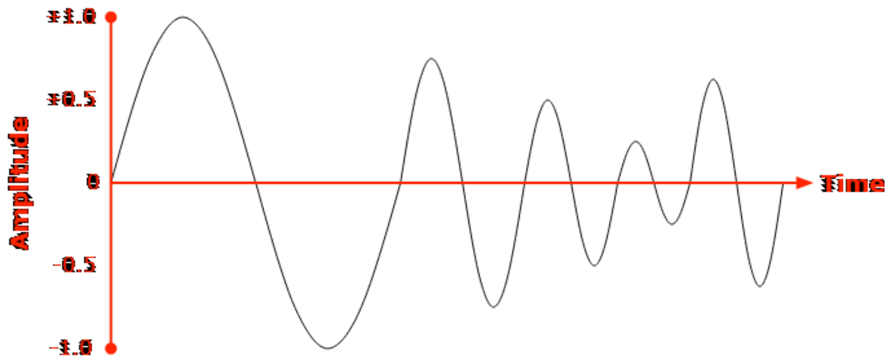
Music Generation

Music composition is one of the main or even the most important aspect of a song. Songs composition cost between $50 to $50,000 per minute of finished music (depend on the experience and the name). In order to compose new popular music artist usually hire composer with the right abilities to compose the specific songs needed for the artist stile. The ability to use the computers to create new and specific music stile can reduce the financial expedites and in the long run reduce dramatically the cost of creating new music/songs. In this project I demonstrate the ability to create new music based on specific genre simple using simple neural network model. The ability to produce artificial music can pave the way for cheaper music generation, moreover by saving large amount of money for the artist should result in larger audience and also in more profitable songs for both the artist and the public.

All data in this project base on Kaggle (https://www.kaggle.com/datasets/andradaolteanu/gtzan-dataset-music-genre-classification?resource=download)). This dataset contains such columns as: "Rock, jazz, classic and more.

Audio Data basic:

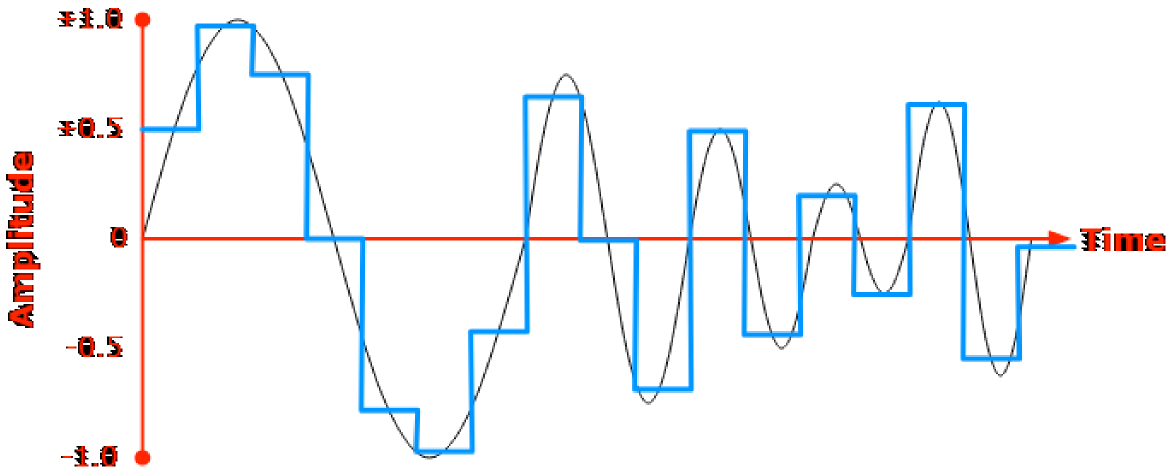
Audio is an inherently analog feature of the natural world. As an object vibrates, it causes the molecules surrounding it to vibrate as well. These molecules affect the ones adjacent to them, and so forth, propagating the vibration in the form of a wave outward from the source until the amplitude of the wave (its volume) fades away with distance. The sounds a person hears every day are, actually vibrations in the air which cause the inner workings of the ear. The farther the air molecules move with each pulse of the wave, the higher the amplitude of the wave, and the louder the sound is. The faster the molecules vibrate, the higher the frequency of the wave.



Computers, however, are digital. In order to represent a sound wave in a way computers can manipulate and work with, the sound has to be converted into a digital form. This process is called analog to digital conversion (A/D for short).

The first factor affecting the fidelity of the captured audio is the audio bandwidth; that is, the range of audio frequencies the A/D converter is capable of capturing and converting into digital form. The audio bandwidth is also affected by the codec, if it chooses to discard any frequency bands while encoding the sound.

This analog signal is then converted into digital form by a circuit that captures the incoming wave's amplitude at regular intervals, converting that data into a number in a form that is understood by the audio recording system. Each of these captured moments is a sample. By chaining all the samples together, you can approximately represent the original wave.



In working with audio data, one of the biggest challenges is preparing an audio file. It has two parts to it, in the time domain and frequency domain. Time is considered an independent variable in the time domain, and its graph shows us how signals are changing over time. In contrast, the frequency domain considers the frequency of the signal as an independent variable. Its graph shows how much of the signal lies in each frequency band over a range of frequencies, making audio data analysis more difficult to perform.

*Obtain from: https://developer.mozilla.org/en-US/docs/Web/Media/Formats/Audio_concepts https://analyticsindiamag.com/a-guide-to-audio-data-preparation-using-tensorflow/

Data Rangling or EDA

This is an example of how to work with audio waveform and spectrogram to analyze audio data.

```
import os
import matplotlib.pyplot as plt

#for loading and visualizing audio files
import librosa
import librosa.display
```

```
#to play audio
import IPython.display as ipd

audio_fpath = "te/Energetic-Drink.wav"
```

Loads and decodes the audio as a time series y, represented as a one-dimensional NumPy floating point array. sr is the sampling rate of y(number of samples per second).

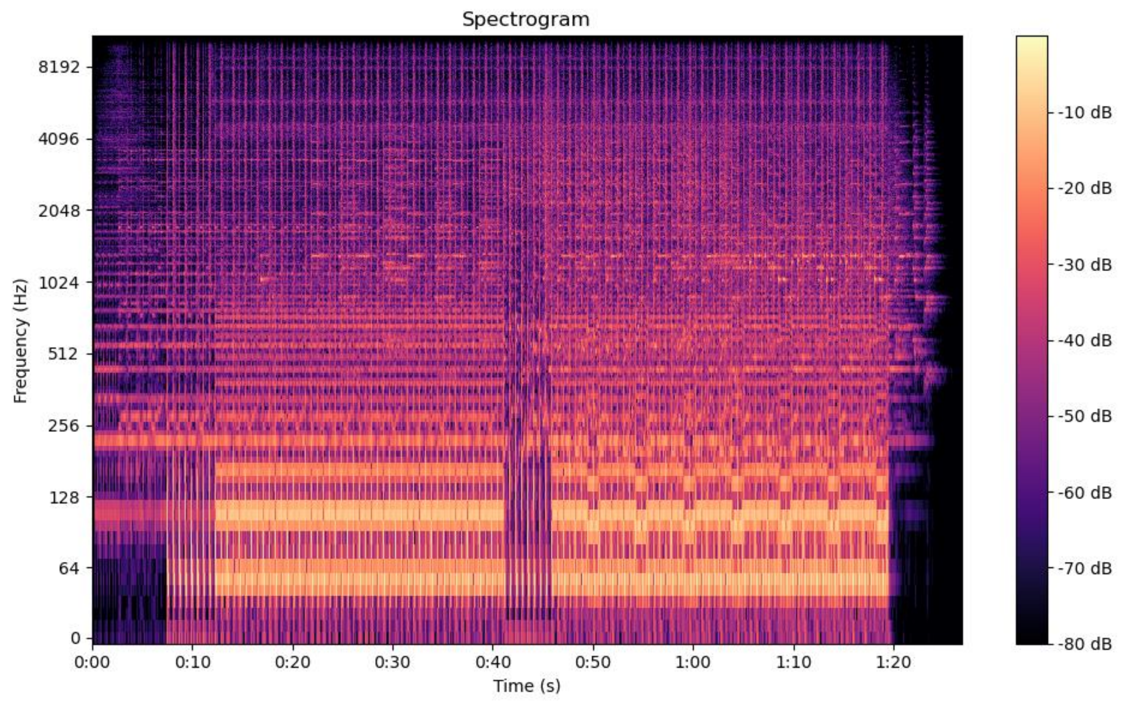
```
y, sr = librosa.load(audio_fpath)
```

Creating a visual way of representing the signal loudness (spectrogram).

```
D = librosa.stft(y)
```

```
import numpy as np
spectrogram = librosa.amplitude_to_db(abs(D), ref=np.max)
```

```
# Display the spectrogram
plt.figure(figsize=(10, 6))
librosa.display.specshow(spectrogram, y_axis='log', x_axis='time')
plt.colorbar(format='%+2.0f dB')
plt.title('Spectrogram')
plt.xlabel('Time (s)')
plt.ylabel('Frequency (Hz)')
plt.tight_layout()
plt.show()
```



```
print(type(y), type(sr))
```

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

Resample from 22KHz (default) to 45.6KHz

```
librosa.load(audio_fpath, sr=45600)
```

```
(array([0., 0., 0., ..., 0., 0., 0.], dtype=float32), 45600)
```

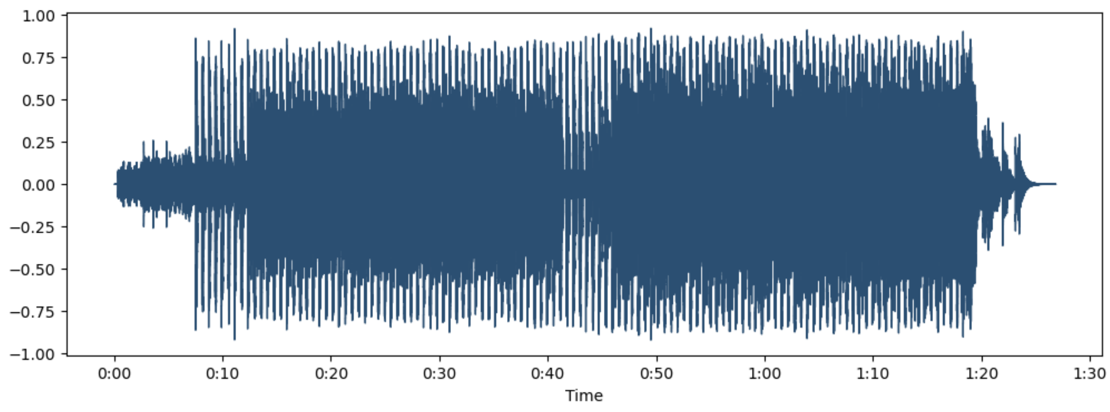
Playing the audio.

```
import IPython
IPython.display.Audio(y, rate=sr)
```

```
<IPython.lib.display.Audio object>
```

Raw Wave plot.

```
plt.figure(figsize=(12, 4))
librosa.display.waveshow(y, color = "#2B4F72")
plt.show()
```

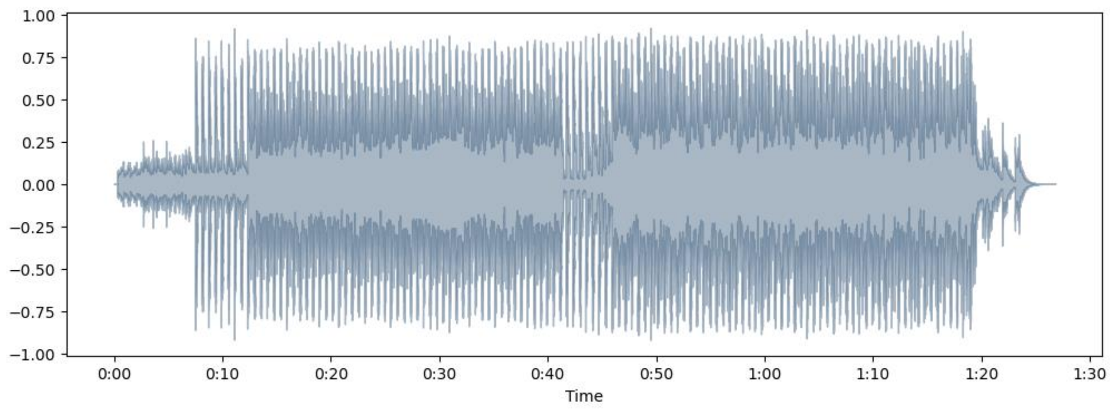


Normelaize spectra

```
from sklearn.preprocessing import normalize
spectral_rolloff = librosa.feature.spectral_rolloff(y+0.01, sr=sr)[0]
plt.figure(figsize=(12, 4))
librosa.display.waveshow(y, sr=sr, alpha=0.4, color = "#2B4F72")
```

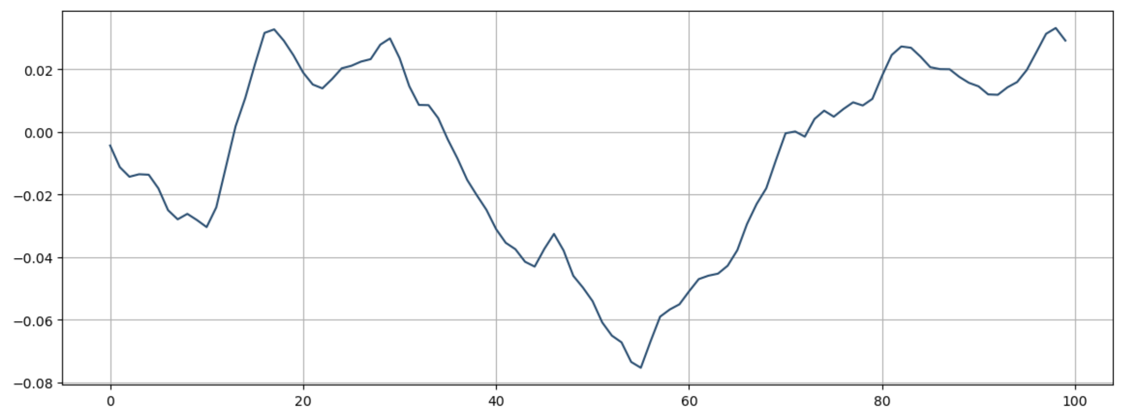
```
C:\Users\moren\AppData\Local\Temp\ipykernel_11468\2580748751.py:2:
FutureWarning: Pass y=[0.01 0.01 0.01 ... 0.01 0.01 0.01] as keyword args.
From version 0.10 passing these as positional arguments will result in an
error
  spectral_rolloff = librosa.feature.spectral_rolloff(y+0.01, sr=sr)[0]
```

```
<librosa.display.AdaptiveWaveplot at 0x1640bea1bb0>
```



Zooming on the wave.

```
n0 = 9000
n1 = 9100
plt.figure(figsize=(14, 5))
plt.plot(y[n0:n1], color="#2B4F72")
plt.grid()
```



Calculate the Zero-crossing rate (the rate or number at which zero crossings occur).

```
zero_crossings = librosa.zero_crossings(y[n0:n1], pad=False)
print("The number of zero-crossings is :",sum(zero_crossings))
```

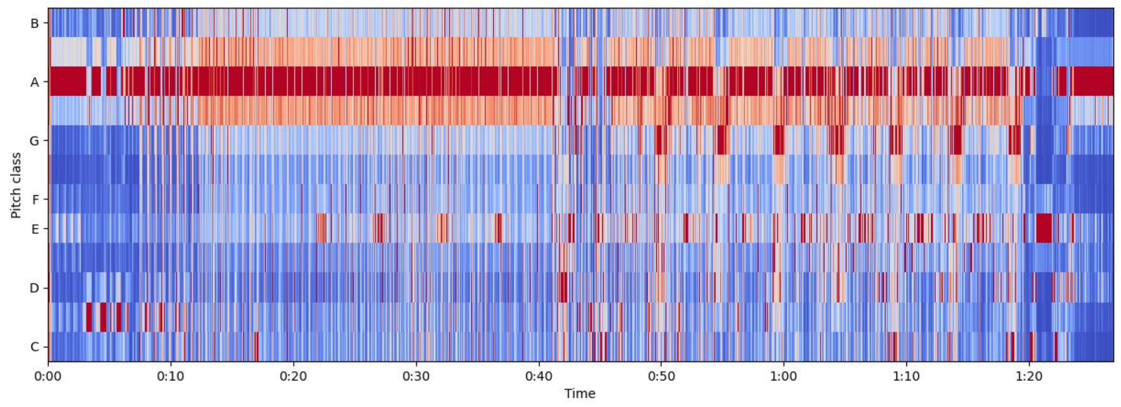
```
The number of zero-crossings is : 5
```

chroma features can capture harmonic and melodic characteristics of music, and thereby it's a powerful tool for analyzing music.

```
chromagram = librosa.feature.chroma_stft(y, sr=sr)
plt.figure(figsize=(15, 5))
librosa.display.specshow(chromagram, x_axis='time', y_axis='chroma',
cmap='coolwarm')
```

```
C:\Users\moren\AppData\Local\Temp\ipykernel_11468\2454330155.py:1:
FutureWarning: Pass y=[0. 0. 0. ... 0. 0. 0.] as keyword args. From version
0.10 passing these as positional arguments will result in an error
  chromagram = librosa.feature.chroma_stft(y, sr=sr)
```

```
<matplotlib.collections.QuadMesh at 0x1640bf9a070>
```



In order to prepare music files for AI we need to convert raw audio data into a suitable format that can be used as input for the generative model. In the context of music, this often means converting audio files into a spectrogram representation, which is a visual representation of the audio frequencies over time. To do so first we need to:

- Choose a Dataset: (in our case I choose rock from gtzan-dataset)
- Convert Audio Files to Spectrograms (Most generative models for music are trained on spectrograms instead of raw audio data).

In this project I used this function to convert wav files to spectrograms:

Converting wav files (or library) to normalized Spectrogram

```
import os
import librosa
import librosa.display
import matplotlib.pyplot as plt
import numpy as np
from sklearn.preprocessing import MinMaxScaler

def wav_to_spectrogram(audio_file_path, output_folder):
    # Load the audio file
    y, sr = librosa.load(audio_file_path)

    # Compute the spectrogram using Short-Time Fourier Transform (STFT)
    D = librosa.stft(y)

    # Convert amplitude to decibels
    spectrogram = librosa.amplitude_to_db(abs(D), ref=np.max)

    # Normalize the spectrogram using Min-Max scaling
    scaler = MinMaxScaler()
    normalized_spectrogram = scaler.fit_transform(spectrogram)


    # Plot and save the spectrogram

    plt.figure(figsize=(10, 6))
    librosa.display.specshow(normalized_spectrogram, y_axis='log',
x_axis='time')
    plt.colorbar(format='%+2.0f dB')
    plt.title('Normalized Spectrogram')
    plt.xlabel('Time (s)')
    plt.ylabel('Frequency (Hz)')
    plt.tight_layout()

    # Get the filename without extension
    filename = os.path.splitext(os.path.basename(audio_file_path))[0]

    # Save the spectrogram as an image
    output_path = os.path.join(output_folder, f'{filename}_spectrogram.png')
    plt.savefig(output_path)
    plt.close()

# Path to the folder containing WAV files
input_folder = 'Data/genres_original/rock'  # Replace with the actual folder
path

# Path to the folder where you want to save the spectrograms
output_folder = 'Data/genres_original/rock_norm'  # Replace with the actual
folder path

# Loop through each WAV file in the input folder and convert to spectrogram
for filename in os.listdir(input_folder):
    if filename.endswith('.wav'):
        audio_file_path = os.path.join(input_folder, filename)
        wav_to_spectrogram(audio_file_path, output_folder)
```

Create Input-Output Pairs:

After converting the wave files into spectrograms, in order to train the generative model, we need to create input-output pairs from the spectrogram data. by taking a small slice of the spectrograms as input and corresponding slice as the output target. This process helps the model learn patterns in the music and generate coherent sequences. Next, we need to save the input-output pairs in a format that can be easily read by the chosen deep learning framework. Common formats include NumPy arrays or TFRecords (TensorFlow). We also need to normalize the data. This step ensures efficient data loading during the training process. Below is the function I used to create the Input-Output Pairs (it is important to mention that I use small batches to overcome memory issues).

```
import os
import numpy as np
from PIL import Image
```

Creating normalized input output pairs

```
# Function to create input-output pairs from a single spectrogram image
def create_input_output_pairs_from_image(spectrogram_image_path, patch_size):
    # Load the spectrogram image
    spectrogram_image = Image.open(spectrogram_image_path)
    spectrogram = np.array(spectrogram_image)

    # Get the number of time steps and frequency bins in the spectrogram
    num_time_steps, num_freq_bins = spectrogram.shape[:2]

    # Initialize lists to store input and output pairs
    input_pairs = []
    output_pairs = []

    # Slide a window of size 'patch_size' across the spectrogram to create
pairs
    for i in range(0, num_time_steps - patch_size * 2 + 1):
        input_patch = spectrogram[i:i+patch_size, :]
        output_patch = spectrogram[i+patch_size:i+2*patch_size, :]
        input_pairs.append(input_patch)
        output_pairs.append(output_patch)

    # Convert lists to numpy arrays
    input_pairs = np.array(input_pairs)
    output_pairs = np.array(output_pairs)

    return np.concatenate((input_pairs, output_pairs), axis=1)

# Path to the library (folder) containing spectrogram images
library_path = 'Data/genres_original/rock_norm'  # Replace with the actual
folder path

# Assuming 'patch_size' is the desired size of each input-output pair patch
patch_size = 64  # Replace with your desired patch size

# Define batch size to process spectrograms in smaller batches
batch_size = 10

input_pairs_list = []

# Iterate over the spectrogram images in the library and create input-output
pairs in batches
for filename in os.listdir(library_path):
    if filename.endswith('.png'):  # Assuming the images are in PNG format
        spectrogram_image_path = os.path.join(library_path, filename)
        input_pairs =
create_input_output_pairs_from_image(spectrogram_image_path, patch_size)
        input_pairs_list.append(input_pairs)

        # Process input pairs in batches
        if len(input_pairs_list) >= batch_size:
            input_pairs_batch = np.concatenate(input_pairs_list, axis=0)
            # Save the batch of input-output pairs as NumPy arrays
            np.save(f'input_pairs_batch_{len(input_pairs_list)}.npy',
input_pairs_batch)
            # Clear the list for the next batch
            input_pairs_list.clear()
```

```
# Save any remaining pairs after processing all spectrograms
if input_pairs_list:
    input_pairs_batch = np.concatenate(input_pairs_list, axis=0)
    np.save(f'input_pairs_batch_{len(input_pairs_list)}.npy',
input_pairs_batch)
    input_pairs_list.clear()
```

```
input_pairs_batch.shape
```

```
(4730, 128, 1000, 4)
```

Training the Mode:

During this project I used a simple neural network model using the Keras library with TensorFlow backend. The model architecture consists of two layers:

- LSTM Layer:
  The first layer is an LSTM (Long Short-Term Memory) layer with 256 units. LSTM is a type of recurrent neural network (RNN) that is well-suited for sequence data, such as time series or sequential data. I used the input shape parameters of: num_time_steps, num_freq_bins * num_channels.
  num_time_steps - represents the number of time steps in the sequence.
  num_freq_bins * num_channels - the number of features in each time step.
  By using (return_sequences=True) the LSTM layer will return the output for each time step in the input sequence.

- Dense Layer:
  The second layer is a Dense layer (fully connected layer) with num_freq_bins * num_channels units. This layer is used to map the output of the LSTM layer to the final output space.
  By using (activation='sigmoid') the sigmoid function, which squashes the output values between 0 and 1.

- The model is compiled using the compile method:
  Loss function is used(loss='mse'), which is suitable for regression tasks where the goal is to minimize the squared differences between predicted and actual values.
  The Adam optimizer (optimizer='adam') to optimize the model's weights during training.

Below is the function I used to train the model and an example:

Using input output pair to train RNN Model

```python
import numpy as np

# Load the input-output pairs from the npy files
input_pairs_all = np.load('input_pairs.npy')

import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import LSTM, Dense

num_time_steps = input_pairs_all.shape[1] // 2
num_freq_bins = input_pairs_all.shape[2]
num_channels = input_pairs_all.shape[3]

# Ensure the input_pairs_all has an even number of time steps
if input_pairs_all.shape[1] % 2 != 0:
    input_pairs_all = input_pairs_all[:, :-1, :, :]

# Define your RNN-based generative model
model = Sequential()
model.add(LSTM(256, input_shape=(num_time_steps, num_freq_bins *
num_channels), return_sequences=True))
model.add(Dense(num_freq_bins * num_channels, activation='sigmoid'))
model.compile(loss='mse', optimizer='adam')

# Define batch size and number of epochs
batch_size = 32
num_epochs = 50

# Create a generator for input-output pairs
def batch_generator(input_pairs, batch_size):
    num_samples = len(input_pairs)
    indices = np.arange(num_samples)
    np.random.shuffle(indices)

    for start_idx in range(0, num_samples, batch_size):
        excerpt = indices[start_idx:start_idx + batch_size]
        batch = input_pairs[excerpt]

        # Split the batch into input and output parts
        input_batch = batch[:, :num_time_steps, :].reshape(-1,
num_time_steps, num_freq_bins * num_channels)
        output_batch = batch[:, num_time_steps:, :].reshape(-1,
num_time_steps, num_freq_bins * num_channels)

        yield input_batch, output_batch

input_pairs_generator = batch_generator(input_pairs_all, batch_size)

# Train the model using the generator
```

```
for epoch in range(num_epochs):
    for input_batch, output_batch in input_pairs_generator:
        model.train_on_batch(input_batch, output_batch)

seed_sequence = input_pairs_all[0:1, :num_time_steps, :, :]
seed_sequence_reshaped = seed_sequence.reshape(-1, num_time_steps,
num_freq_bins * num_channels)

# Generate new music using the trained model
generated_music2 = []
for _ in range(num_time_steps):  # The num_time_steps for the desired length
of the generated music
    predictions = model.predict(seed_sequence_reshaped)
    generated_music2.append(predictions[:, -1:, :])
    seed_sequence_reshaped = np.concatenate([seed_sequence_reshaped[:, 1:,
:], predictions[:, -1:, :]], axis=1)

# Convert the generated_music list to a numpy array
generated_music2 = np.concatenate(generated_music2, axis=1)
```

```
1/1 [==============================] - 0s 40ms/step
1/1 [==============================] - 0s 371ms/step
1/1 [==============================] - 0s 41ms/step
1/1 [==============================] - 0s 42ms/step
1/1 [==============================] - 0s 40ms/step
1/1 [==============================] - 0s 53ms/step
1/1 [==============================] - 0s 37ms/step
1/1 [==============================] - 0s 42ms/step
1/1 [==============================] - 0s 52ms/step
1/1 [==============================] - 0s 51ms/step
1/1 [==============================] - 0s 41ms/step
1/1 [==============================] - 0s 51ms/step
1/1 [==============================] - 0s 40ms/step
1/1 [==============================] - 0s 44ms/step
1/1 [==============================] - 0s 43ms/step
1/1 [==============================] - 0s 41ms/step
1/1 [==============================] - 0s 45ms/step
1/1 [==============================] - 0s 37ms/step
1/1 [==============================] - 0s 57ms/step
1/1 [==============================] - 0s 40ms/step
1/1 [==============================] - 0s 41ms/step
1/1 [==============================] - 0s 68ms/step
1/1 [==============================] - 0s 43ms/step
1/1 [==============================] - 0s 40ms/step
1/1 [==============================] - 0s 40ms/step
1/1 [==============================] - 0s 54ms/step
1/1 [==============================] - 0s 38ms/step
1/1 [==============================] - 0s 44ms/step
1/1 [==============================] - 0s 61ms/step
1/1 [==============================] - 0s 49ms/step
```

```
1/1 [==============================] - 0s 40ms/step
1/1 [==============================] - 0s 45ms/step
1/1 [==============================] - 0s 48ms/step
1/1 [==============================] - 0s 38ms/step
1/1 [==============================] - 0s 40ms/step
1/1 [==============================] - 0s 37ms/step
1/1 [==============================] - 0s 36ms/step
1/1 [==============================] - 0s 43ms/step
1/1 [==============================] - 0s 45ms/step
1/1 [==============================] - 0s 38ms/step
1/1 [==============================] - 0s 42ms/step
1/1 [==============================] - 0s 39ms/step
1/1 [==============================] - 0s 40ms/step
1/1 [==============================] - 0s 38ms/step
1/1 [==============================] - 0s 41ms/step
1/1 [==============================] - 0s 42ms/step
1/1 [==============================] - 0s 42ms/step
1/1 [==============================] - 0s 43ms/step
1/1 [==============================] - 0s 38ms/step
1/1 [==============================] - 0s 39ms/step
1/1 [==============================] - 0s 42ms/step
1/1 [==============================] - 0s 60ms/step
1/1 [==============================] - 0s 39ms/step
1/1 [==============================] - 0s 40ms/step
1/1 [==============================] - 0s 57ms/step
1/1 [==============================] - 0s 41ms/step
1/1 [==============================] - 0s 59ms/step
1/1 [==============================] - 0s 38ms/step
1/1 [==============================] - 0s 40ms/step
1/1 [==============================] - 0s 38ms/step
1/1 [==============================] - 0s 41ms/step
1/1 [==============================] - 0s 37ms/step
1/1 [==============================] - 0s 56ms/step
1/1 [==============================] - 0s 43ms/step
```

Using model to create new music

```python
min_value = 0.0
max_value = 1.0

# Denormalization - Apply the formula element-wise to the entire array
denormalized_music2 = (generated_music2 * (max_value - min_value)) +
min_value

from scipy.io import wavfile

# Set the sampling rate for the generated music

sampling_rate = 44100
```

```
# If 'denormalized_music' contains stereo audio (2 channels), use this:
denormalized_music2 = np.int16(denormalized_music2 * 32767)  # Scale the
denormalized values to 16-bit integers
wavfile.write('generated_music2.wav', sampling_rate, denormalized_music2)

# If 'denormalized_music' contains mono audio (1 channel), use this:
denormalized_music_mono2 = denormalized_music2[:, :, 0]  # Extract the first
channel
denormalized_music_mono2 = np.int16(denormalized_music_mono2 * 32767)  #
Scale the denormalized values to 16-bit integers
wavfile.write('generated_music_mono2.wav', sampling_rate,
denormalized_music_mono2)

import pyaudio

# Set the sampling rate for the generated music

sampling_rate = 44100

# If 'denormalized_music' contains stereo audio (2 channels), use this:
denormalized_music2 = np.int16(denormalized_music2 * 32767)


# Initialize PyAudio
p = pyaudio.PyAudio()

# Open a streaming stream
stream = p.open(format=pyaudio.paInt16,
                channels=2,  # Use 1 for mono or 2 for stereo
                rate=sampling_rate,
                output=True)

# Play the audio
stream.write(denormalized_music2.tobytes())

# Stop the stream and close the PyAudio object
stream.stop_stream()
stream.close()
p.terminate()
```

Conclusion:

In conclusion in this project, I showed the ability to construct simple neural network model for recurrent neural network. During the project I encounter several obstacles like shape of the spectrogram, reshaping the data, issue with the dimensions of the input and output batches, compatibility issue between TensorFlow and the version of NumPy and also librosa and NumPy and also issues with memory of my computer. Overall, I learned a lot from this experience and also did a lot of mistakes that got me forward. This model is far from perfect and considering my computational power I believe that with the right computational power I can achieve better performance. The ability to be able to create music with high efficiency and low budget can reduce the cost of song producing.