analoga alla scala logaritmica e imita l'orecchio umano, avendo una risoluzione di frequenza più alta a quelle più basse e una risoluzione più bassa alle frequenze più alte. Infatti, dal seguente grafico si può notare la natura esponenziale della funzione:



# Capitolo 3

## Architettura e addestramento del modello

I problemi che verranno segnalati nel seguente capitolo non si sono presentati subito ma solo quando si è testata l'accuratezza del modello. Dunque, ci sono paragrafi in cui si parla di esso senza averlo ancora definito.

### 3.1 Machine learning

Usare un algoritmo scritto da un programmatore non è sempre la soluzione migliore in quanto bisogna definire tutti i parametri e i dati necessari alla risoluzione del problema. Infatti, esiste anche un'altra strategia cioè quella di far imparare ad una macchina come risolverlo. La prima cosa che viene spontaneo chiedersi è: come fa una macchina ad imparare? Gli psicologi ci insegnano che l'apprendimento consiste nell'acquisizione o modifica di conoscenze, comportamenti, abilità, valori ed esperienze che sono poi utilizzati per vivere la vita di tutti i giorni. Tramite l'apprendimento creiamo delle regole generali che sono assimilabili in modelli di apprendimento. Questi modelli ci indicano come comportarci in una determinata situazione, ad interagire con gli altri, come leggere, come socializzare e così via. L'apprendimento è un processo iterativo, che continua per tutta la vita e ci permette di migliorare le nostre conoscenze a seconda delle informazioni che raccogliamo. Lo stesso fanno le macchine: dai dati di *input* che analizzano ricavano i modelli di apprendimento.

## 3.2 Librerie utilizzate

I *package* usati per realizzare il progetto sono:

- *numpy, versione 1.17.4*, è una libreria che aggiunge supporto a grandi matrici e *array* multidimensionali insieme a una vasta collezione di funzioni matematiche di alto livello per poter operare efficientemente su queste strutture dati;
- *jams, versione 0.3.4*, è una libreria che legge file *JSON* inerenti al mondo della musica;
- *librosa, versione 0.8.0*, è una libreria per la musica e l'analisi audio. Fornisce gli elementi necessari per recuperare informazioni musicali;
- *scipy, versione 1.4.1*, è una libreria di algoritmi e strumenti matematici che contiene moduli per l'ottimizzazione, per l'algebra lineare, elaborazione di segnali ed immagini e altro;
- *matplotlib, versione 3.1.2*, è una libreria per la creazione di grafici;
- *tensorflow, versione 2.1.0*, è una libreria utilizzata per il *machine learning* che fornisce moduli sperimentati e ottimizzati, utili nella realizzazione di algoritmi per diversi tipi di compiti percettivi e di comprensione del linguaggio;
- *keras, versione 2.3.1*, consente di implementazione algoritmi basati su reti neurali. Permette di sviluppare e prototipare in maniera semplice e veloce modelli nell'ambito del *machine learning* e del *deep learning*. Supporta come *back-end* *Tensorflow* ed è integrata in esso dalla versione 2.

## 3.3 Set dati GuitarSet

Fortunatamente, su Internet abbiamo trovato un *dataset* di *file* audio di chitarra già pronto su cui lavorare. Il *GuitarSet*, chiamato così dal suo creatore, è costituito dai file audio e dai suoi *tab*.

Questo *dataset* contiene trecentosessanta estratti di canzoni della durata di circa trenta secondi l'uno. Quest'ultime sono suonate da sei persone diverse che leggono gli stessi fogli musicali. I fogli musicali sono generati da una combinazione di:

- **5 stili:** rock, cantautore, bossa nova, jazz e funk;
- **3 progressioni:** dodici bar blues, autumn leaves e pachelbel canon;
- **2 Tempi:** lento e veloce.

Gli estratti sono registrati sia con il *pickup esafonico* che con un microfono a condensatore *Neumann U-87*. Ci sono tre registrazioni audio per ogni estratto:

- **hex**: file *.wav* originali a sei canali dal *pickup esafonico*;
- **hex\_cln**: file *.wav* con rimozione delle interferenze applicata;
- **mic**: registrazione monofonica dal microfono di riferimento.

Noi abbiamo usato registrazioni di tipo **mic** perchè sono quelle che più si avvicinano al caso delle registrazioni tramite microfono dello *smartphone*.

Ciascuno dei trecentosessanta estratti ha anche un file *.jams* che memorizza sedici annotazioni:

- Intonazione:
  - 6 annotazioni *pitch\_contour* (1 per stringa)
  - 6 annotazioni *midi\_note* (1 per stringa)
- Beat e tempo:
  - 1 annotazione *beat\_position*
  - 1 annotazione del tempo
- Accordi:
  - 2 annotazioni di accordi (istruite ed eseguite)

Noi useremo le annotazioni *midi\_note*, da cui prenderemo le *tab*.

### 3.3.1 Ricavare le tab dai file *.jams*

Innanzitutto calcoliamo il numero di *frame* per ogni *file* audio, così da poter ricavare un’immagine completa e le corrispondenti *tab* per ogni *frame*. Per calcolare l’istante di tempo per ogni *frame* utilizziamo la funzione *get\_times()*:

```

1 # parameters
2 self.sr_downs = 22050
3 self.hop_length = 512
4 self.n_bins = 192
5 self.bins_per_octave = 24
6
7 self.frameDuration = self.hop_length / self.sr_downs

```

## 12 CAPITOLO 3. ARCHITETTURA E ADDESTRAMENTO DEL MODELLO

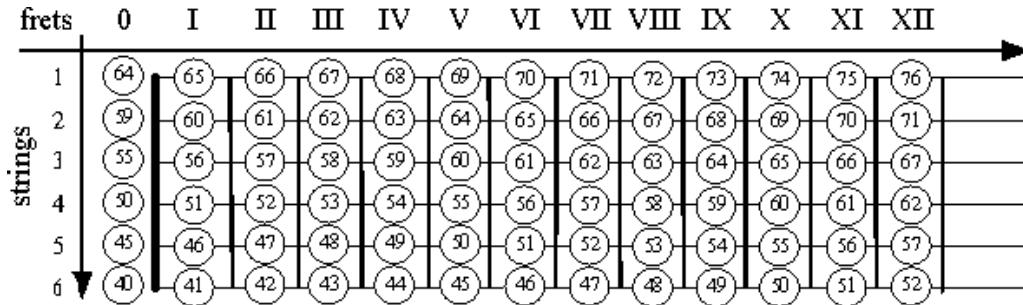
```

8
9 def get_times(self, n):
10     file_audio = os.path.join(self.path_audio, os.listdir(self.path_audio)[n])
11     (source_rate, source_sig) = wavfile.read(file_audio)
12     duration.seconds = len(source_sig) / float(source_rate)
13     totalFrame = math.ceil(duration.seconds / self.frameDuration)
14     self.frame_indices = list(range(totalFrame))
15     times = librosa.frames_to_time(self.frame_indices, sr = self.sr_downs, hop_length =
16                                     self.hop_length)
16     return times

```

Adesso che, per ogni *file* audio, abbiamo una divisione in *frame* di cui conosciamo gli istanti di tempo esatti, possiamo estrarre dai file *.jams* del *dataset* le *tab* corrispondenti.

Più precisamente, dai file *.jams* prendiamo le note *MIDI* e creiamo una matrice 6x19 dove il sei rappresenta il numero di corde mentre il diciannove rappresenta il numero di tasti.



Ogni tasto della matrice ha un valore *MIDI* che varia da 40 a 82.

```

1 [[40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58]
2 [45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63]
3 [50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68]
4 [55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73]
5 [59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77]
6 [64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82]]

```

Ad ogni matrice sostituiamo i valori *MIDI* con una matrice della stessa dimensione

in cui ci sono solo uni o zeri a seconda dei valori *MIDI* che il file *.jams* ci ha restituito. Ad esempio, dato un *frame*, se dal file *.jams* leggiamo che sono state suonate le note 40 della sesta corda, 60 della seconda corda e 67 della prima, avremo la seguente matrice:

```

1      [[1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
2      [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
3      [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
4      [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
5      [0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
6      [0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]]
```

Queste matrici, da ora in avanti, saranno chiamate *labels* e verranno usate come *target y* del modello, per questo è molto importante avere una mappatura di numeri interi "categorica", per cui useremo la codifica *one-hot*, cioè si può trovare un solo "1" in ogni riga. Per finire, aggiungiamo altre due colonne in testa alle *labels* che serviranno a capire se la corda è stata suonata o meno (colonna 0) e se si, se è stata suonata a vuoto oppure o no (colonna 1).

Di conseguenza, la matrice finale di dimensione 6x21 è la seguente:

```

1      [[0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
2      [1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
3      [1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
4      [1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
5      [0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
6      [0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]]
```

Il codice che esegue quanto abbiamo appena descritto è il seguente:

```

1 # labeling parameters
2 self.string_midi_pitches = [40, 45, 50, 55, 59, 64]
3 self.highest_fret = 19
4 self.num_classes = self.highest_fret + 2
5
6 def correct_numbering(self, n):
7     n += 1
8     if n < 0 or n > self.highest_fret:
```

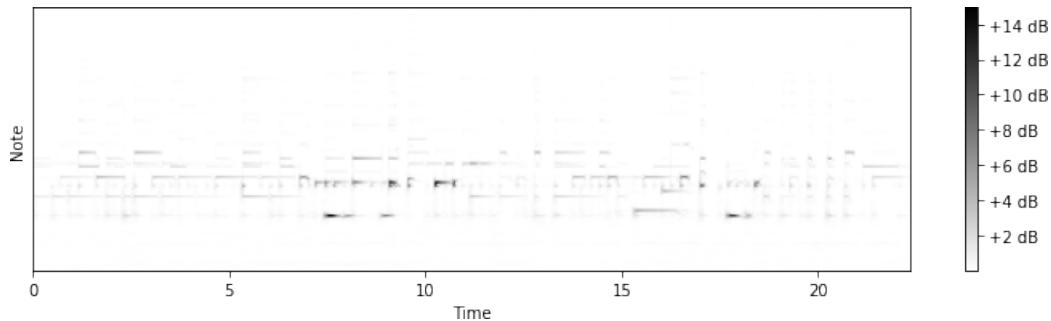
## 14 CAPITOLO 3. ARCHITETTURA E ADDESTRAMENTO DEL MODELLO

```
9         n = 0
10
11     return n
12
13
12 def categorical(self, label):
13     return to_categorical(label, self.num_classes)
14
15
15 def clean_label(self, label):
16     label = [self.correct_numbering(n) for n in label]
17
18     return self.categorical(label)
19
20
19 def clean_labels(self, labs):
21     return np.array([self.clean_label(label) for label in labs])
22
23
22 def spacejam(self, file_num):
24     path = os.path.join(self.path_anno, os.listdir(self.path_anno)[file_num])
25     jam = jams.load(path)
26
27
26     labs = []
28     for string_num in range(6):
29         anno = jam.annotations["note_midi"][string_num]
30         string_label_samples = anno.to_samples(self.times)
31
32         # replace midi pitch values with fret numbers
33         for i in self.frame_indices:
34             if string_label_samples[i] == []:
35                 string_label_samples[i] = -1
36             else:
37                 string_label_samples[i] = int(round(string_label_samples[i][0])) - self.
38                                         string_midi_pitches[string_num]
39
40         labs.append([string_label_samples])
41
42
42     labs = np.array(labs)
43
44
44     # remove the extra dimension
45     labs = np.squeeze(labs)
46     labs = np.swapaxes(labs, 0, 1)
47
45
45     # clean labels
46     labs = self.clean_labels(labs)
47
47     return labs
```

### 3.3.2 Ricavare le immagini dai file audio

Dopo esserci ricavati le *labels* per ogni *frame*, dobbiamo trovare un modo per far "imparare" al modello che un frammento di audio sia associato al corrispondente *label*. Grazie alla libreria *librosa* possiamo analizzare e manipolare il suono, vedere lo spettrogramma della trasformata a Q costante e ricavare per ogni *file* audio un'immagine.

Per esempio, questo è lo spettrogramma della trasformata a Q costante di un audio del *dataset*.



Per utilizzare al meglio il modulo *librosa.cqt()*, dobbiamo impostare alcuni parametri:

- **sr\_downs**: il numero di campioni al secondo del segnale di ingresso;
- **hop\_length**: il numero di campioni tra i fotogrammi;
- **n\_bins**: il numero di bande di frequenza nello spettrogramma risultante;
- **bins\_per\_octave**: il numero di bande di frequenza per ogni ottava.

**Problemi riscontrati:** le funzioni per caricare il *file* audio e la *cqt* restituiscono dei valori negativi e non scalati, per questo bisogna effettuare delle operazioni sulle matrici che potrebbero alterare il risultato. Inoltre, la scelta dei parametri di *simple rate* e *n\_bins* influenzano l'accuratezza del modello.

#### Soluzioni provate:

- Abbiamo avuto la necessità di scrivere delle funzioni per convertire i valori della matrice calcolata. Dunque, è stata ridimensionata in modo che si trovi nell'intervallo 0-255 e poi normalizzata nell'intervallo 0-1;
- All'inizio l'audio è stato campionato a 44100Hz perché è la stessa frequenza usata nel *dataset*. Le immagini generate avevano una frequenza delle note maggiore di quella che avrebbero dovuto avere. Infatti, nella figura in alto la

## 16 CAPITOLO 3. ARCHITETTURA E ADDESTRAMENTO DEL MODELLO

parte scura era meno spessa. Questo provocava delle imprecisioni perchè la rete non riusciva a riconoscere la nota corretta;

- Il valore iniziale scelto di  $n\_bins$  è stato 96. Questo influenzava le dimensioni dell'immagine in quanto è il valore della sua lunghezza e risultava troppo piccola.

**Soluzione finale:** le scelte finali sono ricadute rispettivamente su 22050Hz come *simple rate* e su 192 come  $n\_bins$  perchè dopo aver effettuato diverse ricerche, in ambito di manipolazione del suono, è consigliato utilizzare questi valori. Per quanto riguarda il ridimensionamento abbiamo usato le funzioni *librosa.util.normalize()* e *np.abs()* subito dopo aver caricato i file audio invece di manipolare le matrici in seguito.

```
1 # parameters
2 self.sr_downs = 22050
3 self.hop_length = 512
4 self.n_bins = 192
5 self.bins_per_octave = 24
6
7 def audio_CQT(self, file_num):
8     path = os.path.join(self.path_audio, os.listdir(self.path_audio)[file_num])
9
10    # Perform the Constant-Q Transform
11    data, sr = librosa.load(path, sr = self.sr.downs, mono = True, dtype='float64')
12    data = librosa.util.normalize(data)
13    data = librosa.cqt(data,
14                        sr = self.sr.downs,
15                        hop_length = self.hop_length,
16                        fmin = None,
17                        n_bins = self.n_bins,
18                        bins_per_octave = self.bins_per_octave)
19    CQT = np.abs(data)
20    return CQT
```

In particolare, una volta che l'audio è un oggetto dati *librosa*, *Python* lo vede come un numpy *array* e quindi possiamo evitare di salvarci fisicamente dei classici *file* immagini *.jpeg* o *.png* e continuare a lavorare con le matrici.

I dati (*images* e *labels*) di ogni *file* audio sono stati compressi in archivi *.npz* per organizzare meglio il *dataset* da dare come *input* al modello.

```

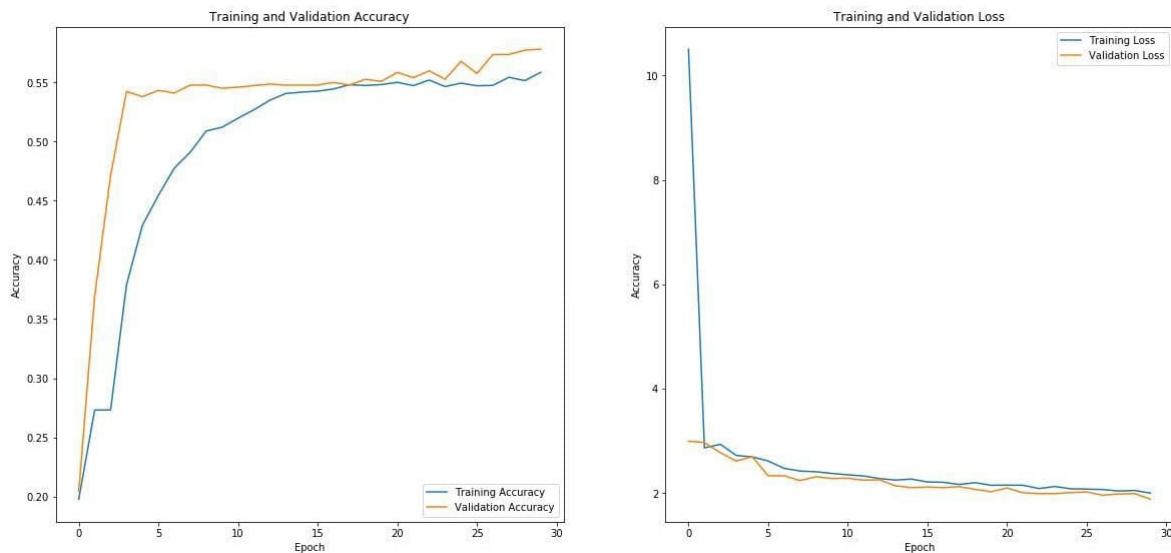
1 def store(self, n, num_frames):
2     save_path = self.save_path
3     filename = self.get_filename(n)
4
5     self.output["imgs"] = self.imgs
6     self.output["labels"] = self.labels
7
8     if not os.path.exists(save_path):
9         os.makedirs(save_path)
10    self.save_data(save_path + filename + ".npz")

```

### 3.3.3 Pre-elaborare i dati

Carichiamo i *file* *.npz* salvati in precedenza.

**Problemi riscontrati:** mentre lavoravamo con il modello ci siamo resi conto che usare una finestra di 1s risultava essere troppo ampia. Se nell'immagine ci sono troppe note e la rete deve riconoscerne poche, non riesce a farlo. Infatti, se non viene scelta correttamente la finestra, l'accuratezza del modello è molto bassa e dopo appena 15 epoche la rete non tende più ad imparare.

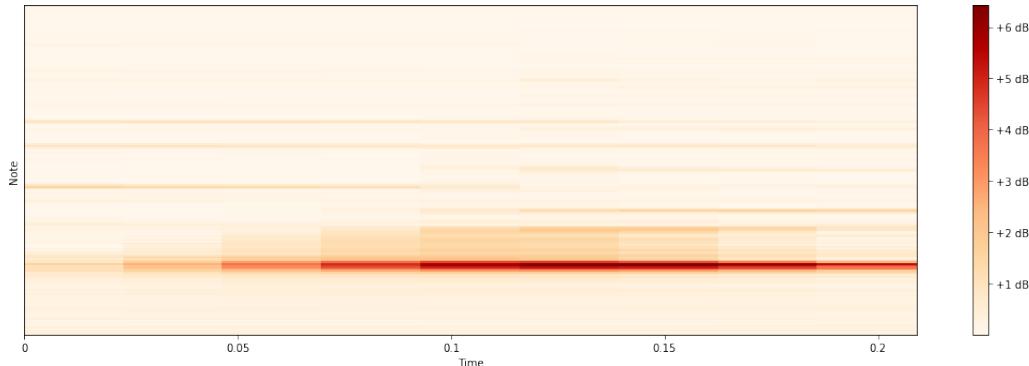


**Soluzioni provate:**

- Abbiamo provato finestre di grandezza compresa tra 0.1s e 0.3s.

**Soluzione finale:** si è visto, sperimentalmente, che un buon arco di tempo per riconoscere una nota è 200ms che, nel nostro caso, corrispondono ad una finestra di nove righe della matrice dell'immagine.

Di conseguenza, quello che dobbiamo fare non è altro che associare, per ogni *frame*, una *label* e un'*immagine* di nove righe (192x9), senza dimenticare di aggiungere un *padding* (quattro zeri nell'*array*, sia all'inizio che alla fine, con la funzione *np.pad()*) perchè così l'ultimo valore, della finestra di nove righe, risulta essere congruo con gli altri.



Il codice che esegue quanto abbiamo appena descritto è il seguente:

```

1 self.con_win_size = 9
2 self.halfwin = con_win_size // 2
3
4 def load_data(self):
5     for i in range(self.n_file):
6         self.log("n." + str(i+1))
7         inp = np.load(os.path.join(self.data_path, os.listdir(self.data_path)[i]))
8
9         full_x = np.pad(inp["imgs"], [(self.halfwin, self.halfwin), (0,0)], mode='constant')
10
11    for frame_idx in range(len(inp['imgs'])):
12        # load a context window centered around the frame index

```

```

13     sample_x = np.swapaxes(full_x[frame_idx : frame_idx+self.con.win_size],0,1)
14     self.training_data.append((sample_x.astype('float64'), inp['labels'][
15         frame_idx][::-1].astype('float64')))
```

Una volta ottenuto l'*array*, chiamato (*training\_data*), di tutte le *images* e i *labels* dei *file* audio, e assegnati rispettivamente le variabili *X* e *y*, lo mescoliamo in modo da avere più imprevedibilità. Infine, suddividiamo questi dati nel seguente modo:

- 10% dei dati li usiamo per la *validation*
- 10% dei dati li usiamo per i *test*
- 80% dei dati li usiamo per il *training*

Il *training set* lo utilizziamo per costruire il modello, il *validation set* per validare i parametri dei *layer* della rete neurale mentre il *test set* per determinare l'accuratezza.

Il codice è il seguente:

```

1 def prepare_data(self, data):
2     inp = []
3     out = []
4
5     for imgs, labels in data:
6         inp.append(imgs)
7         out.append(labels)
8     return inp, out
9
10
10 def partition_data(self):
11     # Randomize training set
12     random.shuffle(self.training_data)
13
14     X_train, y_train = self.prepare_data(self.training_data)
15
16     # Calculate validation and test set sizes
17     val_set_size = int(len(self.training_data) * 0.1)
18     test_set_size = int(len(self.training_data) * 0.1)
19
```

```

20     # Break x apart into train, validation, and test sets
21     self.X_val = X_train[:val_set_size]
22     self.X_test = X_train[val_set_size:(val_set_size + test_set_size)]
23     self.X_train = X_train[(val_set_size + test_set_size):]
24
25     # Break y apart into train, validation, and test sets
26     self.y_val = y_train[:val_set_size]
27     self.y_test = y_train[val_set_size:(val_set_size + test_set_size)]
28     self.y_train = y_train[(val_set_size + test_set_size):]
29
30     self.log("Train set size: " + str(len(self.X_train)))
31     self.log("Validation set size: " + str(len(self.X_val)))
32     self.log("Test set size: " + str(len(self.X_test)))

```

## 3.4 Modello della rete

La difficoltà nell'apprendere i meccanismi di implementazione su *Keras* sono ridotti al minimo grazie alla vasta documentazione presente, arricchita da numerosi esempi sulle più utilizzate configurazioni inerenti il *machine learning*, come le *CNN* (Convolutional Neural Network). Le operazioni di calcolo matriciali possono essere accelerate sia tramite *CPU*, che *GPU* (su hardware *Nvidia* con supporto *CUDA*). Nel nostro caso è stata utilizzata una *GPU* in modo da sfruttare direttamente la sua potenza parallela contenute nelle schede video recenti.

```

1 def gpu():
2     config = ConfigProto()
3     config.gpu_options.allow_growth = True
4     sess = Session(config=config)
5
6     if tf.test.gpu_device_name():
7         log("GPU found")
8     else:
9         log("No GPU found")

```

Le caratteristiche e i vantaggi che ci hanno portato ad utilizzare *Keras* nell'ambito del progetto sono:

- **semplicità:** a differenza di altre *API*, è possibile realizzare modelli complessi scrivendo meno righe di codice, mantenendo nel contempo chiarezza nello sviluppo. Tutto ciò consente allo sviluppatore di mantenere nel tempo il codice in maniera agevole;
- **modularità:** un modello in *Keras* è inteso come una sequenza o un grafo di singoli, compatti e completamente configurabili moduli, che possono lavorare in sinergia tra loro con il minimo numero di restrizioni possibili. Ciò rende il codice estremamente flessibile;
- **estensibilità:** in base alle esigenze dello sviluppatore, è possibile aggiungere facilmente nuovi moduli (ad esempio classi e funzioni) ad un progetto preesistente.

### 3.4.1 Uso di Keras

Prima di dare in *input* al modello il *training set* e il *validation set*, dobbiamo fare delle modifiche alla dimensione del numpy *array* delle immagini *X* in quanto il modello si aspetta una dimensione di *input* di *BATCH* x 192 x 9 x 1.

- *BATCH* sono la quantità di valori dell'intero *training set*;
- 192 è l'altezza dell'immagine (H);
- 9 è la lunghezza dell'immagine (W);
- 1 ci indica che l'immagine è in bianco e nero.

La *y* non ha bisogno di modifiche perchè le dimensioni sono già quelle corrette cioè ha dimensione *BATCH* x 21 x 6.

Il codice che esegue quanto abbiamo appena descritto è il seguente:

```

1 def load_data(self):
2     # Load features sets
3     features_sets = np.load(os.path.join("data/", self.feature_sets_file))
4
5     # Assign feature sets
6     X_train = features_sets['X_train']
7     X_val = features_sets['X_val']
8
9     self.y_train = features_sets['y_train']
10    self.y_val = features_sets['y_val']

```

## 22 CAPITOLO 3. ARCHITETTURA E ADDESTRAMENTO DEL MODELLO

```
11
12     # CNN for TF expects (batch, height, width, channels)
13     # So we reshape the input tensors with a "color" channel of 1
14     self.X_train = X_train.reshape(X_train.shape[0],
15                                     X_train.shape[1],
16                                     X_train.shape[2],
17                                     1)
18     self.X_val = X_val.reshape(X_val.shape[0],
19                               X_val.shape[1],
20                               X_val.shape[2],
21                               1)
```

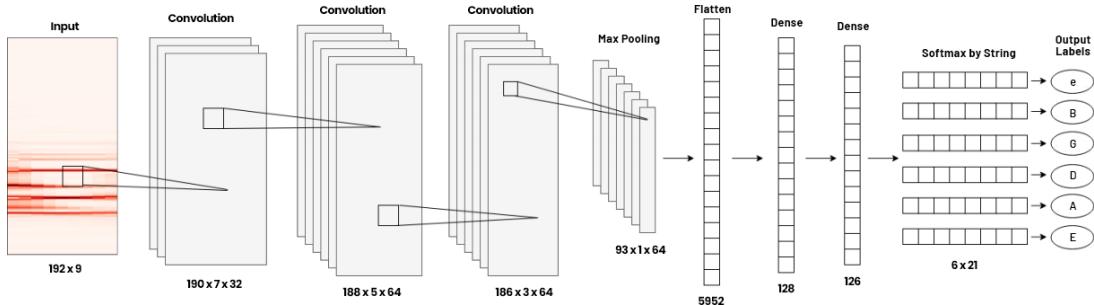
A questo punto definiamo il modello con l'aggiunta dei seguenti *layers*:

- **Conv2D**: mette in evidenza le caratteristiche interessanti dell'immagine. Parametri di *input*: numero di filtri, grandezza filtri, *input shape* e funzione di attivazione;
- **MaxPooling2D**: riduce la dimensione dell'immagine, elimina le informazioni inutili mantenendo quelle più importanti;
- **Dropout**: elimina una percentuale di dati casuali. Ad esempio, elimina rumori di sottofondo e mantiene le informazioni più importanti;
- **Flatten**: appiattisce il tensore e rimuove tutte le dimensioni;
- **Dense**: crea un *layer* di neuroni, ognuno dei quali connesso ad ogni uscita del *layer* precedente;
- **Reshape**: determina la dimensione di uscita. In questo caso è 6x21;
- **Activation**: la funzione di attivazione è una "porta" matematica tra l'*input* che alimenta il neurone corrente e il suo *output* che va allo strato successivo.

L'unità lineare rettificata (*ReLU*) è la funzione di attivazione più comunemente utilizzata nel *deep learning*. La funzione restituisce 0 se l'*input* è negativo, ma per qualsiasi *input* positivo, restituisce quel valore.

La funzione *softmax* è molto utilizzata in statistica e consente di gestire un vettore di uscita normalizzato di n elementi, dove ogni elemento può valere da 0 ad 1 e la somma di tutti gli elementi è pari ad 1. In sostanza il nostro vettore di uscita sarà in una forma simile a quella *one-hot* e ogni posizione corrisponderà alla probabilità

(normalizzata) che l'immagine appartenga a quella specifica classe. La somma di tutte le probabilità sarà  $1 = 100\%$ .



Per compilare il modello abbiamo scelto come ottimizzatore l'algoritmo *Adadelta* che utilizza un metodo di discesa del gradiente stocastico basato sul tasso di apprendimento adattivo per dimensione per affrontare l'inconveniente del continuo declino dei tassi di apprendimento durante la formazione e la necessità di un tasso di apprendimento globale selezionato manualmente.

Il codice è il seguente:

```

1 self.input_shape = (192, 9, 1)
2 self.num_classes = 21
3 self.num_strings = 6
4
5 def build_model(self):
6     model = Sequential()
7     model.add(Conv2D(32, kernel_size=(3, 3), activation='relu', input_shape=self.
8         input_shape))
9     model.add(Conv2D(64, (3, 3), activation='relu'))
10    model.add(Conv2D(64, (3, 3), activation='relu'))
11    model.add(MaxPooling2D(pool_size=(2, 2)))
12    model.add(Dropout(0.25))
13    model.add(Flatten())
14    model.add(Dense(128, activation='relu'))
15    model.add(Dropout(0.5))
16    model.add(Dense(self.num_classes * self.num_strings))
17    model.add(Reshape((self.num_strings, self.num_classes)))
18    model.add(Activation(self.softmax_by_string))

```

## 24 CAPITOLO 3. ARCHITETTURA E ADDESTRAMENTO DEL MODELLO

```
19     model.compile(loss=self.catcross_by_string, optimizer=Adadelta(), metrics=[self.  
20                           avg_acc])  
21  
22     self.model = model
```

La funzione *summary()* ci descrive il modello:

```
1 Model: "sequential"  
2 -----  
3 Layer (type)          Output Shape         Param #  
4 ======  
5 conv2d (Conv2D)        (None, 190, 7, 32)      320  
6 -----  
7 conv2d_1 (Conv2D)      (None, 188, 5, 64)      18496  
8 -----  
9 conv2d_2 (Conv2D)      (None, 186, 3, 64)      36928  
10 -----  
11 max_pooling2d (MaxPooling2D) (None, 93, 1, 64)    0  
12 -----  
13 dropout (Dropout)     (None, 93, 1, 64)      0  
14 -----  
15 flatten (Flatten)     (None, 5952)           0  
16 -----  
17 dense (Dense)         (None, 128)            761984  
18 -----  
19 dropout_1 (Dropout)   (None, 128)           0  
20 -----  
21 dense_1 (Dense)       (None, 126)            16254  
22 -----  
23 reshape (Reshape)     (None, 6, 21)           0  
24 -----  
25 activation (Activation) (None, 6, 21)           0  
26 ======  
27 Total params: 833,982  
28 Trainable params: 833,982  
29 Non-trainable params: 0
```

In particolare, per il modello abbiamo usato delle funzioni personalizzate per la funzione di attivazione finale del modello (*softmax\_by\_string*), per la funzione obiettivo *loss* (*catcross\_by\_string*) e per le metriche di accuratezza (*avg\_acc*) che devono essere valutate dal modello durante l'addestramento.

Esse permettono di utilizzare la funzione di attivazione *softmax* e la funzione di perdita *categorical\_crossentropy* per ogni stringa, e alla fine concatenare i sei risultati in un unico valore.

Il codice che esegue quanto abbiamo appena descritto è il seguente:

```

1 from tensorflow.keras import backend as K
2
3 def softmax_by_string(self, t):
4     sh = K.shape(t)
5     string_sm = []
6     for i in range(self.num_strings):
7         string_sm.append(K.expand_dims(K.softmax(t[:,i,:]), axis=1))
8     return K.concatenate(string_sm, axis=1)
9
10 def catcross_by_string(self, target, output):
11     loss = 0
12     for i in range(self.num_strings):
13         loss += K.categorical_crossentropy(target[:,i,:], output[:,i,:])
14     return loss
15
16 def avg_acc(self, y_true, y_pred):
17     return K.mean(K.equal(K.argmax(y_true, axis=-1), K.argmax(y_pred, axis=-1)))

```

Per avviare l'addestramento del modello eseguiamo la funzione *model.fit()* dove indichiamo con *batch\_size* il numero di campioni per ogni aggiornamento del gradiente e con *epochs* il numero di iterazioni sul quale il modello deve effettuare il *training*. In questo modo partirà la fase di addestramento che andrà ad affinare sempre di più le performance del modello.

Il codice che esegue quanto abbiamo appena descritto è il seguente:

```
1 self.batch_size = 128
```

## 26 CAPITOLO 3. ARCHITETTURA E ADDESTRAMENTO DEL MODELLO

```
2 self.epochs = 500
3
4 def train(self):
5     self.hist = History()
6     self.model.fit(self.X_train,
7                     self.y_train,
8                     batch_size=self.batch_size,
9                     epochs=self.epochs,
10                    verbose=1,
11                    validation_data=(self.X_val, self.y_val),
12                    callbacks=[self.hist])
```

### 3.4.2 Addestramento del modello

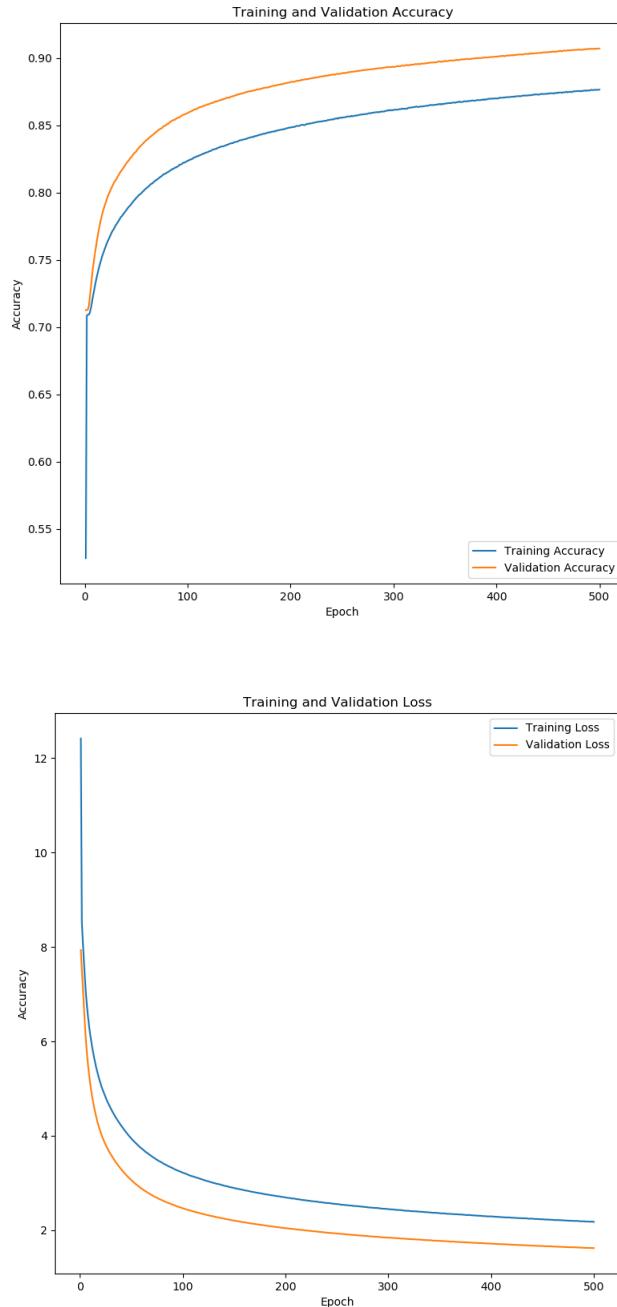
Dopo diverse prove sperimentali, abbiamo deciso di eseguire il modello per cinquecento epoche:

```
Train on 378048 samples, validate on 47256 samples
Epoch 1/500
378048/378048 [=====] - 51s 135us/sample - loss: 12.4176 - avg_acc: 0.5283 - val_loss: 7.9343 - val_avg_acc: 0.7126
Epoch 2/500
378048/378048 [=====] - 47s 124us/sample - loss: 8.5325 - avg_acc: 0.7085 - val_loss: 7.5110 - val_avg_acc: 0.7126
Epoch 3/500
378048/378048 [=====] - 47s 125us/sample - loss: 8.0792 - avg_acc: 0.7092 - val_loss: 7.1406 - val_avg_acc: 0.7129
Epoch 4/500
378048/378048 [=====] - 48s 127us/sample - loss: 7.6807 - avg_acc: 0.7092 - val_loss: 6.7315 - val_avg_acc: 0.7159
Epoch 5/500
378048/378048 [=====] - 47s 125us/sample - loss: 7.3140 - avg_acc: 0.7109 - val_loss: 6.3383 - val_avg_acc: 0.7226
Epoch 6/500
378048/378048 [=====] - 47s 126us/sample - loss: 6.9992 - avg_acc: 0.7142 - val_loss: 6.0077 - val_avg_acc: 0.7298
...
Epoch 496/500
378048/378048 [=====] - 47s 124us/sample - loss: 2.1791 - avg_acc: 0.8764 - val_loss: 1.6230 - val_avg_acc: 0.9068
Epoch 497/500
378048/378048 [=====] - 47s 125us/sample - loss: 2.1769 - avg_acc: 0.8762 - val_loss: 1.6229 - val_avg_acc: 0.9068
Epoch 498/500
378048/378048 [=====] - 47s 124us/sample - loss: 2.1825 - avg_acc: 0.8763 - val_loss: 1.6212 - val_avg_acc: 0.9069
Epoch 499/500
378048/378048 [=====] - 47s 124us/sample - loss: 2.1780 - avg_acc: 0.8763 - val_loss: 1.6195 - val_avg_acc: 0.9070
Epoch 500/500
```

Come si può notare dal *log*, un'epoca (cioè l'addestramento eseguito su un intero *dataset* di 378048 immagini) viene eseguita in circa 47 secondi, usando mediamente 124 microsecondi per immagine. Questo è il risultato ottenuto utilizzando una *GPU NVIDIA RTX 2060*.

Il seguente modello raggiunge una precisione di circa 0,87 su 1 (o 87%) e una perdita del 2% sui dati di addestramento. Invece, sui dati di validazione la precisione supera lo 0,90 (90%) e raggiunge una perdita del 1,6% sui dati di validazione. I *loss* sono la media delle perdite sui dati di *batch* di addestramento.

Il grafico sottostante ci fa comprendere meglio i risultati:



### 3.5 Valutazione del modello

Abbiamo eseguito diversi *test* per mettere alla prova il modello utilizzando il *training test*. La matrice *Answer* è la  $y$  corrispondente alla  $X$  data come *input* alla *predict*. Invece, la *Prediction* corrisponde all'*output* della rete. Il valore più grande della riga corrisponde a dove secondo il modello ci debba essere "1".

Di seguito riportiamo un esempio:

```

33 [1.95525795e-01 1.28646957e-06 5.31254616e-03 4.10607839e-07
34 2.72660202e-07 1.25411839e-06 6.68900725e-07 2.57475822e-05
35 4.56736198e-05 7.99066842e-01 1.76908379e-05 9.00572275e-07
36 1.72483126e-07 6.02985438e-07 5.14156078e-08 3.12393169e-08
37 2.88916464e-08 2.03121537e-08 3.34022587e-08 1.16275478e-08
38 3.34708155e-08]
39 [8.34586322e-01 5.58627034e-05 3.25484907e-05 7.17843577e-05
40 1.15673347e-05 2.41428029e-06 1.23440741e-05 1.65205747e-01
41 1.51618824e-05 1.64955986e-06 8.40611392e-07 3.49875990e-07
42 1.91993990e-07 2.15398154e-06 2.38864061e-07 3.07556469e-08
43 1.46060472e-07 1.96282883e-07 1.66214150e-07 1.29733195e-07
44 2.07605083e-07]]

```

Abbiamo confrontato le prestazioni del modello sul *dataset* di prova e i risultati sono stati quelli previsti. Infatti, l'accuratezza e la perdita dei dati di *test* sono molto simili a quelli dei dati di validazione. Quello che otteniamo è un errore del 10% (quindi un'accuracy del 90%).

```

1 Test loss, Test acc: [1.6260207852918485, 0.90655446]

```

Il codice che esegue quanto abbiamo appena descritto è il seguente:

```

1 def test(self):
2     # TEST: Load model and run it against test set
3     r = random.randint(0, len(self.X_test))
4     for i in range(r, r + 10):
5         print("Answer:", self.y_test[i], "Prediction:",
6               self.model.predict(np.expand_dims(self.X_test[i], 0)))
7
8     self.y_pred = self.model.predict(self.X_test)
9
10 def evaluate(self):
11     # Evaluate model with test set
12     results = self.model.evaluate(x=self.X_test, y=self.y_test)
13     self.log("Test loss, Test acc: " + str(results))

```



# Capitolo 4

## Implementazione su dispositivo Embedded

Per dispositivo *embedded* si intende un dispositivo piccolo e compatto con consumi energetici molto contenuti. Proprio per queste caratteristiche sono usati per il *deployment* di reti neurali.

La scelta è caduta sui dispositivi mobili. Questa decisione è stata vincolante perché disponevamo di solo queste risorse *hardware*.

### 4.1 Conversione del modello da Keras a Tensorflow Lite

Abbiamo convertito il modello usando *Tensorflow Lite*. Il seguente codice consente di caricare il modello addestrato tramite *Tensorflow* e di convertirlo nel formato *.tflite* pronto per essere utilizzato su un dispositivo *embedded*.

```
1 self.model_filename = "model.h5"
2 self.tflite_filename = "model.tflite"
3
4 def convert_tflite(self):
5     model = load_model(self.save_path + self.model_filename, custom_objects={
6         'softmax_by_string': self.softmax_by_string, 'avg_acc': self.avg_acc,
7         'catcross_by_string': self.catcross_by_string})
8     converter = lite.TFLiteConverter.from_keras_model(model)
9     tflite_model = converter.convert()
10    open(self.save_path + self.tflite_filename, "wb").write(tflite_model)
```

Il convertitore ha il compito di ottimizzare il modello riducendo le sue dimensioni e aumentando la sua velocità di esecuzione.

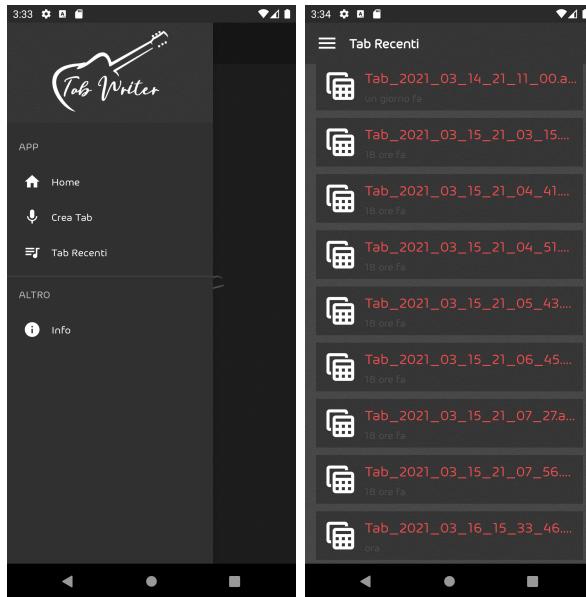
## 4.2 Sviluppo applicazione Android

### 4.2.1 Sviluppo applicazione con Java 1.8 e Android Studio

#### 4.1.2

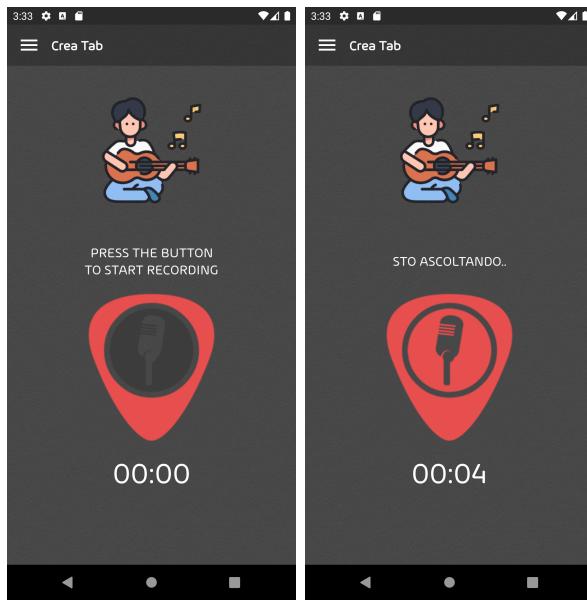
*Java* è una piattaforma che consente di eseguire i programmi scritti in questo linguaggio.

*Android Studio* è un ambiente di sviluppo integrato per lo sviluppo per la piattaforma *Android*.



Dopo aver creato le interfacce con gli elementi grafici, sia in *light mode* che in *dark mode*, è stato implementato un *database* che tiene traccia delle *tab* registrate e predette, in modo da poterle consultare anche in un secondo momento senza dover richiamare l'interprete di *Tensorflow Lite*.

Per avviare la registrazione, basta cliccare sull'immagine centrale su cui è raffigurato un microfono. A questo punto il cellulare registra l'audio e per terminarla bisogna ricliccare sull'immagine.



A questo punto il file verrà salvato sul cellulare e convertito nel formato *.wav*. Una volta aver ricavato le immagini dal *file* audio, grazie al modello della rete è possibile avviare la predizione. Il risultato apparirà in una nuova interfaccia:



**Problemi riscontrati:** puttroppo non sono state trovate librerie in grado di ricalcare, dal *file* audio, la trasformata a Q costante come avevamo fatto su *Python*.

#### Soluzioni provate:

- Abbiamo pensato di utilizzare un *server* che ricavi l'immagine della trasformata dall'audio registrato e la restituisca al dispositivo.

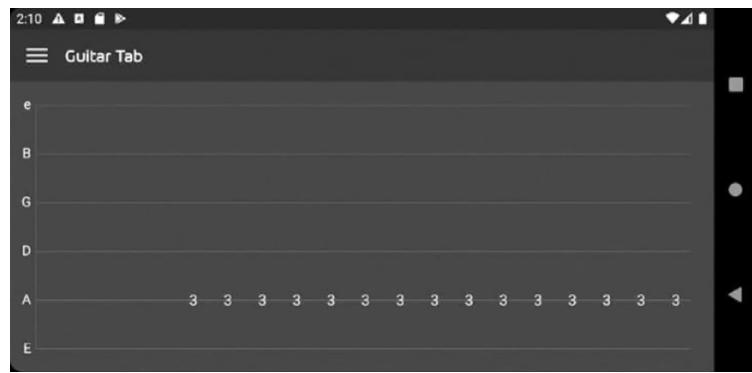
**Soluzione finale:** grazie ai ricevimenti fatti con il professore, ci è stato consigliato di usare *chaquopy*, un *plugin* che consente di implementare codice *Python* all'interno delle applicazioni *Android*.

### 4.2.2 Pre-elaborazione dell'audio

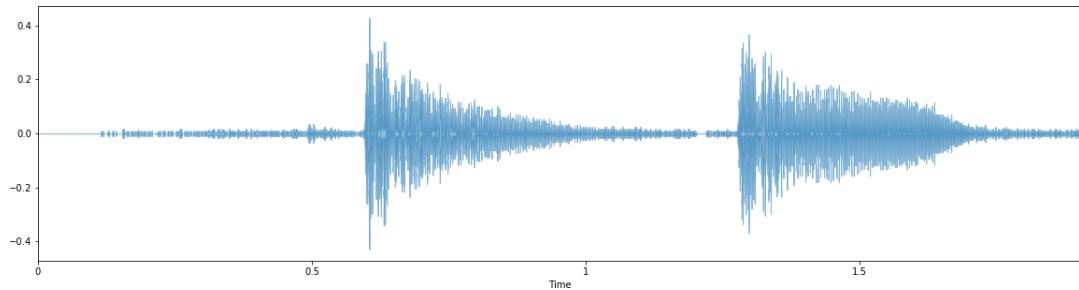
**Problemi riscontrati:** ricavando le immagini per ogni *frame* di audio e successivamente facendo la *predict* per ogni immagine, si ottenevano tante soluzioni quanti erano i *frame*.

Per esempio, suonando il terzo tasto della quinta corda per due volte avremmo sull'interfaccia dell'applicazione una lunga serie di "3", perchè un *frame* ha una durata di pochi millisecondi mentre la nota può avere un *range* anche di qualche secondo. La soluzione corretta sarebbe quella di avere solo due 3.

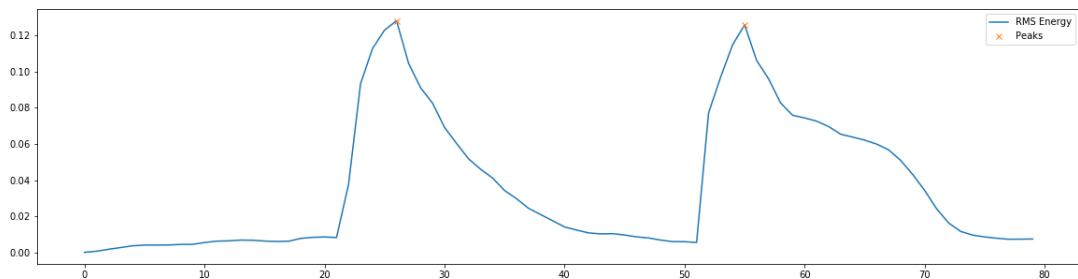
L'immagine sottostante fa vedere quanto è stato descritto:



**Soluzione finale:** visualizzando e analizzando lo spettrogramma, possiamo notare che quando una nota viene suonata ha un'ampiezza massima, come si può notare dal seguente grafico:



Per trovare i punti, in questo caso due perchè è stata suonata due volte la stessa nota, ci viene in aiuto visualizzare l'*RMS Energy* che è l'energia di un segnale. La funzione raggiunge il picco in due punti. In questo modo selezioniamo solo i *frame* più significativi.



Grazie alla funzione `signal.find_peaks()` della libreria `scipy` troviamo tutti i massimi locali mediante un semplice confronto con i valori vicini in modo da ottenere solo i frame più significativi.

```

1 def rms_energy(path):
2     data, sr = librosa.load(path, sr = sr_downs, mono = True, dtype='float64')
3
4     S = librosa.magphase(librosa.stft(data, window=np.ones, center=False))[0]
5     rms = librosa.feature.rms(S=S)
6
7     peaks, _ = find_peaks(rms[0], width=5)
8
9     f = peaks.tolist()
10    for i in range(len(f)):
11        f[i] += 3
12
13    return f

```

Di conseguenza, vedremo solo due volte il tasto ”3”:



### 4.2.3 Implementazione di Tensorflow Lite

Una volta ottenuti gli *input* della rete possiamo eseguire l'inferenza con il modello. Inanzitutto rendiamo accessibile l'interprete di *Tensorflow Lite*, allochiamo i tensori ed estrapoliamo dal modello rispettivamente il tipo e il formato dell'*input* e dell'*output*.

Solo i *frame* più significativi verranno dati in *input* all'interprete tramite la funzione *interpreter.set\_tensor()*. Una volta che vengono definite la dimensione dell'*input* e allocati i tensori, invochiamo l'interprete e tramite la funzione *interpreter.get\_tensor()* otteniamo una copia dei valori provenienti dal tensore di *output*. Infine, i risultati vengono salvati in un *JSON* e conservati nel *database* dell'applicazione.

```

1 path_model = "model.tflite"
2
3 def myconverter(obj):
4     if isinstance(obj, np.integer):
5         return int(obj)
6     elif isinstance(obj, np.floating):
7         return float(obj)
8     elif isinstance(obj, np.ndarray):
9         return obj.tolist()
10    elif isinstance(obj, datetime.datetime):
11        return obj.__str__()
12
13 def predict_model(images, frames):
14     data_json = []
15
16     # Load the TFLite model and allocate tensors.
17     interpreter = tf.lite.Interpreter(model_path=path_model)
18     interpreter.allocate_tensors()
19
20     # Get input and output tensors.
21     input_details = interpreter.get_input_details()
22     output_details = interpreter.get_output_details()
23
24     x = 0
25     for i in range(images.shape[0]):
26         if(i in frames):
27             input_data = images[i].reshape(1, images.shape[1], images.shape[2], images.
28                                         shape[3])

```

```
29     interpreter.set_tensor(input_details[0]['index'], input_data)
30     interpreter.invoke()
31
32     output_data = interpreter.get_tensor(output_details[0]['index'])
33
34     b = np.zeros_like(np.squeeze(output_data))
35     b[np.arange(len(np.squeeze(output_data))), np.argmax(np.squeeze(output_data),
36                                         axis=-1)] = 1
37
38     value = np.argmax(b.astype('uint8'), axis=1)
39     data_json.append({'tab_x' : x, 'value' : value})
40
41     x += 1
42
43 return json.dumps(data_json, default=myconverter)
```

## 4.3 Sviluppo applicazione iOS

#### 4.3.1 Conversione del modello da Keras a CoreML

Per poter usare il modello pre-addestrato sul cellulare abbiamo dovuto convertirlo nel formato *.mlmodel* in modo da poter usare il *framework Core ML*.

**Problemi riscontrati:** durante la conversione del modello sono apparsi diversi errori che impedivano la conversione. Gli errori sono simili a quello riportato di seguito:

```
ValueError: Input 0 of node save/AssignVariableOp was passed float from conv2d/bias:0  
incompatible with expected resource.
```

**Soluzioni provate:**

- Abbiamo preso spunto dal codice che si trova sul blocco di lucidi visti a lezione. Esso usa il *package* *tfcoreml*. Il focus dello *script* è il seguente:

```
1 import tensorflow as tf_converter  
2 tf_converter.convert(  
3     tf_model_path = './output/frozen_model.pb',
```

```

4         mlmodel_path = './output/frozen.mlmodel',
5         output_feature_names = ['Softmax:0'])

```

A questo punto serviva ottenere il file in formato *.pb* da usare come *input*. Questo file prende il nome di modello congelato. Prima di ottenerlo serve salvarci il modello che si ottiene con *Tensorflow*. Esso è formato da quattro file:

- **model-ckpt.meta**: contiene il grafico completo (flusso di dati, le annotazioni per le variabili, le *pipeline* di *input* e altre informazioni);
- **model-ckpt.data-0000-of-00001**: contiene tutti i valori delle variabili (pesi, segnaposto, gradienti, iperparametri, ecc.);
- **model-ckpt.index**: ci sono tutti i metadati. È una tabella immutabile in cui ogni chiave è un nome di un tensore e il suo valore descrive i metadati di un tensore;
- **checkpoint**: tutte le informazioni sul *checkpoint*.

```

1 output_directory_path = './output'
2
3 # the model_dir states where the graph and checkpoint files
4 # will be saved to
5 estimator_model = tf.keras.estimator.model_to_estimator(keras_model = model,
6               model_dir = output_directory.path)
7
8 def input_function(features, labels=None, shuffle=False):
9     input_fn = tf.estimator.inputs.numpy_input_fn(
10         x={"conv2d_1_input": features},
11         y=labels,
12         shuffle=shuffle,
13         batch_size = 128,
14         num_epochs = 30
15     )
16
17     return input_fn
18
19
20 estimator_model.train(input_fn = input_function(X_train,y_train,True))

```

*estimator\_model.train()* serve a verificare che il modello esportato in precedenza sia effettivamente funzionante.

Il modello congelato ci consente di eliminare tutte le informazioni in più che vengono salvate perchè si potrebbe ricaricare quello appena salvato e l'addestramento continua da dove era stato interrotto.

```
1 def freeze_graph(model_dir, output_node_names):
2     if not tf.gfile.Exists(model_dir):
3         raise AssertionError(
4             "Export directory doesn't exists. Please specify an export "
5             "directory: %s" % model_dir)
6
7     if not output_node_names:
8         print("You need to supply the name of a node to output_node_names.")
9     return -1
10
11    checkpoint = tf.train.get_checkpoint_state(model_dir)
12    input_checkpoint = checkpoint.model_checkpoint_path
13
14    absolute_model_dir = "/".join(input_checkpoint.split("/")[:-1])
15    output_graph = absolute_model_dir + "/frozen_model.pb"
16
17    clear_devices = True
18
19    with tf.Session(graph=tf.Graph()) as sess:
20        saver = tf.train.import_meta_graph(
21            input_checkpoint + '.meta', clear_devices=clear_devices)
22
23        saver.restore(sess, input_checkpoint)
24
25        output_graph_def = tf.graph_util.convert_variables_to_constants(
26            sess,
27            tf.get_default_graph().as_graph_def(),
28            output_node_names.split(","))
29
30
31        with tf.gfile.GFile(output_graph, "wb") as f:
32            f.write(output_graph_def.SerializeToString())
```

```

33     print("%d ops in the final graph." % len(output_graph_def.node))
34
35     return output_graph_def
36
37     freeze_graph('./output/','save/restore_all')

```

- Uno dei nuovi tentativi si è basato sul cambio di codice per salvare il modello; abbiamo usato il seguente codice:

```

1 output_directory_path = './output'
2
3 supervisor = tf.train.Supervisor(logdir=output_directory_path)
4
5 with supervisor.managed_session() as session:
6     # train the model here
7     supervisor.saver.save(session, output_directory_path)

```

Tuttavia, i risultati non sono stati quelli sperati.

- Consultando la documentazione di *coreml*, abbiamo scoperto che non sono previsti più aggiornamenti e consigliavano di usare un nuovo *package*. Anche se fossimo stati in grado di convertirlo, non avremmo potuto utilizzare il modello su sistemi operativi *iOS* maggiori di 12. La nuova libreria che abbiamo usato si chiama *coremltools*.

```

1 import coremltools
2 coreml_model = coremltools.converters.keras.convert(model)
3
4 coreml_model.author = 'De Nardi-Tornatore'
5 coreml_model.short_description = 'Recognition guitar music'
6
7 coreml_model.save("Stock.mlmodel")

```

- Successivamente, per esigenze del nuovo modello, sono state inserite alcune funzioni personalizzate. Purtroppo, anche con *coremltools* non siamo stati in grado di convertirlo perché ci appariva un errore in corrispondenza del livello della nuova funzione:

```
TypeError: argument of type 'NoneType' is not iterable
```

**Soluzione finale:** abbiamo deciso di usare *Tensorflow Lite* perchè siamo riusciti a convertire il modello subito senza nessun problema.

### 4.3.2 Sviluppo applicazione con Swift 5 e Xcode 12

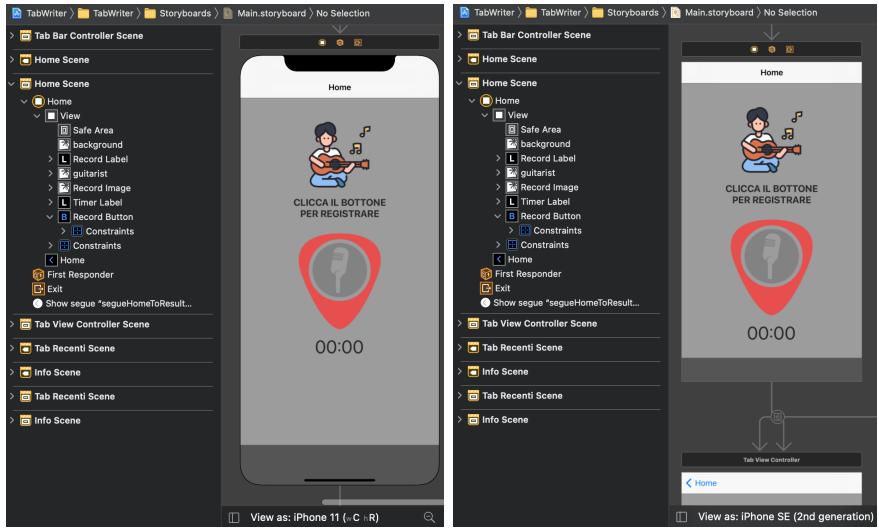
*Swift* è un linguaggio di programmazione *object-oriented* concepito per programmare sui sistemi operativi *Apple*.

*Xcode* è un ambiente di sviluppo integrato completamente sviluppato e mantenuto da *Apple*, che consente di sviluppare *software* per i sistemi *macOS*, *iOS*, *watchOS* e *tvOS*.

Abbiamo dovuto prendere un pò di familiarità con il nuovo linguaggio e il nuovo *IDE* dato che non avevamo mai programmato nel mondo *Apple*. La documentazione messa a disposizione agli sviluppatori è molto vasta e le solide basi apprese a ingegneria hanno fatto il resto. La scelta è ricaduta direttamente sia all'ultima versione del linguaggio che dell'ambiente di sviluppo perché non avevamo vincoli sulla realizzazione dell'applicazione.

Le interfacce si realizzano in modo molto semplice perchè *Xcode* consente di spostare gli elementi grafici con il *mouse* e di posizionarli come si vuole. Tuttavia, è stata la parte che ha richiesto più tempo perchè li abbiamo dovuti configurare nel modo più adatto alle nostre esigenze.

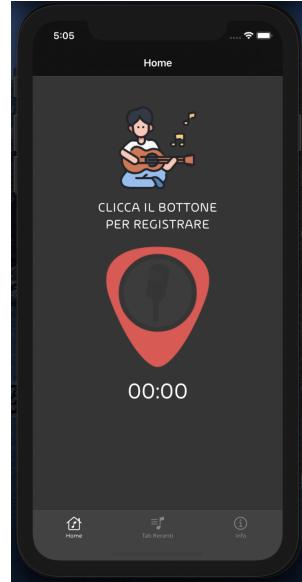
E' stata prestata anche molta attenzione a rendere compatibile l'applicazione su modelli diversi. La dimensione dello schermo influisce molto sul *layout* dell'applicazione. Senza le giuste modifiche è possibile che un elemento venga nascosto o spostato.



Le due immagini precedenti mostrano chiaramente quello appena descritto: i due dispositivi sono diversi e le proporzioni vengono rispettate in entrambi.

L'applicazione che è stata realizzata da un punto di vista estetico è uguale a quella su *Android*. Cambiano solo leggeri particolari che differenziano i due mondi.

Inoltre, una particolare attenzione è stata dedicata anche sulla nuova modalità che sta riscontrando un grandissimo successo: la *dark mode*. Dunque, sono stati presi tutti gli accorgimenti necessari per avere sia l'app compatibile con la versione chiara che con quella scura. L'immagine successiva mostra l'app in versione *dark*:



### 4.3.3 Uso di un server per l'uso della libreria librosa

**Problemi riscontrati:** puttropo non sono state trovate librerie in grado di convertire il *file* audio nella trasformata a Q costante.

**Soluzioni provate:**

- Abbiamo provato ad usare la libreria *PythonKit* senza successo perchè sui dispositivi *iOS* manca l'interprete *Python*.

**Soluzione finale:** per questo motivo ci siamo serviti di un *server* che prende in ingresso la registrazione che è stata effettuata dallo *smartphone* e restituisce in uscita le immagini del *file* audio. Ovviamente la soluzione non è efficiente ma ai fini del progetto può andare più che bene. La predizione viene eseguita sul dispositivo e **non** sul *server*.

Il *server* è stato implementato grazie al *micro-framework Flask* scritto in *Python*.

```
1 import flask
2 import werkzeug
3 import os
4 from preprocessing import preprocessing_file
5 from flask import jsonify
6
7 app = flask.Flask(__name__)
8
9 @app.route('/upload/', methods = ['POST'])
10 def handle_request():
11     audiofile = flask.request.files['file']
12     filename = werkzeug.utils.secure_filename(audiofile.filename)
13     print("\nReceived audio File name : " + audiofile.filename)
14     audiofile.save(filename)
15
16     images, frames = preprocessing_file(filename)
17     print(images.shape)
18     print(frames)
19
20     os.remove(filename)
21
22     return jsonify({"images": images.tolist(), "frames": frames})
```

```

24 if __name__=="__main__":
25     app.run(host="0.0.0.0", port=5000, debug=True)

```

Le stesse operazioni descritte nel *paragrafo 4.2.2* vengono eseguite dal *server* che infatti ritorna solo i *frame* più significativi.

#### 4.3.4 Implementazione di Tensorflow Lite

Per utilizzare il nostro modello *TensorFlow Lite* all'interno dell'applicazione, abbiamo dovuto prima configurare il progetto usando la libreria *Firebase* e poi istanziato l'interprete per poterlo caricare. Nel seguente codice l'interprete viene istanziato grazie a *Interpreter()*.

```

1 static var interpreter: Interpreter!
2
3 static func loadModel('on' controller: UIViewController) -> Bool {
4     guard let modelPath = Bundle.main.path(forResource: "model", ofType: "tflite")
5         else {
6             // Invalid model path
7             return false
8         }
9
10    do {
11        interpreter = try Interpreter(modelPath: modelPath)
12    } catch {
13        return false
14    }
15
16    return true
17 }

```

Per eseguire l'inferenza è necessario effettuare delle trasformazioni sui dati che sono obbligatorie per poterli dare in ingresso al modello. L'*input* della rete deve essere un oggetto *Data* al cui interno sono presenti i dati dell'*array* contenente le immagini del *file* audio (*inputImages*).

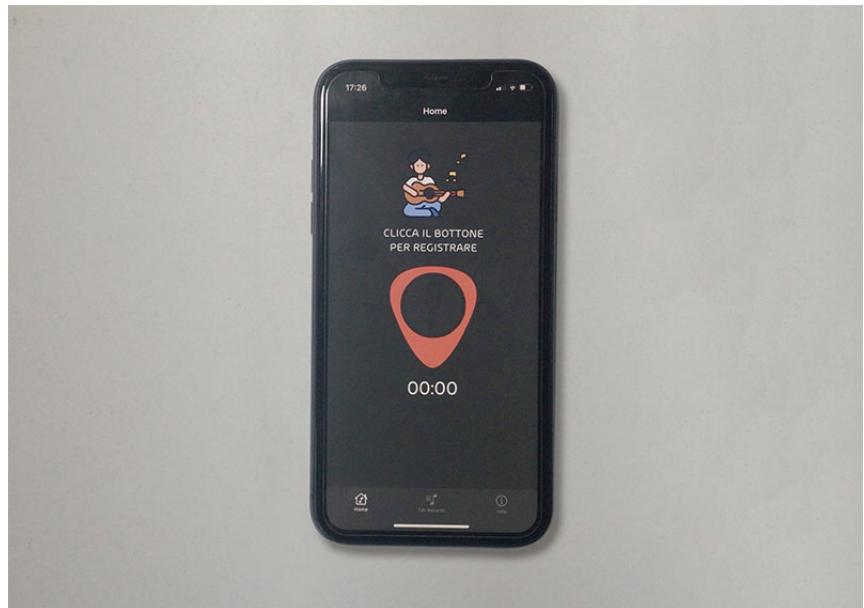
A questo punto si allocano i tensori, si inserisce l'*input* nella rete e si chiama il metodo *invoke()*. È possibile ottenere il tensore di *output* chiamando il metodo

*output(at:)* e gli elementi ottenuti dalla predizione tramite *output.data.copyBytes()*. Inoltre, si può notare che ci sono due cicli: quello esterno serve per leggersi tutte le immagini della registrazione mentre il secondo funge da *check* per verificare che l'immagine corrisponda a uno di quei picchi di cui abbiamo parlato in precedenza.

```
1 // input tensor [1, 192, 9, 1]
2 // output tensor [1, 6, 21]
3 for element in inputImages {
4
5     if (inputFrames.contains(Float32(i))) {
6
7         var inputData = Data()
8         for element2 in element {
9             for element3 in element2 {
10                 for element4 in element3 {
11                     var f = Float32(element4)
12                     //print(f)
13                     let elementSize = MemoryLayout.size(ofValue: f)
14                     var bytes = [UInt8](repeating: 0, count: elementSize)
15                     memcpy(&bytes, &f, elementSize)
16                     inputData.append(&bytes, count: elementSize)
17                 }
18             }
19         }
20
21         try TfliteModel.interpreter.allocateTensors()
22         try TfliteModel.interpreter.copy(inputData, toInputAt: 0)
23         try TfliteModel.interpreter.invoke()
24
25         let output = try TfliteModel.interpreter.output(at: 0)
26         let probabilities =
27             UnsafeMutableBufferPointer<Float32>.allocate(capacity: 126)
28         output.data.copyBytes(to: probabilities)
29
30     } // inputFrames
31
32     // elaborate results
33
34     i += 1
35 } // inputArray
```

### 4.3.5 Installazione su dispositivo fisico

L'applicazione è stata installata non solo sul simulatore ma anche sul dispositivo fisico. Lo *smartphone* è un *Iphone 11* con sistema operativo *iOS 14.1*.



# Capitolo 5

## Conclusioni

Sulle due applicazioni sono stati eseguiti diversi *test* e possiamo affermare che si comportano entrambe abbastanza bene. Certamente, il progetto potrebbe essere migliorato perchè i limiti individuati sono i seguenti:

- Se si suonassero ad esempio, le note 64 e 65 della prima corda in sequenza, la rete potrebbe predire una delle due note, o tutte e due, non sulla stessa corda ma in quella successiva. Questo non sarebbe un errore perchè il suono è lo stesso ma in certi casi sarebbe meglio suonare tasti vicini per una questione di manualità. Ciò accade perchè il sistema determina la tablatura finestra per finestra e non tiene conto della sequenza della registrazione;
- Se la canzone è troppo veloce, cioè ha un tempo o BPM (battiti per minuto) alto, la predizione tende ad essere errata;
- Alcune note potrebbero non essere riconosciute perchè c'è ancora un margine di errore di quasi il 10%.

Nonostante tutto, siamo molto soddisfatti del progetto che è stato realizzato. Grazie a questa materia abbiamo esplorato due mondi che per noi erano sconosciuti come lo sviluppo di applicazioni *mobile*. Siamo consapevoli che abbiamo perlustrato solo una piccola parte di questa vastissima materia ma quello che abbiamo appreso sarà sicuramente usato come base di partenza per progetti futuri.

A fini dimostrativi, forniamo insieme alla documentazione un breve video dell'applicazione in funzione.



# Bibliografia

- [1] *Chitarra classica*. (s.d.). Wikipedia, l'enciclopedia libera. Ultimo accesso: 28 febbraio 2021, <https://it.wikipedia.org/wiki/Chitarra>
- [2] *I tasti della chitarra*. (s.d.). TestoeAccordi. Ultimo accesso: 26 febbraio 2021, <https://www.testoeaccordi.it/menu/tasti.htm>
- [3] *Note manico chitarra* [Image]. (s.d.). Videocorsochitarra. Ultimo accesso: 27 febbraio 2021, <https://videocorsochitarra.it/note-manico-chitarra/>
- [4] Savage. N. (s.d.). *Come Leggere Tablature per Chitarra*. Wikihow. <https://www.wikihow.it/Leggere-Tablature-per-Chitarra>
- [5] Q. Xi, R. Bittner, J. Pauwels, X. Ye, and J. P. Bello, "Guitarset: A Dataset for Guitar Transcription", in *19th International Society for Music Information Retrieval Conference*, Paris, France, Sept. 2018.
- [6] Note Frequency Chart (Download) [Image], Soundonsound.com, Dec 03, 2018 11:32 am.
- [7] *Machine Learning e principio di funzionamento*. (s.d.). Lorenzo Govoni Business e Tecnologia Ultimo accesso: 24 marzo 2021, <https://www.lorenzogovoni.com/machine-learning-e-funzionamento/>
- [8] Drexel University, ExCITe Center, *Expressive and Creative Interaction Technologies*, NEMISIG 2019
- [9] *Utilizza un modello TensorFlow Lite per inferenza con ML Kit su iOS*. (s.d.). Firebase. Ultimo accesso: 03 marzo 2021, <https://firebase.google.com/docs/ml-kit/ios/use-custom-models>