Deep Learning

DATS 6303

Group No. 1

Final Project Report

Topic: Classical Music Generation

INTRODUCTION

In the realm of musical creation, the fusion of art and technology has opened doors to innovative approaches for generating compositions. Our project delves into the fascinating domain of using Neural Network models to craft classical music—a harmonious blend of tradition and innovation. Motivated by our passion for music and driven by the desire to unravel the intricacies of classical compositions, we embarked on a journey to explore the predictability of musical patterns inherent in this genre. Our endeavor was fueled by the aspiration to push the boundaries of music generation and explore the potential of Neural Network models in creating authentic classical compositions. With the aid of advanced techniques and algorithms, we sought to craft pieces that resonate with the rich legacy of classical music, seamlessly blending tradition with contemporary methodologies. For this endeavor, we leveraged a comprehensive dataset comprising classical music compositions in the MIDI format. MIDI files, renowned for their versatility and expressive capabilities, served as the cornerstone of our project, providing a diverse array of musical motifs and arrangements to fuel our exploration. Central to our approach was the utilization of Long Short-Term Memory (LSTM) models—a cutting-edge neural network architecture renowned for its prowess in sequence modeling and generation. By harnessing the power of LSTM, we aimed to unravel the underlying structures and patterns inherent in classical music, paving the way for the creation of compositions that captivate the imagination and evoke emotions.

PROJECT FLOW

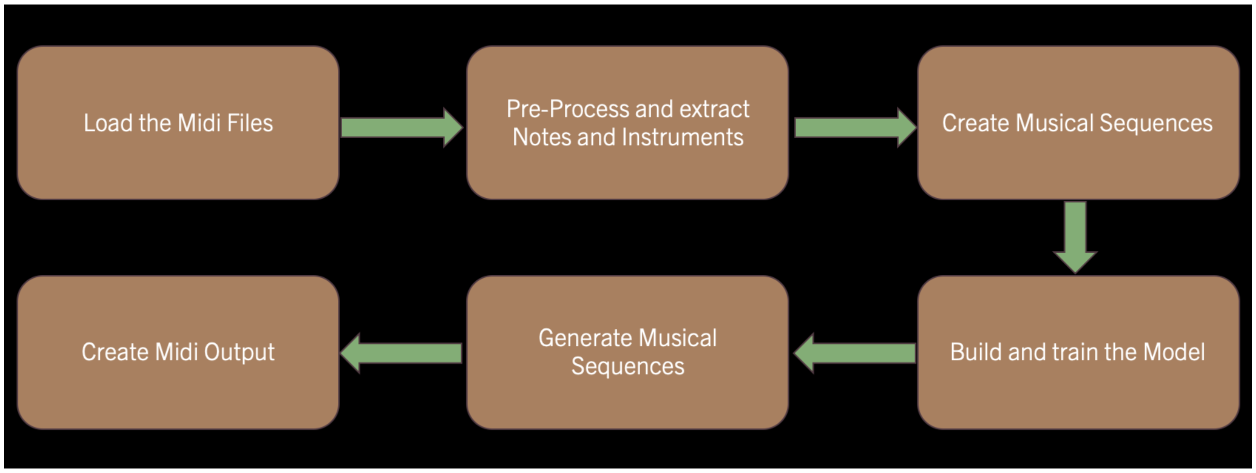

Load the Midi Files
Pre-Process and extract Notes and Instruments
Create Musical Sequences
Build and train the Model
Generate Musical Sequences
Create Midi Output

Figure.1. Project FLow

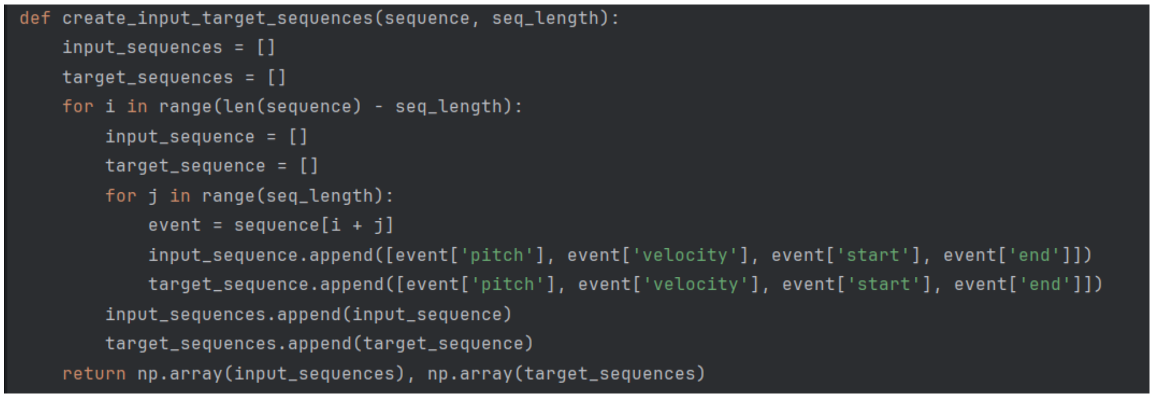
We start by loading MIDI files of a selected composer's classical music compositions. Then, we preprocess these files to extract important musical elements like notes and instruments. Next, we organize these elements into sequential representations, setting the stage for model training. Our neural network model, powered by LSTM, learns temporal dependencies within these sequences during the training phase. Once trained, the model autonomously generates new musical sequences, drawing from its learned patterns. Finally, these generated sequences are converted into MIDI format, resulting in unique classical compositions.

DATA PREPROCESSING

Data preprocessing lays the foundation for any machine learning endeavor, and in our project, it played a pivotal role in understanding and harnessing the intricacies of classical music compositions. With our dataset comprising MIDI files—digitally encoded representations of musical scores—we embarked on a journey to unravel the musical tapestry encapsulated within these files.

Loading MIDI Files for a Specific Composer

The first step in our data preprocessing pipeline involved loading MIDI files corresponding to a specific composer. Leveraging the versatile capabilities of Python libraries such as PrettyMIDI, we traversed through the directory containing MIDI files, selectively retrieving compositions attributed to our chosen composer. When I first tried to load the files it just took in 1 song and wasn't able to append other songs with it, so the model kept training on just the first song of each composer. I had to solve that issue.



```python
def create_input_target_sequences(sequence, seq_length):
    input_sequences = []
    target_sequences = []
    for i in range(len(sequence) - seq_length):
        input_sequence = []
        target_sequence = []
        for j in range(seq_length):
            event = sequence[i + j]
            input_sequence.append([event['pitch'], event['velocity'], event['start'], event['end']])
            target_sequence.append([event['pitch'], event['velocity'], event['start'], event['end']])
        input_sequences.append(input_sequence)
        target_sequences.append(target_sequence)
    return np.array(input_sequences), np.array(target_sequences)
```

Extracting Instruments and Notes Detected Using PrettyMIDI

Once the MIDI files were loaded, we employed PrettyMIDI—a powerful library for MIDI file manipulation—to extract vital information regarding the instruments and notes present in each composition. By dissecting the MIDI files, we gained insights into the orchestration of musical elements, identifying the prominent instrument used—typically the piano—in classical compositions.

Extracting Pitch, Velocity, Start, and End Time for Each Note

Delving deeper, we meticulously extracted crucial attributes of each musical note, including pitch, velocity, start time, and end time. These attributes served as the building blocks for constructing meaningful musical sequences, providing granular insights into the temporal and harmonic nuances of classical compositions.

Defining a Sequence Length

To facilitate the creation of coherent musical sequences, we defined a sequence length—a parameter governing the temporal span of input sequences. This parameter played a crucial role in shaping the dynamics of our model's training process, influencing the depth of contextual information considered during sequence generation. In our case, the sequence length of 30 produced great results.

Creating Input Sequences for Continuous Notes

Armed with a comprehensive understanding of the musical elements present in our dataset, we embarked on the task of creating input sequences for our model. These sequences, composed of 30 continuous notes extracted from MIDI files, formed the raw material upon which our model's training process would operate.

Creating Target Sequences

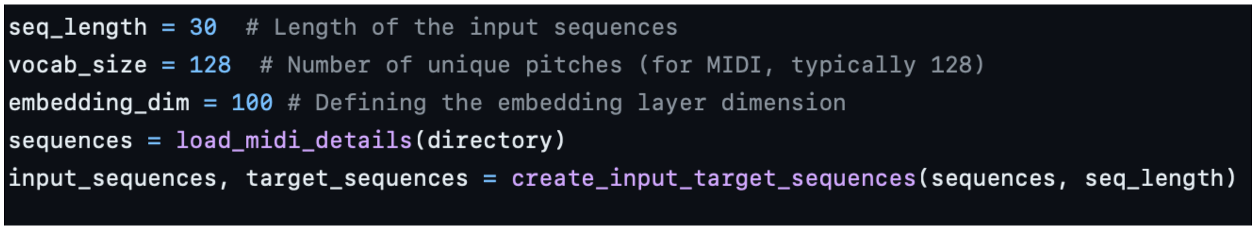
In tandem with input sequence creation, we generated corresponding target sequences—a crucial component in the supervised learning paradigm. The targets were set to be the 31st note a sequence following 30 input notes. These target sequences, aligned with the input sequences, encapsulated the subsequent notes to be predicted by our model, allowing the learning of sequential dependencies within classical compositions.

Converting the Target Sequence to a One-Hot Encoded Vector

As a final preprocessing step, we transformed the target sequences into one-hot encoded vectors—a format conducive to training our model. This encoding scheme facilitated the representation of categorical data, enabling our model to learn and predict the next musical note as a class or category, with great precision.

MODEL 1 (Pitch Classification)

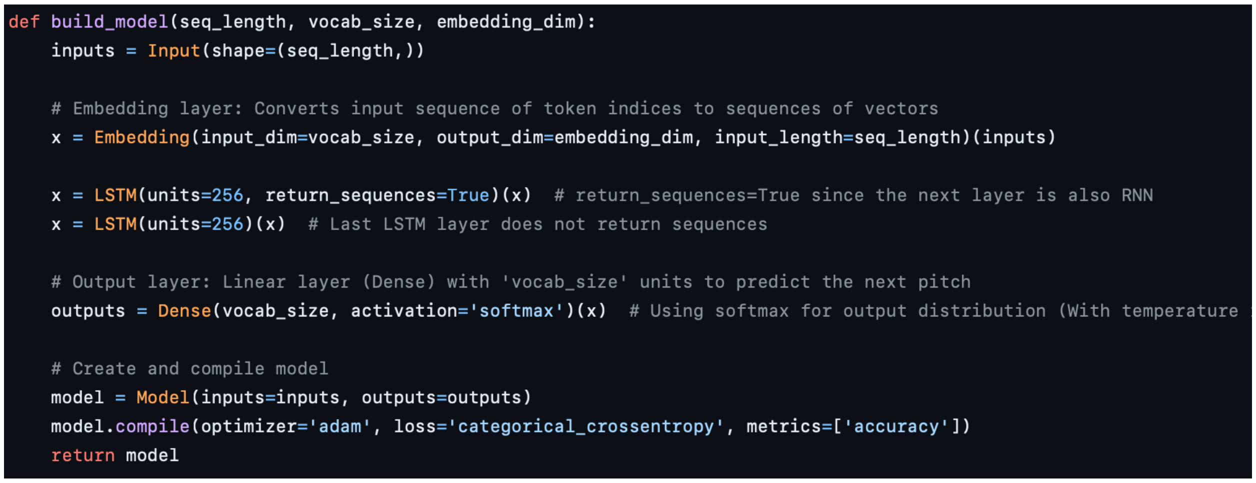
This final code represents a successful attempt to generate classical music using neural networks. The model, based on LSTM architecture, was trained on MIDI files of classical music compositions by Bach. The process involved loading and preprocessing MIDI data to create input and target sequences, followed by building and training the model.



```
seq_length = 30  # Length of the input sequences
vocab_size = 128  # Number of unique pitches (for MIDI, typically 128)
embedding_dim = 100 # Defining the embedding layer dimension
sequences = load_midi_details(directory)
input_sequences, target_sequences = create_input_target_sequences(sequences, seq_length)
```

Figure 2: Setting some critical values

The input sequences, representing sequences of musical notes, were fed into the model to predict the next note in the sequence. The model was trained over multiple epochs to learn the patterns and structures present in the input sequences. Despite challenges such as setting appropriate sequence length and seed values, the model was able to generate musical sequences efficiently and effectively.



```
def build_model(seq_length, vocab_size, embedding_dim):
    inputs = Input(shape=(seq_length,))

    # Embedding layer: Converts input sequence of token indices to sequences of vectors
    x = Embedding(input_dim=vocab_size, output_dim=embedding_dim, input_length=seq_length)(inputs)

    x = LSTM(units=256, return_sequences=True)(x)  # return_sequences=True since the next layer is also RNN
    x = LSTM(units=256)(x)  # Last LSTM layer does not return sequences

    # Output layer: Linear layer (Dense) with 'vocab_size' units to predict the next pitch
    outputs = Dense(vocab_size, activation='softmax')(x)  # Using softmax for output distribution (With temperature

    # Create and compile model
    model = Model(inputs=inputs, outputs=outputs)
    model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])
    return model
```

Figure 3: Model definition

The first layer in our model was an Embedding Layer. Embedding in our model significantly improved the performance by transforming categorical MIDI pitch values into continuous vector representations. This transformation streamlined the learning process by reducing dimensionality and enabling the model to capture the nuanced relationships between musical elements more effectively. As a result, the LSTM model produced more coherent and expressive musical sequences, leading to higher accuracy and better performance overall.

The next 2 layers in our model were LSTM blocks. These layers captured temporal dependencies and musical note relationships, using its ability to remember patterns. Each LSTM cell consists of a forget gate, input gate and output gate. The forget gate determines how much of the past information (or state) should be factored into computing the current state.

$$i_t = \sigma(w_i[h_{t-1}, x_t] + b_i)$$

$$f_t = \sigma(w_f[h_{t-1}, x_t] + b_f)$$

$$o_t = \sigma(w_o[h_{t-1}, x_t] + b_o)$$

$$\begin{aligned} i_t &\rightarrow represents\ input\ gate. \\ f_t &\rightarrow represents\ forget\ gate. \\ o_t &\rightarrow represents\ output\ gate. \\ \sigma &\rightarrow represents\ sigmoid\ function. \\ w_x &\rightarrow weight\ for\ the\ respective\ gate(x)\ neurons. \\ h_{t-1} &\rightarrow output\ of\ the\ previous\ lstm\ block(at\ timestamp\ t-1). \\ x_t &\rightarrow input\ at\ current\ timestamp. \\ b_x &\rightarrow biases\ for\ the\ respective\ gates(x). \end{aligned}$$

The equations above represents how the coefficients for the forget, input and output gates are computed.

$$\tilde{c_t} = tanh(w_c[h_{t-1}, x_t] + b_c)$$

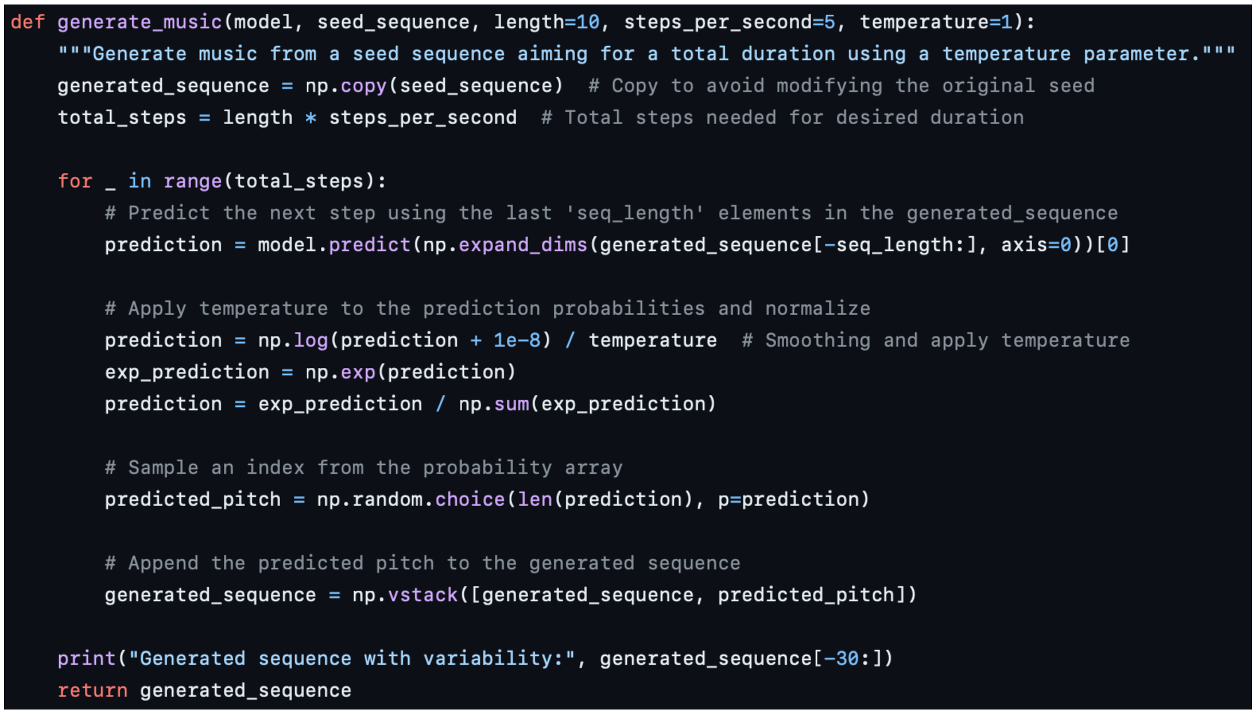
$$c_t = f_t * c_{t-1} + i_t * \tilde{c_t}$$

$$h_t = o_t * tanh(c^t)$$

$$\begin{aligned} c_t &\rightarrow cell\ state(memory)\ at\ timestamp(t). \\ \tilde{c_t} &\rightarrow represents\ candidate\ for\ cell\ state \\ &\quad at\ timestamp(t). \\ &note^*\ others\ are\ same\ as\ above. \end{aligned}$$

These coefficients are used to calculate the cell state and hidden state for each time step.

The softmax activation function played a crucial role in our model by converting the raw output scores into probability distributions over the set of possible MIDI pitch values. This ensured that the output values were normalized and interpretable as probabilities, allowing the model to make more informed decisions about the next pitch in the sequence. By producing a probability distribution, softmax helped the model generate diverse and musically plausible sequences, enhancing the richness and variability of the generated music.



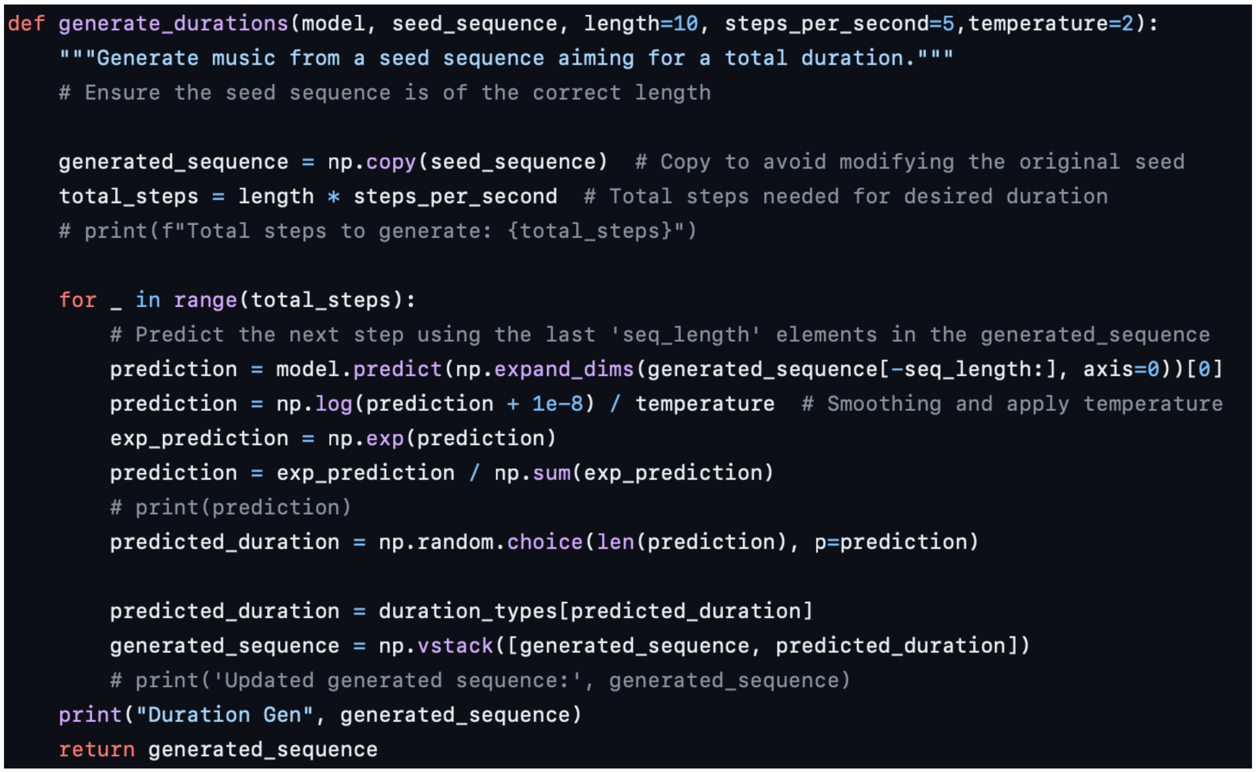
```python
def generate_music(model, seed_sequence, length=10, steps_per_second=5, temperature=1):
    """Generate music from a seed sequence aiming for a total duration using a temperature parameter."""
    generated_sequence = np.copy(seed_sequence)  # Copy to avoid modifying the original seed
    total_steps = length * steps_per_second  # Total steps needed for desired duration

    for _ in range(total_steps):
        # Predict the next step using the last 'seq_length' elements in the generated_sequence
        prediction = model.predict(np.expand_dims(generated_sequence[-seq_length:], axis=0))[0]

        # Apply temperature to the prediction probabilities and normalize
        prediction = np.log(prediction + 1e-8) / temperature  # Smoothing and apply temperature
        exp_prediction = np.exp(prediction)
        prediction = exp_prediction / np.sum(exp_prediction)

        # Sample an index from the probability array
        predicted_pitch = np.random.choice(len(prediction), p=prediction)

        # Append the predicted pitch to the generated sequence
        generated_sequence = np.vstack([generated_sequence, predicted_pitch])

    print("Generated sequence with variability:", generated_sequence[-30:])
    return generated_sequence
```

Figure 4: Function for predicting music

During the generation phase, the model utilized a temperature parameter to add variability and creativity to the generated music. The generated sequences were then converted into MIDI format, producing tangible outputs representing novel classical compositions.

The culmination of our efforts resulted in a remarkable achievement: a maximum accuracy of 90% in generating classical music compositions using neural network models. This substantial accuracy signifies the model's adeptness at discerning intricate patterns and structures inherent in classical music compositions, allowing it to make precise predictions regarding the next musical note based on the input sequences provided. Some of the generated music can be viewed here link

MODEL 2 (Pitch and Duration Classification)



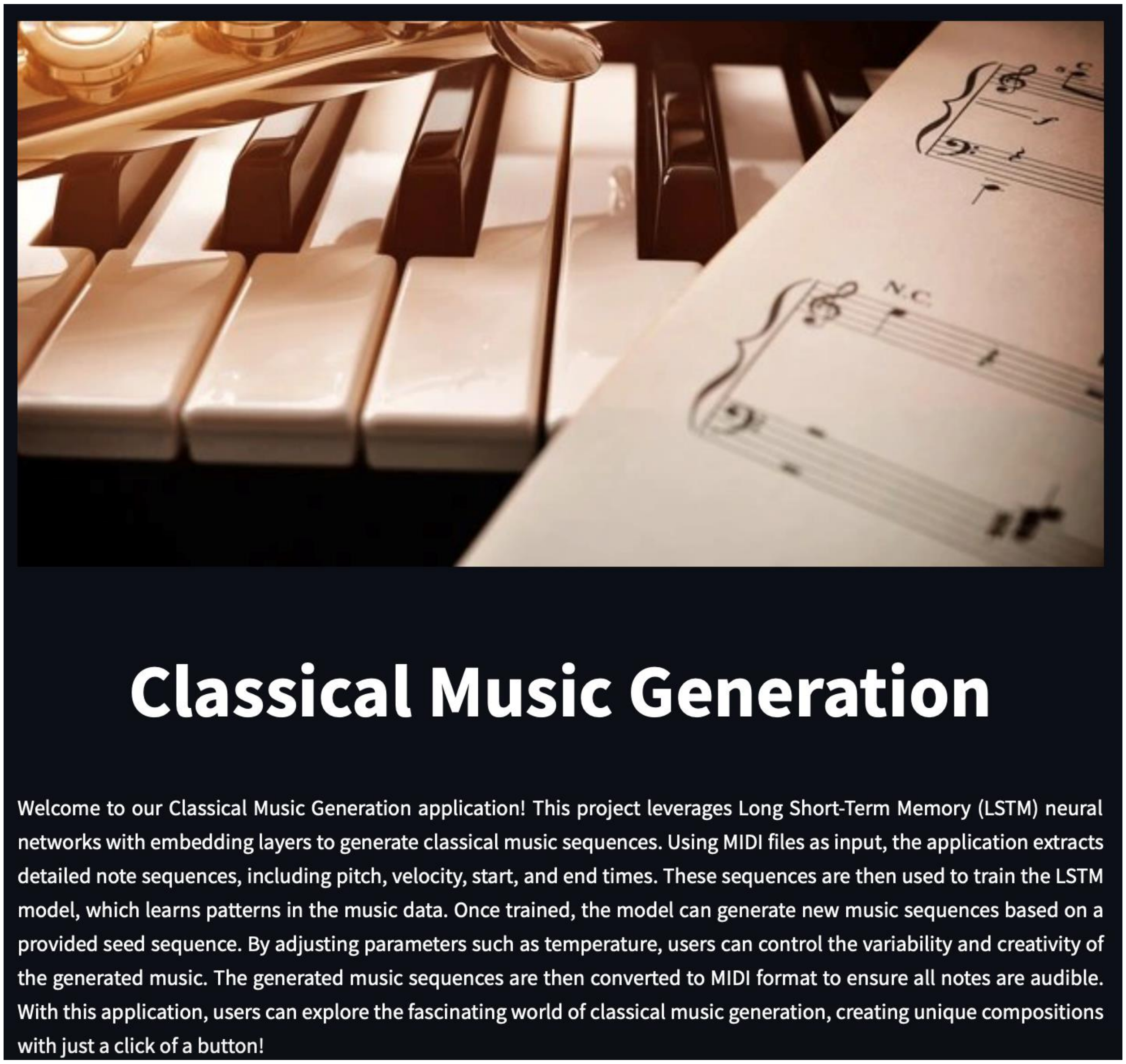
```python
def generate_durations(model, seed_sequence, length=10, steps_per_second=5,temperature=2):
    """Generate music from a seed sequence aiming for a total duration."""
    # Ensure the seed sequence is of the correct length

    generated_sequence = np.copy(seed_sequence)  # Copy to avoid modifying the original seed
    total_steps = length * steps_per_second  # Total steps needed for desired duration
    # print(f"Total steps to generate: {total_steps}")

    for _ in range(total_steps):
        # Predict the next step using the last 'seq_length' elements in the generated_sequence
        prediction = model.predict(np.expand_dims(generated_sequence[-seq_length:], axis=0))[0]
        prediction = np.log(prediction + 1e-8) / temperature  # Smoothing and apply temperature
        exp_prediction = np.exp(prediction)
        prediction = exp_prediction / np.sum(exp_prediction)
        # print(prediction)
        predicted_duration = np.random.choice(len(prediction), p=prediction)

        predicted_duration = duration_types[predicted_duration]
        generated_sequence = np.vstack([generated_sequence, predicted_duration])
        # print('Updated generated sequence:', generated_sequence)
    print("Duration Gen", generated_sequence)
    return generated_sequence
```

Figure 5: Generating music with tempo

In this version, we extended our model to predict both pitch and duration of musical notes. Similar to predicting pitches, the durations were set up to be distinct classes. We introduced a one-hot encoding scheme for duration prediction and built a separate LSTM model to handle duration prediction. Finally, we updated the MIDI generation function to incorporate predicted durations alongside pitch values. These modifications allowed us to experiment with diverse generation settings, enhancing the model's capabilities despite not achieving perfect performance by lacking in rhythmic depth. A generated result can be found here

STREAMLIT APP

In our Streamlit app, we've integrated key features to facilitate classical music generation and exploration. Upon launching the app, users are greeted with an image, title, and introduction, setting the stage for their experience. The app's central functionality revolves around two dropdown menus: the first allows users to select a composer for training music data, while the second enables the selection of a composer for generating music predictions.

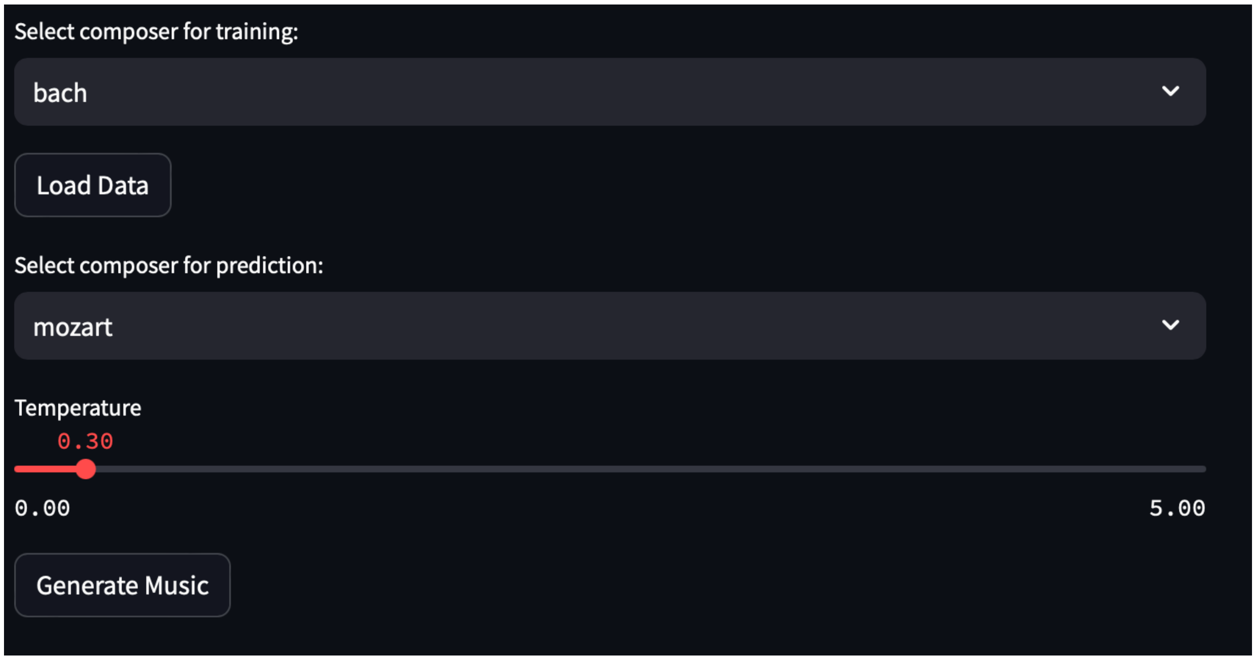


Classical Music Generation

Welcome to our Classical Music Generation application! This project leverages Long Short-Term Memory (LSTM) neural networks with embedding layers to generate classical music sequences. Using MIDI files as input, the application extracts detailed note sequences, including pitch, velocity, start, and end times. These sequences are then used to train the LSTM model, which learns patterns in the music data. Once trained, the model can generate new music sequences based on a provided seed sequence. By adjusting parameters such as temperature, users can control the variability and creativity of the generated music. The generated music sequences are then converted to MIDI format to ensure all notes are audible. With this application, users can explore the fascinating world of classical music generation, creating unique compositions with just a click of a button!

Figure 6: Title and Intro in app

Once a composer is selected for training, clicking the "Load Data" button triggers the loading of the corresponding trained model, which has been pre-trained and stored for each composer. This ensures that the app is equipped with the appropriate model for generating predictions based on the selected composer's style and patterns.


Select composer for training:

bach

Select composer for prediction:

mozart

Temperature

0.30

0.00 5.00

Figure 7: Main functions in app

Following the model loading step, users can choose a composer from the second dropdown menu. This selection decides the initial seed of notes from which the model generated music. Upon selection, the app is primed to generate music predictions based on the loaded model. To initiate the music generation process, users simply need to click the "Generate Music" button, prompting the app to produce an audio representation of the predicted music.

Additionally, we've added a slider for adjusting the "temperature," a parameter that controls the variability and novelty in the generated music. The slider ranges from 1 to 5 with increments of 0.1, allowing users to fine-tune the creativity of the music generation process and explore new musical patterns.

However, we encountered a challenge with displaying the audio output within the app. Since our music is in .mid format, which Streamlit does not support for direct playback, we had to devise a workaround. Specifically, we implemented a conversion process to transform the .mid files into .wav format, which is compatible with Streamlit's audio playback capabilities. Despite this hurdle, the app remains functional and offers users a seamless experience for exploring and generating classical music compositions.

CONCLUSION

In conclusion, our project offers valuable insights and suggests several promising directions for future research in neural network-based classical music generation. One avenue for exploration involves experimenting with diverse architectures, including different LSTM configurations and alternative models such as Variational Autoencoders (VAEs) or Generative Adversarial Networks (GANs). By diversifying model architectures, we can potentially enhance the capabilities of music generation systems to produce more diverse and complex compositions. Additionally, there is room for improvement in data representation techniques, where incorporating more sophisticated methods, such as encoding additional musical features or utilizing different embedding strategies, could lead to richer musical representations and more nuanced compositions. Another promising area is the incorporation of attention mechanisms, which could enable models to focus on relevant parts of the input sequence, capturing long-range dependencies more effectively and potentially improving the coherence and structure of generated music. Lastly, extending the model's ability to generate longer sequences could lead to the creation of more intricate and elaborate musical compositions, capturing broader musical structures and offering greater creative possibilities. Overall, these avenues for future work hold promise for advancing the field of neural network-based classical music generation and pushing the boundaries of what is possible in algorithmic composition.

REFERENCES

[1]. https://github.com/andfanilo/streamlit-midi-to-wav
[2]. https://medium.com/@alexissa122/generating-original-classical-music-with-an-lstm-neural-network-and-attention-abf03f9ddcb4
[3]. https://arxiv.org/abs/1901.04696
[4]. https://arxiv.org/abs/2012.01231
[5]. https://www.proquest.com/docview/2678689154?pq-origsite=gscholar&fromopenview=true&sourcetype=Dissertations%20&%20Theses
[6]. https://www.andrews.edu/services/honors/research/papers/nathaniel-patterson_symposium-presentation.pdf
[7]. https://medium.com/analytics-vidhya/music-generation-with-lstm-based-rnn-3fa967bc1f37
[8]. https://towardsdatascience.com/lstms-for-music-generation-8b65c9671d35
[9]. https://www.hindawi.com/journals/cin/2022/8336616/
[10]. https://towardsdatascience.com/pyzzicato-piano-instrumental-music-generation-using-deep-neural-networks-ed9e89320bf6
[11]. https://www.mdpi.com/2076-3417/13/7/4543
[12]. https://discuss.streamlit.io/t/run-streamlit-from-pycharm/21624
[13]. https://github.com/andfanilo/streamlit-midi-to-wav