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

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, l'aria si riempe delle sue note.
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 come è fatta 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 e gli obiettivi raggiunti.

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

Capitolo 1

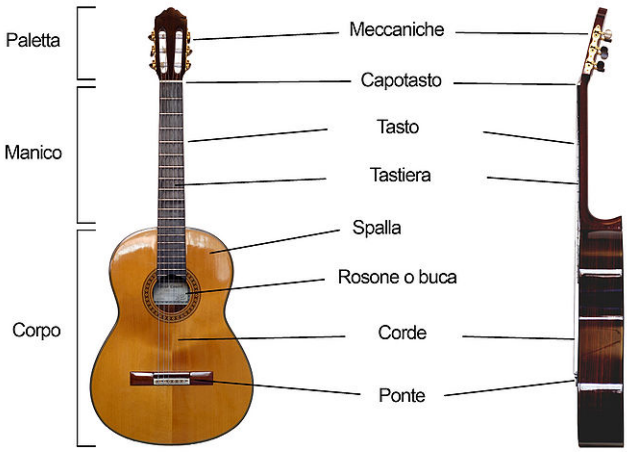
Chitarra

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 chiatarra 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

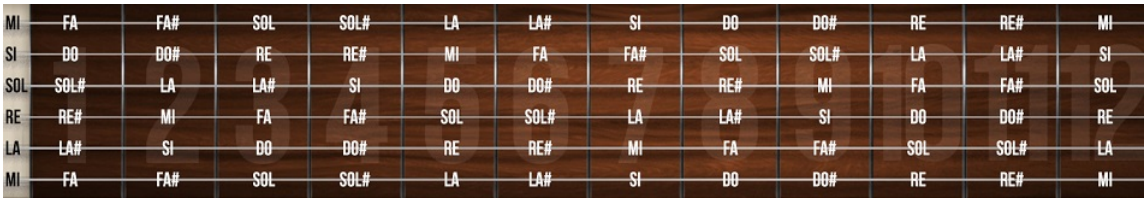
Le corde delle chitarre moderne sono sei e sono ordinate dall'alto verso il basso nel seguente modo:

- 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.

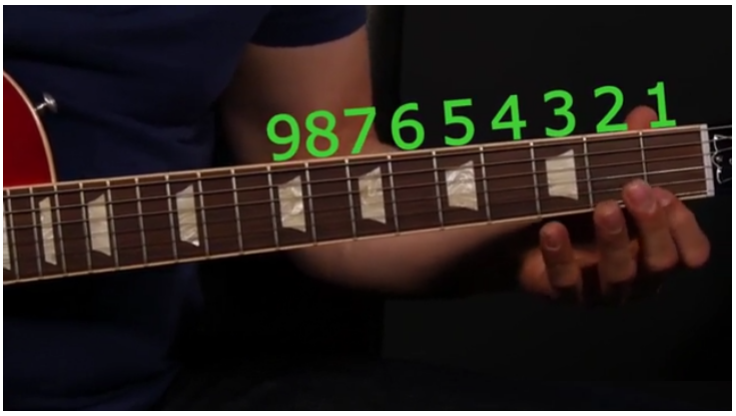
1.0.2 Tasti

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 19 tasti, possiamo fare 19 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: la chitarra deve essere quindi 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.

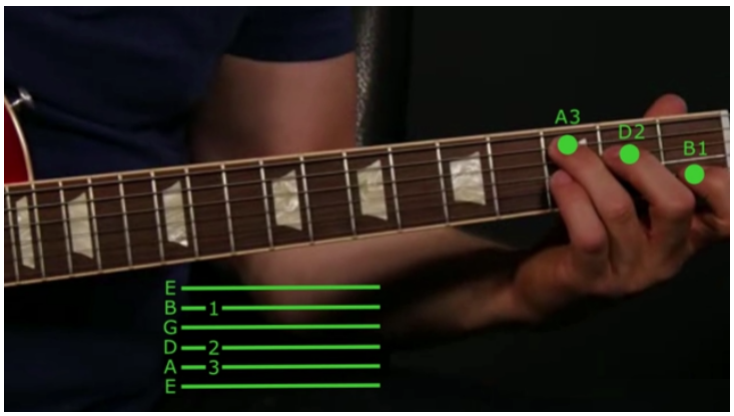


1.0.3 Tab

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 superiore a zero, 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)	NOTE	FREQ (Hz)	NOTE	FREQ (Hz)	NOTE	FREQ (Hz)	NOTE	FREQ (Hz)	NOTE	FREQ (Hz)	NOTE	FREQ (Hz)	NOTE	FREQ (Hz)	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	3322.44		
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 8 note posizionate a frequenza diversa nella scala musicale. Le frequenze intermedie sono altre sei note. Per esempio, il *la3* (A4) ha frequenza di 440 Hz, il *la* posto un'ottava sopra ha frequenza 880 Hz, quello un'ottava sotto ha frequenza 220 Hz.

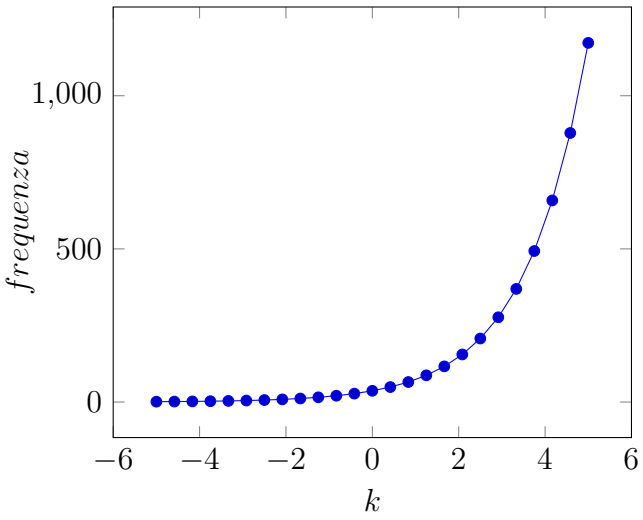
Se si rappresentassero le prime sei ottave della nota C potremmo vedere che la sua frequenza raddoppia ad ogni ottava.

I tasti che intercorrono fra gli estremi della stessa ottava (esempio *do3-do4*) sono dodici note per cui la frequenza deve raddoppiare ogni dodici note. Si può rappresentare quanto detto dalla seguente formula:

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

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

3.1 Librerie utilizzate

- *NumPy* è una libreria che aggiunge supporto a grandi matrici e array multi-dimensionali insieme a una vasta collezione di funzioni matematiche di alto livello per poter operare efficientemente su queste strutture dati;

- *Jams* è una libreria che legge file JSON inerenti al mondo della musica;

- *Librosa* è una libreria per la musica e l'analisi audio. Fornisce gli elementi necessari per recuperare informazioni musicali;

- *SciPy* 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* è una libreria per la creazione di grafici;

- *Tensorflow* 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* 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 essa dalla versione 2.

3.2 Set dati GuitarSet

Fortunatamente, su Internet abbiamo trovato un *data set* 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 *data set* contiene 360 estratti di canzoni della durata di circa 30 secondi l'uno. Essi sono il risultato delle seguenti combinazioni:

- 6 persone suonano ciascuno gli stessi 30 fogli
- Vengono registrate 2 versioni diverse: comping e soloing

I 30 fogli sono generati da una combinazione di

- **5 stili**: Rock, Cantautore, Bossa Nova, Jazz e Funk
- **3 progressioni**: 12 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 originale a 6 canali dal pickup esafonico;
- **hex_cln**: file wav esadecimali 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 360 estratti ha anche un file .jams che memorizza 16 annotazioni da cui prenderemo le tab:

- 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).

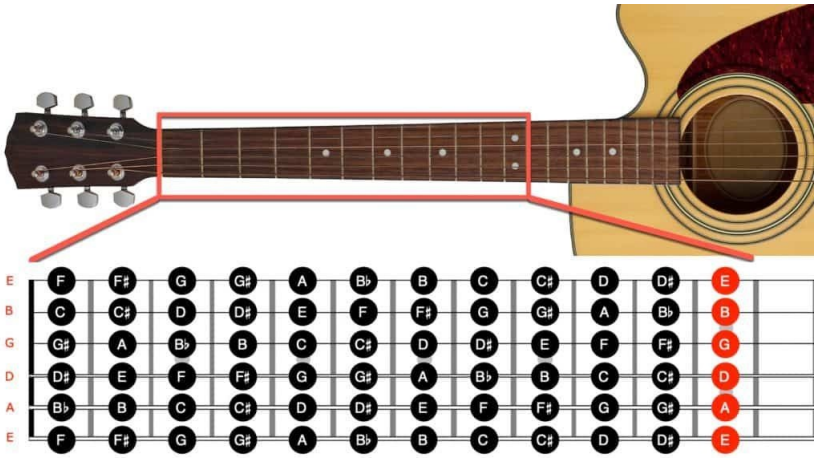
Noi useremo le annotazioni *midi_note*.

3.2.1 Ricavare le tab dai file .jams

Innanzitutto calcoliamo i *frame* per ogni *file* audio, così da poter ricavare un immagine e la corrispondente *tab* per ogni *frame*. Per calcolare l'istante di tempo per ogni *frame* utilizziamo la funzione *get_times()*

```
# parameters
self.sr_downs = 22050
self.hop_length = 512
self.n_bins = 192
self.bins_per_octave = 24

self.frameDuration = self.hop_length / self.sr_downs

def get_times(self, n):
    file_audio = os.path.join(self.path_audio,
                              os.listdir(self.path_audio)[n])
    (source_rate, source_sig) = wavfile.read(file_audio)
    duration_seconds = len(source_sig) / float(source_rate)
    totalFrame = math.ceil(duration_seconds / self.frameDuration)
    self.frame_indices = list(range(totalFrame))
    times = librosa.frames_to_time(self.frame_indices,
                                   sr = self.sr_downs,
                                   hop_length = self.hop_length)
    return times
```

Adesso che, per ogni file audio, abbiamo una divisione in frame di cui conosciamo gli istanti di tempo esatti, possiamo estarre dai file *.jams* del *dataset* le tab corrispondenti.
Più precisamente, dai file *.jams* prendiamo le *MIDI* note e creiamo una matrice 6x19 dove il 6 rappresenta il numero di corde mentre il 19 rappresenta il numero di tasti.



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

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

Ad ogni matrice sostituiamo i *MIDI* valori con una matrice della stessa dimensione in cui ci sono solo 1 o 0 a seconda dei *MIDI* valori che il *file .jams* ci ha restituito. Ad esempio, se dal file *.jams* abbiamo che in quel frame sono state suonate le note 40, 62 e 65 avremmo la seguente matrice:

```
[[1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1]
 [0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 1 0 0]
 [0 0 0 0 0 0 0 1 0 0 1 0 0 0 0 0 0 0]
 [0 0 0 1 0 0 1 0 0 0 0 0 0 0 0 0 0 0]
 [0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]]
```

Dato che una nota si può trovare su più corde, alla fine bisogna sceglierne soltanto una perchè il suono è lo stesso. Dunque, gli "1" in più devono essere cancellati. Inoltre, siccome queste matrici che chiameremo *labels* le daremo in input al modello è importante avere i dati categorici, quindi devono essere mappati come numeri interi per cui useremo la codifica *one-hot*, cioè si può trovare un "1" solo in ogni riga. A questa matrice aggiungiamo altre due colonne che servono a capire se la corda

è stata suonata (colonna 0) e se si, se è stata suonata a vuoto (colonna 1) oppure no.

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

```
[[0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
 [1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
 [1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
 [1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]]
```

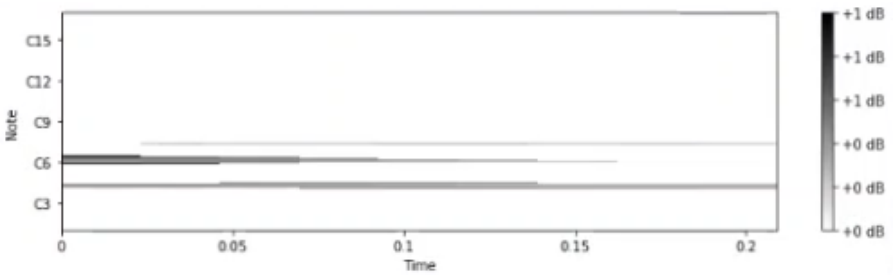
Il codice che esegue quanto abbiamo appena descritto è il seguente:

```
 # labeling parameters
self.string_midi_pitches = [40,45,50,55,59,64]
self.highest_fret = 19
self.num_classes = self.highest_fret + 2

def correct_numbering(self, n):
    n += 1
    if n < 0 or n > self.highest_fret:
        n = 0
    return n

def categorical(self, label):
    return to_categorical(label, self.num_classes)

def clean_label(self, label):
    label = [self.correct_numbering(n) for n in label]
    return self.categorical(label)

def clean_labels(self, labs):
    return np.array([self.clean_label(label) for label in labs])

def spacejam(self, file_num):
    path = os.path.join(self.path_anno, os.listdir(self.path_anno)[file_num])
    jam = jams.load(path)

    labs = []
    for string_num in range(6):
```

```
        anno = jam.annotations["note_midi"][string_num]
        string_label_samples = anno.to_samples(self.times)

        # replace midi pitch values with fret numbers
        for i in self.frame_indices:
            if string_label_samples[i] == []:
                string_label_samples[i] = -1
            else:
                string_label_samples[i] = int(round(string_label_samples[i][0]) -
                                              self.string_midi_pitches[string_num])
        labs.append([string_label_samples])

    labs = np.array(labs)

    # remove the extra dimension
    labs = np.squeeze(labs)
    labs = np.swapaxes(labs, 0, 1)

    # clean labels
    labs = self.clean_labels(labs)
    return labs
```

3.2.2 Ricavare le immagini dai file audio

Dopo aver trovato i *label* per ogni *frame*, dobbiamo trovare un modo per far imparare al modello che un frammento di audio sia associato al corrispondente *label*. Grazie a *librosa* possiamo convertire i *file* audio in immagini. Per far ciò usiamo la trasformazione a Q Costante che ci viene fornita da questa libreria e che ricava per ogni file audio un'immagine.



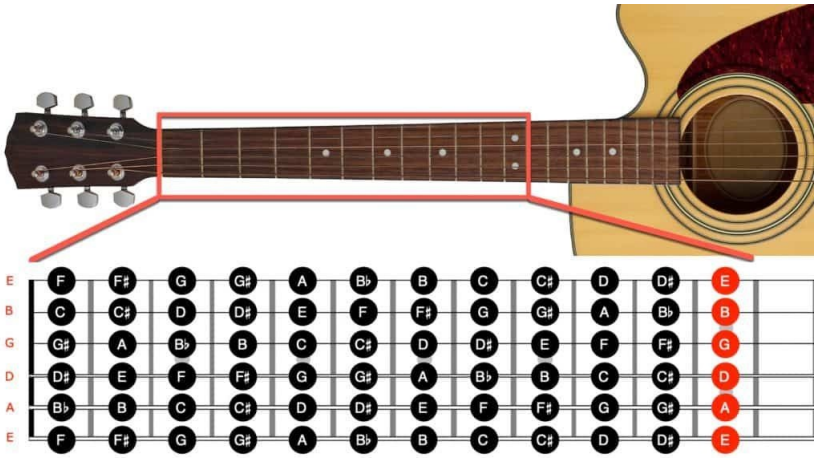

```
# parameters
```

```
self.sr_downs = 22050
self.hop_length = 512
self.n_bins = 192
self.bins_per_octave = 24

def audio_CQT(self, file_num):
    path = os.path.join(self.path_audio, os.listdir(self.path_audio)[file_num])

    # Perform the Constant-Q Transform
    data, sr = librosa.load(path, sr = self.sr_downs, mono = True, dtype='float64')
    data = librosa.util.normalize(data)
    data = librosa.cqt(data,
                       sr = self.sr_downs,
                       hop_length = self.hop_length,
                       fmin = None,
                       n_bins = self.n_bins,
                       bins_per_octave = self.bins_per_octave)
    CQT = np.abs(data)
    return CQT
```

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.

3.2.3 Pre-elaborare i dati

Carichiamo i file .npz e per ogni immagine aggiungiamo 4 zeri nell'array dell'immagine sia all'inizio che alla fine. Questo perchè per ogni frame salviamo 9 righe alla volta che corrispondono a 0.2 secondi.
Sperimentalmente, abbiamo visto che per ottenere un buon valore di accuratezza nella rete, le immagini devono essere stampate in questo arco temporale. Se il tempo fosse stato inferiore avremmo avuto immagini di file audio con solo note singole e quando la rete avrebbe dovuto riconoscere più note non sarebbestata in grado di farlo. Nell'immagine seguente è possibile vedere che le lettere sulle note si ripetono. Ad esempio, la lettera F si trova sulla corda F e D. Se la mano è posizionata sulla corda A, è impossibile che si riesca a suonare la F della corda G. Dunque, un istante di tempo troppo corto non aiutava la rete a riconoscerela nota giusta.



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

```
self.con_win_size = 9
self.halfwin = con_win_size // 2

def load_data(self):
    for i in range(self.n_file):
        self.log("n." + str(i+1))
        inp = np.load(os.path.join(self.data_path, os.listdir(self.data_path)[i]))

        full_x = np.pad(inp["imgs"], [(self.halfwin,self.halfwin), (0,0)],
                        mode='constant')

        for frame_idx in range(len(inp['imgs'])):
            # load a context window centered around the frame index
            sample_x = np.swapaxes(full_x[frame_idx : frame_idx+self.con_win_size],0,1)
            self.training_data.append((sample_x.astype('float64'),
                                       inp['labels'][frame_idx][::-1].astype('float64')))
```

Una volta ottenuto l'array di tutte le immagini e i label dei file audio lo mescoliamo in modo da avere 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 *traning*

Per costruire il modello usiamo il *traning set*, il validation set per validarlo mentre il *test set* per determinare l'accuratezza.

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

```
def prepare_data(self, data):
        inp = []
        out = []

        for imgs, labels in data:
                inp.append(imgs)
                out.append(labels)
        return inp, out

def partition_data(self):
        # Randomize training set
        random.shuffle(self.training_data)

        X_train, y_train = self.prepare_data(self.training_data)

        # Calculate validation and test set sizes
        val_set_size = int(len(self.training_data) * 0.1)
        test_set_size = int(len(self.training_data) * 0.1)

        # Break x apart into train, validation, and test sets
        self.X_val = X_train[:val_set_size]
        self.X_test = X_train[val_set_size:(val_set_size + test_set_size)]
        self.X_train = X_train[(val_set_size + test_set_size):]

        # Break y apart into train, validation, and test sets
        self.y_val = y_train[:val_set_size]
        self.y_test = y_train[val_set_size:(val_set_size + test_set_size)]
        self.y_train = y_train[(val_set_size + test_set_size):]

        self.log("Train set size: " + str(len(self.X_train)))
        self.log("Validation set size: " + str(len(self.X_val)))
        self.log("Test set size: " + str(len(self.X_test)))
```

3.3 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 CNN (Convolutional Neural Network). Le operazioni di calcolo matriciali possono essere accelerate sia tramite CPU, che GPU (su hardware Nvidia con supporto CUDA). Le caratteristiche e i vantaggi che ci hanno portato ad utilizzarlo 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.

3.3.1 Uso di Keras

Prima di dare in input al modello traning set e validation set, dobbiamo fare delle modifiche alla dimensione del numpy array di nome X (images) in quanto il modello si aspetta un input di BATCHx192x9x1.

- batch sono la quantità di valori dell'intero traning set;

- 192 è l'altezza dell'immagine;

- 9 è la lunghezza dell'immagine;

- 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 BATCHx21x6.
Il codice che esegue quanto abbiamo appena descritto è il seguente:

```
def load_data(self):
    # Load features sets
    features_sets = np.load(os.path.join("data/", self.feature_sets_file))

    # Assign feature sets
    X_train = features_sets['X_train']
    X_val = features_sets['X_val']

    self.y_train = features_sets['y_train']
    self.y_val = features_sets['y_val']

    # CNN for TF expects (batch, height, width, channels)
    # So we reshape the input tensors with a "color" channel of 1
    self.X_train = X_train.reshape(X_train.shape[0],
                                   X_train.shape[1],
                                   X_train.shape[2],
                                   1)
    self.X_val = X_val.reshape(X_val.shape[0],
                               X_val.shape[1],
                               X_val.shape[2],
                               1)
```

A questo punto definiamo il modello:

- **Conv2D**: serve per mettere 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. parametri di input: matrice di input;
- **Dropout** elimina una percentuale di dati casuali. Ad esempio, elimina rumori di sottofondo e mantiene le informazioni più importanti. Parametri di input: percentuale.
- **Flattern** appiattisce il tensore e rimuove tutte le dimensioni;
- **Dense**: decide il numero di numero di neuroni in uscita.
- **Reshape**: determina la dimensione di uscita. In questo caso è 6x21;

- **Activation** serve affinchè la somma di ogni elemento di uscita sia uno.

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 che esegue quanto abbiamo appena descritto è il seguente:

```
self.input_shape = (192, 9, 1)
self.num_classes = 21
self.num_strings = 6

def build_model(self):
    model = Sequential()
    model.add(Conv2D(32, kernel_size=(3, 3),
              activation='relu',
              input_shape=self.input_shape))
    model.add(Conv2D(64, (3, 3), activation='relu'))
    model.add(Conv2D(64, (3, 3), activation='relu'))
    model.add(MaxPooling2D(pool_size=(2, 2)))
    model.add(Dropout(0.25))
    model.add(Flatten())
    model.add(Dense(128, activation='relu'))
    model.add(Dropout(0.5))
    model.add(Dense(self.num_classes * self.num_strings))
    model.add(Reshape((self.num_strings, self.num_classes)))
    model.add(Activation(self.softmax_by_string))

    model.compile(loss=self.catcross_by_string,
                  optimizer=Adadelta(),
                  metrics=[self.avg_acc])

    self.model = model
```

Questo è il *summary* del modello:

```
Model: "sequential"
_________________________________________________________________
```

```
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 190, 7, 32)        320
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 188, 5, 64)        18496
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 186, 3, 64)        36928
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 93, 1, 64)         0
_________________________________________________________________
dropout (Dropout)            (None, 93, 1, 64)         0
_________________________________________________________________
flatten (Flatten)            (None, 5952)              0
_________________________________________________________________
dense (Dense)                (None, 128)               761984
_________________________________________________________________
dropout_1 (Dropout)          (None, 128)               0
_________________________________________________________________
dense_1 (Dense)              (None, 126)               16254
_________________________________________________________________
reshape (Reshape)            (None, 6, 21)             0
_________________________________________________________________
activation (Activation)      (None, 6, 21)             0
=================================================================
Total params: 833,982
Trainable params: 833,982
Non-trainable params: 0
```

In particolare, per il modello abbiamo usato delle funzioni personalizzate per la funzione di attivazione finale del modello (*sofmax_by_string*), per la funzione obiettivo *loss* (*catcross_by_string*) e per le metriche (*avg_acc*) che devono essere valutate del modello durante l'addestramento e il modello.

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

```
def softmax_by_string(self, t):
    sh = K.shape(t)
    string_sm = []
    for i in range(self.num_strings):
        string_sm.append(K.expand_dims(K.softmax(t[:,i,:]), axis=1))
```

```
    return K.concatenate(string_sm, axis=1)

def catcross_by_string(self, target, output):
    loss = 0
    for i in range(self.num_strings):
        loss += K.categorical_crossentropy(target[:,i,:], output[:,i,:])
    return loss

def avg_acc(self, y_true, y_pred):
    return K.mean(K.equal(K.argmax(y_true, axis=-1), K.argmax(y_pred, axis=-1)))
```

Per avviare l'addestramento del modello eseguiamo il comando *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 *traning*.

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

```
self.batch_size = 128
self.epochs = 500

def train(self):
    self.hist = History()
    self.model.fit(self.X_train,
                   self.y_train,
                   batch_size=self.batch_size,
                   epochs=self.epochs,
                   verbose=1,
                   validation_data=(self.X_val, self.y_val),
                   callbacks=[self.hist])
```

3.3.2 Addestramento del modello

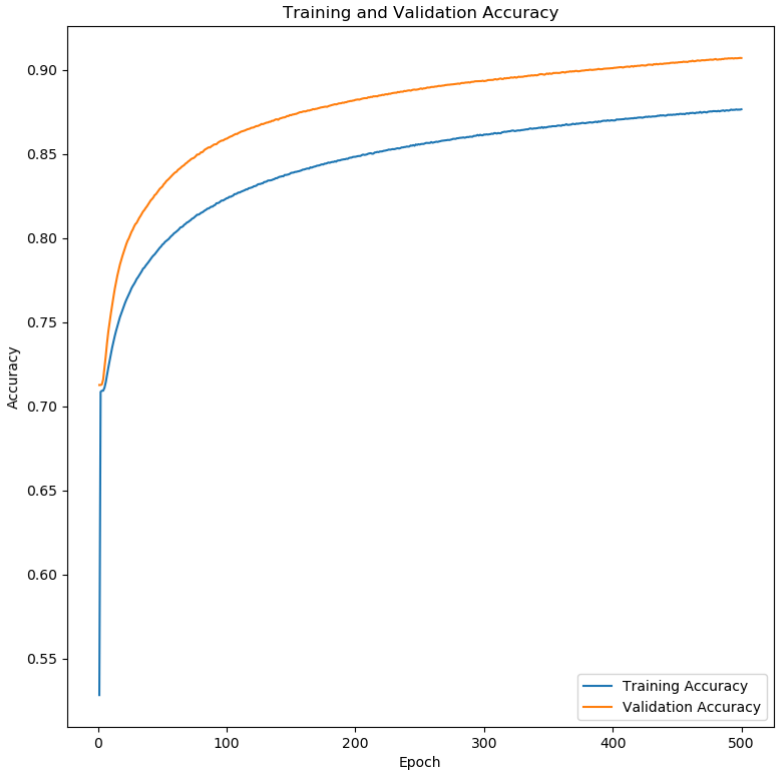
Dopo diverse prove sperimentali, abbiamo deciso di eseguire il modello per 500 epoche:

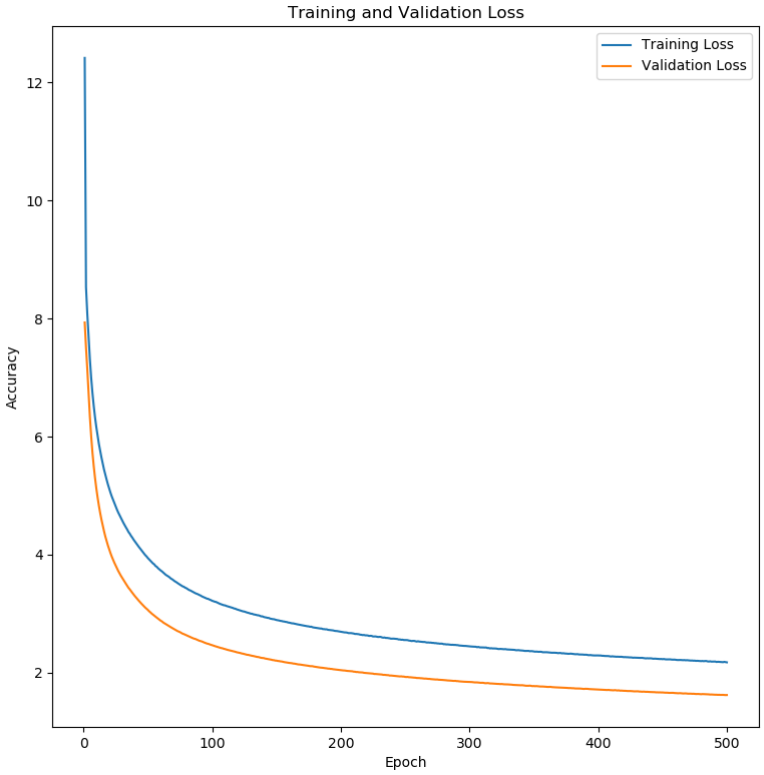
```
Train on 378048 samples, validate on 47256 samples
Epoch 1/500
378048/378048 [==============================] - 51s 135us/sample - loss: 12.4176 - avg
```

```
Epoch 2/500
378048/378048 [==============================] — 47s 124us/sample — loss: 8.5325 — avg_
Epoch 3/500
378048/378048 [==============================] — 47s 125us/sample — loss: 8.0792 — avg_
Epoch 4/500
378048/378048 [==============================] — 48s 127us/sample — loss: 7.6807 — avg_
Epoch 5/500
378048/378048 [==============================] — 47s 125us/sample — loss: 7.3140 — avg_
...
Epoch 496/500
378048/378048 [==============================] — 47s 124us/sample — loss: 2.1791 — avg_
Epoch 497/500
378048/378048 [==============================] — 47s 125us/sample — loss: 2.1769 — avg_
Epoch 498/500
378048/378048 [==============================] — 47s 124us/sample — loss: 2.1825 — avg_
Epoch 499/500
378048/378048 [==============================] — 47s 124us/sample — loss: 2.1780 — avg_
Epoch 500/500
378048/378048 [==============================] — 48s 128us/sample — loss: 2.1744 — avg_
```

Questo modello raggiunge una precisione di circa 0,87 (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. Con questi dati sembra avere ottimi risultati in quanto non abbiamo la presenza di *overfitting*.

Il grafico sottostante ci fa comprendere meglio i risultati:


Training and Validation Accuracy
Accuracy
Epoch
0.90
0.85
0.80
0.75
0.70
0.65
0.60
0.55
0
100
200
300
400
500
Training Accuracy
Validation Accuracy



3.4 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 l'1.

Di seguito riportiamo un esempio:

```
Answer:
[[1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]]
Prediction:
```

```
[[[9.92360413e−01 2.17636057e−06 9.15135024e−04 5.14742032e−06
2.43473296e−05 1.06439222e−06 7.78801041e−05 6.46503735e−03
1.63315246e−06 9.63436905e−05 1.24673179e−05 3.12326738e−05
2.25798635e−06 1.63281229e−07 3.24770076e−06 4.84239081e−07
1.52655346e−07 5.45700573e−07 8.95105003e−08 8.23356672e−08
1.47931530e−07]
[9.39490139e−01 3.07771461e−06 6.14974033e−06 4.04494698e−04
2.00287148e−04 4.00353492e−05 6.59466750e−05 5.94259761e−02
8.67796407e−05 5.67219940e−05 7.53599452e−05 1.00899604e−04
4.14257047e−05 1.81559798e−07 2.50978928e−07 6.39639268e−07
1.08545635e−06 7.57909859e−08 1.36756853e−07 2.80448461e−07
2.88561992e−08]
[6.80675685e−01 9.86625196e−07 9.40187310e−05 9.99796339e−06
3.16168219e−01 2.45542527e−04 2.77317440e−06 6.20479288e−04
8.65152571e−04 1.44455684e−04 3.30817638e−05 1.13483705e−03
5.68294695e−07 2.63474021e−07 1.93704693e−07 7.46157014e−07
2.58557316e−06 5.98578467e−08 1.14819862e−07 8.72972166e−08
7.63703412e−08]
[2.83732861e−01 1.93928386e−07 1.61119090e−06 1.88041362e−04
2.22303987e−01 5.61906018e−05 3.06668881e−05 9.38944722e−05
2.24740448e−04 4.93272096e−01 7.19120653e−05 1.72053715e−05
2.75737580e−06 2.60463275e−06 6.42855795e−08 5.21738400e−07
7.07197287e−08 1.48094529e−07 9.68439124e−08 5.93339031e−08
2.30616735e−07]
[1.95525795e−01 1.28646957e−06 5.31254616e−03 4.10607839e−07
2.72660202e−07 1.25411839e−06 6.68900725e−07 2.57475822e−05
4.56736198e−05 7.99066842e−01 1.76908379e−05 9.00572275e−07
1.72483126e−07 6.02985438e−07 5.14156078e−08 3.12393169e−08
2.88916464e−08 2.03121537e−08 3.34022587e−08 1.16275478e−08
3.34708155e−08]
[8.34586322e−01 5.58627034e−05 3.25484907e−05 7.17843577e−05
1.15673347e−05 2.41428029e−06 1.23440741e−05 1.65205747e−01
1.51618824e−05 1.64955986e−06 8.40611392e−07 3.49875990e−07
1.91993990e−07 2.15398154e−06 2.38864061e−07 3.07556469e−08
1.46060472e−07 1.96282883e−07 1.66214150e−07 1.29733195e−07
2.07605083e−07]]]
```

Abbiamo confrontato le prestazioni del modello sul set di dati di test 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.

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

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

```
def test(self):
    # TEST: Load model and run it against test set
    r = random.randint(0, len(self.X_test))
    for i in range (r, r + 10):
        print("Answer:", self.y_test[i], "Prediction:",
              self.model.predict(np.expand_dims(self.X_test[i], 0)))

    self.y_pred = self.model.predict(self.X_test)

def evaluate(self):
    # Evaluate model with test set
    results = self.model.evaluate(x=self.X_test, y=self.y_test)
    self.log("Test loss, Test acc: " + str(results))
```

Capitolo 4

Implementazione su dispositivo Embedded

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

4.1 Sviluppo applicazione iOS

4.1.1 Conversione del modello da Keras a CoreML

tempo di lavoro: 12 giorni

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 r
esource.
```

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:

```
import tfcoreml as tf_converter
tf_converter.convert(
            tf_model_path = './output/frozen_model.pb',
```

```
            mlmodel_path = './output/frozen.mlmodel',
            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*.

```
outpututput_directory_path = './output'

# the model_dir states where the graph and checkpoint files
# will be saved to
estimator_model = tf.keras.estimator.model_to_estimator(
                        keras_model = model,
                        model_dir = output_directory_path)

def input_function(features,labels=None,shuffle=False):
    input_fn = tf.estimator.inputs.numpy_input_fn(
        x={"conv2d_1_input": features},
        y=labels,
        shuffle=shuffle,
        batch_size = 128,
        num_epochs = 30
    )
    return input_fn

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.

```
def freeze_graph(model_dir, output_node_names):
  if not tf.gfile.Exists(model_dir):
    raise AssertionError(
      "Export directory doesn't exists. Please specify an export "
      "directory: %s" % model_dir)

  if not output_node_names:
    print("You need to supply the name of a node to output_node_names.")
    return -1

  checkpoint = tf.train.get_checkpoint_state(model_dir)
  input_checkpoint = checkpoint.model_checkpoint_path

  absolute_model_dir = "/".join(input_checkpoint.split('/')[:-1])
  output_graph = absolute_model_dir + "/frozen_model.pb"

  clear_devices = True

  with tf.Session(graph=tf.Graph()) as sess:
    saver = tf.train.import_meta_graph(
        input_checkpoint + '.meta', clear_devices=clear_devices)

    saver.restore(sess, input_checkpoint)

    output_graph_def = tf.graph_util.convert_variables_to_constants(
      sess,
      tf.get_default_graph().as_graph_def(),
      output_node_names.split(",")
    )

    with tf.gfile.GFile(output_graph, "wb") as f:
      f.write(output_graph_def.SerializeToString())
    print("%d ops in the final graph." % len(output_graph_def.node))

  return output_graph_def

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:

```
output_directory_path = './output'

supervisor = tf.train.Supervisor(logdir=output_directory_path)

with supervisor.managed_session() as session:
    # train the model here
    supervisor.saver.save(session, output_directory_path)
```

  Tuttavia, i risultati non sono stati quelli sperati.

- Cercando nella 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 utilizzato su sistemi operativi *iOS* maggiori di 12. La nuova libreria che abbiamo usato si chiama *coremltools*.

```
import coremltools
coreml_model = coremltools.converters.keras.convert(model)

coreml_model.author = 'De Nardi—Tornatore'
coreml_model.short_description = 'Recognition guitar music'

coreml_model.save("Stock.mlmodel")
```

- Per non avere altri errori che ci apparivano, abbiamo usato i livelli di convoluzione presi direttamente da *keras* e non da *tensorflow*. E' stato molto difficile superare tutti questi ostacoli perché i messaggi di errore non aiutavano a capire bene che cosa bisognasse modificare.

- Successivamente, per aumentare l'accuratezza, è stata inserita una funziona di attivazione personalizzata. Purtroppo, anche con *coremltools* non siamo stati in grado di convertirlo perchè 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.

4.1.2 Conversione del modello da Keras a Tensorflow Lite

tempo di lavoro: 1h

Come accennato nel paragrafo precedente, abbiamo convertito il modello usando *Tensorflow Lite*.

```
model = tf.keras.models.load_model("saved/model.h5",
    custom_objects={'softmax_by_string': self.softmax_by_string,
    'avg_acc': self.avg_acc, 'catcross_by_string': self.catcross_by_string})
                converter = lite.TFLiteConverter.from_keras_model(model)
                tflite_model = converter.convert()
                open("saved/model.tflite", "wb").write(tflite_model)
```

Il seguente codice consente di caricare il modello addestrato tramite *Tensorflow* e di convertirlo nel formato *.tflite* pronto per essere usato su un dispositivo mobile nel nostro caso.

4.1.3 Sviluppo applicazione con Swift 5 e Xcode 12

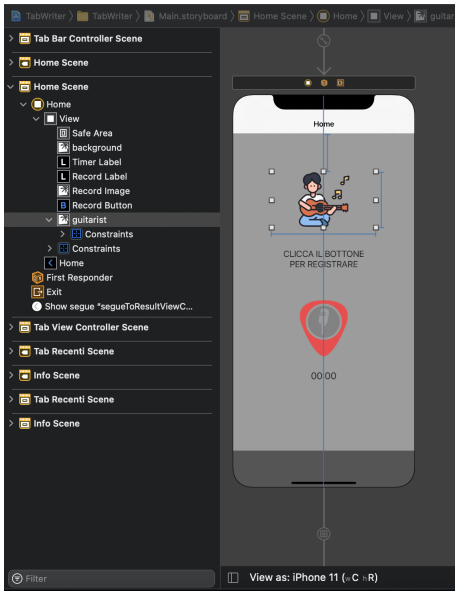
tempo di lavoro: 20 giorni

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

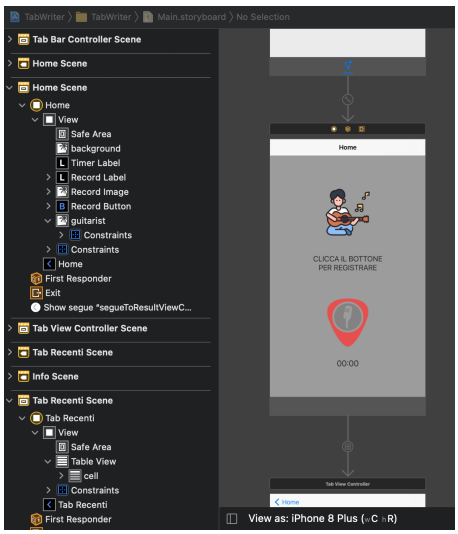
Xcode è un ambiente di sviluppo integrato completamente sviluppato e mantenuto da *Apple*, che consente di sviluppare *software* per i sistemi *macOS*, *iOS*, *watchOS* e *tvOS*.

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.



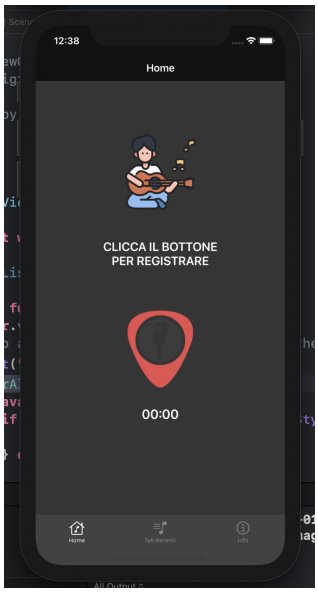
E' stata prestata anche molta attenzione a rendere compatibile l'applicazione su modelli diversi. La dimensione dello schermo influisce molto sul *layout* dell'applicazione. Senza le giuste modifiche è possibile che un elemento venga nascosto o spostato.



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

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*:



4.1.4 Uso di TensorFlow Lite su iOS

Abbiamo eseguito i seguenti passaggi:

- Registrato l'*app* sul sito *Firebase* perchè bisogna monitorarla (passaggio obbligatorio come scritto nella guida);
- Scaricato e aggiunto il file di configurazione che si ottiene dopo la registrazione su *Firebase*;
- Aggiunto *Firebase* all'*app*;
- Inizializzato *Firebase* nel progetto *iOS* e usate le sue *API* per usare il modello sullo *smartphone*.

4.1.5 Uso di un server per l'uso della libreria Librosa

tempo di lavoro: 5 giorni

Problemi riscontrati: putroppo 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 scritto grazie al *framework* di *Python* che si chiama *flask*

```
import flask

app = Flask(__name__)

@app.route('/upload/', methods = ['POST'])
def handle_request():
        #when the request arrives

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

4.2 Sviluppo applicazione Android

4.2.1 Conversione del modello da Keras a Tensorflow Lite

tempo di lavoro: 1h

Tutto ha funzionato al primo colpo senza nessun problema.

```
model = tf.keras.models.load_model("saved/model.h5",
    custom_objects={'softmax_by_string': self.softmax_by_string,
    'avg_acc': self.avg_acc, 'catcross_by_string': self.catcross_by_string})
                converter = lite.TFLiteConverter.from_keras_model(model)
                tflite_model = converter.convert()
                open("saved/model.tflite", "wb").write(tflite_model)
```

Il seguente codice consente di caricare il modello addestrato tramite *Tensorflow* e di convertirlo nel formato *.tflite* pronto per essere usato su un dispositivo mobile nel nostro caso.

4.2.2 Sviluppo applicazione con Java 1.8 e Android Studio 4.1.2

tempo di lavoro: 12 giorni

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*.

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

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

Soluzioni provate:

- Abbiamo usato la soluzione del *server* come in *iOS*.

Soluzione finale: grazie ai ricevimenti fatti con il professore, ci è stato consigliato di usare *chaquopy*.

Capitolo 5

Conclusioni

Sulle due applicazioni sono stati eseguiti diversi *test* e possiamo affermare che funziona tutto correttamente. Siamo molto soddisfatti del progetto che è stato realizzato. Siamo consapevoli che abbiamo esplorato solo una piccola parte di questa vastissima materia ma quello che abbiamo appreso sarà sicuramente usato come base di partenza in futuri progetti.

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] Drexel University, ExCITe Center, *Expressive and Creative Interaction Technologies*, NEMISIG 2019

[8] *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