Final Project

Development of a model that generates rhythmic drum music matching the individual's heartbeat

Prepared by Group 5:
110511137 劉資浩
111101013 蔡長頤
111550079 方品仁
111550205 葉莉莎

INTRODUCTION

In today’s business world, more and more people often work too much, sleep poorly, stress out, have bad habits and a broken routine. Meditation is a well known mental exercise, relaxation and rebooting will be much easier if a person starts practicing it. One can reduce stress, increase calmness and clarity, and promote happiness by listening to special kinds of music while meditating.

Therefore, the development of a model, which will help to practice meditation, improve overall well-being on one’s own and calm your mind is **relevant**.

The core idea is creating music that aligns with an individual's heartbeat, thus personalizing the rhythm to the listener's physiological state.

Tasks of the work:

1. Research existence applications on reducing stress with music.
2. Finding ways to improve existing applications and create something new.
3. Choose an appropriate dataset of music.
4. Creating a program algorithm.
5. Development of a Python program that generates rhythmic drum music matching the individual's heartbeat.
6. User trial.

To solve the problems, research methods are used, namely:

– Analysis and generalization of information from the Internet.

– Analysis of the properties of existing applications.

Practical value of this work lies in the possibility of applying the developed model in meditation devices and tools, as well as a soundtrack for exercise or fitness routines, and in the fact that the model created is ready for use.

MUSIC FOR MEDITATION. EXISTING METHODS AND APPLICATIONS

1.1 Overview of meditation gadgets

There are now a large number of devices and tools for practicing meditation.

In order to deepen the topic of using music for meditation, a detailed overview of existing applications was conducted.

1. Sensate Relaxation Device

The Sensate relaxation device represents an innovative solution designed to assist individuals in calming their nervous systems and achieving a state of relaxation for both the mind and body. Users simply need to lay down and place the device on their chest. The vibrations emitted by the device are engineered to stimulate the Vagus Nerve, facilitating relaxation and optimizing the nervous system for an ideal state conducive to meditation. See in Figure 1.1.

Figure 1.1 Sensate Relaxation Device

2. Core by Hyperice Meditation Trainer

This innovative gadget functions as a personalized meditation trainer. When held in one's hands, Core sends gentle vibrations to help center attention during meditation. Beyond this, Core provides valuable feedback on the body's physical responses, enhancing the overall meditation experience. See in Figure 1.2.

Figure 1.2 Core by Hyperice Meditation Trainer

3. OPUS Soundbed

This is a vibroacoustic bed. It uses seven vibration channels to send frequencies through the body and bring it to the perfect state of relaxation [1].

See in Figure 1.3.

Figure 1.3 OPUS Soundbed

We could apply the developed model in such meditation devices and tools.

METHODOLOGY

In the project, to generate the meditation music based on heart beat, a methodology was used that includes a well-known machine learning LSTM model approach.

Long Short Term Memory networks – usually just called “LSTMs” – are a special kind of RNN, capable of learning long-term dependencies [2].

The project's methodology revolves around using LSTM to generate drum music sequences.

2.1 Overview of sequential nature of music

1. Temporal patterns:

Drum patterns in music have inherent temporal structures. These include rhythms, beats, and the timing of drum hits, which form a sequence over time.

2. Dependencies:

Certain drum beats or patterns often depend on preceding beats. For example, a drum fill might typically lead into a chorus, or certain rhythms might be characteristic of specific musical genres.

The motivation for building a LSTM based neural network originates from the thought that drum patterns are highly repetitive and can therefore be modeled quite well by LSTMs since they do not rely on recurring themes of any kind.

2.2 How LSTMs Work with Drum Sequences

1. Learning Drum Patterns:

An LSTM model can be trained on a dataset of drum sequences (like the Groove MIDI dataset) to learn the typical patterns and structures in drum music. This involves understanding timing, velocity, and which drum components (snare, bass, hi-hat, etc.) are struck in sequences.

2. Memory cells:

LSTMs have memory cells that can maintain information over longer sequences. This feature is particularly useful in music, where the relevance of a past beat or rhythm might extend over several measures.

3. Predicting the next beat:

After training, the LSTM model can predict the next beat in a sequence based on the previous beats. This predictive capability is the core of generating new drum music.

2.3 Generating Drum Music with LSTMs

1. Starting Seed:

The generation process begins with a seed input, which can be a set of initial beats or rhythms.

2. Sequential Generation:

The model uses this seed to generate the next beat. This new output is then fed back into the model as part of the input for the next step. This loop continues, allowing the LSTM to generate a sequence of drum beats.

3. Variability and Creativity:

LSTMs can introduce variability and creativity in the generated music. While they learn from existing patterns, the stochastic nature of the prediction process can lead to new and interesting drum sequences.

2.4. Application in Synchronizing with Heartbeat

1. Adaptation:

The LSTM model can adapt the tempo or intensity of the generated drum pattern to match the user's heartbeat. This requires additional processing to align the model's output with real-time heartbeat data.

2. Real-time Interaction:

For effective synchronization, the model needs to operate in real-time or near-real-time, adjusting to the changing heartbeat data.

By harnessing LSTMs' ability to handle and predict sequential data, we wish to create a dynamic and responsive system that not only generates rhythmic drum music but also synchronizes it with physiological data, creating a personalized musical experience.

IMPLEMENTATION

3.1 The Dataset

We used the Groove MIDI Dataset [3] for the purpose of building generative models. It is composed of 13.6 hours of aligned MIDI and (synthesized) audio of human-performed, tempo-aligned expressive drumming. The dataset contains 1,150 MIDI files and over 22,000 measures of drumming. But training on that many MIDI files would take too much time, therefore we used a subset of it. The dataset consists of a great variety of genres, reaching from Afrobeat to Pop. When training the models, we used MIDI files from numerous genres to achieve an evenly distributed set of possible outputs. Example of the data you can see in Figure 3.1.

Pitch	Roland Mapping	GM Mapping	Paper Mapping	Frequency
36	Kick	Bass Drum 1	Bass (36)	88067
38	Snare (Head)	Acoustic Snare	Snare (38)	102787
40	Snare (Rim)	Electric Snare	Snare (38)	22262
37	Snare X-Stick	Side Stick	Snare (38)	9696
48	Tom 1	Hi-Mid Tom	High Tom (50)	13145

Figure 3.1 Data in the dataset

3.2 Preprocessing the Data

Music generation

Extracted musical features from MIDI files, focusing on notes, velocities, and timings. This step involved parsing MIDI data to retrieve these elements, which are crucial for music generation. See Python code in Figure 3.2, 3.3.

```
# Define the process_midi_data function
def process_midi_data(midi_info):
        genre = midi_info['genre'].numpy()    # Assuming genre is a TensorFlow tensor
        tempo = midi_info['tempo']
        time_signature = midi_info['time_signature']
        note_list = midi_info['notes']

        processed_notes = []
        for note in note_list:
                note_number = note['note']
                velocity = note['velocity']
                time = note['time']
                processed_note = [note_number, velocity, time]
                processed_notes.append(processed_note)

        processed_notes_array = np.array(processed_notes)
        return genre, tempo, time_signature, processed_notes_array


# Initialize a list to store the processed data
processed_data = []
```

Figure 3.2

```
# Iterate through the dataset
for features in dataset_subset:
        midi_data = features['midi']
        genre = features["style"]["primary"]
        midi_bytes = midi_data.numpy()
        midi_stream = io.BytesIO(midi_bytes)

        try:
                midi_file = mido.MidiFile(file=midi_stream)
                midi_info = {"genre": genre, "tempo": None, "time_signature": None, "notes": []}

                for track in midi_file.tracks:
                        for msg in track:
                                if msg.is_meta:
                                        if msg.type == 'set_tempo':
                                                tempo = mido.tempo2bpm(msg.tempo)
                                                midi_info["tempo"] = tempo
                                        elif msg.type == 'time_signature':
                                                time_signature = f"{msg.numerator}/{msg.denominator}"
                                                midi_info["time_signature"] = time_signature
                                else:
                                        if msg.type in ['note_on', 'note_off']:
                                                midi_info["notes"].append({"note": msg.note, "velocity": msg.velocity, "time": msg.time})

                # Process the extracted MIDI data
                processed_midi_data = process_midi_data(midi_info)
                processed_data.append(processed_midi_data)

        except Exception as e:
                print(f"Error reading MIDI file: {e}")
```

Figure 3.3

3.3 Preparation for Model Input

The processed data was formatted into sequences suitable for LSTM input, preserving the temporal nature of the music. See Python code in Figure 3.4.

```python
def process_batch(dataset_batch):
    input_sequences = []
    output_notes = []

    # `dataset_batch` is a batch of features
    midi_data_batch = dataset_batch['midi']

    for midi_data in midi_data_batch:
        midi_bytes = midi_data.numpy()
        midi_stream = io.BytesIO(midi_bytes)

        try:
            midi_file = mido.MidiFile(file=midi_stream)
            notes = []
            for track in midi_file.tracks:
                for msg in track:
                    if msg.type in ['note_on', 'note_off']:
                        notes.append([msg.note, msg.velocity, msg.time])

            # Create sequences for each MIDI file
            for i in range(0, len(notes) - sequence_length):
                input_seq = notes[i:i + sequence_length]
                output_note = notes[i + sequence_length]
                input_sequences.append(input_seq)
                output_notes.append(output_note)

        except Exception as e:
            print(f"Error reading MIDI file: {e}")

    return input_sequences, output_notes
```

Figure 3.4

3.4 Feature Normalization and one-hot encoding

This step is crucial to ensure that the model receives data within a consistent range, enhancing the learning process. See Python code in Figure 3.5, 3.6.

```
all_notes = [note[0] for sequence in all_input_sequences for note in sequence]
all_velocities = [note[1] for sequence in all_input_sequences for note in sequence]
all_times = [note[2] for sequence in all_input_sequences for note in sequence]


normalized_notes = simple_normalize(np.array(all_notes), 127)    # Normalize notes
normalized_velocities = simple_normalize(np.array(all_velocities), 127)    # Normalize velocities

# For time, determine an appropriate max value based on your data
max_time_value = max(all_times)    # or a predefined value if you have one
normalized_times = simple_normalize(np.array(all_times), max_time_value)    # Normalize times


normalized_dataset = []
for i in range(0, len(all_notes), sequence_length):
        sequence = [[normalized_notes[i + j], normalized_velocities[i + j], normalized_times[i + j]]
                            for j in range(sequence_length)]
        normalized_dataset.append(sequence)

normalized_dataset = np.array(normalized_dataset)
```

Figure 3.5

```
# Initialize the output array with the appropriate shape
num_unique_notes = len(note_to_int)
one_hot_encoded_output = np.zeros((len(all_output_notes), num_unique_notes + 2))    # +2 for velocity and time

# Define the maximum value for time normalization
max_time_value = max([note[2] for note in all_output_notes])    # Or set a predefined maximum time value

for i, note in enumerate(all_output_notes):
        # One-hot encode the note
        note_index = note_to_int[note[0]]
        one_hot_encoded_output[i, note_index] = 1

        # Simple normalization for velocity and time
        normalized_velocity = simple_normalize(note[1], 127)
        normalized_time = simple_normalize(note[2], max_time_value)

        # Store the normalized values
        one_hot_encoded_output[i, -2] = normalized_velocity    # Normalized velocity
        one_hot_encoded_output[i, -1] = normalized_time         # Normalized time
```

Figure 3.6

3.5 Model Architecture

LSTM Network: Chose LSTM due to its effectiveness in learning from sequences. The network consists of several LSTM layers, each followed by dropout layers to prevent overfitting.

Output Layer Design: Implemented separate output layers for predicting notes (categorical output) and continuous features like velocity and time (linear output). See Python code in Figure 3.7.

```
# Input layer
input_layer = Input(shape=(sequence_length, 3))  # 3 features: note, velocity, time

# LSTM layers
x = LSTM(256, return_sequences=True)(input_layer)
x = Dropout(0.3)(x)
x = LSTM(256, return_sequences=True)(x)
x = Dropout(0.3)(x)
x = LSTM(256)(x)
x = Dense(256)(x)
x = Dropout(0.3)(x)

# Separate output layers
output_note = Dense(len(unique_notes), activation='softmax', name='output_note')(x)
output_continuous = Dense(2, activation='linear', name='output_continuous')(x)  # 2 units for velocity and time

# Concatenate outputs
final_output = Concatenate()([output_note, output_continuous])

# Define the model
model = Model(inputs=input_layer, outputs=final_output)
```

Figure 3.7

3.6 Custom Loss Function

Composition: Combined categorical cross-entropy (for note predictions) and mean squared error (for velocity and time). This hybrid approach allows the model to effectively learn both discrete and continuous aspects of music. See Python code in Figure 3.8.

```
def custom_loss(y_true, y_pred):
    # Assuming the first part of y_pred is categorical (one-hot encoded notes) and the rest is continuous
    note_pred = y_pred[:, :22]
    continuous_pred = y_pred[:, 22:]#All 22s sould be replaced by len(unique_notes)

    note_true = y_true[:, :22]
    continuous_true = y_true[:, 22:]

    # Categorical loss (for notes)
    cat_loss = categorical_crossentropy(note_true, note_pred)

    # Mean squared error (for velocity and time)
    mse_loss = mean_squared_error(continuous_true, continuous_pred)

    # Combine the losses
    combined_loss = cat_loss + mse_loss

    return combined_loss
```

Figure 3.8

3.7 Model Training

Training Process: Feed the normalized sequences into the LSTM model. Trained the model over multiple epochs, adjusting parameters based on validation loss. See Python code in Figure 3.9.

```
history = model.fit(X_train, y_train, epochs=50, batch_size=64, validation_data=(X_test, y_test))
```

Figure 3.9

3.8 Model Evaluation

Assessed the model's performance using a held-out test dataset, focusing on its ability to accurately predict musical sequences. See Python code in Figure 3.10.

```
# Evaluate the model
test_loss = model.evaluate(X_test, y_test)

print(f"Test Loss: {test_loss}")
```

```
414/414 [==============================] - 2s 4ms/step - loss: 1.9246
Test Loss: 1.9246209859848022
```

Figure 3.10

3.9 Music Generation and Post-processing

Seed Sequence: Used a predefined MIDI sequence as the starting point for generation, ensuring that the generated music has a base reference.

See Python code in Figure 3.11, 3.12.

```
afrobeat_seed_sequence = midi_to_seed_sequence('/content/drive/My Drive/Colab Notebooks/Python ML/Final project/groove/drummer1/se
blues_seed_sequence = midi_to_seed_sequence('/content/drive/My Drive/Colab Notebooks/Python ML/Final project/groove/drummer4/sessi
country_seed_sequence = midi_to_seed_sequence('/content/drive/My Drive/Colab Notebooks/Python ML/Final project/groove/drummer1/ses
dance_seed_sequence = midi_to_seed_sequence('/content/drive/My Drive/Colab Notebooks/Python ML/Final project/groove/drummer1/sessi
funk_seed_sequence = midi_to_seed_sequence('/content/drive/My Drive/Colab Notebooks/Python ML/Final project/groove/drummer1/sessio
gospel_seed_sequence = midi_to_seed_sequence('/content/drive/My Drive/Colab Notebooks/Python ML/Final project/groove/drummer1/sess
hiphop_seed_sequence = midi_to_seed_sequence('/content/drive/My Drive/Colab Notebooks/Python ML/Final project/groove/drummer1/eval
jazz_seed_sequence = midi_to_seed_sequence('/content/drive/My Drive/Colab Notebooks/Python ML/Final project/groove/drummer1/sessio
rock_seed_sequence = midi_to_seed_sequence('/content/drive/My Drive/Colab Notebooks/Python ML/Final project/groove/drummer1/eval_s
```

Figure 3.11

```
def midi_to_seed_sequence(midi_file, sequence_length=50):
    """
    Converts a MIDI file to a seed sequence formatted like the training data.

    :param midi_file: Path to the MIDI file
    :param sequence_length: The length of the seed sequence
    :return: A seed sequence extracted from the MIDI file
    """
    midi = mido.MidiFile(midi_file)

    notes = []
    for track in midi.tracks:
        for msg in track:
            if not msg.is_meta and msg.type in ['note_on', 'note_off']:
                # Extract note number, velocity, and time
                note_data = [msg.note, msg.velocity, msg.time]
                notes.append(note_data)
                if len(notes) >= sequence_length:
                    break
        if len(notes) >= sequence_length:
            break

    # Pad with zeros if the sequence is shorter than desired
    if len(notes) < sequence_length:
        padding = sequence_length - len(notes)
        notes.extend([[0, 0, 0]] * padding)

    return np.array(notes)
```

Figure 3.12

The seed sequence can be chosen by the user by modifying the line of code below. Genres that are available include rock music, blue music, pop music etc. See Python code in Figure 3.13.

```
################################################################################
seed_sequence = jazz_seed_sequence
################################################################################
```

Figure 3.13

The seed sequence then goes through a normalization process like the training data.

3.10 Heartbeat Synchronization

Incorporated a mechanism to adjust the timing of the generated music to align with a specified heartbeat rate. This involved modifying the time feature of the music based on the desired BPM.

Note that MIDI timing is typically based on ticks per quarter note (TPQN), also known as the MIDI clock. The "time" in a MIDI message indicates the delay from the previous event in ticks, not in seconds or beats. Therefore, to adjust MIDI event timing to a specific BPM, you need to calculate the number of ticks that correspond to a beat at that BPM. This requires knowledge of the MIDI file's TPQN and the tempo (BPM). The formula to convert BPM to ticks per beat is:

$$ticks\_per\_beat = \frac{60}{BPM} \cdot TPQN$$

```
def get_tpqn_from_midi(midi_data):
    midi_stream = io.BytesIO(midi_data.numpy())
    midi_file = mido.MidiFile(file=midi_stream)
    return midi_file.ticks_per_beat

# Iterate through the dataset to find TPQN
for features in dataset.take(1):  # Just checking the first file for example
    midi_data = features['midi']
    tpqn = get_tpqn_from_midi(midi_data)
    print("Ticks Per Quarter Note (TPQN):", tpqn)
    break  # Only check the first file
```

```
Ticks Per Quarter Note (TPQN): 480
```

Figure 3.14

▲By iterating through the dataset, we found that the TPQN of the dataset is 480.

```
def adjust_timing_to_heartbeat(continuous_data, heartbeat_bpm):
    # Adjust only the time component of the continuous data
    beat_duration_in_ticks = (60 / heartbeat_bpm)*480
    #beat_duration_in_ticks = heartbeat_bpm
    continuous_data[:, -1] = beat_duration_in_ticks    # Assuming last column is time
    return continuous_data
```

Figure 3.19

▲To convert BPM to ticks per beat is:

$$ticks\_per\_beat = \frac{60}{BPM} \cdot TPQN = \frac{60}{BPM} \cdot 480$$

```
def generate_heartbeat_synced_music(seed_sequence, heartbeat_bpm, num_steps, sequence_length):
    generated_sequence = seed_sequence
    current_sequence = seed_sequence

    for _ in range(num_steps):
        # Reshape current_sequence for prediction
        current_sequence_reshaped = current_sequence[-sequence_length:].reshape(1, sequence_length, 3)

        # Predict the next part of the sequence
        next_part = model.predict(current_sequence_reshaped)
        print(next_part)

        # Convert one-hot encoded notes to single note values
        next_notes = np.argmax(next_part[:, :22], axis=1)    # Get the index of the '1' in one-hot
        next_continuous = next_part[:, 22:]    # Velocity and time

        # Adjust the timing to match the heartbeat
        adjusted_continuous = adjust_timing_to_heartbeat(next_continuous, heartbeat_bpm)

        # Combine note with continuous features
        adjusted_next_part = np.column_stack([next_notes, adjusted_continuous])

        # Update the generated sequence
        generated_sequence = np.concatenate([generated_sequence, adjusted_next_part])

        # Update the current sequence for the next prediction
        current_sequence = np.concatenate([current_sequence, adjusted_next_part])

    return generated_sequence
```

Figure 3.20

3.11 Denormalization and MIDI Conversion

Transformed the model's output back to the original scale and converted it into MIDI format for playback.

```
def sequence_to_midi(denormalized_sequence, output_file='generated_music.mid'):
        mid = MidiFile()
        track = MidiTrack()
        mid.tracks.append(track)
        track.append(Message('program_change', program=12, time=0))

        last_time = 0
        for note_data in denormalized_sequence:
                note, velocity, time = note_data

                # Convert time to integer as MIDI ticks
                midi_time = int(time * mid.ticks_per_beat)

                # Calculate time delta
                time_delta = midi_time - last_time
                if time_delta < 0:
                        time_delta = 0

                # Create note_on or note_off message
                if velocity > 0:
                        track.append(Message('note_on', note=note, velocity=velocity, time=time_delta))
                else:
                        track.append(Message('note_off', note=note, velocity=0, time=time_delta))

                last_time = midi_time

        mid.save(output_file)
```

Figure 3.21

RESULTS AND DISCUSSIONS

In terms of reaching our goal **”generating rhythmic drum music synchronized with the user's heartbeat to help people promote relaxation, reduce stress, and enhance overall well-being”**, the resulting music beats are considered pretty successful.

The model takes in the heartbeat rate (bpm) of the user, and generates music that has a low beat corresponding to their heartbeat. An example music generated from our project is in the link below:

https://drive.google.com/file/d/1sr9ZPu3ZUkBBmNw2t9qGN8zAEC2FZXoe/view?usp=drive_link

Besides generating physiologically synchronized music, we wanted an additional feature, that is, **the user can choose what kind of music he or she wants to listen to by choosing the corresponding seed sequence.** For example, rock music, blue music, pop music etc. However, the development of this feature wasn’t successful. We have tried different model architectures, but could not really generate versatile genres of beats.

The challenge in generating versatile genres of beats in an AI-based music generation project can be attributed to several factors:

1. Data Diversity and Volume:

Because LSTM training is extremely time consuming, and the Groove dataset is big, we only trained our model with a subset of the whole dataset. This tradeoff may decrease the richness of the training dataset. And if the dataset lacks

diversity in musical genres or is limited in volume, the model may not learn enough variety to generate different genres effectively.

2. Model Complexity and Architecture:

The chosen LSTM model, while powerful for sequence prediction, might not capture the full complexity of musical composition, especially when it comes to capturing the essence of different genres. Exploring more complex or different types of neural networks could yield better genre differentiation.

3. Feature Representation:

The way musical features (notes, velocity, timing) are represented and processed might not fully encapsulate the nuances that differentiate musical genres. For instance, subtle variations in rhythm or harmony that are crucial in genre distinction may not be adequately captured.

4. Heartbeat Synchronization:

The process of synchronizing music with a heartbeat might oversimplify or alter the natural rhythm and flow of different music genres. This synchronization process could overshadow genre-specific rhythmic patterns.

5. Post-Processing and Denormalization:

The steps involved in converting the model's output back to MIDI format, especially the denormalization process, might introduce distortions or inaccuracies, affecting the musical quality and genre fidelity.

6. Subjectivity in Music:

Musical genres are often subjectively interpreted and can vary widely. What constitutes a particular genre can be nuanced and influenced by small stylistic elements that a model might not capture.

In conclusion, enhancing the diversity and volume of the dataset, experimenting with different model architectures, refining feature representation, and carefully managing the post-processing steps are key areas for improvement. Additionally, considering the inherent subjectivity in music, obtaining user feedback and iteratively refining the model based on this feedback could be beneficial.

‘

REFERENCES

1. https://peaceinside.me/cool-meditation-gadgets-and-technologies
2. https://colah.github.io/posts/2015-08-Understanding-LSTMs/
3. https://www.google.com/url?q=https://magenta.tensorflow.org/datasets/groove&sa=D&source=docs&ust=1705234806632844&usg=AOvVaw1Wv6Glet9LvQHoImaZivL2