MATH 494A Final Project

Yifan Wu

April 20, 2023

Abstract

In this project, our objective is to utilize an LSTM neural network to predict a sequence of notes for the popular song "Jingle Bells," using the original MIDI file as training data. By converting the MIDI file into a suitable format, we extract essential musical attributes and normalize them for model input. Our LSTM model is designed to capture the temporal patterns present in the music data, enabling accurate prediction of subsequent notes. We evaluate the performance of the model by comparing the generated notes to the actual song and converting the predicted notes into MIDI format for auditory assessment, showcasing the model's capability in producing musically coherent and precise predictions.

1 Introduction and Overview

Music generation and prediction is a captivating area of research, with applications spanning entertainment, education, and the creative arts. Long Short-Term Memory (LSTM) neural networks have demonstrated exceptional success in various fields, including time series forecasting, natural language processing, and sequential data modeling. This positions LSTM networks as an ideal choice for modeling complex patterns present in musical compositions.

Although the primary focus of this project is forecasting notes in "Jingle Bells," the methods employed here can potentially be extended to other musical compositions. Understanding and exploring the adaptability of LSTM networks in handling different musical styles and structures presents a broader perspective of their applications.

This study illustrates the power of LSTM networks in music prediction and generation, paving the way for further research and applications in creative domains and providing insights for developing advanced models in other sequential data-based disciplines. By showcasing the capabilities of LSTM networks in predicting musical notes, we aim to inspire future studies that delve deeper into the potential of these networks in diverse scientific and engineering domains.

2 Theoretical Background

2.1: Universal Function Approximators

Universal Function Approximators revolves around the concept of approximating continuous functions using specific classes of functions. The foundational work in this area was done by Cybenko (1989), who demonstrated that any continuous function could be approximated to arbitrary accuracy using linear combinations of sigmoid functions. Sigmoid functions are a class of monotone increasing functions that exhibit an 'S' shape in their graph. Cybenko's theorem, known as the Universal Approximation Theorem,

states that for any continuous function $f: [0,1]^M \rightarrow R.$ it can be approximated to arbitrary accuracy in the $||.||_1$ norm by functions of the form:

$$g(x) = \sum_{k=1}^{n} v_k \sigma(W_k * x + b_k) \qquad \textbf{[2].}$$

Where $n \geq 1, W_k \in R^M . v_k, b_k \in R \; and \; \sigma \; is \; a \; sigmoid \; function$ [1]. Over time, researchers have expanded upon Cybenko's original result, and it is now understood that any non-polynomial function can be used as an activation function for approximating continuous functions. This has led to the development of various universal approximation theorems that provide the theoretical foundation for the approximation properties of modern neural networks.

The accuracy of the approximation can be improved by increasing the width (n) or the depth (d) of the function. Width refers to the hidden dimension inside the activation function, while depth refers to the number of compositions of functions in the form of:

$$g(x) = V * \sigma(Wx + b) \qquad \textbf{[3]}$$

Where $W \in R^{nxM}, \; V \in R^{Nxn}, b \in R^n \; and \; \sigma$ is referred to as an activation function in the context of neural networks.

2.2: Neural Network & Loss Function

In the context of neural networks, we use activation functions (e.g., sigmoid or ReLU) to create multiple layers of neurons that form the network. A single-layer NN is given by:

$$y = \sigma(Wx + b) \qquad \textbf{[4]}$$

Where $\sigma$is the activation function, W is the weight matrix, and b is the bias vector. For a deeper NN, we can compose multiple layers as follows:

$$y = \sigma_d(W_d * \sigma_{d-1}(W_{d-1} \ldots . . (W_2 * \sigma_1(W_1 x + b_1) + b_2) + \cdots .) + b_{d-1}) + b_d) \qquad \textbf{[5]}$$

The goal of training a NN is to minimize the loss function (e.g., mean squared error) between the desired output and the NN's output. This is achieved by adjusting the network's weights (W) and biases (b) using an optimization algorithm, such as Gradient Descent. The loss function L can be defined as:

$$L = ||X - g(Y)||_{mse} \coloneqq \frac{1}{2k} \sum_{k=1}^{k} ||y_k - g(X_k)||_2^2 \qquad \textbf{[6]}$$

where K is the number of data points, y_k is the desired output, and $g(X_k)$ is the NN's output.

Gradient Descent is an iterative optimization algorithm that updates the network's weights and biases by computing the gradients of the loss function with respect to these parameters. The updates are performed as follows:

$$W_j^{new} = W_j^{old} - \Upsilon \frac{\partial L}{\partial W_j} \qquad \textbf{[7]}$$

$$b_j^{new} = b_j^{old} - \Upsilon \frac{\partial L}{\partial b_j}$$

Where Υ is the learning rate, and j represents the index of the layer. The network is trained by minimizing a loss function using an optimization algorithm, such as Gradient Descent, which updates the weights and biases based on the gradients of the loss function.

2.3: Recurrent Neural Networks (RNNs)&Long Short-Term Memory (LSTM)

Recurrent Neural Networks (RNNs) are particularly suitable for modeling sequential data, such as music, compared to Feedforward Neural Networks (FNNs). FNNs are unable to capture temporal dependencies between input data points due to their static structure, while RNNs maintain an internal state that allows them to process input sequences in a dynamic manner. This ability to model the temporal patterns in sequential data makes RNNs more effective in music forecasting tasks.

However, standard RNNs suffer from vanishing and exploding gradient problems, which hinder their ability to learn long-term dependencies in the data. These issues arise during the backpropagation process when gradients become either too small or too large, causing the model's learning to be either too slow or unstable.

Long Short-Term Memory (LSTM) networks are a special type of RNN designed to handle long-term dependencies in sequential data effectively. LSTMs address the vanishing and exploding gradient issues by introducing a memory cell and three gating mechanisms: the input gate, the output gate, and the forget gate. The memory cell stores information over time, while the gates control the flow of information into and out of the cell. These gating mechanisms allow LSTMs to selectively update, remember, and access information, making them capable of learning long-term dependencies in the data.

$$i_t = \sigma(W_i * [h_{t-1}, x_t] + b_i) \quad \textbf{[8]}$$

$$f_t = \sigma(W_f * [h_{t-1}, x_t] + b_f) \quad \textbf{[9]}$$

$$o_t = \sigma(W_o * [h_{t-1}, x_t] + b_o) \quad \textbf{[10]}$$

$$g_t = tanh(W_g * [h_{t-1}, x_t] + b_g) \quad \textbf{[11]}$$

$$c_t = f_t \odot c_{t-1} + i_t \odot g_t \quad \textbf{[12]}$$

$$h_t = o_t \odot \tanh(c_t) \quad \textbf{[13]}$$

where i_t , f_t , o_t represent the input, forget, and output gates, respectively; g_t *is the candidate memory cell;* c_t *is the updated memory cell;* h_t *is the updated hidden state where W and b are the weight matrices and bias vectors for each gate and* $\odot$ *denotes the element-wise multiplication*

3 Algorithm Implementation and Development

3.1 Data Preparation

We preprocess the 'jingle-bells-keyboard.mid' MIDI file, extracting note information, including pitch, velocity, and start time. The extracted notes are represented as tuples (pitch, velocity, start_time), where pitch is the note's pitch, velocity is its intensity, and start_time is the time the note starts. After sorting the notes by start time, we normalize the start times by dividing them by the maximum start time in the dataset. We then organize the notes into an (m, 3) matrix, where m is the total number of notes in the MIDI file, and each row in the matrix represents a single note with its pitch, velocity, and normalized start time. This matrix is used as the basis for generating input and target sequences for the neural network.

3.2 Sequence Generation and Data Splitting

We create input and target sequences for the neural network using input lengths of 110 notes and target lengths

of 37 notes. These sequence lengths can be adjusted depending on the desired granularity or complexity of the model. For example, increasing the input length to 220 notes or target length to 60 notes could provide more context for the model to capture musical patterns, but it may also increase the computational requirements for training. It is important to experiment with different sequence lengths to find the optimal balance between model performance and computational efficiency.

Once the input and target sequences are generated, we split the dataset into training and validation sets with an 80-20 ratio using the train_test_split function from the sklearn library. This separation allows us to evaluate the model's performance on unseen data, ensuring that it generalizes well to new musical patterns and does not overfit the training data.

3.3 Neural Network Model Initialization

We initialize a Long Short-Term Memory (LSTM) neural network model using TensorFlow. The model consists of the following layers:

An LSTM layer with 64 units.

A RepeatVector layer for repeating the LSTM output.

Another LSTM layer with 64 units, set to return sequences.

A TimeDistributed Dense layer with a linear activation function.

The model is compiled using the Mean Squared Error (MSE) loss function and the Adam optimizer. The Adam optimizer is an advanced variant of the stochastic gradient descent (SGD) algorithm, which is widely used for optimizing neural networks. It incorporates adaptive learning rate mechanisms, allowing for individual adjustment of learning rates for each weight in the model. This results in faster convergence and more robust performance compared to traditional SGD.

Additionally, the Adam optimizer combines the benefits of two other popular optimization algorithms, AdaGrad and RMSProp, by using both first-order momentum and second-order adaptive learning rates. This combination helps mitigate problems such as vanishing gradients and noisy weight updates, ultimately leading to more effective and efficient training of the neural network model.

In this implementation, we utilize a linear activation function in the TimeDistributed Dense layer. A linear activation function simply maintains the input values without any transformation, making it suitable for regression tasks, such as predicting the pitch, velocity, and start time of MIDI notes. This choice of activation function allows the model to output continuous values, which are necessary for accurately predicting the properties of the generated MIDI notes.

3.4 Model Training

In the training process, we train the model for 10,000 epochs with a batch size of 8. We provide the training and validation datasets to the model, and the model learns to predict the next 37 notes given an input sequence of 110 notes.

Batch size is an important parameter in the training process, as it determines the number of samples used to compute the gradient in each update. Using a smaller batch size provides a more accurate estimate of the gradient, but requires more iterations to process the entire dataset, leading to increased computational costs. On the other hand, a larger batch size can result in faster training, but the gradient estimation may be less accurate, potentially leading to slower convergence or suboptimal solutions.

In our implementation, we use a batch size of 8, which provides a good balance between accurate gradient estimation and computational efficiency. The model is trained for a total of 10,000 epochs, allowing it to learn the complex relationships between the input and target sequences.

During training, the model computes the Mean Squared Error (MSE) loss function, which measures the average squared differences between the predicted and actual output data. The MSE loss function is commonly used for regression tasks, as it emphasizes larger errors and is sensitive to outliers. This encourages the model to learn accurate predictions, as it seeks to minimize the MSE loss.

The Adam optimizer is employed to minimize the MSE loss through the process of gradient descent. At each training step, the optimizer computes the gradients of the loss with respect to the model's trainable weights and updates the weights accordingly. This iterative process of computing gradients and updating weights continues throughout the training, ultimately leading to an improved model capable of accurately predicting the next 37 notes based on a given input sequence of 110 notes.

3.5 Music Generation

After training, we use the trained model to generate new music. We use an existing input sequence as the seed and generate 5 new notes using the model's predictions. The generated sequence is post-processed to obtain note values, velocities, and start times in their original format.

3.6 Evaluation

To evaluate the model's performance, we compare the true notes in the validation dataset with the predicted notes generated by the model. We plot the comparison, showcasing the model's ability to predict new notes based on the given input sequence.

4 Computational Results

4.1 hidden_layers=2, neurons_per_layer=64: target_length = 8 notes.

In the following visualization, we showcase the architecture of our LSTM model alongside a comparison between the predicted and original notes for a sequence of 8 notes. This enables us to evaluate the model's performance in accurately predicting the notes based on the given input sequence while also providing an overview of the model structure.

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 lstm (LSTM)                 (None, 64)                17408

 repeat_vector (RepeatVector  (None, 8, 64)            0
 )

 lstm_1 (LSTM)               (None, 8, 64)             33024

 time_distributed (TimeDistr  (None, 8, 3)             195
 ibuted)

=================================================================
Total params: 50,627
Trainable params: 50,627
Non-trainable params: 0
_________________________________________________________________
```

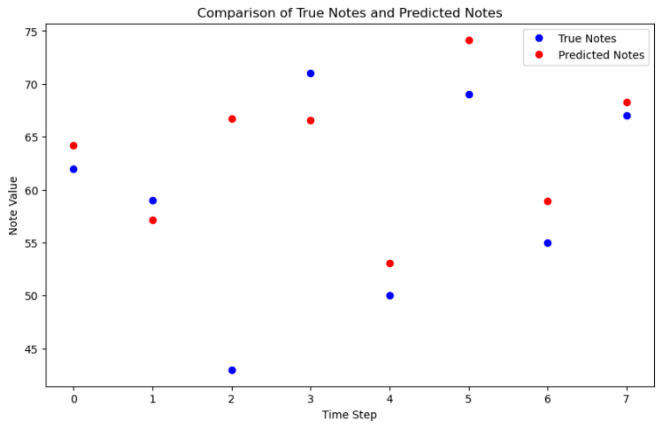

Figure 1: left plot is my LSTM model of 8 notes; the right is the comparison of 8 notes.

```
2/2 [==============================] - 0s 33ms/step - loss: 0.0013 - val_loss: 42.4554
Epoch 14992/15000
2/2 [==============================] - 0s 27ms/step - loss: 0.0012 - val_loss: 42.4781
Epoch 14993/15000
2/2 [==============================] - 0s 38ms/step - loss: 0.0013 - val_loss: 42.4729
Epoch 14994/15000
2/2 [==============================] - 0s 35ms/step - loss: 0.0012 - val_loss: 42.4917
Epoch 14995/15000
2/2 [==============================] - 0s 28ms/step - loss: 0.0014 - val_loss: 42.5803
Epoch 14996/15000
2/2 [==============================] - 0s 32ms/step - loss: 0.0013 - val_loss: 42.5757
Epoch 14997/15000
2/2 [==============================] - 0s 34ms/step - loss: 0.0014 - val_loss: 42.4806
Epoch 14998/15000
2/2 [==============================] - 0s 38ms/step - loss: 0.0013 - val_loss: 42.4689
Epoch 14999/15000
2/2 [==============================] - 0s 32ms/step - loss: 0.0012 - val_loss: 42.5139
Epoch 15000/15000
2/2 [==============================] - 0s 34ms/step - loss: 0.0014 - val_loss: 42.4963
```

Figure 2: Loss function Result of 8 notes

The prediction of 8 notes is quite satisfactory, considering the resulting plot and loss value of 0.0014 after 150,000 epochs. In Appendix A, a 3D plot illustrating the prediction performance, as well as a link to the GitHub repository containing the actual output MIDI file, can be found for further reference and evaluation.

4.2: hidden_layers=2, neurons_per_layer=64: target_length = 37 notes.

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 lstm (LSTM)                 (None, 64)                17408

 repeat_vector (RepeatVector  (None, 37, 64)           0
 )

 lstm_1 (LSTM)               (None, 37, 64)            33024

 time_distributed (TimeDistr  (None, 37, 3)            195
 ibuted)

=================================================================
Total params: 50,627
Trainable params: 50,627
Non-trainable params: 0
_________________________________________________________________
```

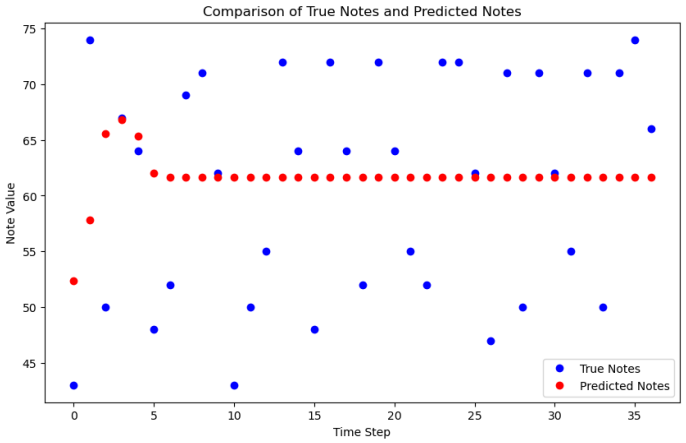

Figure 3: left plot is my LSTM model of 37 notes; the right is the comparison of 37 notes.

```
1/1 [==============================] - 0s 52ms/step - loss: 24.8338 - val_loss: 32.1375
Epoch 14992/15000
1/1 [==============================] - 0s 57ms/step - loss: 24.8167 - val_loss: 32.1360
Epoch 14993/15000
1/1 [==============================] - 0s 56ms/step - loss: 24.8082 - val_loss: 32.1239
Epoch 14994/15000
1/1 [==============================] - 0s 67ms/step - loss: 24.8233 - val_loss: 32.1566
Epoch 14995/15000
1/1 [==============================] - 0s 57ms/step - loss: 24.8514 - val_loss: 32.1068
Epoch 14996/15000
1/1 [==============================] - 0s 67ms/step - loss: 24.9222 - val_loss: 32.1372
Epoch 14997/15000
1/1 [==============================] - 0s 59ms/step - loss: 24.8522 - val_loss: 32.1805
Epoch 14998/15000
1/1 [==============================] - 0s 56ms/step - loss: 24.9202 - val_loss: 32.1180
Epoch 14999/15000
1/1 [==============================] - 0s 53ms/step - loss: 24.8297 - val_loss: 32.1181
Epoch 15000/15000
1/1 [==============================] - 0s 38ms/step - loss: 24.8436 - val_loss: 32.1898
```

Figure 4: Loss function Result of 37 notes.

After conducting 150,000 epochs on the model, we observed a gradual decrease in the loss. However, as the results indicate, the decrease in loss is slow and may not be satisfactory due to the significant increase in the target notes to 37 instead of 8. This larger target length could potentially impact the model's performance, leading to the observed higher loss value even after 150,000 epochs. To

improve the model's performance and reduce the loss further, we may need to consider other strategies, such as increasing the number of epochs, adjusting the model architecture, or modifying the learning rate to better capture the nuances in the data and ultimately achieve our desired goal. In Appendix A, we have included a 3D plot illustrating the prediction performance. The plot shows that for the first 8 notes, the results are quite good, but beyond that, the predicted notes essentially become a horizontal line, which indicates that the model consistently generates only one note. This observation further highlights the need for refining our model to enhance its performance on the larger target length.

5 Summary and Conclusions

In this project, we explored the use of LSTM neural networks for music generation, focusing on predicting musical notes from input sequences. Our model achieved satisfactory predictions for the first 8 notes, demonstrating its potential in generating short sequences of music. However, for a target length of 37 notes, the results were less satisfactory, indicating a need for refining the model to enhance its performance in generating longer musical sequences.

Possible directions for improvement include increasing the number of epochs, adjusting the model architecture, or exploring alternative data preprocessing techniques. Additionally, future research could investigate the use of attention mechanisms or other advanced neural network architectures to better capture temporal dependencies in the data.

This study showcases the potential of neural networks in music generation and highlights the importance of continued research to improve prediction accuracy and extend the forecasting horizon. By refining our models and techniques, we can contribute to the development of more sophisticated music generation systems that can inspire and enrich the creative process for musicians and composers.

References

[1] Jose Nathan Kutz. *Data-Driven Modeling scientific computation: methods for complex systems and big data. Oxford University Press, 2013*

Appendix A Additional Figure

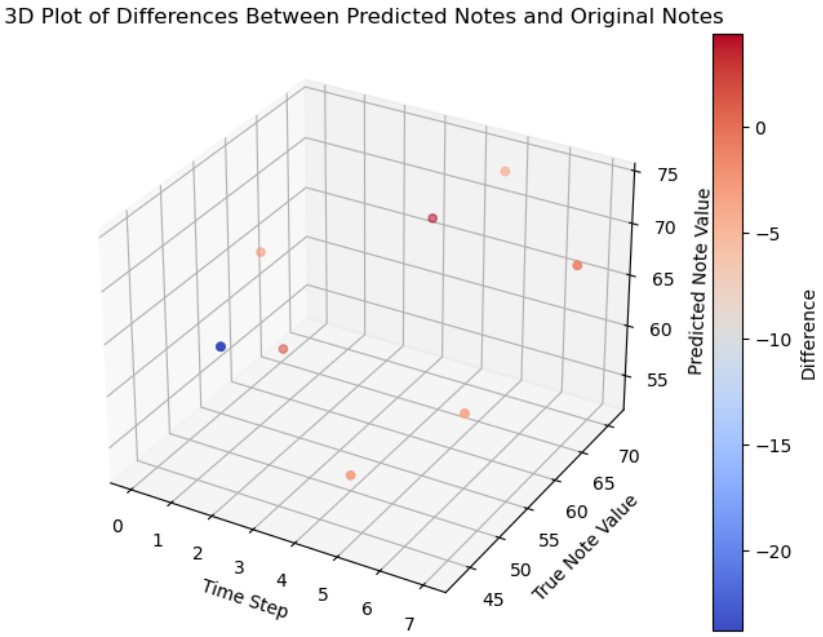

Figure 5: The image shows the 3D plot of notes 8.
Github link of predicted MIDI file:
https://github.com/YifanWu1993/Data_Driven/blob/main/Final%20project:%20Forecast%20piano%20music%20by%20LSTM%20NN/8notes_predictied.midi

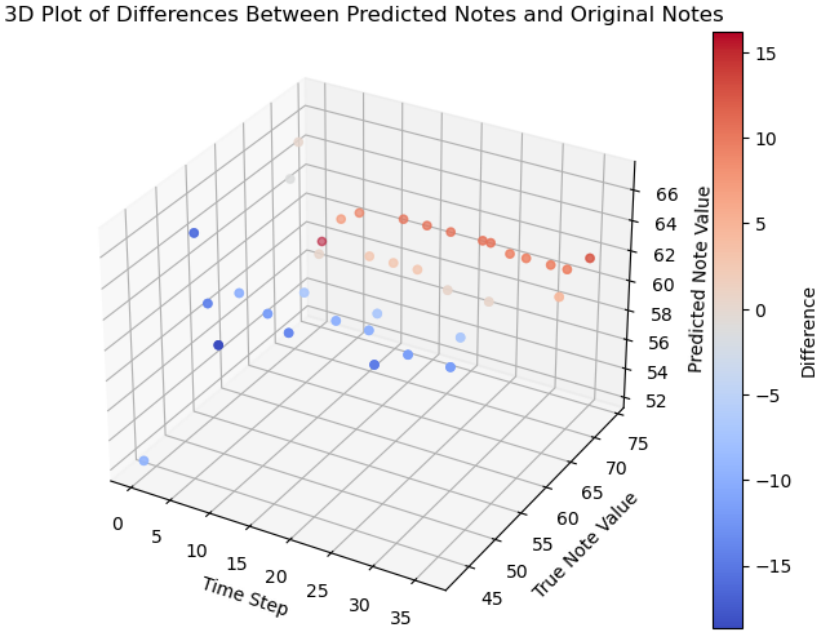

Figure 6: The image shows the 3D plot of notes 37.

Appendix B Python Functions

1: create_lstm_model(input_shape, lstm_units, output_shape)
model.add(tf.keras.layers.Dense(4))

2: Calculate the mean squared error loss. mean_squared_error

3: model.fit(X_train, y_train, epochs=epochs, batch_size=batch_size, validation_data=(X_val, y_val))

4: model.predict(input_sequence)

5: from time import time. t0 = time()

Appendix C Python Code

```
import mido
from mido import MidiFile, MidiTrack, Message

def preprocess_midi(filename):
    mid = MidiFile(filename)
    notes = []
    time = 0

    for track in mid.tracks:
        for msg in track:
            time += msg.time

            if msg.type == 'note_on':
                velocity = msg.velocity / 127.0
                notes.append((msg.note, velocity, time))

    # Sort notes by start time
    notes.sort(key=lambda x: x[2])

    # Normalize start times
```

```
    max_start_time = max(notes, key=lambda x: x[2])[2]
    normalized_notes = [(note, velocity, start_time / max_start_time) for note,
velocity, start_time in notes]

    return normalized_notes

filename = 'jingle-bells-keyboard.mid'
preprocessed_data = preprocess_midi(filename)

import numpy as np

def create_sequences(data, input_length, target_length):
    input_sequences = []
    target_sequences = []

    for i in range(0, len(data) - input_length - target_length, input_length):
        input_sequences.append(data[i:i+input_length])
        target_sequences.append(data[i+input_length:i+input_length+target_length])

    return np.array(input_sequences), np.array(target_sequences)

input_length = 35
target_length = 8

input_sequences, target_sequences = create_sequences(preprocessed_data,
input_length, target_length)

len(preprocessed_data)

import numpy as np
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import LSTM, Dense, TimeDistributed
from tensorflow.keras.layers import RepeatVector
```

```
# Define the model architecture
def create_lstm_model(input_shape, lstm_units, output_shape):
    model = Sequential()
    model.add(LSTM(lstm_units, input_shape=input_shape))
    model.add(RepeatVector(output_shape[0]))
    model.add(LSTM(lstm_units, return_sequences=True))
    model.add(TimeDistributed(Dense(output_shape[1], activation='linear')))

    # Compile the model
    model.compile(loss='mean_squared_error', optimizer='adam')

    return model

# Set the model parameters
input_shape = input_sequences.shape[1:]
lstm_units = 64

# Create the model
model = create_lstm_model(input_shape, lstm_units, target_sequences.shape[1:])

# Print a summary of the model architecture
model.summary()

from sklearn.model_selection import train_test_split

# Split the data into training and validation sets
X_train, X_val, y_train, y_val = train_test_split(input_sequences, target_sequences,
test_size=0.2, random_state=42)

# Set the training parameters
epochs = 15000
batch_size = 8

# Train the model
history = model.fit(
```

```
    X_train,
    y_train,
    epochs=epochs,
    batch_size=batch_size,
    validation_data=(X_val, y_val)
)


def postprocess_notes(notes):
    max_start_time = max(notes, key=lambda x: x[2])[2]
    return [(note, int(velocity * 127), start_time * max_start_time) for note, velocity,
start_time in notes]

# Use an existing sequence as the seed
seed_sequence = input_sequences[0]

# Number of notes to generate
num_notes_to_generate = 5

# Generate new notes
generated_sequence = seed_sequence.copy()
for _ in range(num_notes_to_generate):
    input_sequence = np.expand_dims(generated_sequence[-input_length:], axis=0)
    predicted_note = model.predict(input_sequence)
    generated_sequence = np.vstack([generated_sequence, predicted_note[0]])

# Post-process the generated sequence
generated_sequence = postprocess_notes(generated_sequence)

# Print the generated sequence
print(generated_sequence)

def create_midi_file(notes, output_filename):
    # Create a new MIDI file
    mid = MidiFile()
```

```
    track = MidiTrack()
    mid.tracks.append(track)

    current_time = 0
    duration = 185  # You can adjust this value to change note durations
    for note, velocity, start_time in notes:
        delta_time = int(start_time - current_time)
        note = int(note)
        velocity = int(velocity * 127)  # Rescale the velocity back to the original range
        velocity = max(min(velocity, 127), 0)  # Clip the velocity to the valid range (0-
127)
        track.append(Message('note_on', note=note, velocity=velocity,
time=delta_time))
        track.append(Message('note_off', note=note, velocity=0, time=duration))
        current_time = start_time

    # Save the MIDI file
    mid.save(output_filename)

output_filename = '8notes'
create_midi_file(generated_sequence, output_filename)

import matplotlib.pyplot as plt

# Predict the first sequence in the validation set
val_sequence = X_val[0]
val_sequence = np.expand_dims(val_sequence, axis=0)
predicted_output = model.predict(val_sequence)

# Extract the note values from the target and predicted sequences
true_notes = y_val[0][:, 0]
predicted_notes = predicted_output[0][:, 0]

# Plot the target notes and predicted notes
plt.figure(figsize=(10, 6))
```

```
plt.plot(true_notes, 'bo', label='True Notes')
plt.plot(predicted_notes, 'ro', label='Predicted Notes')
plt.xlabel('Time Step')
plt.ylabel('Note Value')
plt.title('Comparison of True Notes and Predicted Notes')
plt.legend()
plt.show()


import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

# Calculate the differences between true and predicted notes
diff_notes = true_notes - predicted_notes

# Create the x, y, and z values for the 3D plot
x = np.arange(len(true_notes))
y = true_notes
z = predicted_notes

# Create a 3D scatter plot
fig = plt.figure(figsize=(10, 6))
ax = fig.add_subplot(111, projection='3d')
ax.scatter(x, y, z, c=diff_notes, marker='o', cmap='coolwarm')

ax.set_xlabel('Time Step')
ax.set_ylabel('True Note Value')
ax.set_zlabel('Predicted Note Value')
ax.set_title('3D Plot of Differences Between Predicted Notes and Original Notes')

# Add a colorbar to represent the differences
cbar = plt.colorbar(ax.scatter(x, y, z, c=diff_notes, marker='o', cmap='coolwarm'))
cbar.set_label('Difference')
```

```
plt.show()
```