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 2021/2022

- RELAZIONE FINALE - RICONOSCIMENTO DI GENERI MUSICALI TRAMITE L'USO DI CNN

Esame di Attività Progettuale di Sistemi Digitali M

Autore:
Gabriele Tornatore

Indice

Introduzione

Come mai prima d'ora, il web è diventato un luogo di condivisione di lavori creativi, come la musica, tra una comunità globale di artisti e amanti dell'arte. Sebbene le raccolte di musica precedono la nascita del Web, esso ha consentito raccolte su scala molto più ampia. Mentre prima le persone possedevano i dischi in vinile o i CD, al giorno d'oggi hanno accesso istantaneo a tutti i contenuti musicali pubblicati tramite piattaforme di streaming online come Spotify, iTunes, Youtube ecc.
Un aumento così drastico delle dimensioni delle raccolte musicali ha dato vita a due sfide:

- la necessità di organizzare automaticamente una raccolta (poiché utenti ed editori non possono più gestirle manualmente);
- la necessità di consigliare automaticamente nuove canzoni a un utente che conosce le proprie abitudini di ascolto;

Un compito fondamentale in entrambe queste sfide è essere in grado di raggruppare le canzoni in categorie semantiche. I generi musicali sono categorie che sono sorte attraverso una complessa interazione di culture, artisti e forze di mercato per caratterizzare le somiglianze tra le composizioni e organizzare le raccolte musicali. Eppure i confini tra i generi rimangono ancora confusi, rendendo il problema del riconoscimento del genere musicale un compito non banale.
Il progetto ha l'obiettivo di riconoscere in modo automatico il genere di un brano musicale di cui è disponibile solo una registrazione tramite l'utilizzo di reti neurali convoluzionali (*CNN*). Il tutto funziona tramite l'ausilio di un semplice *smartphone*.

La relazione è articolata come segue:

- nel **capitolo 1** viene presentato il dataset utilizzato;
- nel **capitolo 2** viene descritto il modello della rete, l'addestramento che è stato eseguito e la sua accuratezza;
- nel **capitolo 3** viene realizzata l'implementazione sul dispositivo *embedded*, gli *smartphone*;

- il **capitolo 4** 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, i modelli, i *log* e l'applicazione di tutto il progetto possono essere reperiti nella seguente repository git: https://github.com/it9tst/music-genre-recognition

Capitolo 1

Set dati utilizzati

Un genere musicale è una categoria convenzionale che identifica e classifica i brani e le composizioni in base a criteri di affinità. Le musiche possono essere raggruppate in base alle loro convenzioni formali e stilistiche, alla tradizione in cui si inseriscono, allo spirito dei loro temi, alla loro destinazione o, se presente, al loro testo. L'indeterminatezza di alcuni di questi parametri rende spesso la divisione della musica in generi controversa e arbitraria. Un genere musicale può a sua volta dividersi in sottogeneri che ne ampliano la complessità. Fortunatamente, su Internet si possono trovare tanti *dataset* adatti al progetto che si vuole svolgere.

1.1 Set dati FMA

In un primo momento si è deciso di utilizzare il set dati *FMA (Free Music Archive)* un set di dati open source e facilmente accessibile, adatto per valutare diverse attività di *MIR (Music Information Retrieval)*. Esso contiene 8000 brani di 30 secondi l'uno divisi equamente in 8 generi:

- Electronic
- Experimental
- Folk
- Hip-Hop
- Instrumental
- International
- Pop

- Rock

Inoltre viene dato in dotazione anche un file *.csv* contenente tutti i metadata dei brani come ID, titolo, artista, genere, tags, etc.
Questo *dataset* è stato utilizzato per i modelli basati su alberi decisionali, che si trovano nella cartella *models*, che vanno dal numero 1 al numero 6.
Nota a margine, invece di 8000 brani se ne sono potuti utilizzare solo 7994 perchè sei file erano corrotti.

1.2 Set dati GTZAN

Successivamente si è utilizzato il set dati *GTZAN* (detto anche il *MNIST of sounds*) che, a differenza del primo, contiene 1000 brani di 30 secondi l'uno divisi in 100 file audio per ogni genere. Quindi questa volta abbiamo a disposizione 10 generi:

- Blues
- Classical
- Country
- Disco
- Hip-Hop
- Jazz
- Metal
- Pop
- Reggae
- Rock

Il set di dati *GTZAN* è il set di dati open source più utilizzato con il machine learning per il riconoscimento del genere musicale. I file sono stati raccolti tra il 2000 e il 2001 da una varietà di fonti tra cui CD personali, radio e registrazioni microfoniche, al fine di rappresentare una varietà di condizioni di registrazione.
Questo *dataset* è stato utilizzato per i modelli basati su alberi decisionali, che si trovano nella cartella *models*, che vanno dal numero 7 al numero 11.
Nota a margine, invece di 1000 brani se ne sono potuti utilizzare solo 999 perchè un file era corrotto.

Capitolo 2

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.

2.1 Librerie utilizzate

I *package* usati per realizzare il progetto sono:

- *JupyterLab, versione 3.1.7* è un'interfaccia utente basata sul Web. Offre un ambiente di sviluppo interattivo per lavorare con i notebook Jupyter, il codice e i dati.

- *numpy, versione 1.19.5*, 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;

- *librosa, versione 0.8.1*, è una libreria per la musica e l'analisi audio. Fornisce gli elementi necessari per recuperare informazioni musicali;

- *matplotlib, versione 3.4.3*, è una libreria per la creazione di grafici;

- *scipy, versione 1.7.1*, 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;

- *seaborn, versione 0.11.2*, è una libreria che permette la creazione di grafici, molto diffuso tra data scientist e data analys;

- *tensorflow, versione 2.5.0*, 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;

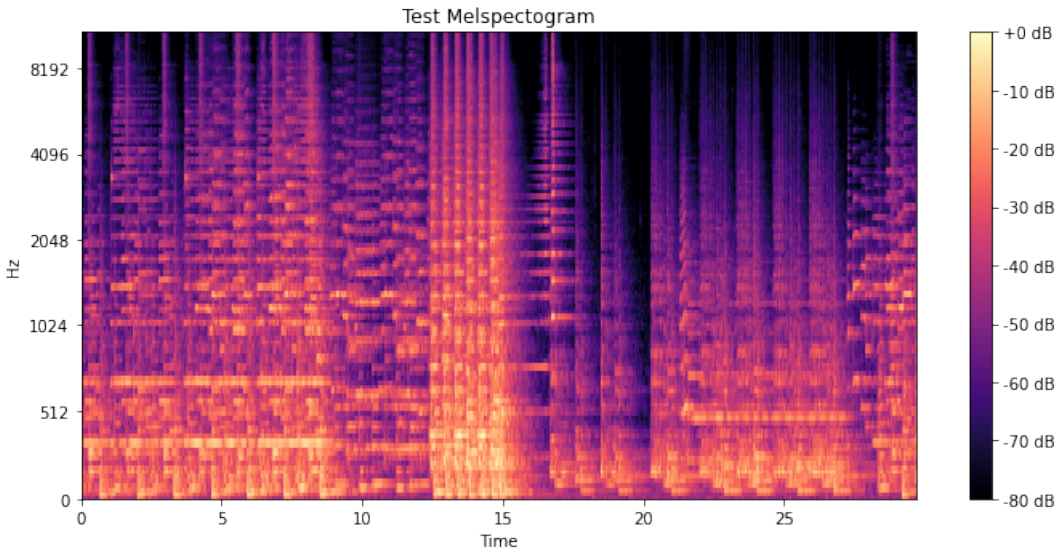
- *tensorboard, versione 2.5.0*, che consente di verificare visivamente e interpretare le esecuzioni e i grafici di TensorFlow.

2.2 Preparazione del dataset

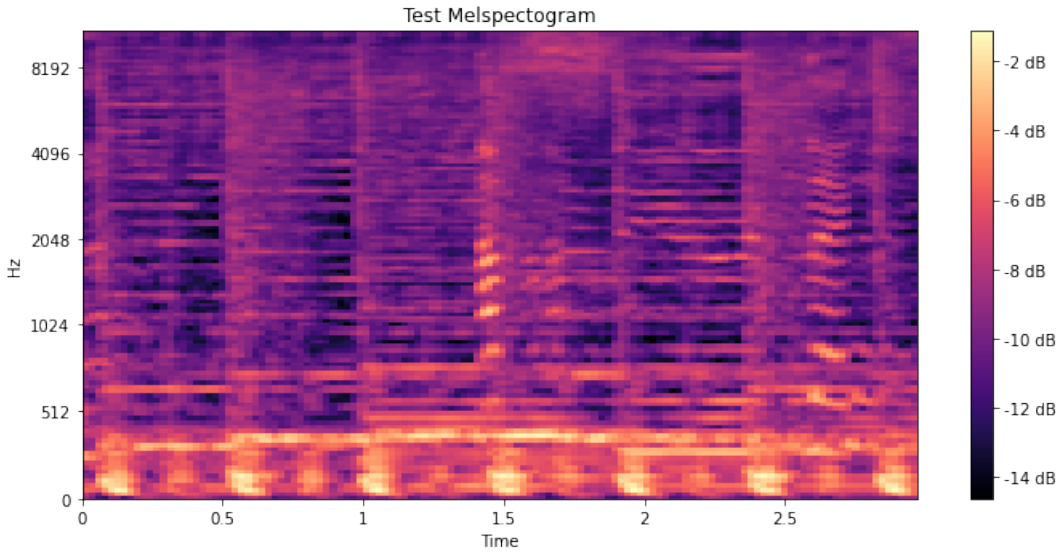
Ciò che segue è servito per preparare ed elaborare sia il primo set di dati *(FMA)* che il secondo *(GTZAN)*. Per evitare ripetizioni, descrivendo i dati nel dettaglio, mi riferirò solo al secondo e definitivo *dataset* utilizzato. Il Jupyter notebook dove si trova tutto il codice eseguito, per questa fase, è il file *"1 - Load dataset.ipynb"*.

Per prima cosa è stato creato un dizionario con tutti i generi del dataset. Successivamente per ogni file audio si è ricavato il corrispettivo spettrogramma grazie alla libreria *librosa*, con cui si può analizzare e manipolare il suono.
Per sopperire alla poca quantità di brani disponibili per genere e per evitare di dare al modello troppe informazioni in una volta, si è deciso di dividere ogni brano di 30 secondi (1280x128) in dieci file di 3 secondi l'uno (128x128), così da avere non più 100, ma 1000 file per genere per un totale complessivo di 7990 file (un brano era corrotto (splittato in 10 file) e quindi non è stato inserito).

Questo è un esempio di spettrogramma di un file audio di 30 secondi.



Invece questo è un esempio di spettrogramma di un file audio di 3 secondi.



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

```
def create_spectogram(audio_path):
    y, sr = librosa.load(audio_path)
    spect = librosa.feature.melspectrogram(y=y, sr=sr)
    spect = librosa.power_to_db(spect, ref=np.max)
    return spect.T

def data_split(X_data, y_data):
    X_data_split = []
    for x in X_data:
        X_data_split.extend(np.split(x, 10))

    X_data_split = np.array(X_data_split)
    y_data_split = np.repeat(y_data, 10, axis=0)
    print(X_data_split.shape, y_data_split.shape)

    return X_data_split, y_data_split

def create_array(g):
    genres = []
    X_spect = np.empty((0, 1280, 128))
    count = 0
```

```
    #Code skips records in case of errors
    print(g)
    for filename in os.listdir(os.path.join('data/genres_original/',f'{g}')):
        try:
            count += 1
            audio_path = os.path.join(f'data/genres_original/{g}',f'{filename}')
            spect = create_spectogram(audio_path)

            # Normalize for small shape differences
            spect = spect[:1280, :]
            X_spect = np.append(X_spect, [spect], axis=0)
            genres.append(dict_genres[g])
            if count % 100 == 0:
                print("Currently processing: ", count)
        except:
            print("Couldn't process: ", count)
            continue
    y_arr = np.array(genres)

    X_spect, y_arr = data_split(X_spect, y_arr)

    return X_spect, y_arr

dict_genres = {'Blues':0, 'Classical':1, 'Country':2, 'Disco':3, 'Hip-Hop':4, 'Jazz':5,
     'Metal':6, 'Pop':7, 'Reggae':8, 'Rock':9}
X_data_genre = [[],[],[],[],[],[],[],[],[],[]]
y_data_genre = [[],[],[],[],[],[],[],[],[],[]]

for g, i in list(dict_genres.items()):
    X_data_genre[i], y_data_genre[i] = create_array(g)
```

Una volta ottenuti gli *array* degli spettrogrammi e dei corrispettivi generi, si passa alla fase in cui mescoliamo i dati, in modo da avere più imprevedibilità, e di suddivisione 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*

Il *traning 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 che esegue quanto è stato appena descritto è il seguente:

```
def shuffle_data(X_data, y_data):
    training_data = []
    for i in range(X_data.shape[0]):
        training_data.append((X_data[i], y_data[i]))

    random.shuffle(training_data)

    return training_data

def prepare_data(data):
    X = []
    y = []

    for frames, labels in data:
        X.append(frames)
        y.append(labels)
    return X, y

def partition_data(training_data):
    X_train, y_train = prepare_data(training_data)

    # Calculate validation and test set sizes
    val_set_size = 100
    test_set_size = 100

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

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

```
35
36      print("Train set size: " + str(len(X_train)))
37      print("Validation set size: " + str(len(X_valid)))
38      print("Test set size: " + str(len(X_test)))
39
40      return np.array(X_train), np.array(y_train), np.array(X_valid), np.array(y_valid),
            np.array(X_test), np.array(y_test)
41
42  X_train_genre = [[],[],[],[],[],[],[],[],[],[]]
43  y_train_genre = [[],[],[],[],[],[],[],[],[],[]]
44  X_valid_genre = [[],[],[],[],[],[],[],[],[],[]]
45  y_valid_genre = [[],[],[],[],[],[],[],[],[],[]]
46  X_test_genre = [[],[],[],[],[],[],[],[],[],[]]
47  y_test_genre = [[],[],[],[],[],[],[],[],[],[]]
48
49  for g, i in list(dict_genres.items()):
50      print(g)
51      training_data = shuffle_data(X_data_genre[i], y_data_genre[i])
52      X_train_genre[i], y_train_genre[i], X_valid_genre[i], y_valid_genre[i],
            X_test_genre[i], y_test_genre[i] = partition_data(training_data)
```

Eseguiamo una conversione da spettrogrammi in scala dB in spettrogrammi di potenza.

```
1  X_train_raw = librosa.core.db_to_power(X_train, ref=1.0)
2  X_train_log = np.log(X_train_raw)
3  print(np.amin(X_train_raw), np.amax(X_train_raw), np.mean(X_train_raw))
4  print(np.amin(X_train_log), np.amax(X_train_log), np.mean(X_train_log))
5
6  X_valid_raw = librosa.core.db_to_power(X_valid, ref=1.0)
7  X_valid_log = np.log(X_valid_raw)
8  print(np.amin(X_valid_raw), np.amax(X_valid_raw), np.mean(X_valid_raw))
9  print(np.amin(X_valid_log), np.amax(X_valid_log), np.mean(X_valid_log))
10
11 X_test_raw = librosa.core.db_to_power(X_test, ref=1.0)
12 X_test_log = np.log(X_test_raw)
13 print(np.amin(X_test_raw), np.amax(X_test_raw), np.mean(X_test_raw))
14 print(np.amin(X_test_log), np.amax(X_test_log), np.mean(X_test_log))
```

Usiamo una codifica *one-hot* per i labels in modo da avere una mappatura di numeri "categorica", cioè si può trovare un solo "1" in ogni riga; e successivamente salviamo i dati in archivi compressi *.npz* per organizzare meglio il *dataset* da dare come *input* al modello.

```
y_train = utils.to_categorical(y_train, num_classes=10).astype(int)
y_valid = utils.to_categorical(y_valid, num_classes=10).astype(int)
y_test = utils.to_categorical(y_test, num_classes=10).astype(int)

np.savez('data/all_targets_sets_train_new', X_train, y_train)
np.savez('data/all_targets_sets_valid_new', X_valid, y_valid)
np.savez('data/all_targets_sets_test_new', X_test, y_test)
```

2.3 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 le *CNN* (Convolutional Neural Network). Le operazioni di calcolo matriciali possono essere accelerate sia tramite *CPU*, che *GPU* (su *hardware Nvidia* con supporto *CUDA*). In questo caso è stata utilizzata una *GPU* in modo da sfruttare direttamente la sua potenza parallela contenute nelle schede video recenti.

```
config = ConfigProto()
config.gpu_options.allow_growth = True
sess = Session(config=config)

if tf.test.gpu_device_name():
    print("GPU found")
else:
    print("No GPU found")
```

2.3.1 Uso di Keras

Il Jupyter notebook dove si trova tutto il codice eseguito, per questa fase, sono i file riguardanti il modello che cominciano con il numero *"2"* e *"3"*.
Dopo aver caricato il *traning set* e il *validation set* possiamo definire 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;
- **Flattern**: appiattisce il tensore e rimuove tutte le dimensioni;
- **Dense**: crea un *layer* di neuroni, ognuno dei quali connesso ad ogni uscita del *layer* precedente e determina la dimensione di uscita;
- **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 è stato scelto come ottimizzatore l'algoritmo $ADAM$ in quanto è genericamente raccomandato perchè riesce a "smussare" i passaggi di discesa del gradiente in modo che il percorso da seguire sia meno rumoroso e la convergenza più veloce.

Il codice che definisce il modello è il seguente:

```
num_classes = 10
nb_filters1=32
nb_filters2=64
nb_filters3=128
nb_filters4=512
```

```
ksize = (3,3)
pool_size_1= (2,2)

def conv_recurrent_model_build(model_input):
    print('Building model...')
    layer = model_input

    ### Convolutional blocks
    conv_1 = Conv2D(filters = nb_filters1, kernel_size = ksize, strides=1, padding='
        valid', activation='relu', name='conv_1')(layer)
    pool_1 = MaxPooling2D(pool_size_1)(conv_1)

    conv_2 = Conv2D(filters = nb_filters2, kernel_size = ksize, strides=1, padding='
        valid', activation='relu', name='conv_2')(pool_1)
    pool_2 = MaxPooling2D(pool_size_1)(conv_2)

    conv_3 = Conv2D(filters = nb_filters3, kernel_size = ksize, strides=1, padding='
        valid', activation='relu', name='conv_3')(pool_2)
    pool_3 = MaxPooling2D(pool_size_1)(conv_3)

    conv_4 = Conv2D(filters = nb_filters4, kernel_size = ksize, strides=1, padding='
        valid', activation='relu', name='conv_4')(pool_3)
    pool_4 = MaxPooling2D(pool_size_1)(conv_4)

    flatten1 = Flatten()(pool_4)

    output = Dense(num_classes, activation='softmax', name='preds')(flatten1)

    model_output = output
    model = Model(model_input, model_output)

    opt = Adam(learning_rate=0.001)

    model.compile(loss='categorical_crossentropy', optimizer=opt, metrics=['accuracy'])
    print(model.summary())
    return model
```

Prima di avviare l'addestramento del modello, definiamo alcune funzioni *callback* che possono essere eseguite in momenti specifici del ciclo di training, ad esempio

alla fine di ogni epoca:

- **TensorBoard**: per usare Tensorboard;
- **Checkpoint**: consente di salvare il modello, con i suoi pesi, al verificarsi di specificate condizioni;
- **Reduce Learning Rate on Plateau**: consente di attuare una strategia flessibile di training, riducendo il learning rate se le prestazioni del modello non migliorano (plateau).

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 è il seguente:

```
batch_size = 32
n_features = X_train.shape[2]
n_time = X_train.shape[1]
EPOCH_COUNT = 30

def train_model(x_train, y_train, x_val, y_val):
    n_frames = 128
    n_frequency = 128

    #reshape and expand dims for conv2d
    x_train = np.expand_dims(x_train, axis = -1)
    x_val = np.expand_dims(x_val, axis = -1)

    input_shape = (n_frames, n_frequency, 1)
    model_input = Input(input_shape, name='input')

    model = conv_recurrent_model_build(model_input)

    log_dir = "./logs/without_tree/"+datetime.now().strftime("%Y%m%d-%H%M%S")
    tb_callback = TensorBoard(log_dir=log_dir, histogram_freq=1, write_graph=True,
        write_images=False, update_freq='batch', profile_batch=2, embeddings_freq=0,
        embeddings_metadata=None)

```

```
22      checkpoint_callback = ModelCheckpoint('./models/without_tree/weights.best.h5',
            monitor='val_accuracy', verbose=1, save_best_only=True, mode='max')
23
24      reducelr_callback = ReduceLROnPlateau(monitor='val_accuracy', factor=0.5, patience
            =10, min_delta=0.01, verbose=1)
25      callbacks_list = [tb_callback, checkpoint_callback, reducelr_callback]
26
27      # Fit the model and get training history.
28      print('Training...')
29      history = model.fit(x_train, y_train, batch_size=batch_size, epochs=EPOCH_COUNT,
            validation_data=(x_val, y_val), verbose=1, callbacks=callbacks_list)
30
31      return model, history
```

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

```
1  Model: "model"
2  _________________________________________________________________
3  Layer (type)                 Output Shape              Param #
4  =================================================================
5  input (InputLayer)           [(None, 128, 128, 1)]     0
6  _________________________________________________________________
7  conv_1 (Conv2D)              (None, 126, 126, 32)      320
8  _________________________________________________________________
9  max_pooling2d (MaxPooling2D) (None, 63, 63, 32)        0
10 _________________________________________________________________
11 conv_2 (Conv2D)              (None, 61, 61, 64)        18496
12 _________________________________________________________________
13 max_pooling2d_1 (MaxPooling2 (None, 30, 30, 64)        0
14 _________________________________________________________________
15 conv_3 (Conv2D)              (None, 28, 28, 128)       73856
16 _________________________________________________________________
17 max_pooling2d_2 (MaxPooling2 (None, 14, 14, 128)       0
18 _________________________________________________________________
19 conv_4 (Conv2D)              (None, 12, 12, 512)       590336
20 _________________________________________________________________
21 max_pooling2d_3 (MaxPooling2 (None, 6, 6, 512)         0
22 _________________________________________________________________
23 flatten (Flatten)            (None, 18432)             0
```

```
_________________________________________________________________
preds (Dense)                (None, 10)                184330
=================================================================
Total params: 867,338
Trainable params: 867,338
Non-trainable params: 0
```

2.3.2 Addestramento e valutazione del modello

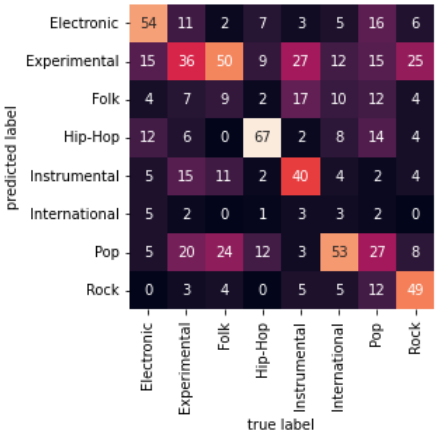
Durante il progetto sono stati effettuati diversi addestramenti che si trovano tutti nella cartella *models* con i corrispettivi log che si trovano nella cartella *logs*.

Problemi riscontrati: i primi test effettuati con il *dataset FMA* hanno riportato risultati abbastanza deludenti con valori di accuratezza sul *test set* intorno al 40%.

Soluzioni provate: una prima soluzione è stata quella di utilizzare un albero decisionale binario in modo da suddividere i dati in "isole" separate ricorsivamente e scegliendo, per ogni nodo, tra le possibili combinazioni, quella con il valore di accuratezza sul *validation set* maggiore. Sono stati testati 6 alberi che variano per settaggi del modello e tipologia (alberi bilanciati o sbilanciati - alberi numero 2 e numero 3). Purtroppo non ci sono stati miglioramenti rispetto al modello singolo senza albero, con i valori di accuratezza sul *test set* che si sono aggirati intorno al 35% - 40%.

Di seguito uno dei tanti report e *confusion matrix* ottenuti.

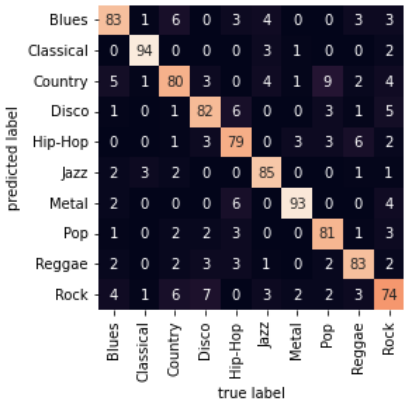
	precision	recall	f1-score	support
Electronic	0.52	0.54	0.53	100
Experimental	0.19	0.36	0.25	100
Folk	0.14	0.09	0.11	100
Hip-Hop	0.59	0.67	0.63	100
Instrumental	0.48	0.40	0.44	100
International	0.19	0.03	0.05	100
Pop	0.18	0.27	0.21	100
Rock	0.63	0.49	0.55	100
accuracy			0.36	800
macro avg	0.36	0.36	0.35	800
weighted avg	0.36	0.36	0.35	800



Soluzione finale: dopo svariati tentativi, il cambiamento di *dataset* da *FMA* a *GTZAN* è stata la soluzione più ottimale trovata. Infatti usando il secondo, con il modello senza albero, si sono visti risultati molto soddisfacenti con valori di accuratezza sul *test set* intorno al 79%. Anche questa volta, per migliorare ulteriormente l'accuratezza, è tornato in gioco l'utilizzo dell'albero decisionale dove con alberi bilanciati (test albero numeri 7, 8 e 9) il risultato è stato pressocchè uguale a quello senza albero, mentre utilizzando alberi sbilanciati (test albero numeri 10 e 11) si è finalmente giunti ad un miglioramento, seppur non di molto, con i valori di accuratezza sul *test set* che hanno superato l'83%.

Di seguito il report e la *confusion matrix* dell'albero numero 11, il definitivo.

	precision	recall	f1-score	support
Blues	0.81	0.83	0.82	100
Classical	0.94	0.94	0.94	100
Country	0.73	0.80	0.77	100
Disco	0.83	0.82	0.82	100
Hip-Hop	0.81	0.79	0.80	100
Jazz	0.90	0.85	0.88	100
Metal	0.89	0.93	0.91	100
Pop	0.87	0.81	0.84	100
Reggae	0.85	0.83	0.84	100
Rock	0.73	0.74	0.73	100
accuracy			0.83	1000
macro avg	0.84	0.83	0.83	1000
weighted avg	0.84	0.83	0.83	1000

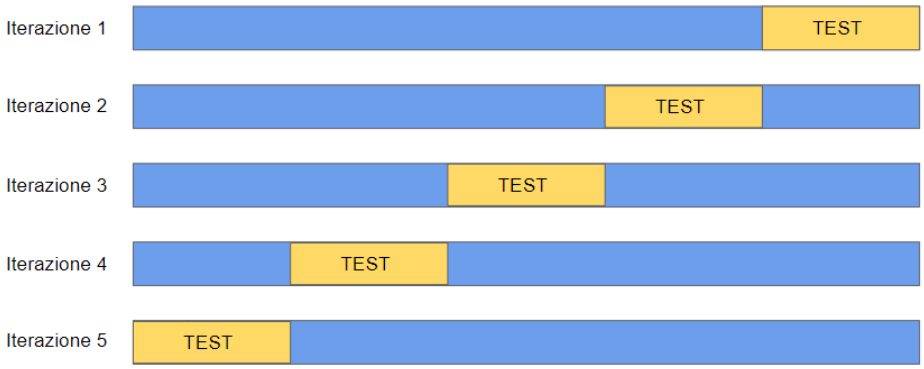


2.3.3 K-Fold Cross Validation

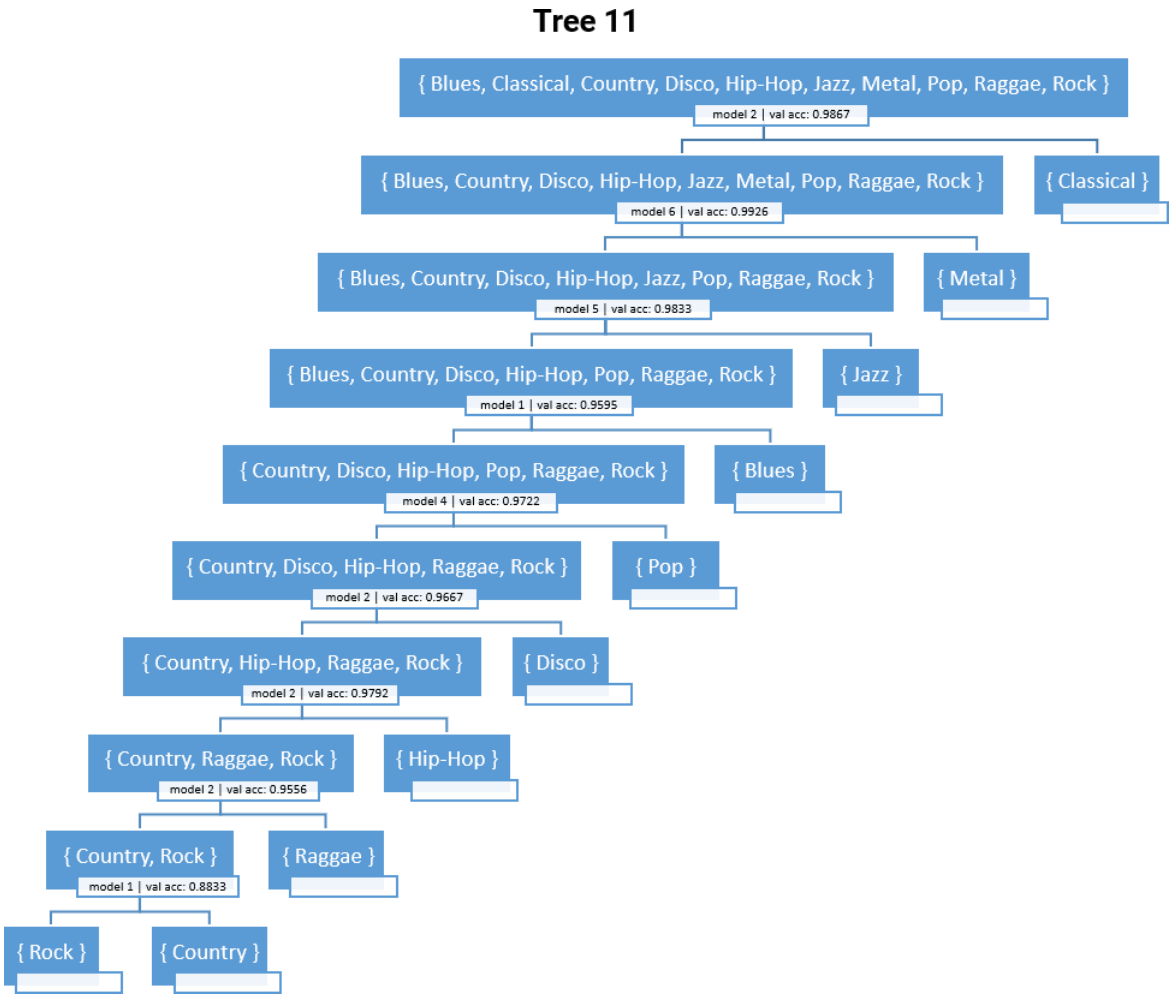
Per la creazione dell'albero decisionale e quindi per la scelta del miglior modello per ogni nodo, si è adottata la tecnica del *Cross Validation* (o validazione incrociata) che è una tecnica statistica, spesso chiamata anche k-fold cross validation perché il dataset iniziale viene diviso in una serie di porzioni uguali di dati (k-campi) dove vengono usati iterativamente un tot per il *train set* e un tot per il *validation set*.

In questo modo si è in grado di limitare i danni nel caso di dati sporchi nel *training set*. Il risultato finale sarà poi una media delle performances delle varie iterazioni.
Non c'è da preoccuparsi se le performances cambieranno da iterazione a interazione, è proprio quello il senso di usare ogni volta dati differenti per addestrare e valutare il modello.

Nella *Cross Validation* si vede bene come per ogni iterazione si utilizzano porzioni diverse per il *train set* (parte blu) e per il *validation set* (parte gialla).



Ed ecco l'albero sbilanciato ottenuto, con specificato il numero di modello migliore scelto, per ogni nodo:



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

```
X_train_cross = np.concatenate((X_train, X_valid), axis=0)
y_train_cross = np.concatenate((y_train, y_valid), axis=0)

num_splits = 10
n_splits = 15

kfold = KFold(n_splits, shuffle=True)

fold_no = 1
for train, test in kfold.split(X_train_cross, y_train_cross):
    dict_genres_list = list(dict_genres.values())
    split = [[dict_genres_list.pop(fold_no - 1)], dict_genres_list]

    y_train_cross_binary = np.argmax(y_train_cross, axis=1)
    y_train_cross_binary = np.in1d(y_train_cross_binary, split[0])
    y_train_cross_binary = utils.to_categorical(y_train_cross_binary*1, num_classes=2)

    print("Folder num {}".format(fold_no))
    genres_name = [[reverse_map[g] for g in split_l] for split_l in split]
    print("Genre split {}".format(genres_name))

    model, history  = train_model(X_train_cross[train], y_train_cross_binary[train],
        X_train_cross[test], y_train_cross_binary[test], "folder{}".format(fold_no),
        genres_name)
    show_summary_stats(history)

    fold_no += 1
    if fold_no > num_splits:
        break
```

Capitolo 3

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.

3.1 Conversione del modello da Keras a Tensorflow Lite

Il modello è stato convertito in *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*.

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

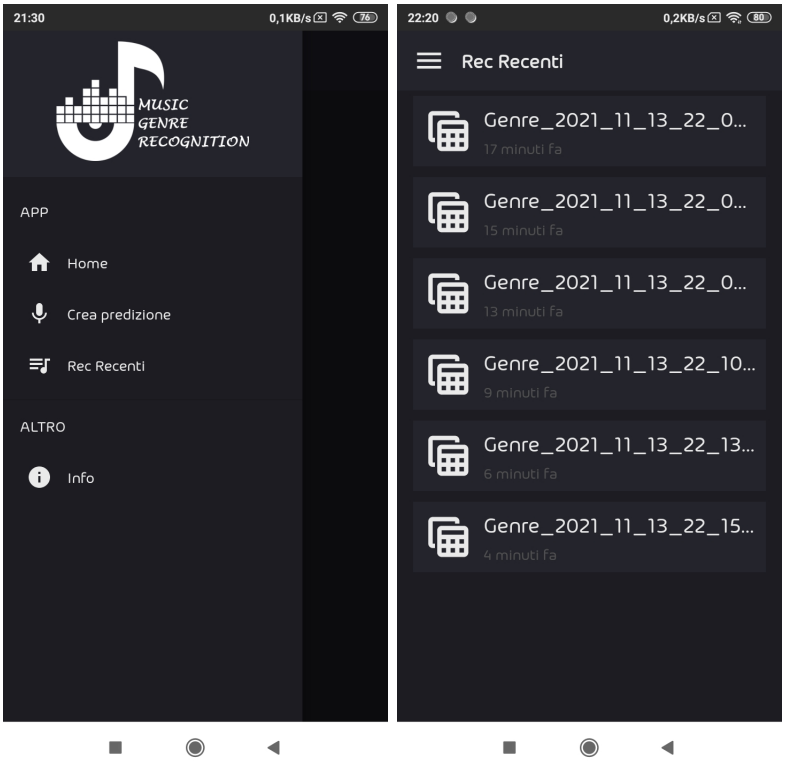
```
from tensorflow import lite
from tensorflow.keras.models import load_model

model_root_0 = load_model('models/tree_11/model_finals/model_root.h5')
converter = lite.TFLiteConverter.from_keras_model(model_root_0)
tflite_model = converter.convert()
open("model_root_0.tflite", "wb").write(tflite_model)
```

Il Jupyter notebook dove si trova il codice anche per gli altri modelli dell'albero si trova nel file *"5 - Model tflite.ipynb"*.

3.2 Sviluppo applicazione Android

3.2.1 Sviluppo applicazione con Java 1.8 e Android Studio 4.1.2

L'applicazione è stata creata in *Java* con *Android Studio*, che è un ambiente di sviluppo integrato per lo sviluppo per la piattaforma *Android*.



Sono state create un paio di interfacce con elementi grafici ed è stato implementato un *database* che tiene traccia del risultato delle registrazioni 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 sul pulsante centrale su cui è raffigurato un microfono. A questo punto il cellulare registra l'audio e per terminarla bisogna ricliccare sul pulsante.

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. Dopodiché il risultato apparirà in una nuova interfaccia.

Per ricavare lo spettrogramma dalle registrazioni audio e per gestire la predizione è stato usato *chaquopy*, un *plugin* che consente di implementare codice *Python* all'interno delle applicazioni *Android*.

3.2.2 Implementazione di Tensorflow Lite

Una volta ottenuti gli *input* della rete, possiamo eseguire l'inferenza con il modello. Per predire correttamente il modello con l'albero decisionale è stato necessario creare una classe *ModelNodeTree* che, essendo dinamica, funzioni con qualsiasi tipo e grandezza di albero.
Una volta passato il path del modello, relativo al nodo da predire, 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*. Passiamo il batch da predire tramite la funzione *interpreter.set_tensor()*, invochiamo l'interprete e tramite la funzione *interpreter.get_tensor()* otteniamo una copia dei valori provenienti dal tensore di *output*.

In base al risultato, continuiamo la discesa dell'albero verso il ramo destro o sinistro, ricorsivamente, fino al nodo radice.

```
class ModelNodeTree():
    def __init__(self, model, left, right, translate = None):
        self._model = model
        self._left = left
        self._right = right
        self._translate = translate

    def predict(self, batch):
        interpreter = tf.lite.Interpreter(model_path=self._model)
        interpreter.allocate_tensors()

        input_details = interpreter.get_input_details()
        output_details = interpreter.get_output_details()

        interpreter.set_tensor(input_details[0]['index'], batch)
        interpreter.invoke()
        y = interpreter.get_tensor(output_details[0]['index'])

        y = np.argmax(y, axis=-1)
        y_copy = np.copy(y)

        if self._translate is not None:
            for k, t in self._translate.items():
                y[y_copy == t] = k
        else:
            if np.sum(y_copy == 0) > 0:
                if type(self._left) == int:
                    y[y_copy == 0] = self._left
                else:
                    y[y_copy == 0] = self._left.predict(batch[y_copy == 0])

            if np.sum(y_copy == 1) > 0:
                if type(self._right) == int:
                    y[y_copy == 1] = self._right
                else:
                    y[y_copy == 1] = self._right.predict(batch[y_copy == 1])
        return y
```

Infine, i risultati vengono salvati in un *JSON* e conservati nel *database* dell'applicazione.

```
from model_node_tree import ModelNodeTree

f8 = ModelNodeTree(path_model_8, 9, 2)
f7 = ModelNodeTree(path_model_7, f8, 8)
f6 = ModelNodeTree(path_model_6, f7, 4)
f5 = ModelNodeTree(path_model_5, f6, 3)
f4 = ModelNodeTree(path_model_4, f5, 7)
f3 = ModelNodeTree(path_model_3, f4, 0)
f2 = ModelNodeTree(path_model_2, f3, 5)
f1 = ModelNodeTree(path_model_1, f2, 6)
root = ModelNodeTree(path_model_0, f1, 1)

def predict_model(images):
    data_json = []

    for i in range(images.shape[0]):
        input_data = images[i].reshape(1, images.shape[1], images.shape[2], images.
            shape[3])
        genre = root.predict(input_data)
        data_json.append({'genre' : genre})

    return json.dumps(data_json, default=myconverter)
```

Per visionare il risultato ottenuto nello *smartphone*, si è scelto di mostrare a video solo i generi predetti con percentuali maggiori del 20%.

Capitolo 4

Conclusioni

Sull'applicazione sono stati eseguiti diversi *test* con risultati abbastanza buoni. Il fatto che ci voglia una finestra minima di 3 secondi per avere almeno un'immagine predetta, fa si che bisogni registrare un bel po di canzone per avere un risultato più accurato.
Alcune predizioni sbagliate potrebbero dipendere dal fatto che ci possa essere confusione tra le classi. Spesso le canzoni appartengono a più generi e sottogeneri che si somigliano tra loro e quindi questo potrebbe portare ad una predizione errata.
Espandere il campione originale è stato determinante per avere risultati migliori, nonostante ciò anche 1000 spettrogrammi per genere potrebbero essere un campione molto piccolo poiché stiamo addestrando dei modelli da zero. Un set di dati ancora più grande dovrebbe migliorarne i risultati.
Sarebbe interessante indagare meglio sul perchè il set di dati *FMA* non abbia raggiunto i risultati del set di dati *GTZAN*. Forse il primo è più impegnativo.
Una cosa certa è che l'utilizzo dell'albero decisionale ha contribuito a migliorare l'accuratezza del modello.

Nonostante tutto, il progetto realizzato è stato molto soddisfacente. Grazie a questa materia, prima con *Sistemi Digitali M*, e poi con la corrispondente *Attività Progettuale* si è potuto imparare ed esplorare un mondo bellissimo e vasto come quello del *Machine Learning*.

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

Bibliografia

[1] *Genere musicale*. (s.d.). Wikipedia, l'enciclopedia libera. Ultimo accesso: 11 novembre 2021, https://it.wikipedia.org/wiki/Genere_musicale

[2] *Generi musicali, teorie e criteri di classificazione*. (s.d.). Note tra le righe. Ultimo accesso: 11 novembre 2021, https://www.notetralerighe.it/teoria-musicale/generi-musicali

[3] *Learning to Recognize Musical Genre from Audio*. (s.d.). Papers With Code. Ultimo accesso: 11 novembre 2021, https://paperswithcode.com/paper/learning-to-recognize-musical-genre-from

[4] *Tecniche di ottimizzazione II*. (s.d.). Aaron Defazio. https://atcold.github.io/pytorch-Deep-Learning/it/week05/05-2/

[5] *Comprendere la discesa del gradiente e l'ottimizzazione di Adam*. (s.d.). Lorenzo Govoni Business e Tecnologia Ultimo accesso: 13 novembre 2021, https://ichi.pro/it/comprendere-la-discesa-del-gradiente-e-l-ottimizzazione-di-adam-78249617493008

[6] *Machine Learning e principio di funzionamento*. (s.d.). ichi.pro. Ultimo accesso: 13 novembre 2021, https://ichi.pro/it/comprendere-la-discesa-del-gradiente-e-l-ottimizzazione-di-adam-78249617493008

[7] K-fold Cross Validation (Download) [Image], Pulp Learning, Dec 03, 2018 11:32 am.

[8] Bob L. Sturm. The GTZAN dataset: Its contents, its faults, their effects on evaluation, and its future use. arXiv:1306.1461v2