AI Music Generation Project

Overview

The AI Music Generation Project is designed to create music using deep learning techniques, specifically utilizing Long Short-Term Memory (LSTM) networks. The model learns patterns in MIDI files, allowing it to generate new musical sequences that mimic the style of the input data. This document provides a detailed breakdown of the code, the methodologies used, and instructions for running the project.

Table of Contents

Installation

Before running the project, ensure you have the required libraries installed. Use the following command in your terminal:

```
bash
Copy code
pip install music21 keras tensorflow mido
```

These libraries are necessary for handling MIDI files, building the neural network, and generating music.

Data Preparation

MIDI File Parsing

MIDI (Musical Instrument Digital Interface) files are used to store music performance data. This project requires a dataset of MIDI files, which can be collected from various sources such as public domain music archives or your personal music collection.

The code begins by setting the path to the dataset:

```
python
Copy code
data_path = r"D:\# DATA SCIENCE\# PROJECTS\- PROJECTS INTERNSHIPS\CODEALPHA
- AI ENGINEERING\Music Generation with AI Project\Data"
```

Note Extraction

The function `get_notes(data_path)` is defined to extract notes and chords from the MIDI files:

```
python
Copy code
def get_notes(data_path):
    notes = []

    for folder in os.listdir(data_path):
        folder_path = os.path.join(data_path, folder)
        if os.path.isdir(folder_path):
            for file in glob.glob(os.path.join(folder_path, "*.mid")):
                midi = converter.parse(file)

                notes_to_parse = None
                parts = instrument.partitionByInstrument(midi)

                if parts:  # File has instrument parts
                    notes_to_parse = parts.parts[0].recurse()
                else:  # File has flat notes
                    notes_to_parse = midi.flat.notes

                for element in notes_to_parse:
                    if isinstance(element, note.Note):
                        notes.append(str(element.pitch))
                    elif isinstance(element, chord.Chord):
                        notes.append('.'.join(str(n) for n in
element.normalOrder))

    return notes
```

- **MIDI Parsing**: The `converter.parse(file)` function from the `music21` library parses each MIDI file.
- **Instrument Parts**: The code checks if the MIDI file contains different instrument parts and retrieves the notes from the first part. If no parts are found, it retrieves the flat notes.
- **Note and Chord Handling**: It distinguishes between single notes and chords. If the element is a note, it appends its pitch to the `notes` list. If it's a chord, it joins the notes in the chord and appends them as well.

After running this function, the total number of extracted notes is printed:

```
python
Copy code
notes = get_notes(data_path)
print(f"Number of notes extracted: {len(notes)}")
```

Data Preprocessing

Mapping Notes to Integers

Once the notes are extracted, they need to be converted into a format suitable for training the LSTM model. This involves mapping each unique note to an integer.

```
python
Copy code
unique_notes = list(set(notes))
n_vocab = len(unique_notes)

# Map notes to integers
note_to_int = {note: i for i, note in enumerate(unique_notes)}
int_notes = [note_to_int[note] for note in notes]
```

- **Unique Notes**: A set is created from the notes to find all unique entries.
- **Mapping**: Each unique note is assigned a unique integer using a dictionary comprehension.

Creating Sequences for Training

Next, sequences of notes are created for training the model. This is crucial because LSTM networks require input sequences to learn temporal dependencies.

```
python
Copy code
sequence_length = 100
network_input = []
network_output = []

for i in range(0, len(int_notes) - sequence_length):
    seq_in = int_notes[i:i + sequence_length]
    seq_out = int_notes[i + sequence_length]

    network_input.append(seq_in)
    network_output.append(seq_out)
```

- **Input Sequences**: The loop creates input sequences of length `sequence_length` (100 in this case).
- **Output Notes**: The output is the note immediately following the input sequence.

Finally, the input data is reshaped for the LSTM model:

```
python
Copy code
n_patterns = len(network_input)

# Reshape for LSTM model (RNN)
network_input = np.reshape(network_input, (n_patterns, sequence_length, 1))
network_input = network_input / float(n_vocab)

# One-hot encode the output
network_output = to_categorical(network_output)
```

- **Reshape**: The input data is reshaped into the format `(number of patterns, sequence length, 1)` required by the LSTM model.

- **Normalization**: The input data is normalized by dividing by the number of unique notes.
- **One-Hot Encoding**: The output is one-hot encoded to convert the integer labels into a binary matrix.

Model Architecture

The next step is to define the architecture of the LSTM model. This is accomplished using the `keras` library:

```
python
Copy code
from keras.models import Sequential
from keras.layers import LSTM, Dense, Dropout, Activation
from keras.optimizers import Adam

# Build the LSTM model
model = Sequential()
model.add(LSTM(512, input_shape=(network_input.shape[1],
network_input.shape[2]), return_sequences=True))
model.add(Dropout(0.3))
model.add(LSTM(512, return_sequences=False))
model.add(Dropout(0.3))
model.add(Dense(256))
model.add(Activation('relu'))
model.add(Dropout(0.3))
model.add(Dense(n_vocab))
model.add(Activation('softmax'))

model.compile(loss='categorical_crossentropy',
optimizer=Adam(learning_rate=0.001))

# Print model summary
model.summary()
```

Explanation of Layers

- **LSTM Layers**: The model consists of two LSTM layers. The first LSTM layer has 512 units and returns sequences to the next layer. The second LSTM layer also has 512 units but does not return sequences.
- **Dropout Layers**: Dropout is applied after each LSTM layer to prevent overfitting. This randomly sets a fraction of the input units to 0 during training.
- **Dense Layer**: After the LSTM layers, a Dense layer with 256 units and ReLU activation is added.
- **Output Layer**: The final Dense layer has a number of units equal to the vocabulary size (`n_vocab`) with softmax activation to output a probability distribution over the possible notes.

The model is compiled with categorical cross-entropy loss and the Adam optimizer.

Training the Model

Once the model architecture is defined, it is time to train the model using the prepared input and output data.

```
python
Copy code
model.fit(network_input, network_output, epochs=10, batch_size=64)
```

- **Epochs**: The model is trained for 10 epochs.
- **Batch Size**: A batch size of 64 is used, meaning that the model processes 64 sequences at a time.

During this phase, the model learns to predict the next note in a sequence based on the preceding notes.

Music Generation

After training, the model can be used to generate new music. The function `generate_music` function generates a sequence of musical notes based on the trained LSTM model:

```
python
Copy code
def generate_music(model, network_input, int_to_note, n_vocab,
output_length=500, temperature=1.0):
    start = np.random.randint(0, len(network_input) - 1)
    pattern = network_input[start]
    prediction_output = []

    # Generate output_length notes
    for _ in range(output_length):
        prediction_input = np.reshape(pattern, (1, len(pattern), 1))
        prediction_input = prediction_input / float(n_vocab)

        prediction = model.predict(prediction_input, verbose=0)
        prediction = np.log(prediction + 1e-6) / temperature
        prediction = np.exp(prediction)
        prediction = prediction / np.sum(prediction)
        index = np.random.choice(range(n_vocab), p=prediction[0])
        result = int_to_note[index]
        prediction_output.append(result)

        pattern = np.append(pattern, index)
        pattern = pattern[1:]

    return prediction_output
```

Explanation of the Function

- **Start Point**: The function begins by selecting a random starting point in the input data. This seed will be used to generate music.
- **Prediction Loop**: A loop runs for `output_length` iterations (default is 500), where in each iteration:
  - The current `pattern` is reshaped to fit the model's input shape.
  - The input is normalized by dividing by `n_vocab`.

    - The model predicts the probabilities of the next note based on the current pattern.
- **Temperature Control**: Temperature is a parameter that controls the randomness of predictions:
    - A higher temperature (>1) makes the predictions more random, while a lower temperature (<1) makes the predictions more deterministic.
    - The prediction probabilities are adjusted using the logarithm and exponential functions to control the temperature before selecting the next note.
- **Selecting the Next Note**: The `np.random.choice` function is used to sample the next note index based on the adjusted probabilities. The predicted note is then converted back to its original form using the `int_to_note` mapping.
- **Pattern Update**: The new note index is appended to the current pattern, and the oldest note is removed to maintain the sequence length.

Finally, the function returns the generated sequence of notes.

Generating MIDI Files

Once the notes are generated, they need to be converted into MIDI format so that they can be played or saved as files. This is accomplished with the `create_midi` function:

```
python
Copy code
def create_midi(prediction_output, output_file):
    offset = 0
    output_notes = []

    for pattern in prediction_output:
        if ('.' in pattern) or pattern.isdigit():
            notes_in_chord = pattern.split('.')
            notes = []
            for current_note in notes_in_chord:
                note_obj = note.Note(int(current_note))
                note_obj.storedInstrument = instrument.Piano()
                notes.append(note_obj)
            new_chord = chord.Chord(notes)
            new_chord.offset = offset
            output_notes.append(new_chord)
        else:
            new_note = note.Note(pattern)
            new_note.offset = offset
            new_note.storedInstrument = instrument.Piano()
            output_notes.append(new_note)

        offset += np.random.uniform(0.5, 1.0)  # Randomize offset

    midi_stream = stream.Stream(output_notes)
    midi_stream.write('midi', fp=output_file)
```

Explanation of the Function

- **Output Notes**: The function initializes an empty list `output_notes` to hold the notes and chords that will be added to the MIDI file.
- **Iterating Over Generated Patterns**: It loops through the generated musical patterns:

    - If the pattern represents a chord (identified by the presence of a '.'), it splits the pattern into individual notes, creates note objects, and constructs a chord object.
    - If the pattern is a single note, it creates a note object directly.
- **Setting Instrument and Offset**: Each note or chord is assigned a `storedInstrument` (in this case, Piano) and an offset which is randomized to create a more natural rhythm.
- **Creating the MIDI Stream**: After all notes are processed, a `stream.Stream` object is created, and the MIDI file is written using the `write` method.

Saving Generated Music

To generate and save multiple pieces of music, a loop is implemented:

```
python
Copy code
for i in range(5):  # Generate 5 different pieces
    predicted_notes = generate_music(model, network_input, int_to_note,
n_vocab, output_length=500, temperature=1.0)
    output_file = f'test_output_{i}.mid'
    create_midi(predicted_notes, output_file)
    print(f"MIDI file {output_file} saved.")
```

- **Loop**: This loop generates five different pieces of music.
- **Output Filename**: Each piece is saved with a unique filename (e.g., `test_output_0.mid`).

Saving and Loading the Model

After training and generating music, it’s important to save the model for future use:

```
python
Copy code
# Save the model
model.save('music_generation_model.h5')
```

This command saves the trained model to a file named `music_generation_model.h5`.

To load the saved model later, the following code can be used:

```
python
Copy code
from keras.models import load_model

# Load the saved model
model = load_model('music_generation_model.h5')
```

This allows you to continue generating music without needing to retrain the model.

Load & Generate a New Song

After loading the model, you can generate new music and save it:

```
python
Copy code
# Path to the folder where you want to save the generated music files
save_path = r'D:\# DATA SCIENCE\# PROJECTS\- PROJECTS INTERNSHIPS\CODEALPHA
- AI ENGINEERING\Music Generation with AI Project\Generated Music'

# Ensure the folder exists, create it if it doesn't
os.makedirs(save_path, exist_ok=True)

# Regenerate a new piece of music
predicted_notes = generate_music(model, network_input, int_to_note,
n_vocab, output_length=500, temperature=1.0)

# Generate a unique filename using the current timestamp
output_file = os.path.join(save_path,
f'new_song_{time.strftime("%Y%m%d_%H%M%S")}.mid')

# Save the MIDI file with the unique filename in the specified folder
create_midi(predicted_notes, output_file)
print(f"New MIDI file {output_file} saved to {save_path}.")
```

Explanation of the Process

- **Output Path**: The path where generated MIDI files will be saved is specified, and the directory is created if it doesn't already exist.
- **Music Generation**: A new piece of music is generated using the previously loaded model.
- **Unique Filename**: The output filename is generated based on the current timestamp, ensuring that each new piece is saved with a unique name.
- **Saving the File**: The generated MIDI file is saved using the `create_midi` function.

Conclusion and Future Work

The AI Music Generation Project successfully demonstrates the use of LSTM networks to generate music based on training data from MIDI files. Here are some potential areas for future development:

1. **Experimenting with Different Architectures**: Different neural network architectures, such as GRU or attention-based models, could be explored to potentially improve the quality of generated music.
2. **Data Augmentation**: Increasing the dataset size through data augmentation techniques (e.g., transposing notes, changing tempos) may help the model learn more robust features.
3. **Interactive Interface**: Developing a user interface that allows users to customize the parameters of music generation, such as the length of the piece or the temperature for randomness, would enhance user experience.
4. **Incorporating Additional Features**: Integrating other features, such as rhythm or dynamics, could lead to more complex and interesting compositions.
5. **Evaluation Metrics**: Implementing evaluation metrics for assessing the musicality of generated compositions could help in fine-tuning the model and guiding future improvements.

6. **Real-Time Generation**: Investigating methods for real-time music generation could open up exciting possibilities for live performances and interactive music applications.