Music Genre Classification - Classifying the genre of a music using deep neural networks

Introduction

Music Genre classification is one of the branches of *Music Information Retrieval (MIR)*. A robust recommendation system begins with the categorization of music genres. Sound processing is a huge reaseach area through which we can find solutions to various medical or mental issues through music theraphy solutions. There are various music applications such as Spotify, Google Play, Apple Music, etc., but for implementation, one of the most important steps is to classify the genre of a music which requires audio processing, it is one of the most complex tasks that involves time signal processing, time series, spectrograms, spectral coefficients, and audio feature extraction to feed a neural network.

Dataset description

The dataset used is GTZAN (the famous GTZAN dataset, the MNIST of sounds)

The GTZAN dataset contains 1000 audio files. Contains a total of 10 genres, each genre contains 100 audio files

1.Blues

2.Classical

3.Country

4.Disco

5.Hip-hop

6.Jazz

7.Metal

8.Pop

9.Reggae

10.Rock

Genres original

A compilation of ten genres, each with 100 audio recordings, each lasting 30 seconds (the famous GTZAN dataset, the MNIST of sounds)

Images original

Each audio file has a visual representation. Neural networks are one technique to classify data because they usually take in some form of picture representation.

CSV files

The audio files' features are contained within. Each song lasts for 30 seconds long has a mean and variance computed across several features taken from an audio file in one file. The songs are separated into 3 second audio files in the other file, which has the same format.

Tensorflow

TensorFlow is a python's open source library developed by google which provides a collection of workflows to develop and train models using Python or JavaScript, and to easily deploy in the cloud, on-prem, in the browser, or on-device no matter what language you use. The tf. data API enables you to build complex input pipelines from simple, reusable pieces. It is used ease the process of acquiring data, training models, serving predictions, and refining future results.

Tensorflow makes it easy to work on our machine learning and deep learning models, hence We used tensorflow and keras in our notebook to train and test the deep learning model.

Import libraries

In [1]:
```
# Importing all the required libraries

import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import scipy
import os
import pickle
import librosa
import librosa.display
import IPython.display as ipd
from IPython.display import Audio
from sklearn.model_selection import train_test_split
```

```
from sklearn.preprocessing import LabelEncoder
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras.models import Sequential
```

```
2024-04-19 17:58:22.749544: I tensorflow/core/platform/cpu_feature_guard.c
c:210] This TensorFlow binary is optimized to use available CPU instructio
ns in performance-critical operations.
To enable the following instructions: AVX2 FMA, in other operations, rebui
ld TensorFlow with the appropriate compiler flags.
```

In [2]:
```
# Reading the csv file
df = pd.read_csv("Data/features_3_sec.csv")
df.head()
```

Out[2]:

	filename	length	chroma_stft_mean	chroma_stft_var	rms_mean	rms
0	blues.00000.0.wav	66149	0.335406	0.091048	0.130405	0.00
1	blues.00000.1.wav	66149	0.343065	0.086147	0.112699	0.00
2	blues.00000.2.wav	66149	0.346815	0.092243	0.132003	0.00
3	blues.00000.3.wav	66149	0.363639	0.086856	0.132565	0.00
4	blues.00000.4.wav	66149	0.335579	0.088129	0.143289	0.00

5 rows × 60 columns

In [3]:
```
# Shape of the data
df.shape
```

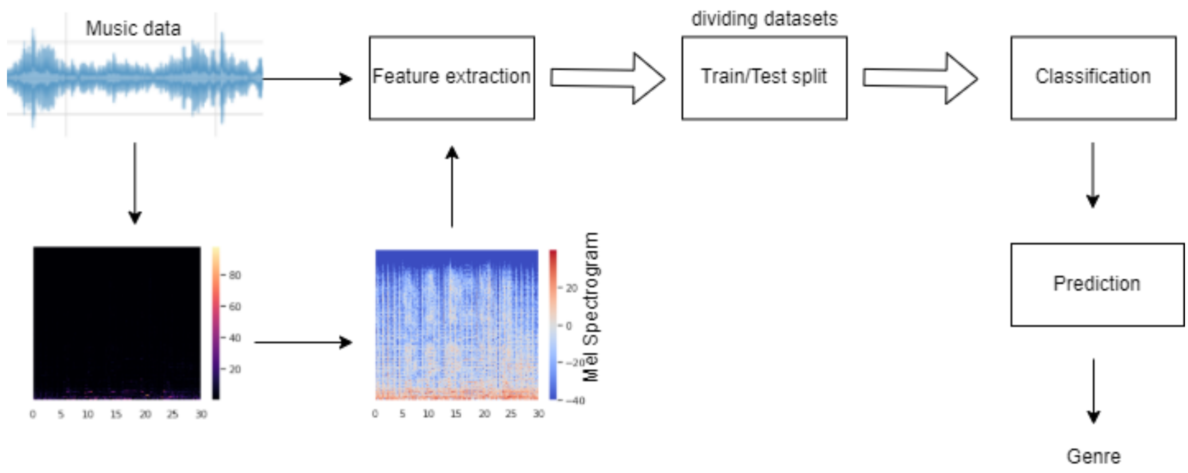
Out[3]: (9990, 60)

In [4]:
```
# Data type of the data
df.dtypes
```

Out[4]:
```
filename                    object
length                       int64
chroma_stft_mean           float64
chroma_stft_var            float64
rms_mean                   float64
rms_var                    float64
spectral_centroid_mean     float64
spectral_centroid_var      float64
spectral_bandwidth_mean    float64
spectral_bandwidth_var     float64
rolloff_mean               float64
rolloff_var                float64
zero_crossing_rate_mean    float64
zero_crossing_rate_var     float64
harmony_mean               float64
harmony_var                float64
```

```
perceptr_mean              float64
perceptr_var               float64
tempo                      float64
mfcc1_mean                 float64
mfcc1_var                  float64
mfcc2_mean                 float64
mfcc2_var                  float64
mfcc3_mean                 float64
mfcc3_var                  float64
mfcc4_mean                 float64
mfcc4_var                  float64
mfcc5_mean                 float64
mfcc5_var                  float64
mfcc6_mean                 float64
mfcc6_var                  float64
mfcc7_mean                 float64
mfcc7_var                  float64
mfcc8_mean                 float64
mfcc8_var                  float64
mfcc9_mean                 float64
mfcc9_var                  float64
mfcc10_mean                float64
mfcc10_var                 float64
mfcc11_mean                float64
mfcc11_var                 float64
mfcc12_mean                float64
mfcc12_var                 float64
mfcc13_mean                float64
mfcc13_var                 float64
mfcc14_mean                float64
mfcc14_var                 float64
mfcc15_mean                float64
mfcc15_var                 float64
mfcc16_mean                float64
mfcc16_var                 float64
mfcc17_mean                float64
mfcc17_var                 float64
mfcc18_mean                float64
mfcc18_var                 float64
mfcc19_mean                float64
mfcc19_var                 float64
mfcc20_mean                float64
mfcc20_var                 float64
label                       object
dtype: object
```

Proposed Methodology



In [5]:
```
# Loading a sample audio from the dataset
audio ="Data/genres_original/reggae/reggae.00010.wav"
data,sr=librosa.load(audio)
print(type(data),type(sr))
```

```
<class 'numpy.ndarray'> <class 'int'>
```

In order to work with audio data we use Librosa, a python package used for audio and music analysis. It is a powerful package widely used for audio visualization and for building MIR systems. We will be using the package for loading and visualizing the audio data.

In [6]:
```
# Initializing sample rate to 45600 we obtain the signal value array
librosa.load(audio,sr=45600)
```

Out[6]:
```
(array([-0.00555292, -0.00768963, -0.00668519, ...,  0.08035275,
         0.0663713 ,  0.03239053], dtype=float32),
 45600)
```

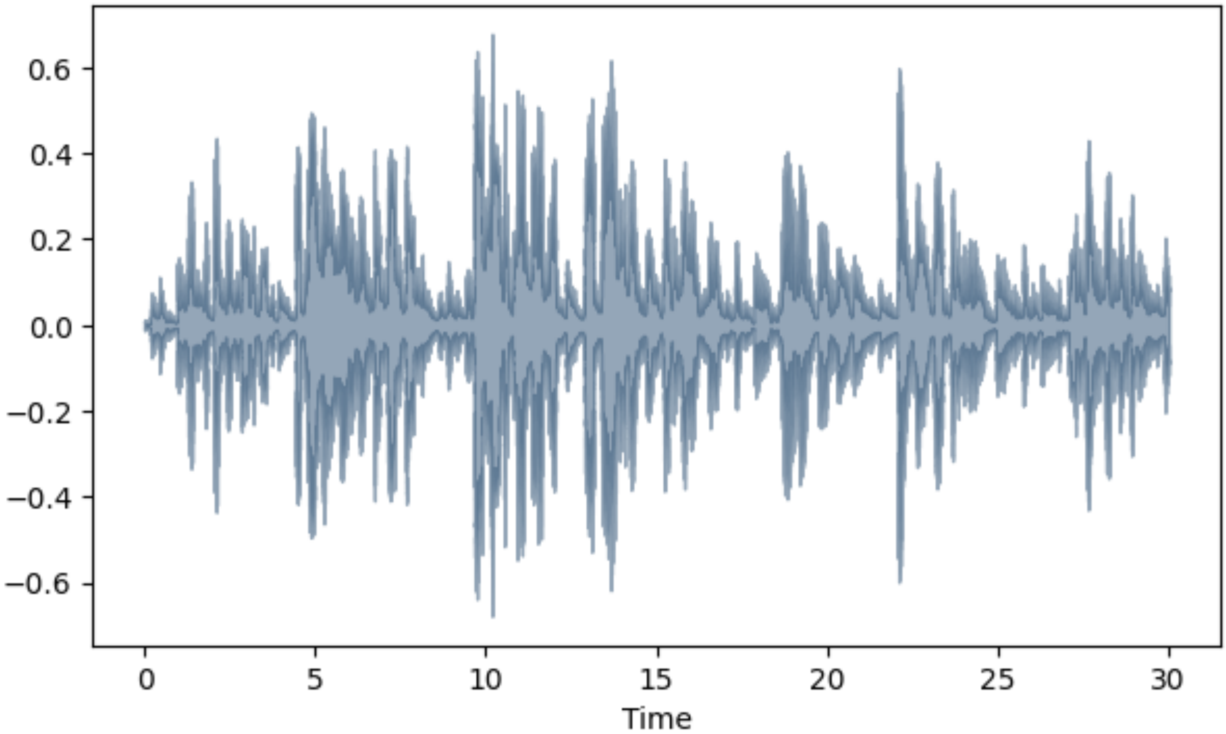
In [7]:
```
# Taking Short-time Fourier transform of the signal
y = librosa.stft(data)
S_db = librosa.amplitude_to_db(np.abs(y), ref=np.max)
```

In [8]:
```
# Playing audio file
import IPython
IPython.display.Audio(data,rate=sr)
```

Out[8]: 00:00

It is important to note that while working with any kind of audio data to solve any kind of problem statement, using only .wav format audio files is appropriate to analyze the data. If you are given audio files with .mp3 format you have to batch convert the data to waveforms using online software as .wav is the standard way of representing the audio files and it is the only way to work with audio data. Below is the wave form representation on the audio

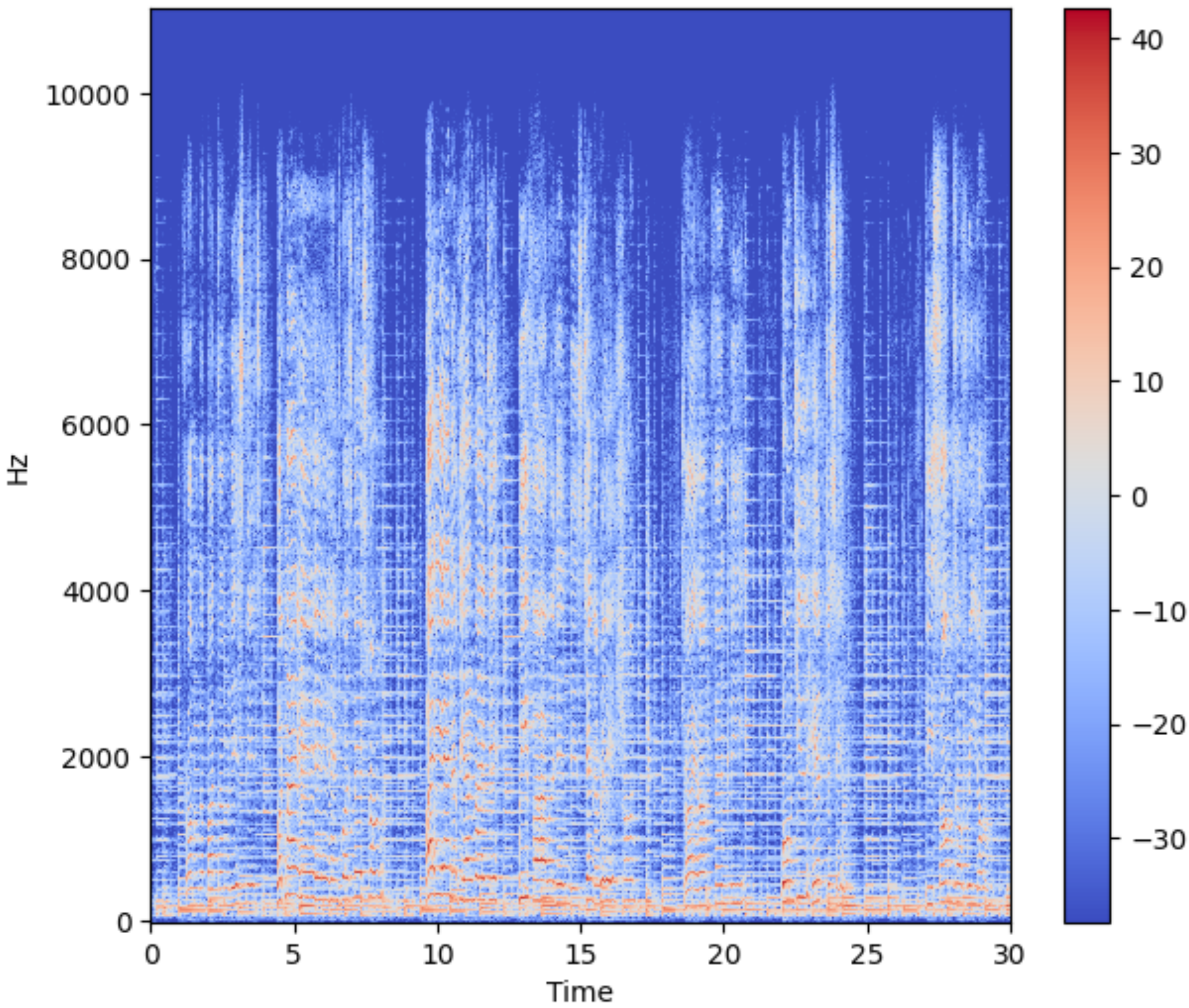
In [9]:
```
# Wave form of the audio
plt.figure(figsize=(7,4))
librosa.display.waveshow(data,color="#2B4F72", alpha = 0.5)
plt.show()
```



A spectrogram is a visual representation of the signal loudness of a signal over time at different frequencies included in a certain waveform. We can examine increase or decrease of energy over period of time. Spectrograms are also known as sonographs, voiceprints, and voicegrams. We can also know how energy levels change over time period.

In [10]:
```
# Spectrogram of the audio
stft=librosa.stft(data)
stft_db=librosa.amplitude_to_db(abs(stft))
plt.figure(figsize=(7,6))
librosa.display.specshow(stft_db,sr=sr,x_axis='time',y_axis='hz')
plt.colorbar()
```

Out[10]:
```
<matplotlib.colorbar.Colorbar at 0x13bc09610>
```



Data Pre Processing

Extracting Audio features

The process of extraction of features from the data to utilize them for analysis is known as feature extraction. Each audio signal consists of various audio features however we must extract features that are relevant to the problem that we are solving. Here are some features listed which are used in our project.

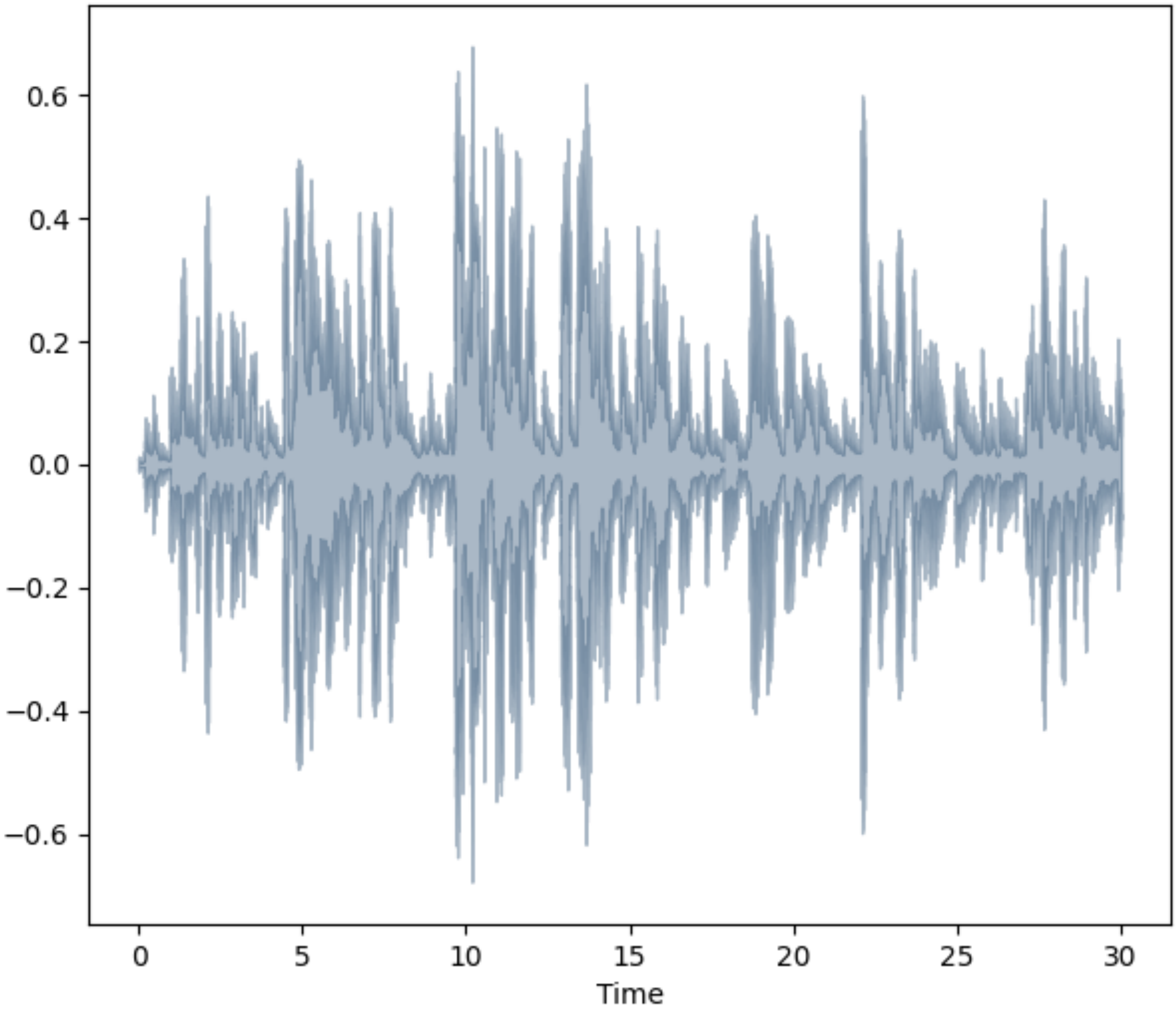
Spectral roll off

It computes the rolloff frequency for each frame in a given signal. The frequency under which some percentage (cutoff) of the total energy of a spectrum is obtained. It can be used to differentiate between the harmonic and noisy sounds. Spectral Roll off

In [11]:
```
spectral_rolloff=librosa.feature.spectral_rolloff(y=data,sr=sr)[0]
```

```
plt.figure(figsize=(7,6))
librosa.display.waveshow(data,sr=sr,alpha=0.4,color="#2B4F72")
```

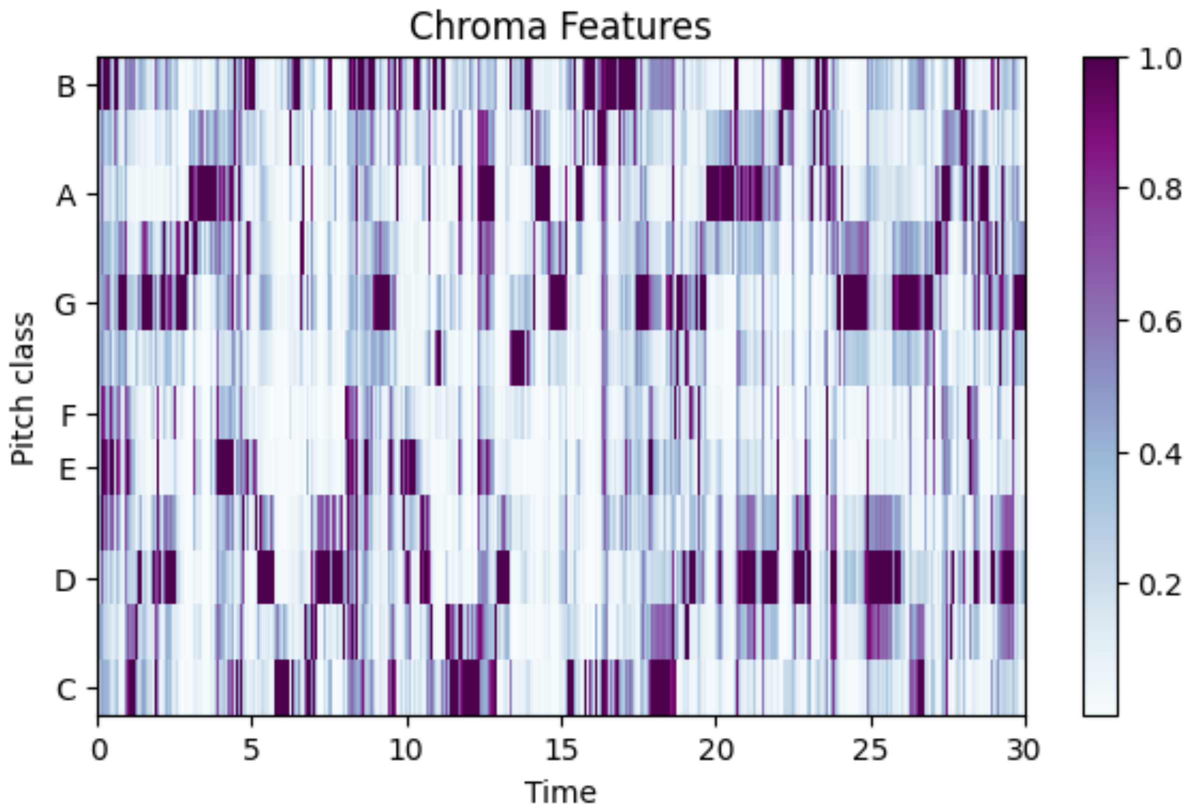
Out[11]: <librosa.display.AdaptiveWaveplot at 0x13bc4e810>



Chroma feature

It closely relates with the twelve different pitch classes. Chroma based features are also called as pitch class profiles. It is the powerful tool for analyzing and categorizing them. Harmonic and melodic characteristics of music are captured by them. Chroma featue

In [12]:
```
import librosa.display as lplt
chroma = librosa.feature.chroma_stft(y=data,sr=sr)
plt.figure(figsize=(7,4))
lplt.specshow(chroma,sr=sr,x_axis="time",y_axis="chroma",cmap="BuPu")
plt.colorbar()
plt.title("Chroma Features")
plt.show()
```

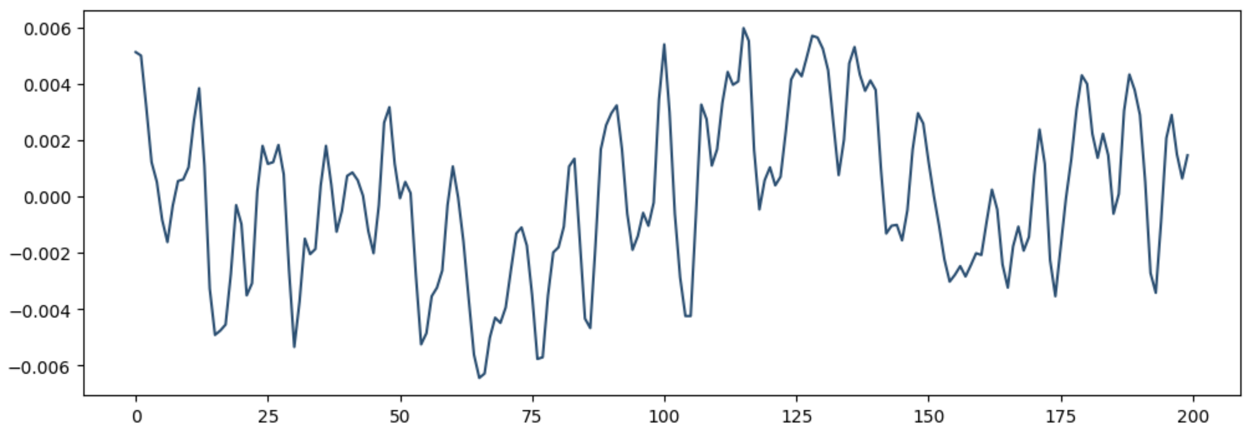


Zero Crossing Rate

It is the rate at which a signal transitions from positive to zero to negative or from negative to zero or simply said the number of times the signal crosses x-axis is as the zero-crossing rate (ZCR).

In [13]:
```
start=1000
end=1200
plt.figure(figsize=(12,4))
plt.plot(data[start:end],color="#2B4F72")
```

Out[13]: [<matplotlib.lines.Line2D at 0x13c029760>]



In [14]:
```
# Printing the number of times signal crosses the x-axis
zero_cross_rate=librosa.zero_crossings(data[start:end],pad=False)
```

```
print("The number of zero_crossings are :", sum(zero_cross_rate))
```

```
The number of zero_crossings are : 36
```

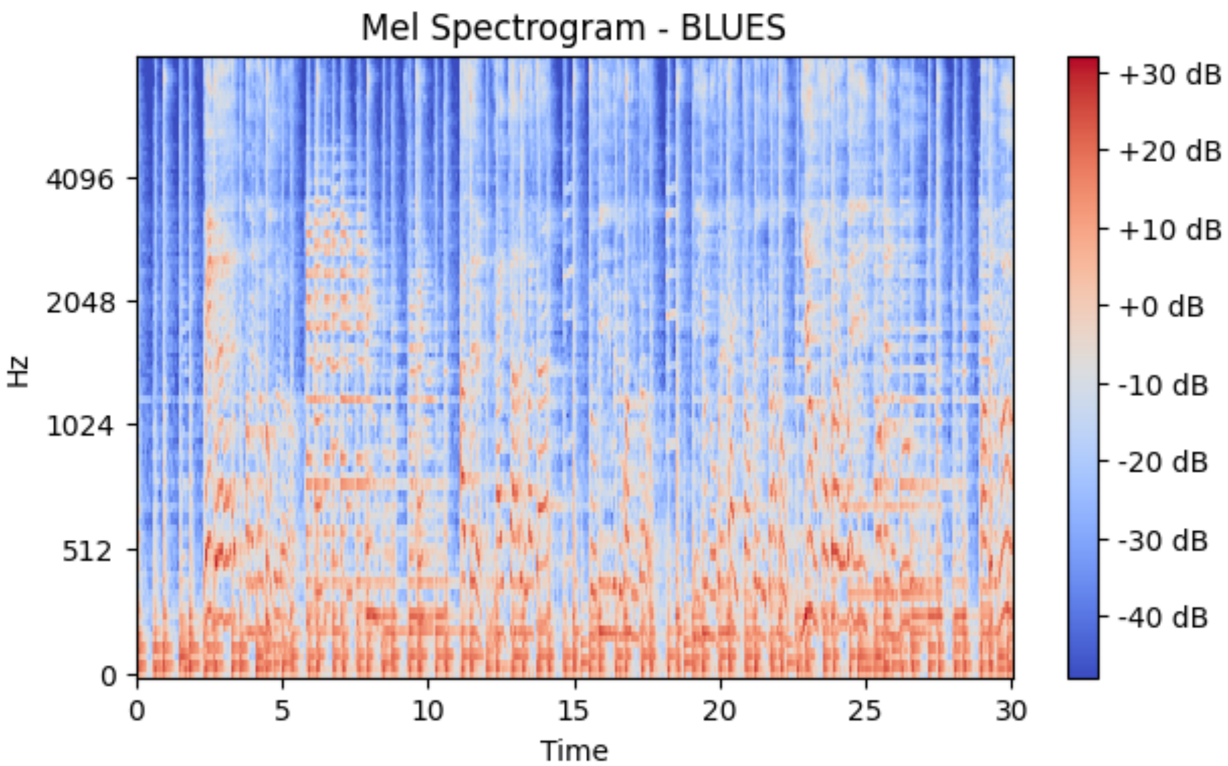
Exploratory Data Analysis(EDA)

Vizualizing the audio files, wave plots and spectrograms for all the 10 genre classes

In [15]:
```
# EDA for all the music genre classes

# 1. BLUES
audio1= 'Data/genres_original/blues/blues.00001.wav'
data, sr = librosa.load(audio1)
plt.figure(figsize=(7, 3))
#librosa.display.waveshow(data, sr=sr,alpha=0.4,)
#plt.title('Waveplot - BLUES')

# Creating log mel spectrogram
plt.figure(figsize=(7, 4))
spectrogram = librosa.feature.melspectrogram(y=data, sr=sr, n_mels=128,fm
spectrogram = librosa.power_to_db(spectrogram)
librosa.display.specshow(spectrogram, y_axis='mel', fmax=8000, x_axis='ti
plt.title('Mel Spectrogram - BLUES')
plt.colorbar(format='%+2.0f dB');
# playing audio
ipd.Audio(audio1)
```

Out[15]: 00:00

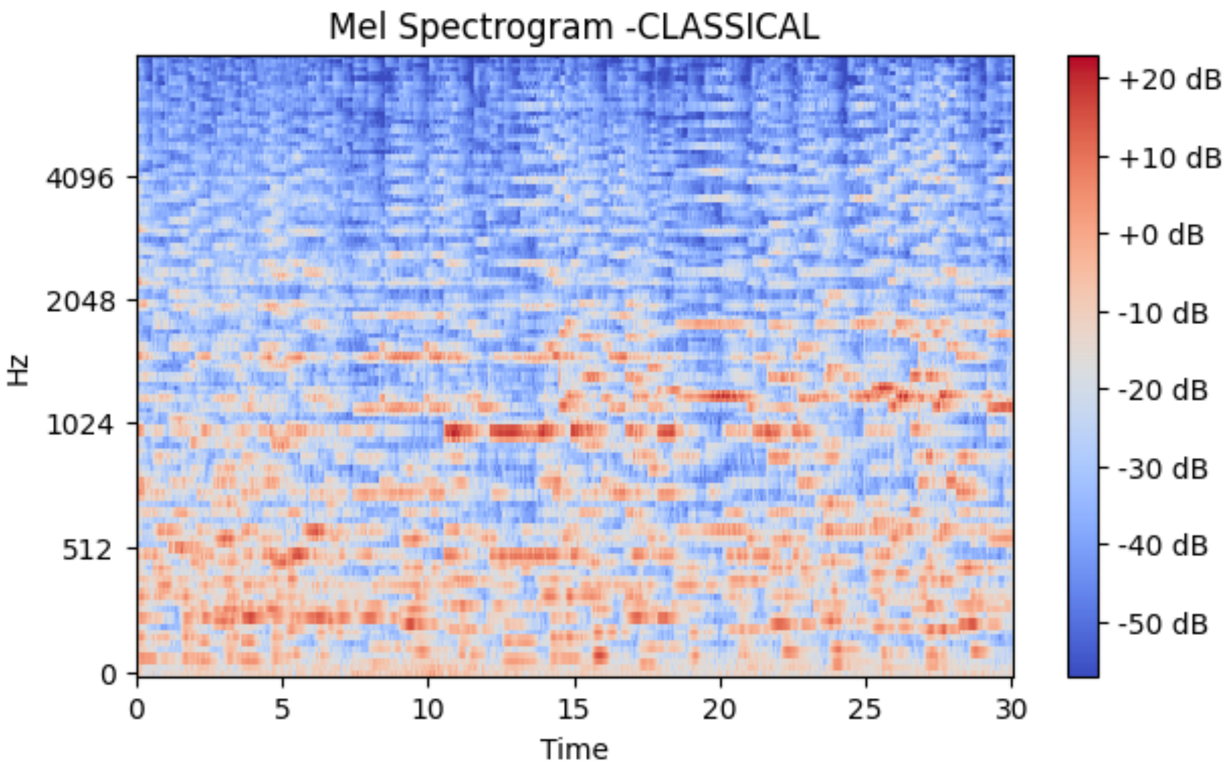
```
<Figure size 700x300 with 0 Axes>
```



In [16]:
```
# 2. CLASSICAL -
audio1= 'Data/genres_original/classical/classical.00001.wav'
data, sr = librosa.load(audio1)
plt.figure(figsize=(7, 3))
#librosa.display.waveshow(data, sr=sr,alpha=0.4)
#plt.title('Waveplot - CLASSICAL')

# Creating log mel spectrogram
plt.figure(figsize=(7, 4))
spectrogram = librosa.feature.melspectrogram(y=data, sr=sr, n_mels=128,fm
spectrogram = librosa.power_to_db(spectrogram)
librosa.display.specshow(spectrogram, y_axis='mel', fmax=8000, x_axis='ti
plt.title('Mel Spectrogram -CLASSICAL')
plt.colorbar(format='%+2.0f dB');

# playing audio
ipd.Audio(audio1)
```

Out[16]: 00:00

```
<Figure size 700x300 with 0 Axes>
```

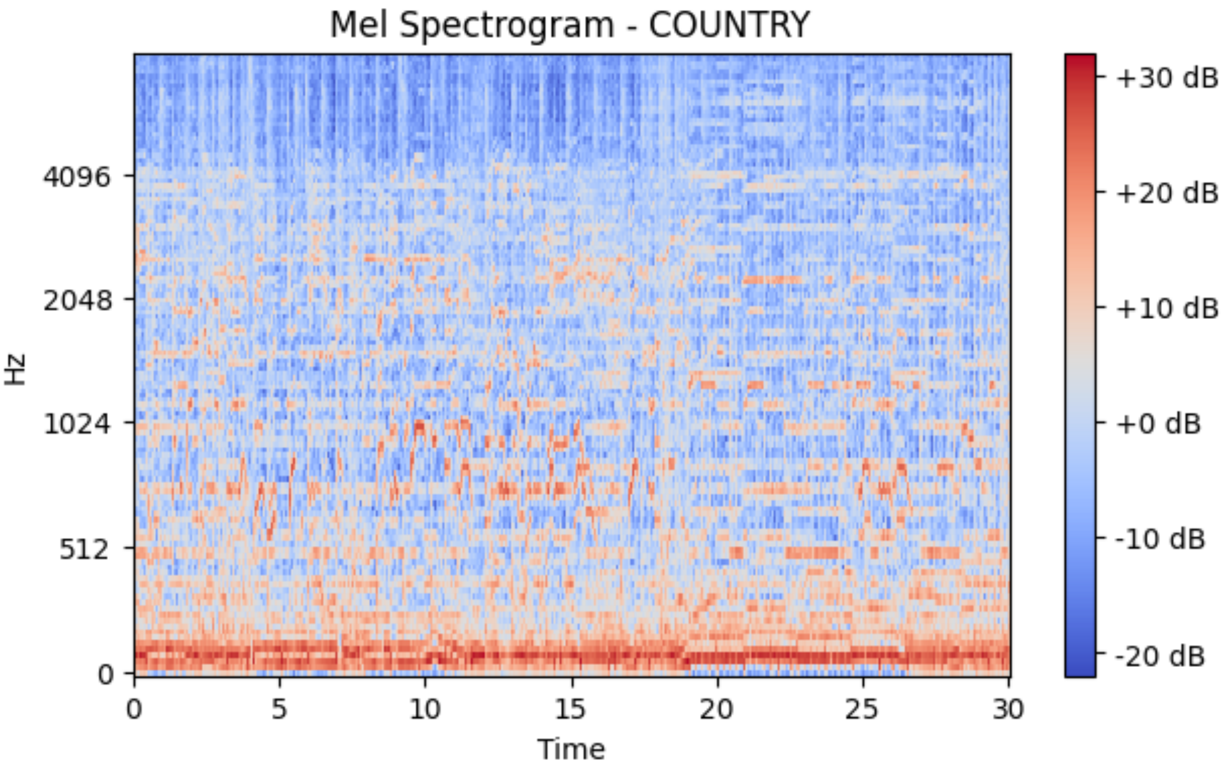


In [17]:
```
# 3. COUNTRY
audio1= 'Data/genres_original/country/country.00001.wav'
data, sr = librosa.load(audio1)
plt.figure(figsize=(7, 3))
#librosa.display.waveshow(data, sr=sr,alpha=0.4)
#plt.title('Waveplot - COUNTRY')

# Ccreating log mel spectrogram
plt.figure(figsize=(7, 4))
```

```
spectrogram = librosa.feature.melspectrogram(y=data, sr=sr, n_mels=128,fm
spectrogram = librosa.power_to_db(spectrogram)
librosa.display.specshow(spectrogram, y_axis='mel', fmax=8000, x_axis='ti
plt.title('Mel Spectrogram - COUNTRY')
plt.colorbar(format='%+2.0f dB');

# playing audio
ipd.Audio(audio1)
```

Out[17]: 00:00

<Figure size 700x300 with 0 Axes>

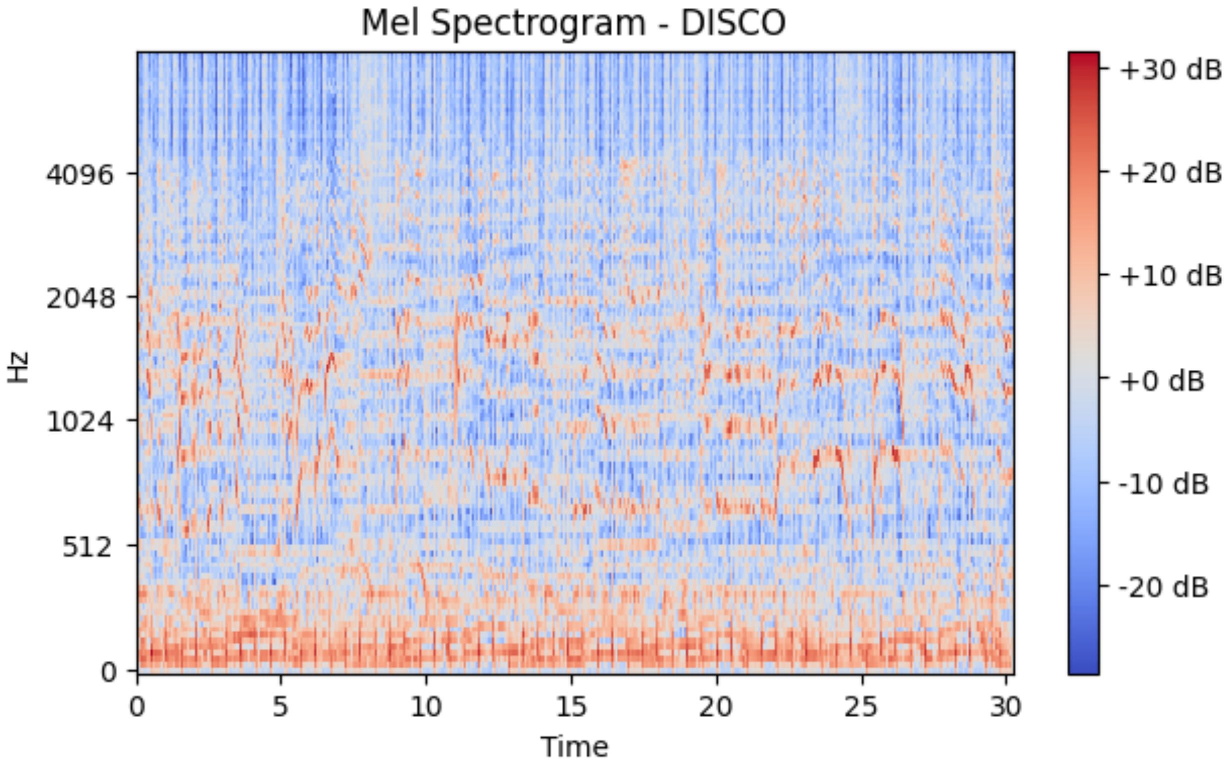


In [18]:

```
# 4. DISCO
audio1= 'Data/genres_original/disco/disco.00001.wav'
data, sr = librosa.load(audio1)
plt.figure(figsize=(7, 3))
#librosa.display.waveshow(data, sr=sr,alpha=0.4)
#plt.title('Waveplot - DISCO')

# Creating log mel spectrogram
plt.figure(figsize=(7, 4))
spectrogram = librosa.feature.melspectrogram(y=data, sr=sr, n_mels=128,fm
spectrogram = librosa.power_to_db(spectrogram)
librosa.display.specshow(spectrogram, y_axis='mel', fmax=8000, x_axis='ti
plt.title('Mel Spectrogram - DISCO')
plt.colorbar(format='%+2.0f dB');
# playing audio
ipd.Audio(audio1)
```

Out[18]: 00:00

<Figure size 700x300 with 0 Axes>

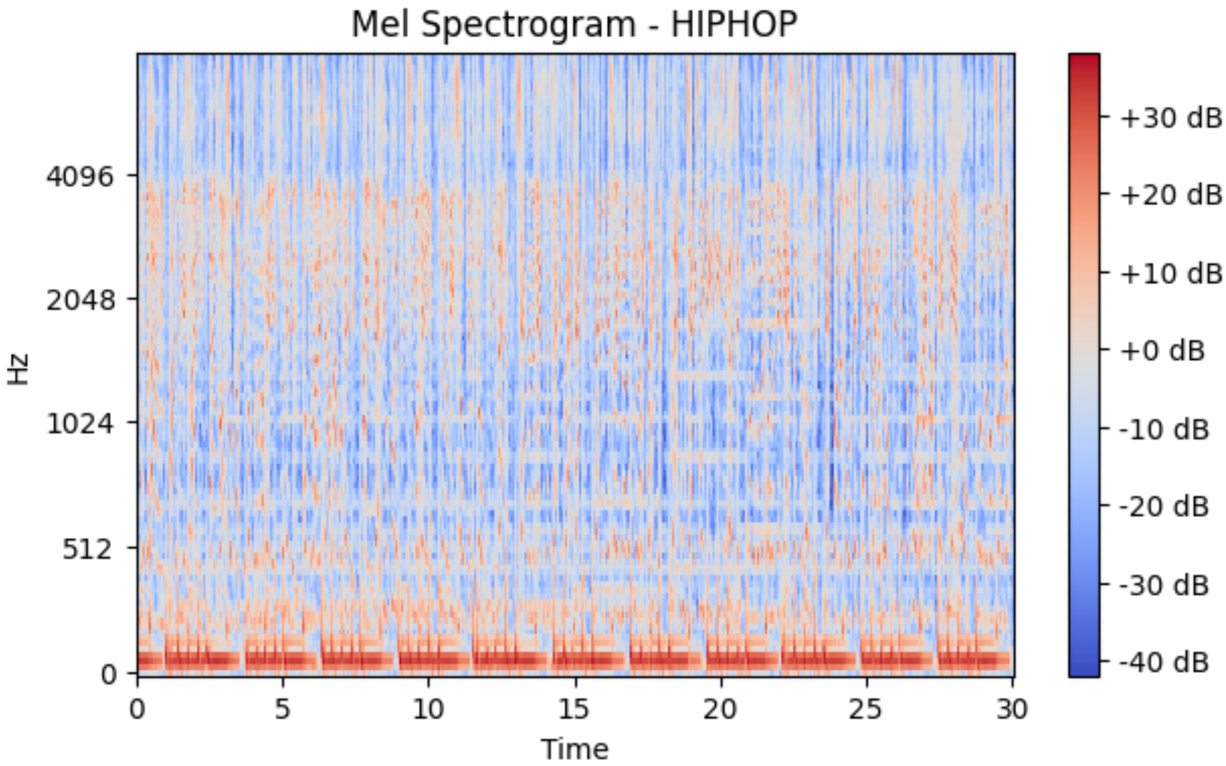


In [19]:

```
# 5. HIPHOP
audio1= 'Data/genres_original/hiphop/hiphop.00001.wav'
data, sr = librosa.load(audio1)
plt.figure(figsize=(7, 3))
#$librosa.display.waveshow(data, sr=sr, alpha = 0.4)
#plt.title('Waveplot - HIPHOP')

# Creating log mel spectrogram
plt.figure(figsize=(7, 4))
spectrogram = librosa.feature.melspectrogram(y=data, sr=sr, n_mels=128,fm
spectrogram = librosa.power_to_db(spectrogram)
librosa.display.specshow(spectrogram, y_axis='mel', fmax=8000, x_axis='ti
plt.title('Mel Spectrogram - HIPHOP')
plt.colorbar(format='%+2.0f dB');

# playing audio
ipd.Audio(audio1)
```

Out[19]: 00:00

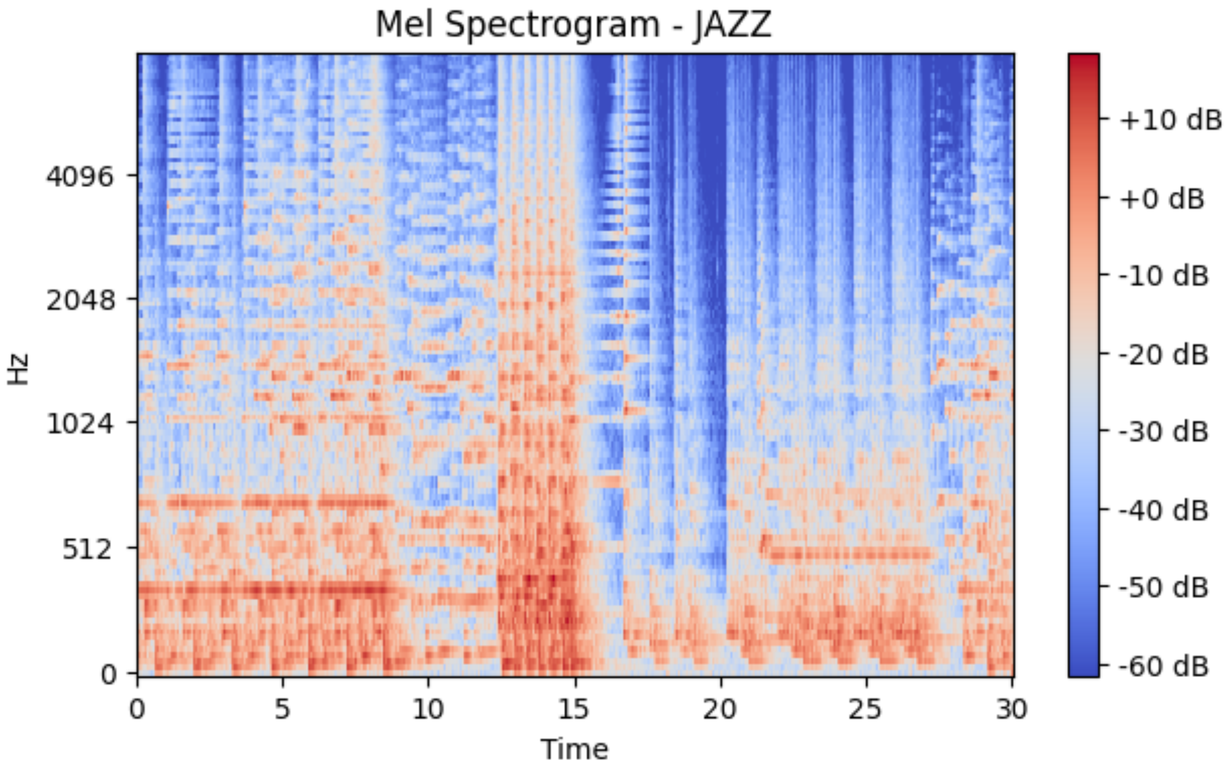
<Figure size 700x300 with 0 Axes>



In [20]:
```
# 6. JAZZ
audio1= 'Data/genres_original/jazz/jazz.00001.wav'
data, sr = librosa.load(audio1)
plt.figure(figsize=(7, 3))
#librosa.display.waveshow(data, sr=sr,alpha=0.4)
#plt.title('Waveplot - JAZZ')

# Creating log mel spectrogram
plt.figure(figsize=(7, 4))
spectrogram = librosa.feature.melspectrogram(y=data, sr=sr, n_mels=128,fm
spectrogram = librosa.power_to_db(spectrogram)
librosa.display.specshow(spectrogram, y_axis='mel', fmax=8000, x_axis='ti
plt.title('Mel Spectrogram - JAZZ')
plt.colorbar(format='%+2.0f dB');

# playing audio
ipd.Audio(audio1)
```

Out[20]: 00:00

```
<Figure size 700x300 with 0 Axes>
```

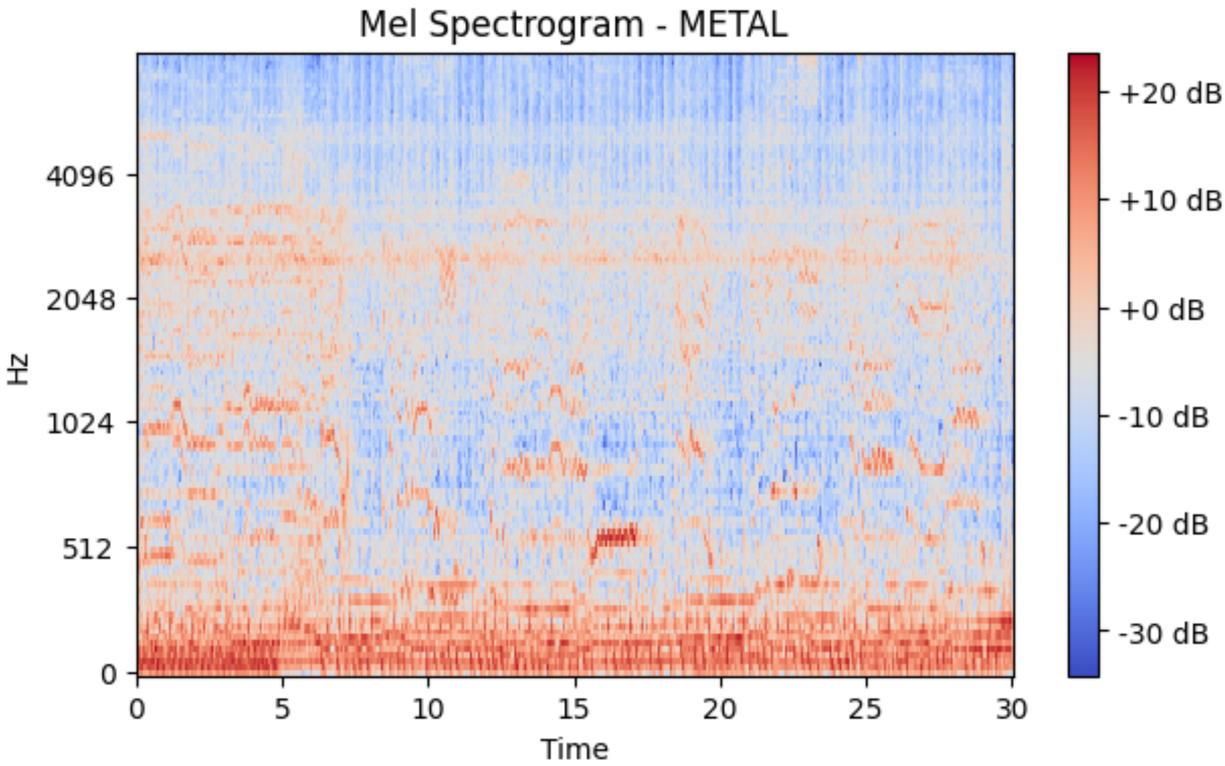


In [21]:

```
# 7. METAL
audio1= 'Data/genres_original/metal/metal.00001.wav'
data, sr = librosa.load(audio1)
plt.figure(figsize=(7, 3))
#librosa.display.waveshow(data, sr=sr,alpha=0.4)
#plt.title('Waveplot - METAL')

# creating log mel spectrogram
plt.figure(figsize=(7, 4))
spectrogram = librosa.feature.melspectrogram(y=data, sr=sr, n_mels=128,fm
spectrogram = librosa.power_to_db(spectrogram)
librosa.display.specshow(spectrogram, y_axis='mel', fmax=8000, x_axis='ti
plt.title('Mel Spectrogram - METAL')
plt.colorbar(format='%+2.0f dB');

# playing audio
ipd.Audio(audio1)
```

Out[21]: 00:00

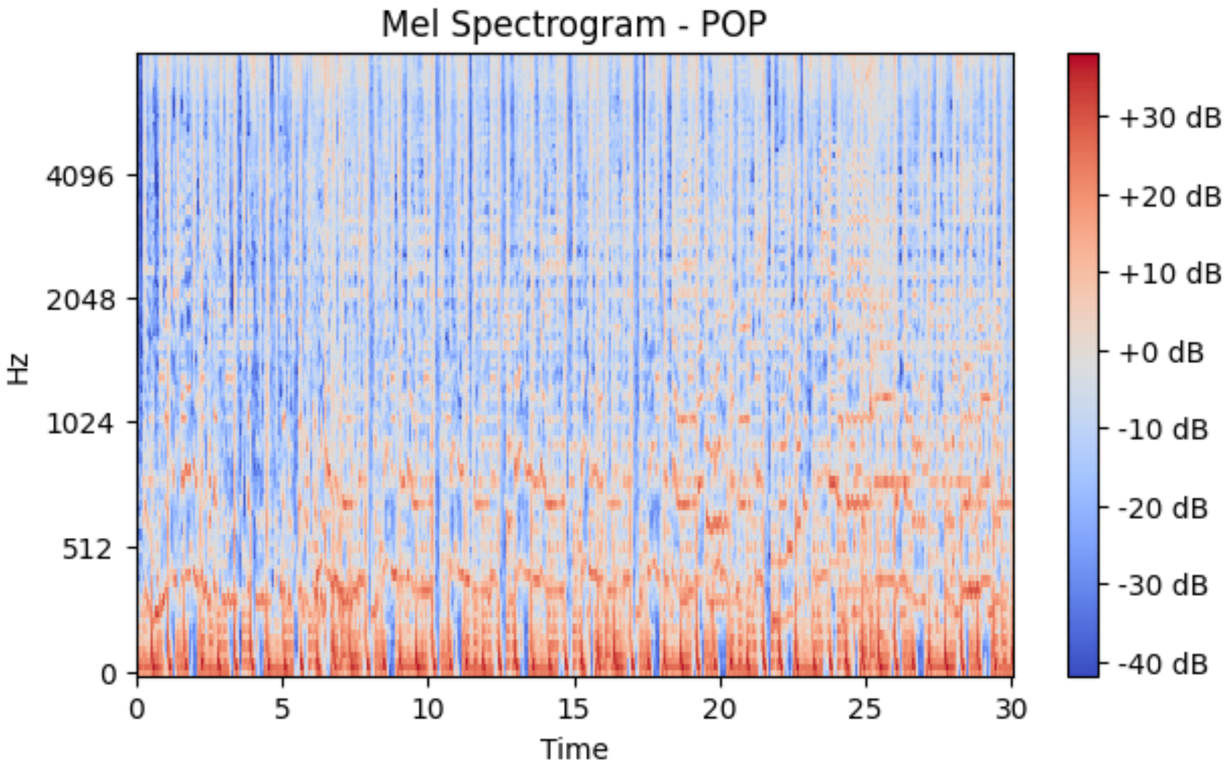
```
<Figure size 700x300 with 0 Axes>
```



In [22]:
```
# 8. POP
audio1= 'Data/genres_original/pop/pop.00001.wav'
data, sr = librosa.load(audio1)
plt.figure(figsize=(8, 3))
#librosa.display.waveshow(data, sr=sr,alpha=0.4)
#plt.title('Waveplot - POP')

# Creating log mel spectrogram
plt.figure(figsize=(7, 4))
spectrogram = librosa.feature.melspectrogram(y=data, sr=sr, n_mels=128,fm
spectrogram = librosa.power_to_db(spectrogram)
librosa.display.specshow(spectrogram, y_axis='mel', fmax=8000, x_axis='ti
plt.title('Mel Spectrogram - POP')
plt.colorbar(format='%+2.0f dB');

# playing audio
ipd.Audio(audio1)
```

Out[22]: 00:00

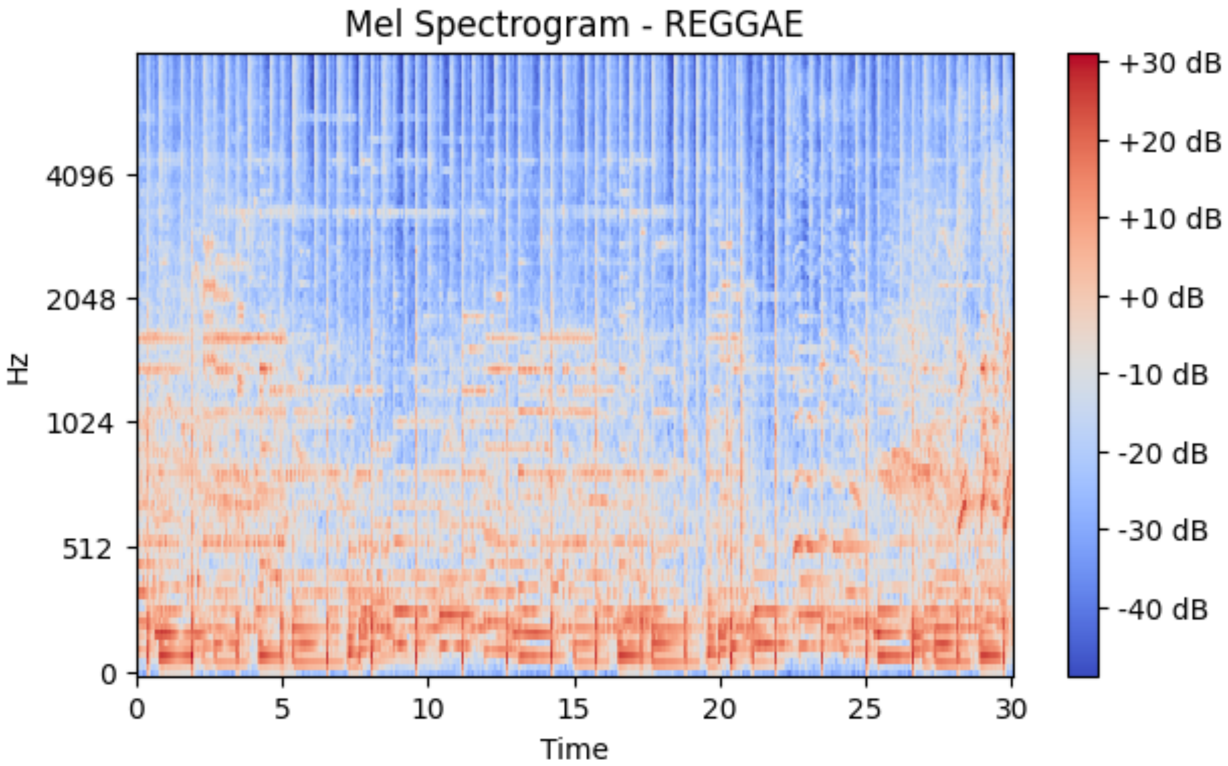
```
<Figure size 800x300 with 0 Axes>
```



In [23]:
```
# 9. REGGAE
audio1= 'Data/genres_original/reggae/reggae.00001.wav'
data, sr = librosa.load(audio1)
plt.figure(figsize=(7, 3))
#librosa.display.waveshow(data, sr=sr,alpha=0.4)
#plt.title('Waveplot - REGGAE')

# Creating log mel spectrogram
plt.figure(figsize=(7, 4))
spectrogram = librosa.feature.melspectrogram(y=data, sr=sr, n_mels=128,fm
spectrogram = librosa.power_to_db(spectrogram)
librosa.display.specshow(spectrogram, y_axis='mel', fmax=8000, x_axis='ti
plt.title('Mel Spectrogram - REGGAE')
plt.colorbar(format='%+2.0f dB');

# playing audio
ipd.Audio(audio1)
```

Out[23]: 00:00

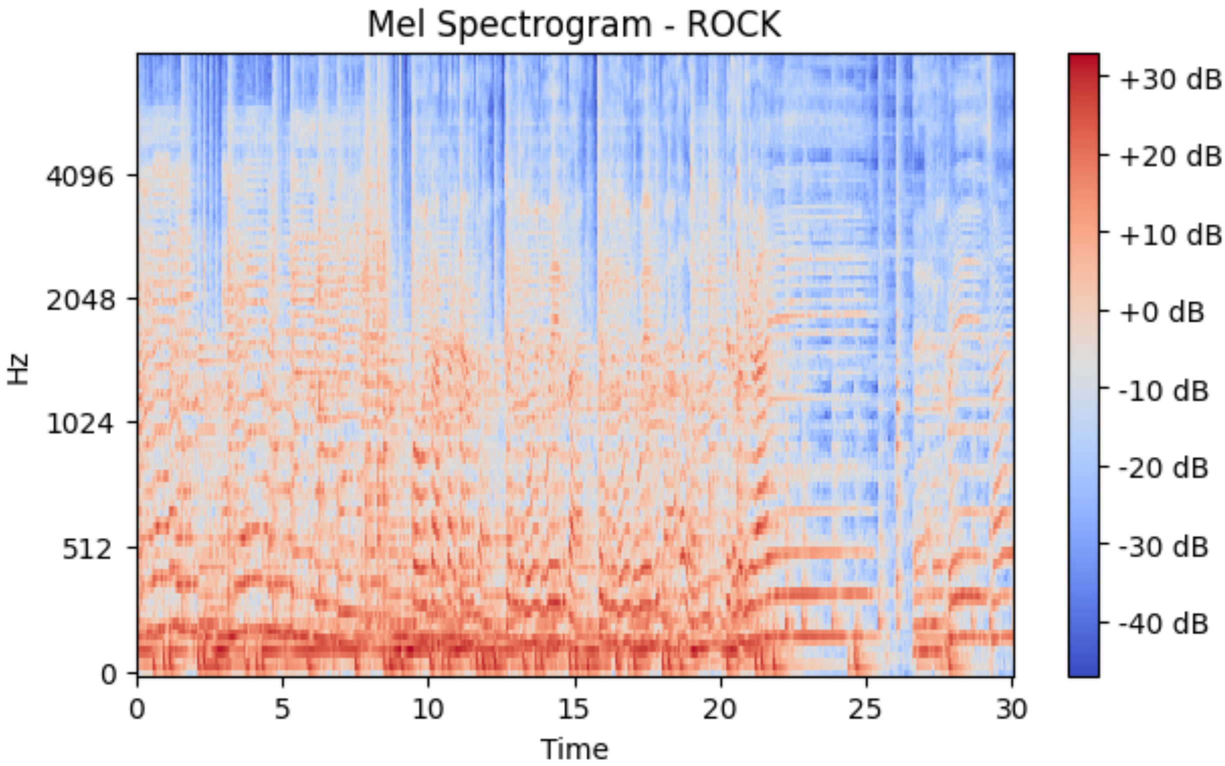
<Figure size 700x300 with 0 Axes>



In [24]:
```
# 10.ROCK
audio1= 'Data/genres_original/rock/rock.00001.wav'
data, sr = librosa.load(audio1)
plt.figure(figsize=(7, 3))
#librosa.display.waveshow(data, sr=sr,alpha=0.4)
#plt.title('Waveplot - ROCK')

# Creating log mel spectrogram
plt.figure(figsize=(7, 4))
spectrogram = librosa.feature.melspectrogram(y=data, sr=sr, n_mels=128,fm
spectrogram = librosa.power_to_db(spectrogram)
librosa.display.specshow(spectrogram, y_axis='mel', fmax=8000, x_axis='ti
plt.title('Mel Spectrogram - ROCK')
plt.colorbar(format='%+2.0f dB');

# Playing audio
ipd.Audio(audio1)
```

Out[24]: 00:00

```
<Figure size 700x300 with 0 Axes>
```



In [25]:
```
# Finding misssing values
# Find all columns with any NA values
print("Columns containing missing values",list(df.columns[df.isnull().any
```

```
Columns containing missing values []
```

In [26]:
```
# Label Encoding - encod the categorical classes with numerical integer v

# Blues - 0
# Classical - 1
# Country - 2
# Disco - 3
# Hip-hop - 4
# Jazz - 5
# Metal - 6
# Pop - 7
# Reggae - 8
# Rock - 9

class_encod=df.iloc[:,-1]
converter=LabelEncoder()
y=converter.fit_transform(class_encod)
y
```

Out[26]:
```
array([0, 0, 0, ..., 9, 9, 9])
```

In [27]:
```
#features
print(df.iloc[:,:-1])
```

```
                    filename   length  chroma_stft_mean  chroma_stft_var   rms_me
an  \
```

```
0     blues.00000.0.wav   66149          0.335406         0.091048  0.1304
05
1     blues.00000.1.wav   66149          0.343065         0.086147  0.1126
99
2     blues.00000.2.wav   66149          0.346815         0.092243  0.1320
03
3     blues.00000.3.wav   66149          0.363639         0.086856  0.1325
65
4     blues.00000.4.wav   66149          0.335579         0.088129  0.1432
89
...                 ...     ...               ...              ...
...
9985   rock.00099.5.wav   66149          0.349126         0.080515  0.0500
19
9986   rock.00099.6.wav   66149          0.372564         0.082626  0.0578
97
9987   rock.00099.7.wav   66149          0.347481         0.089019  0.0524
03
9988   rock.00099.8.wav   66149          0.387527         0.084815  0.0664
30
9989   rock.00099.9.wav   66149          0.369293         0.086759  0.0505
24

       rms_var  spectral_centroid_mean  spectral_centroid_var  \
0     0.003521             1773.065032          167541.630869
1     0.001450             1816.693777           90525.690866
2     0.004620             1788.539719          111407.437613
3     0.002448             1655.289045          111952.284517
4     0.001701             1630.656199           79667.267654
...        ...                     ...                    ...
9985  0.000097             1499.083005          164266.886443
9986  0.000088             1847.965128          281054.935973
9987  0.000701             1346.157659          662956.246325
9988  0.000320             2084.515327          203891.039161
9989  0.000067             1634.330126          411429.169769

      spectral_bandwidth_mean  spectral_bandwidth_var  ...  mfcc16_mean  \
0                 1972.744388           117335.771563  ...    -2.853603
1                 2010.051501            65671.875673  ...     4.074709
2                 2084.565132            75124.921716  ...     4.806280
3                 1960.039988            82913.639269  ...    -1.359111
4                 1948.503884            60204.020268  ...     2.092937
...                       ...                     ...  ...          ...
9985              1718.707215            85931.574523  ...     5.773784
9986              1906.468492            99727.037054  ...     2.074155
9987              1561.859087           138762.841945  ...    -1.005473
9988              2018.366254            22860.992562  ...     4.123402
9989              1867.422378           119722.211518  ...     1.342274

      mfcc16_var  mfcc17_mean  mfcc17_var  mfcc18_mean  mfcc18_var  \
0      39.687145    -3.241280   36.488243     0.722209   38.099152
1      64.748276    -6.055294   40.677654     0.159015   51.264091
2      67.336563    -1.768610   28.348579     2.378768   45.717648
```

```
3       47.739452    -3.841155   28.337118    1.218588   34.770935
4       30.336359     0.664582   45.880913    1.689446   51.363583
...           ...          ...         ...         ...         ...
9985    42.485981    -9.094270   38.326839   -4.246976   31.049839
9986    32.415203   -12.375726   66.418587   -3.081278   54.414265
9987    78.228149    -2.524483   21.778994    4.809936   25.980829
9988    28.323744    -5.363541   17.209942    6.462601   21.442928
9989    38.801735   -11.598399   58.983097   -0.178517   55.761299

      mfcc19_mean  mfcc19_var  mfcc20_mean  mfcc20_var
0       -5.050335   33.618073    -0.243027   43.771767
1       -2.837699   97.030830     5.784063   59.943081
2       -1.938424   53.050835     2.517375   33.105122
3       -3.580352   50.836224     3.630866   32.023678
4       -3.392489   26.738789     0.536961   29.146694
...           ...         ...          ...         ...
9985    -5.625813   48.804092     1.818823   38.966969
9986   -11.960546   63.452255     0.428857   18.697033
9987     1.775686   48.582378    -0.299545   41.586990
9988     2.354765   24.843613     0.675824   12.787750
9989    -6.903252   39.485901    -3.412534   31.727489

[9990 rows x 59 columns]
```

In [28]:
```python
# Drop the column filename as it is no longer required for training
df=df.drop(labels="filename",axis=1)
```

In [29]:
```python
#scaling
from sklearn.preprocessing import StandardScaler
fit=StandardScaler()
X=fit.fit_transform(np.array(df.iloc[:,:-1],dtype=float))
X.shape
```

Out[29]:
```
(9990, 58)
```

In [30]:
```python
# splitting 70% data into training set and the remaining 30% to test set
#X_train,X_test,y_train,y_test
X_train, X_test, y_train, y_test = train_test_split(X,y,test_size=0.3)
```

In [31]:
```python
print(X_train[0], X_test[0], y_train[0], y_test[0])
```

```
[ 0.          0.17454748 -0.16786045  1.26092319 -0.29961028  0.82415139
 -0.6242867   0.58360511 -0.9494246   0.65381434 -0.75781388  1.34351573
 -0.30800052  0.14945459  1.03444194 -1.05709535  0.38012461  2.74822978
  1.21257523 -0.78602996 -0.96962748 -0.74783852 -0.11808877 -0.93551374
 -0.05414628 -0.84477981  0.88837226 -0.65831567 -0.61309431 -0.95213151
  1.00246053 -0.4524252   0.09700885 -0.64781563  0.68053705 -0.8137713
  0.60153521 -0.67793108  0.77671635  0.1857818  -0.69417786  0.3125858
  1.1061198   0.36456843  0.37271764 -0.62455541  0.07564346 -0.54787233
 -0.03863196 -0.4301779  -0.44068051 -0.35699287 -0.71560312  0.59458176
 -0.16351972  0.35339598 -0.6093537   1.29029887] [ 0.         -1.77448677
-0.52543134 -1.37805968 -0.71880322 -1.40065413
 -0.84896357 -1.91070523 -0.82149533 -1.59899381 -0.9673883  -0.76541305
 -0.640455    0.2100789  -0.89501302  0.36079715 -0.83896781 -0.38105596
 -1.69831324 -0.55492136  1.8260873   0.12984088 -1.48276218 -0.554724
 -1.70856967 -0.88929862 -1.01951772 -0.8556794  -2.41184055 -0.77723765
 -1.17476749 -1.07071606 -3.09901232  0.34396813 -2.04484578  0.78010671
 -2.84325768 -0.29401836 -0.01538973 -0.030968   -2.0889579   1.78013118
  0.34102936  1.29585669 -0.67557162  0.16413691  0.22483146  0.97340266
  0.10717857  0.87716777  1.41172981 -0.02921626  0.70961553  0.15502093
  1.09975177  0.09621615  1.88720554 -0.58640717] 3 1
```

In [32]:

```
# test data size
len(y_test)
X_train.shape
X_test.shape
```

Out[32]:

```
(2997, 58)
```

In [33]:

```
# size of training data
len(y_train)
```

Out[33]:

```
6993
```

K-Nearest Neighbors (KNN)

KNN is a fundamental Machine learning algorithm that is most commonly used among all kinds of problems. It classifies the data points based on the point that is near them by finding the euclidians distance given by d = ((x2-x1)^2 - (y2-y1)^2)^1/2 as a metric.

In [34]:

```
# Applying K nearest Neighbour algorithm to predict the results
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import classification_report, confusion_matrix
import matplotlib.pyplot as plt
import seaborn as sns

clf1=KNeighborsClassifier(n_neighbors=3)
clf1.fit(X_train,y_train)
y_pred=clf1.predict(X_test)
print("Training set score: {:.3f}".format(clf1.score(X_train, y_train)))
```

```
print("Test set score: {:.3f}".format(clf1.score(X_test, y_test)))
cf_matrix = confusion_matrix(y_test, y_pred)
sns.set(rc = {'figure.figsize':(8,3)})
sns.heatmap(cf_matrix, annot=True)
print(classification_report(y_test,y_pred))
```

```
Training set score: 0.952
Test set score: 0.887
              precision    recall  f1-score   support

           0       0.85      0.93      0.89       308
           1       0.89      0.96      0.93       303
           2       0.77      0.82      0.80       306
           3       0.86      0.93      0.89       308
           4       0.93      0.88      0.91       311
           5       0.92      0.82      0.87       301
           6       0.98      0.95      0.96       295
           7       0.94      0.85      0.89       304
           8       0.88      0.94      0.90       284
           9       0.89      0.79      0.84       277

    accuracy                           0.89      2997
   macro avg       0.89      0.89      0.89      2997
weighted avg       0.89      0.89      0.89      2997
```

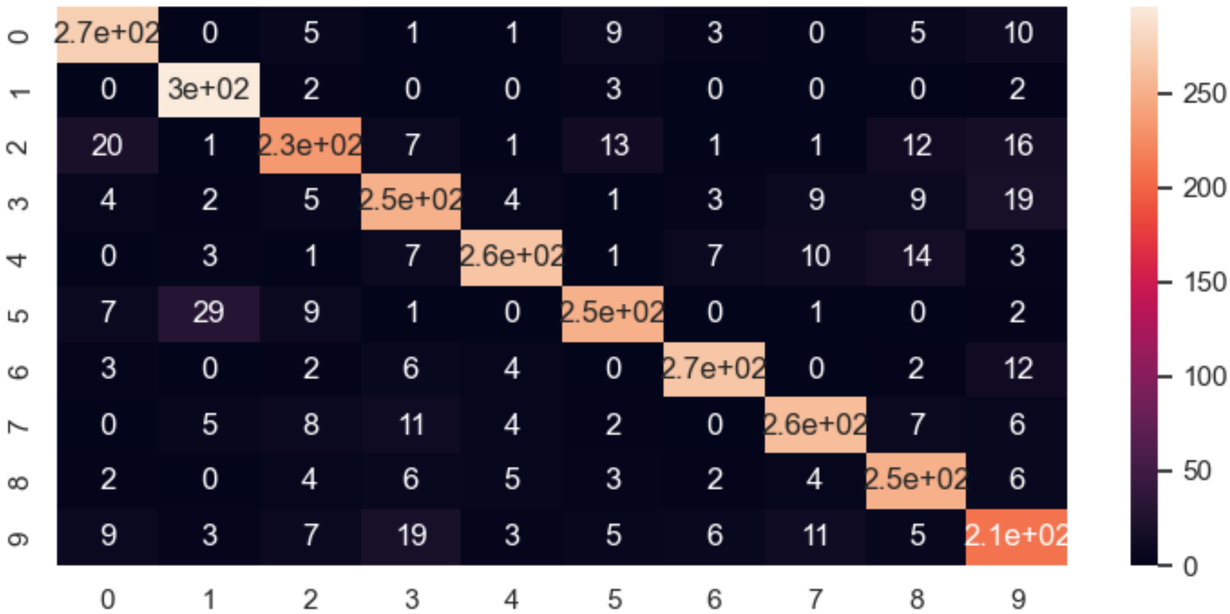
Support Vector Machine (SVM)

SVM is one of the best machine learning models. Since the data is not linearly separable, we have used the SVM kernel function as sigmoid. The sigmoid function is given by K(yn,yi) = tanh(-gamma*(yn,yi)+r)

In [35]:
```
# Applying Support Vector Machines to predict the results
from sklearn.svm import SVC
svclassifier = SVC(kernel='rbf', degree=8)
svclassifier.fit(X_train, y_train)
```

```
print("Training set score: {:.3f}".format(svclassifier.score(X_train, y_t
print("Test set score: {:.3f}".format(svclassifier.score(X_test, y_test))
y_pred = svclassifier.predict(X_test)
cf_matrix3 = confusion_matrix(y_test, y_pred)
sns.set(rc = {'figure.figsize':(9,4)})
sns.heatmap(cf_matrix3, annot=True)
print(classification_report(y_test, y_pred))
```

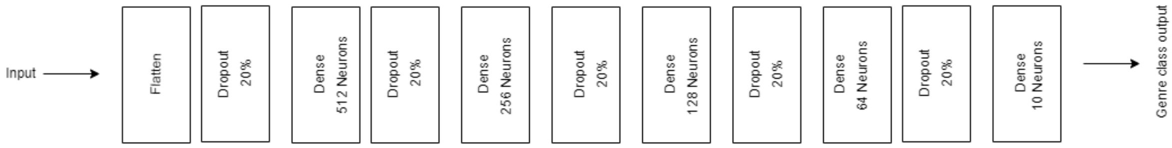
```
Training set score: 0.917
Test set score: 0.855
              precision    recall  f1-score   support

           0       0.86      0.89      0.87       308
           1       0.87      0.98      0.92       303
           2       0.84      0.76      0.80       306
           3       0.81      0.82      0.82       308
           4       0.92      0.85      0.89       311
           5       0.87      0.84      0.85       301
           6       0.92      0.90      0.91       295
           7       0.88      0.86      0.87       304
           8       0.82      0.89      0.85       284
           9       0.73      0.75      0.74       277

    accuracy                           0.85      2997
   macro avg       0.85      0.85      0.85      2997
weighted avg       0.86      0.85      0.85      2997
```



Convolutional Neural Networks (CNN)

Using neural networks is the best way to classify huge data to draw predictions. Convolutions can solve the given problem very precisely and the algorithm has already been used most widely in classifying the image data.

Model Architecture



In [36]:
```
# Training the model using the following parameters
# metrics = accuracy
# epochs = 600
# loss = sparse_categorical_crossentropy
# batch_size = 256
# optimizer = adam

def train_model(model, epochs, optimizer, X_train, y_train, X_test, y_tes
    batch_size=32
    model.compile(optimizer=optimizer,loss='categorical_crossentropy',met
    return model.fit(X_train, y_train, validation_data=(X_test, y_test),
```

In [37]:
```
def Validation_plot(history):
    print("Validation Accuracy",max(history.history["val_accuracy"]))
    pd.DataFrame(history.history).plot(figsize=(12,6))
    plt.show()
```

Keras is the high-level API of TensorFlow 2: an approachable, highly-productive interface for solving machine learning problems, with a focus on modern deep learning. It provides essential abstractions and building blocks for developing and shipping machine learning solutions with high iteration velocity.

In [38]:
```
X.shape[1]
```

Out[38]: 58

In [39]:
```
from keras.models import Sequential, Model, load_model
from keras.layers import Input, Dense, Dropout, Flatten, LSTM
from keras.layers import Conv2D, MaxPooling2D, Conv1D, MaxPooling1D, MaxP
from tensorflow.keras.applications import EfficientNetB0
# Reshape input data for Conv2D layer
X_train = X_train.reshape(X_train.shape[0], X_train.shape[1], 1)  # Resha
X_test = X_test.reshape(X_test.shape[0], X_test.shape[1], 1)
```

In [40]:
```
# We used different layers to train the neural network by importing keras
# for input and hidden neurons we use the most widly used activation func
model=tf.keras.models.Sequential([
    tf.keras.layers.Flatten(input_shape=(X.shape[1],)),
    tf.keras.layers.Dropout(0.2),

    tf.keras.layers.Dense(512,activation='relu'),
    keras.layers.Dropout(0.2),

    tf.keras.layers.Dense(256,activation='relu'),
```

```
    tf.keras.layers.Dropout(0.2),

    tf.keras.layers.Dense(128,activation='relu'),
    tf.keras.layers.Dropout(0.2),

    tf.keras.layers.Dense(64,activation='relu'),
    tf.keras.layers.Dropout(0.2),

    tf.keras.layers.Dense(32,activation='relu'),
    tf.keras.layers.Dropout(0.2),

    tf.keras.layers.Dense(10,activation='softmax'),
])

optimizer = tf.keras.optimizers.Adam(learning_rate=0.000146)
model.compile(optimizer=optimizer,
             loss="sparse_categorical_crossentropy",
              metrics=["accuracy"])
model.summary()
#model_history=train_model(model=model,epochs=600,optimizer='adam')
```

```
/Library/Frameworks/Python.framework/Versions/3.12/lib/python3.12/site-packages/keras/src/layers/reshaping/flatten.py:37: UserWarning: Do not pass an `input_shape`/`input_dim` argument to a layer. When using Sequential models, prefer using an `Input(shape)` object as the first layer in the model instead.
  super().__init__(**kwargs)
```

Model: "sequential"

Layer (type)	Output Shape	
flatten (Flatten)	(None, 58)	
dropout (Dropout)	(None, 58)	
dense (Dense)	(None, 512)	
dropout_1 (Dropout)	(None, 512)	
dense_1 (Dense)	(None, 256)	
dropout_2 (Dropout)	(None, 256)	
dense_2 (Dense)	(None, 128)	
dropout_3 (Dropout)	(None, 128)	
dense_3 (Dense)	(None, 64)	
dropout_4 (Dropout)	(None, 64)	
dense_4 (Dense)	(None, 32)	
dropout_5 (Dropout)	(None, 32)	
dense_5 (Dense)	(None, 10)	

Total params: 205,098 (801.16 KB)
Trainable params: 205,098 (801.16 KB)
Non-trainable params: 0 (0.00 B)

In [41]:
```
from keras.utils import to_categorical

# Convert target labels to one-hot encoded format
y_train_one_hot = to_categorical(y_train)
y_test_one_hot = to_categorical(y_test)

# Now, train the model using one-hot encoded target labels
model_history = train_model(model=model, epochs=200, optimizer='adam', X_
#keras output
```

```
Epoch 1/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 4s 8ms/step - accuracy: 0.2865 - loss: 1.9389
- val_accuracy: 0.5899 - val_loss: 1.1939
Epoch 2/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 1s 6ms/step - accuracy: 0.4934 - loss: 1.3784
- val_accuracy: 0.6473 - val_loss: 1.0074
Epoch 3/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 1s 6ms/step - accuracy: 0.5693 - loss: 1.2392
- val_accuracy: 0.6940 - val_loss: 0.8896
Epoch 4/200
```

```
219/219 ━━━━━━━━━━━━━━━━━━━━ 3s 6ms/step - accuracy: 0.6017 - loss: 1.1599
- val_accuracy: 0.7231 - val_loss: 0.8010
Epoch 5/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 1s 6ms/step - accuracy: 0.6452 - loss: 1.0615
- val_accuracy: 0.7277 - val_loss: 0.7934
Epoch 6/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 1s 6ms/step - accuracy: 0.6604 - loss: 1.0268
- val_accuracy: 0.7464 - val_loss: 0.7309
Epoch 7/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 1s 6ms/step - accuracy: 0.6721 - loss: 0.9960
- val_accuracy: 0.7724 - val_loss: 0.6914
Epoch 8/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 1s 6ms/step - accuracy: 0.6916 - loss: 0.9352
- val_accuracy: 0.7824 - val_loss: 0.6479
Epoch 9/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 1s 6ms/step - accuracy: 0.7007 - loss: 0.8906
- val_accuracy: 0.7895 - val_loss: 0.6349
Epoch 10/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 1s 6ms/step - accuracy: 0.7167 - loss: 0.8786
- val_accuracy: 0.8038 - val_loss: 0.5892
Epoch 11/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 1s 6ms/step - accuracy: 0.7269 - loss: 0.8258
- val_accuracy: 0.8105 - val_loss: 0.5811
Epoch 12/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 1s 6ms/step - accuracy: 0.7425 - loss: 0.7948
- val_accuracy: 0.8141 - val_loss: 0.5793
Epoch 13/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 1s 6ms/step - accuracy: 0.7483 - loss: 0.7964
- val_accuracy: 0.8232 - val_loss: 0.5319
Epoch 14/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 1s 6ms/step - accuracy: 0.7421 - loss: 0.7756
- val_accuracy: 0.8245 - val_loss: 0.5393
Epoch 15/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 3s 6ms/step - accuracy: 0.7471 - loss: 0.7616
- val_accuracy: 0.8255 - val_loss: 0.5304
Epoch 16/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 1s 6ms/step - accuracy: 0.7576 - loss: 0.7241
- val_accuracy: 0.8315 - val_loss: 0.5031
Epoch 17/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 1s 6ms/step - accuracy: 0.7658 - loss: 0.7228
- val_accuracy: 0.8335 - val_loss: 0.4951
Epoch 18/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 1s 6ms/step - accuracy: 0.7767 - loss: 0.6799
- val_accuracy: 0.8412 - val_loss: 0.4850
Epoch 19/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 1s 6ms/step - accuracy: 0.7780 - loss: 0.6744
- val_accuracy: 0.8502 - val_loss: 0.4696
Epoch 20/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 1s 6ms/step - accuracy: 0.7814 - loss: 0.6798
- val_accuracy: 0.8515 - val_loss: 0.4714
Epoch 21/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 1s 6ms/step - accuracy: 0.7809 - loss: 0.6751
- val_accuracy: 0.8525 - val_loss: 0.4644
```

```
Epoch 22/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 1s 6ms/step - accuracy: 0.7893 - loss: 0.6588
- val_accuracy: 0.8562 - val_loss: 0.4472
Epoch 23/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 1s 6ms/step - accuracy: 0.7902 - loss: 0.6376
- val_accuracy: 0.8495 - val_loss: 0.4455
Epoch 24/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 1s 6ms/step - accuracy: 0.7969 - loss: 0.6428
- val_accuracy: 0.8612 - val_loss: 0.4245
Epoch 25/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 1s 6ms/step - accuracy: 0.8140 - loss: 0.5833
- val_accuracy: 0.8672 - val_loss: 0.3989
Epoch 26/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 1s 6ms/step - accuracy: 0.8013 - loss: 0.6125
- val_accuracy: 0.8672 - val_loss: 0.4061
Epoch 27/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 1s 6ms/step - accuracy: 0.8140 - loss: 0.5725
- val_accuracy: 0.8585 - val_loss: 0.4245
Epoch 28/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 1s 7ms/step - accuracy: 0.8134 - loss: 0.6015
- val_accuracy: 0.8712 - val_loss: 0.4014
Epoch 29/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 1s 6ms/step - accuracy: 0.8129 - loss: 0.5838
- val_accuracy: 0.8729 - val_loss: 0.3833
Epoch 30/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 1s 6ms/step - accuracy: 0.8184 - loss: 0.5602
- val_accuracy: 0.8549 - val_loss: 0.4268
Epoch 31/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 1s 6ms/step - accuracy: 0.8170 - loss: 0.5843
- val_accuracy: 0.8765 - val_loss: 0.3886
Epoch 32/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 1s 6ms/step - accuracy: 0.8132 - loss: 0.5746
- val_accuracy: 0.8719 - val_loss: 0.4029
Epoch 33/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 1s 6ms/step - accuracy: 0.8267 - loss: 0.5326
- val_accuracy: 0.8745 - val_loss: 0.3796
Epoch 34/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 2s 7ms/step - accuracy: 0.8239 - loss: 0.5554
- val_accuracy: 0.8809 - val_loss: 0.3737
Epoch 35/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 1s 7ms/step - accuracy: 0.8230 - loss: 0.5573
- val_accuracy: 0.8836 - val_loss: 0.3703
Epoch 36/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 1s 7ms/step - accuracy: 0.8218 - loss: 0.5359
- val_accuracy: 0.8869 - val_loss: 0.3455
Epoch 37/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 1s 6ms/step - accuracy: 0.8380 - loss: 0.5055
- val_accuracy: 0.8725 - val_loss: 0.3695
Epoch 38/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 2s 8ms/step - accuracy: 0.8335 - loss: 0.5442
- val_accuracy: 0.8789 - val_loss: 0.3781
Epoch 39/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 2s 7ms/step - accuracy: 0.8287 - loss: 0.5275
```

```
- val_accuracy: 0.8832 - val_loss: 0.3676
Epoch 40/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 2s 9ms/step - accuracy: 0.8388 - loss: 0.5250
- val_accuracy: 0.8862 - val_loss: 0.3604
Epoch 41/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 2s 8ms/step - accuracy: 0.8452 - loss: 0.4878
- val_accuracy: 0.8862 - val_loss: 0.3535
Epoch 42/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 2s 9ms/step - accuracy: 0.8438 - loss: 0.4895
- val_accuracy: 0.8926 - val_loss: 0.3528
Epoch 43/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 2s 8ms/step - accuracy: 0.8571 - loss: 0.4527
- val_accuracy: 0.8922 - val_loss: 0.3373
Epoch 44/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 2s 8ms/step - accuracy: 0.8462 - loss: 0.5047
- val_accuracy: 0.8939 - val_loss: 0.3297
Epoch 45/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 2s 8ms/step - accuracy: 0.8390 - loss: 0.4858
- val_accuracy: 0.8926 - val_loss: 0.3213
Epoch 46/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 2s 8ms/step - accuracy: 0.8485 - loss: 0.4985
- val_accuracy: 0.9009 - val_loss: 0.3217
Epoch 47/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 2s 8ms/step - accuracy: 0.8494 - loss: 0.4749
- val_accuracy: 0.8956 - val_loss: 0.3189
Epoch 48/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 2s 8ms/step - accuracy: 0.8572 - loss: 0.4551
- val_accuracy: 0.8936 - val_loss: 0.3417
Epoch 49/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 3s 12ms/step - accuracy: 0.8528 - loss: 0.475
2 - val_accuracy: 0.8949 - val_loss: 0.3330
Epoch 50/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 2s 8ms/step - accuracy: 0.8554 - loss: 0.4664
- val_accuracy: 0.8969 - val_loss: 0.3138
Epoch 51/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 3s 8ms/step - accuracy: 0.8361 - loss: 0.5102
- val_accuracy: 0.8799 - val_loss: 0.3555
Epoch 52/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 2s 8ms/step - accuracy: 0.8610 - loss: 0.4529
- val_accuracy: 0.8972 - val_loss: 0.3134
Epoch 53/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 2s 7ms/step - accuracy: 0.8553 - loss: 0.4818
- val_accuracy: 0.8906 - val_loss: 0.3369
Epoch 54/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 2s 7ms/step - accuracy: 0.8627 - loss: 0.4403
- val_accuracy: 0.8956 - val_loss: 0.3283
Epoch 55/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 3s 7ms/step - accuracy: 0.8583 - loss: 0.4313
- val_accuracy: 0.8959 - val_loss: 0.3308
Epoch 56/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 2s 7ms/step - accuracy: 0.8611 - loss: 0.4391
- val_accuracy: 0.9009 - val_loss: 0.3215
Epoch 57/200
```

```
219/219 ━━━━━━━━━━━━━━━━━━━━ 2s 7ms/step - accuracy: 0.8549 - loss: 0.4448
- val_accuracy: 0.9026 - val_loss: 0.3168
Epoch 58/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 2s 7ms/step - accuracy: 0.8613 - loss: 0.4267
- val_accuracy: 0.9002 - val_loss: 0.3008
Epoch 59/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 2s 7ms/step - accuracy: 0.8620 - loss: 0.4578
- val_accuracy: 0.9042 - val_loss: 0.3096
Epoch 60/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 2s 7ms/step - accuracy: 0.8667 - loss: 0.4247
- val_accuracy: 0.9112 - val_loss: 0.3034
Epoch 61/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 2s 7ms/step - accuracy: 0.8603 - loss: 0.4341
- val_accuracy: 0.9066 - val_loss: 0.3000
Epoch 62/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 2s 8ms/step - accuracy: 0.8684 - loss: 0.4311
- val_accuracy: 0.9072 - val_loss: 0.2977
Epoch 63/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 2s 7ms/step - accuracy: 0.8685 - loss: 0.4306
- val_accuracy: 0.9036 - val_loss: 0.3073
Epoch 64/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 2s 8ms/step - accuracy: 0.8653 - loss: 0.4145
- val_accuracy: 0.9106 - val_loss: 0.3045
Epoch 65/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 2s 7ms/step - accuracy: 0.8756 - loss: 0.3862
- val_accuracy: 0.9059 - val_loss: 0.3049
Epoch 66/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 3s 8ms/step - accuracy: 0.8764 - loss: 0.3941
- val_accuracy: 0.9022 - val_loss: 0.2964
Epoch 67/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 2s 7ms/step - accuracy: 0.8687 - loss: 0.4044
- val_accuracy: 0.9032 - val_loss: 0.2987
Epoch 68/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 2s 8ms/step - accuracy: 0.8642 - loss: 0.4243
- val_accuracy: 0.9019 - val_loss: 0.3067
Epoch 69/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 2s 7ms/step - accuracy: 0.8618 - loss: 0.4352
- val_accuracy: 0.9019 - val_loss: 0.2950
Epoch 70/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 2s 7ms/step - accuracy: 0.8764 - loss: 0.3930
- val_accuracy: 0.8892 - val_loss: 0.3331
Epoch 71/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 2s 8ms/step - accuracy: 0.8638 - loss: 0.4204
- val_accuracy: 0.9039 - val_loss: 0.2954
Epoch 72/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 2s 8ms/step - accuracy: 0.8736 - loss: 0.3962
- val_accuracy: 0.9039 - val_loss: 0.2908
Epoch 73/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 2s 9ms/step - accuracy: 0.8769 - loss: 0.3890
- val_accuracy: 0.9106 - val_loss: 0.2819
Epoch 74/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 2s 8ms/step - accuracy: 0.8727 - loss: 0.4094
- val_accuracy: 0.9099 - val_loss: 0.2844
```

```
Epoch 75/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 2s 8ms/step - accuracy: 0.8859 - loss: 0.3719
- val_accuracy: 0.9139 - val_loss: 0.2728
Epoch 76/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 2s 8ms/step - accuracy: 0.8851 - loss: 0.3735
- val_accuracy: 0.9102 - val_loss: 0.2904
Epoch 77/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 2s 7ms/step - accuracy: 0.8815 - loss: 0.3866
- val_accuracy: 0.9076 - val_loss: 0.2955
Epoch 78/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 2s 7ms/step - accuracy: 0.8875 - loss: 0.3812
- val_accuracy: 0.9082 - val_loss: 0.2831
Epoch 79/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 2s 7ms/step - accuracy: 0.8820 - loss: 0.3856
- val_accuracy: 0.9109 - val_loss: 0.2835
Epoch 80/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 2s 7ms/step - accuracy: 0.8783 - loss: 0.3803
- val_accuracy: 0.9102 - val_loss: 0.2925
Epoch 81/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 2s 7ms/step - accuracy: 0.8740 - loss: 0.4140
- val_accuracy: 0.9149 - val_loss: 0.2882
Epoch 82/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 3s 8ms/step - accuracy: 0.8844 - loss: 0.3851
- val_accuracy: 0.9136 - val_loss: 0.2835
Epoch 83/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 2s 8ms/step - accuracy: 0.8821 - loss: 0.4056
- val_accuracy: 0.9156 - val_loss: 0.2725
Epoch 84/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 2s 7ms/step - accuracy: 0.8766 - loss: 0.3917
- val_accuracy: 0.9069 - val_loss: 0.2976
Epoch 85/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 1s 7ms/step - accuracy: 0.8797 - loss: 0.3685
- val_accuracy: 0.9126 - val_loss: 0.2744
Epoch 86/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 2s 7ms/step - accuracy: 0.8768 - loss: 0.3804
- val_accuracy: 0.9196 - val_loss: 0.2683
Epoch 87/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 2s 8ms/step - accuracy: 0.8841 - loss: 0.3532
- val_accuracy: 0.9112 - val_loss: 0.2865
Epoch 88/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 2s 7ms/step - accuracy: 0.8810 - loss: 0.3591
- val_accuracy: 0.9186 - val_loss: 0.2692
Epoch 89/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 2s 7ms/step - accuracy: 0.8793 - loss: 0.3795
- val_accuracy: 0.9162 - val_loss: 0.2711
Epoch 90/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 2s 7ms/step - accuracy: 0.8889 - loss: 0.3618
- val_accuracy: 0.9189 - val_loss: 0.2726
Epoch 91/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 2s 8ms/step - accuracy: 0.8857 - loss: 0.3588
- val_accuracy: 0.9152 - val_loss: 0.2717
Epoch 92/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 3s 8ms/step - accuracy: 0.8870 - loss: 0.3527
```

```
- val_accuracy: 0.9162 - val_loss: 0.2811
Epoch 93/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 2s 7ms/step - accuracy: 0.8900 - loss: 0.3629
- val_accuracy: 0.9179 - val_loss: 0.2797
Epoch 94/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 2s 7ms/step - accuracy: 0.8917 - loss: 0.3693
- val_accuracy: 0.9169 - val_loss: 0.2740
Epoch 95/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 2s 7ms/step - accuracy: 0.8849 - loss: 0.3664
- val_accuracy: 0.9132 - val_loss: 0.2855
Epoch 96/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 2s 8ms/step - accuracy: 0.8952 - loss: 0.3313
- val_accuracy: 0.9116 - val_loss: 0.2855
Epoch 97/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 2s 7ms/step - accuracy: 0.8919 - loss: 0.3378
- val_accuracy: 0.9156 - val_loss: 0.2753
Epoch 98/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 2s 7ms/step - accuracy: 0.8852 - loss: 0.3572
- val_accuracy: 0.9132 - val_loss: 0.2795
Epoch 99/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 2s 8ms/step - accuracy: 0.8924 - loss: 0.3458
- val_accuracy: 0.9219 - val_loss: 0.2527
Epoch 100/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 2s 8ms/step - accuracy: 0.8987 - loss: 0.3174
- val_accuracy: 0.9142 - val_loss: 0.2803
Epoch 101/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 2s 7ms/step - accuracy: 0.8924 - loss: 0.3536
- val_accuracy: 0.9193 - val_loss: 0.2477
Epoch 102/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 2s 8ms/step - accuracy: 0.8962 - loss: 0.3425
- val_accuracy: 0.9246 - val_loss: 0.2505
Epoch 103/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 2s 9ms/step - accuracy: 0.8912 - loss: 0.3557
- val_accuracy: 0.9206 - val_loss: 0.2532
Epoch 104/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 2s 7ms/step - accuracy: 0.8976 - loss: 0.3480
- val_accuracy: 0.9193 - val_loss: 0.2664
Epoch 105/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 2s 7ms/step - accuracy: 0.8979 - loss: 0.3281
- val_accuracy: 0.9236 - val_loss: 0.2576
Epoch 106/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 1s 7ms/step - accuracy: 0.8946 - loss: 0.3427
- val_accuracy: 0.9239 - val_loss: 0.2708
Epoch 107/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 1s 6ms/step - accuracy: 0.8952 - loss: 0.3216
- val_accuracy: 0.9169 - val_loss: 0.2695
Epoch 108/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 2s 8ms/step - accuracy: 0.8970 - loss: 0.3333
- val_accuracy: 0.9159 - val_loss: 0.2824
Epoch 109/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 2s 8ms/step - accuracy: 0.8934 - loss: 0.3505
- val_accuracy: 0.9156 - val_loss: 0.2846
Epoch 110/200
```

```
219/219 ━━━━━━━━━━━━━━━━━━━━ 2s 7ms/step - accuracy: 0.8926 - loss: 0.3366
- val_accuracy: 0.9189 - val_loss: 0.2674
Epoch 111/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 2s 7ms/step - accuracy: 0.8946 - loss: 0.3482
- val_accuracy: 0.9259 - val_loss: 0.2553
Epoch 112/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 2s 8ms/step - accuracy: 0.8953 - loss: 0.3383
- val_accuracy: 0.9213 - val_loss: 0.2649
Epoch 113/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 2s 7ms/step - accuracy: 0.9107 - loss: 0.3009
- val_accuracy: 0.9259 - val_loss: 0.2605
Epoch 114/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 2s 8ms/step - accuracy: 0.8970 - loss: 0.3240
- val_accuracy: 0.9206 - val_loss: 0.2573
Epoch 115/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 2s 7ms/step - accuracy: 0.9011 - loss: 0.3142
- val_accuracy: 0.9129 - val_loss: 0.2827
Epoch 116/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 2s 7ms/step - accuracy: 0.8950 - loss: 0.3462
- val_accuracy: 0.9216 - val_loss: 0.2616
Epoch 117/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 1s 6ms/step - accuracy: 0.8945 - loss: 0.3392
- val_accuracy: 0.9246 - val_loss: 0.2631
Epoch 118/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 2s 8ms/step - accuracy: 0.8930 - loss: 0.3386
- val_accuracy: 0.9173 - val_loss: 0.2694
Epoch 119/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 2s 8ms/step - accuracy: 0.9009 - loss: 0.3269
- val_accuracy: 0.9186 - val_loss: 0.2638
Epoch 120/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 2s 8ms/step - accuracy: 0.9041 - loss: 0.3201
- val_accuracy: 0.9169 - val_loss: 0.2574
Epoch 121/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 2s 7ms/step - accuracy: 0.8975 - loss: 0.3249
- val_accuracy: 0.9219 - val_loss: 0.2563
Epoch 122/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 1s 6ms/step - accuracy: 0.9030 - loss: 0.2968
- val_accuracy: 0.9173 - val_loss: 0.2701
Epoch 123/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 2s 7ms/step - accuracy: 0.8975 - loss: 0.3191
- val_accuracy: 0.9209 - val_loss: 0.2782
Epoch 124/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 2s 8ms/step - accuracy: 0.8978 - loss: 0.3199
- val_accuracy: 0.9273 - val_loss: 0.2542
Epoch 125/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 2s 7ms/step - accuracy: 0.9011 - loss: 0.3338
- val_accuracy: 0.9266 - val_loss: 0.2569
Epoch 126/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 2s 7ms/step - accuracy: 0.9081 - loss: 0.3258
- val_accuracy: 0.9236 - val_loss: 0.2462
Epoch 127/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 2s 7ms/step - accuracy: 0.9006 - loss: 0.3355
- val_accuracy: 0.9239 - val_loss: 0.2570
```

```
Epoch 128/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 2s 7ms/step - accuracy: 0.9036 - loss: 0.3430
- val_accuracy: 0.9246 - val_loss: 0.2567
Epoch 129/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 1s 6ms/step - accuracy: 0.9033 - loss: 0.3325
- val_accuracy: 0.9269 - val_loss: 0.2530
Epoch 130/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 2s 7ms/step - accuracy: 0.9054 - loss: 0.3021
- val_accuracy: 0.9283 - val_loss: 0.2480
Epoch 131/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 2s 7ms/step - accuracy: 0.9090 - loss: 0.3104
- val_accuracy: 0.9276 - val_loss: 0.2490
Epoch 132/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 2s 7ms/step - accuracy: 0.8953 - loss: 0.3339
- val_accuracy: 0.9259 - val_loss: 0.2510
Epoch 133/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 1s 7ms/step - accuracy: 0.8997 - loss: 0.3169
- val_accuracy: 0.9239 - val_loss: 0.2580
Epoch 134/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 1s 7ms/step - accuracy: 0.9030 - loss: 0.3243
- val_accuracy: 0.9239 - val_loss: 0.2495
Epoch 135/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 2s 7ms/step - accuracy: 0.9116 - loss: 0.3037
- val_accuracy: 0.9203 - val_loss: 0.2547
Epoch 136/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 2s 7ms/step - accuracy: 0.8975 - loss: 0.3172
- val_accuracy: 0.9239 - val_loss: 0.2485
Epoch 137/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 1s 6ms/step - accuracy: 0.9065 - loss: 0.2979
- val_accuracy: 0.9253 - val_loss: 0.2521
Epoch 138/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 1s 7ms/step - accuracy: 0.9004 - loss: 0.3393
- val_accuracy: 0.9273 - val_loss: 0.2558
Epoch 139/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 1s 7ms/step - accuracy: 0.8945 - loss: 0.3429
- val_accuracy: 0.9309 - val_loss: 0.2348
Epoch 140/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 2s 7ms/step - accuracy: 0.9021 - loss: 0.3107
- val_accuracy: 0.9269 - val_loss: 0.2452
Epoch 141/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 1s 7ms/step - accuracy: 0.9038 - loss: 0.3015
- val_accuracy: 0.9279 - val_loss: 0.2453
Epoch 142/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 2s 7ms/step - accuracy: 0.9031 - loss: 0.3085
- val_accuracy: 0.9246 - val_loss: 0.2539
Epoch 143/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 2s 7ms/step - accuracy: 0.9078 - loss: 0.3043
- val_accuracy: 0.9293 - val_loss: 0.2341
Epoch 144/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 2s 7ms/step - accuracy: 0.9201 - loss: 0.2774
- val_accuracy: 0.9206 - val_loss: 0.2537
Epoch 145/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 2s 7ms/step - accuracy: 0.9009 - loss: 0.3318
```

```
- val_accuracy: 0.9253 - val_loss: 0.2438
Epoch 146/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 2s 7ms/step - accuracy: 0.9050 - loss: 0.2991
- val_accuracy: 0.9256 - val_loss: 0.2561
Epoch 147/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 2s 7ms/step - accuracy: 0.9129 - loss: 0.2830
- val_accuracy: 0.9186 - val_loss: 0.2735
Epoch 148/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 2s 7ms/step - accuracy: 0.8992 - loss: 0.2991
- val_accuracy: 0.9213 - val_loss: 0.2625
Epoch 149/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 2s 7ms/step - accuracy: 0.8994 - loss: 0.3272
- val_accuracy: 0.9243 - val_loss: 0.2499
Epoch 150/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 2s 7ms/step - accuracy: 0.9026 - loss: 0.3156
- val_accuracy: 0.9236 - val_loss: 0.2530
Epoch 151/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 2s 7ms/step - accuracy: 0.9069 - loss: 0.3081
- val_accuracy: 0.9269 - val_loss: 0.2484
Epoch 152/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 2s 7ms/step - accuracy: 0.9149 - loss: 0.2804
- val_accuracy: 0.9283 - val_loss: 0.2436
Epoch 153/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 1s 7ms/step - accuracy: 0.9121 - loss: 0.2978
- val_accuracy: 0.9323 - val_loss: 0.2352
Epoch 154/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 1s 7ms/step - accuracy: 0.9099 - loss: 0.2806
- val_accuracy: 0.9323 - val_loss: 0.2423
Epoch 155/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 2s 7ms/step - accuracy: 0.9077 - loss: 0.3210
- val_accuracy: 0.9319 - val_loss: 0.2373
Epoch 156/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 2s 7ms/step - accuracy: 0.9197 - loss: 0.2703
- val_accuracy: 0.9326 - val_loss: 0.2311
Epoch 157/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 1s 7ms/step - accuracy: 0.9118 - loss: 0.2860
- val_accuracy: 0.9366 - val_loss: 0.2348
Epoch 158/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 1s 6ms/step - accuracy: 0.9151 - loss: 0.2997
- val_accuracy: 0.9303 - val_loss: 0.2500
Epoch 159/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 2s 7ms/step - accuracy: 0.9054 - loss: 0.3001
- val_accuracy: 0.9266 - val_loss: 0.2523
Epoch 160/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 2s 7ms/step - accuracy: 0.9184 - loss: 0.2665
- val_accuracy: 0.9339 - val_loss: 0.2449
Epoch 161/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 2s 7ms/step - accuracy: 0.9068 - loss: 0.2986
- val_accuracy: 0.9323 - val_loss: 0.2482
Epoch 162/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 1s 7ms/step - accuracy: 0.9083 - loss: 0.2897
- val_accuracy: 0.9303 - val_loss: 0.2486
Epoch 163/200
```

```
219/219 ━━━━━━━━━━━━━━━━━━━━ 2s 7ms/step - accuracy: 0.9052 - loss: 0.3144
- val_accuracy: 0.9283 - val_loss: 0.2436
Epoch 164/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 1s 6ms/step - accuracy: 0.9162 - loss: 0.2931
- val_accuracy: 0.9326 - val_loss: 0.2359
Epoch 165/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 2s 7ms/step - accuracy: 0.9158 - loss: 0.2873
- val_accuracy: 0.9309 - val_loss: 0.2527
Epoch 166/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 2s 7ms/step - accuracy: 0.9082 - loss: 0.2974
- val_accuracy: 0.9283 - val_loss: 0.2420
Epoch 167/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 2s 7ms/step - accuracy: 0.9187 - loss: 0.2752
- val_accuracy: 0.9246 - val_loss: 0.2498
Epoch 168/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 2s 7ms/step - accuracy: 0.9090 - loss: 0.2794
- val_accuracy: 0.9306 - val_loss: 0.2421
Epoch 169/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 2s 7ms/step - accuracy: 0.9177 - loss: 0.2781
- val_accuracy: 0.9249 - val_loss: 0.2510
Epoch 170/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 1s 6ms/step - accuracy: 0.9135 - loss: 0.2800
- val_accuracy: 0.9233 - val_loss: 0.2620
Epoch 171/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 2s 7ms/step - accuracy: 0.9095 - loss: 0.2867
- val_accuracy: 0.9219 - val_loss: 0.2702
Epoch 172/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 2s 7ms/step - accuracy: 0.9113 - loss: 0.2860
- val_accuracy: 0.9333 - val_loss: 0.2428
Epoch 173/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 2s 7ms/step - accuracy: 0.9104 - loss: 0.2873
- val_accuracy: 0.9316 - val_loss: 0.2553
Epoch 174/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 2s 7ms/step - accuracy: 0.9134 - loss: 0.2880
- val_accuracy: 0.9269 - val_loss: 0.2483
Epoch 175/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 1s 7ms/step - accuracy: 0.9104 - loss: 0.2968
- val_accuracy: 0.9289 - val_loss: 0.2612
Epoch 176/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 2s 7ms/step - accuracy: 0.9074 - loss: 0.3163
- val_accuracy: 0.9259 - val_loss: 0.2592
Epoch 177/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 2s 7ms/step - accuracy: 0.9243 - loss: 0.2668
- val_accuracy: 0.9179 - val_loss: 0.2641
Epoch 178/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 1s 6ms/step - accuracy: 0.9105 - loss: 0.2932
- val_accuracy: 0.9289 - val_loss: 0.2325
Epoch 179/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 1s 7ms/step - accuracy: 0.9135 - loss: 0.2886
- val_accuracy: 0.9289 - val_loss: 0.2512
Epoch 180/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 1s 6ms/step - accuracy: 0.9162 - loss: 0.2930
- val_accuracy: 0.9266 - val_loss: 0.2512
```

```
Epoch 181/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 2s 7ms/step - accuracy: 0.9155 - loss: 0.2679
- val_accuracy: 0.9216 - val_loss: 0.2612
Epoch 182/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 2s 7ms/step - accuracy: 0.9146 - loss: 0.2829
- val_accuracy: 0.9233 - val_loss: 0.2463
Epoch 183/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 2s 7ms/step - accuracy: 0.9067 - loss: 0.2969
- val_accuracy: 0.9229 - val_loss: 0.2551
Epoch 184/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 2s 7ms/step - accuracy: 0.9208 - loss: 0.2707
- val_accuracy: 0.9303 - val_loss: 0.2477
Epoch 185/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 2s 7ms/step - accuracy: 0.9117 - loss: 0.2909
- val_accuracy: 0.9263 - val_loss: 0.2490
Epoch 186/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 2s 7ms/step - accuracy: 0.9119 - loss: 0.2863
- val_accuracy: 0.9259 - val_loss: 0.2379
Epoch 187/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 1s 7ms/step - accuracy: 0.9153 - loss: 0.2871
- val_accuracy: 0.9273 - val_loss: 0.2441
Epoch 188/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 1s 6ms/step - accuracy: 0.9198 - loss: 0.2666
- val_accuracy: 0.9273 - val_loss: 0.2434
Epoch 189/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 2s 7ms/step - accuracy: 0.9124 - loss: 0.2717
- val_accuracy: 0.9259 - val_loss: 0.2469
Epoch 190/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 2s 7ms/step - accuracy: 0.9213 - loss: 0.2769
- val_accuracy: 0.9279 - val_loss: 0.2326
Epoch 191/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 1s 7ms/step - accuracy: 0.9216 - loss: 0.2752
- val_accuracy: 0.9316 - val_loss: 0.2447
Epoch 192/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 1s 6ms/step - accuracy: 0.9231 - loss: 0.2516
- val_accuracy: 0.9299 - val_loss: 0.2346
Epoch 193/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 2s 8ms/step - accuracy: 0.9079 - loss: 0.3015
- val_accuracy: 0.9329 - val_loss: 0.2350
Epoch 194/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 2s 7ms/step - accuracy: 0.9203 - loss: 0.2608
- val_accuracy: 0.9283 - val_loss: 0.2463
Epoch 195/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 2s 8ms/step - accuracy: 0.9156 - loss: 0.2545
- val_accuracy: 0.9223 - val_loss: 0.2525
Epoch 196/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 2s 7ms/step - accuracy: 0.9190 - loss: 0.2597
- val_accuracy: 0.9246 - val_loss: 0.2553
Epoch 197/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 1s 7ms/step - accuracy: 0.9092 - loss: 0.3010
- val_accuracy: 0.9286 - val_loss: 0.2494
Epoch 198/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 2s 7ms/step - accuracy: 0.9180 - loss: 0.2709
```

```
- val_accuracy: 0.9276 - val_loss: 0.2442
Epoch 199/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 2s 7ms/step - accuracy: 0.9137 - loss: 0.2867
- val_accuracy: 0.9289 - val_loss: 0.2599
Epoch 200/200
219/219 ━━━━━━━━━━━━━━━━━━━━ 2s 7ms/step - accuracy: 0.9117 - loss: 0.2773
- val_accuracy: 0.9259 - val_loss: 0.2587
```

In [45]:

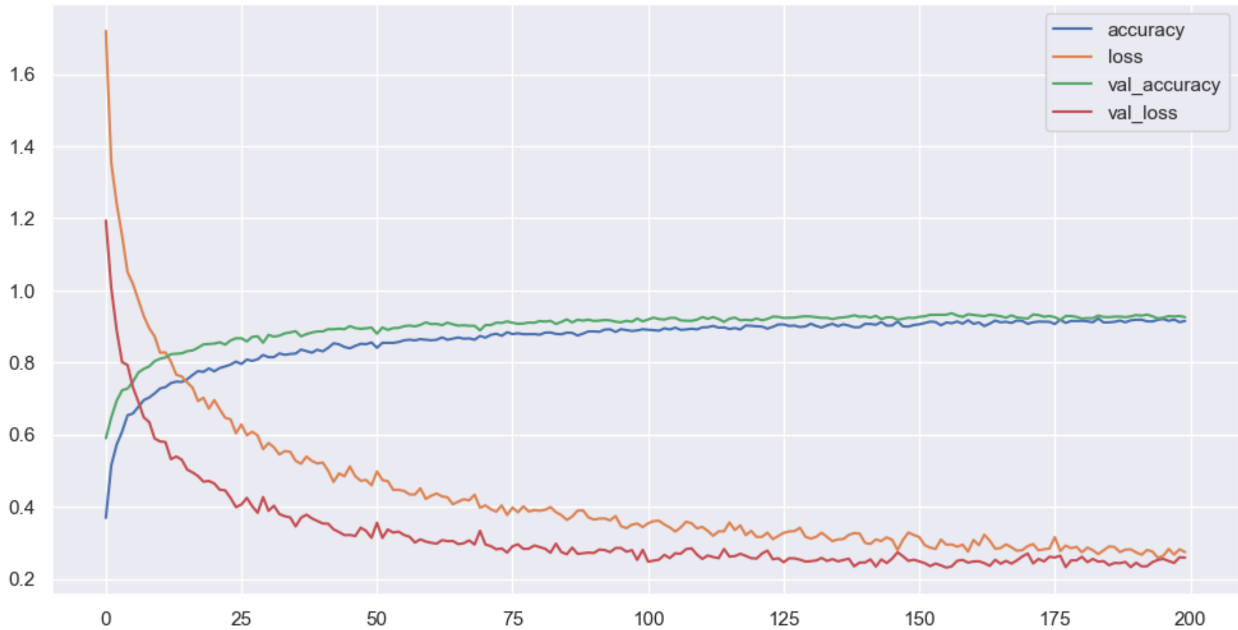
```
test_loss,test_acc=model.evaluate(X_test,y_test_one_hot,batch_size=256)
print("The test loss is ",test_loss)
print("The best accuracy is: ",test_acc*100)
```

```
12/12 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - accuracy: 0.9291 - loss: 0.2465
The test loss is  0.2587079703807831
The best accuracy is:  92.59259104728699
```

In [46]:

```
# The plot dipicts how training and testing data performed
Validation_plot(model_history)
```

```
Validation Accuracy 0.9366032481193542
```



In [47]:

```
# Sample testing
sample = y_test_one_hot
sample = sample[np.newaxis, ...]
prediction = model.predict(X_test)
predicted_index = np.argmax(prediction, axis = 1)
print("Expected Index: {}, Predicted Index: {}".format(y_test, predicted_
```

```
94/94 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step
Expected Index: [1 6 0 ... 7 8 7], Predicted Index: [1 6 0 ... 3 8 7]
```

In [48]:

```
# Plotting the confusion matrix for analizing the true positives and nega
import seaborn as sn
import matplotlib.pyplot as plt
pred_x = model.predict(X_test)
from sklearn.metrics import confusion_matrix
```

```
cm = confusion_matrix(y_test,predicted_index )
cm
```

```
94/94 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step
```

Out[48]:
```
array([[290,   0,   4,   1,   1,   3,   1,   0,   6,   2],
       [  0, 293,   2,   0,   0,   7,   0,   0,   1,   0],
       [  6,   1, 272,   4,   1,   8,   0,   2,   9,   3],
       [  2,   1,   1, 289,   3,   1,   1,   5,   3,   2],
       [  0,   0,   3,   5, 289,   0,   1,   5,   7,   1],
       [  3,  22,   8,   0,   0, 267,   0,   0,   0,   1],
       [  2,   0,   0,   6,   1,   0, 276,   0,   3,   7],
       [  0,   0,   3,   5,   7,   1,   0, 282,   5,   1],
       [  1,   0,   2,   1,   2,   0,   0,   1, 274,   3],
       [  0,   2,   7,   8,   2,   1,   3,   6,   5, 243]])
```

Conclusion

As expected CNN outperformed KNN and SVM. It produced best results in both testing and taring data. As we increased the number of epochs the loss percentage decreased with a gradual increase in accuracy scores. It can be clearly seen in the above validation plot in which the curves almost coincided with each other.

References

1.https://www.tensorflow.org/datasets/catalog/gtzan

2.https://www.kaggle.com/code/dapy15/music-genre-classification

3.https://www.clairvoyant.ai/blog/music-genre-classification-using-cnn

4.https://github.com/alikaratana/Music-Genre-Classification

In [ ]:

In [ ]: