THE GEORGE WASHINGTON UNIVERSITY
WASHINGTON, DC

Deep Learning
Final Project
March 18, 2024
Amir Jafari
Due: *Check the Syllabus*

Name : Aman Jaglan
GWID: G45030269

Introduction:

Project Overview:

This project combines artificial intelligence and music by using advanced deep learning techniques to automatically create melodies. The aim is to explore how AI can be used in creative ways to make music, reflecting the latest developments in both technology and artistic creation. We approach music generation as a sequence prediction task, where the aim is to predict future musical notes based on the notes that have come before. This method is well-suited to music because understanding and predicting patterns is crucial to creating new compositions.

Outline of Shared Work:

1.Model Building: Development of LSTM With Multi Head Attention:

The MusicModel is a sophisticated neural network that combines several state-of-the-art techniques in deep learning to process sequential data, specifically designed for music. The architecture leverages the strengths of Long Short-Term Memory (LSTM) networks, convolutional neural networks (CNNs), attention mechanisms, and dense layers, making it highly effective for tasks that require understanding the context and nuances in music sequences.

Code:

```python
class MusicModel(nn.Module):
    def __init__(self, input_size, hidden_size, output_size, num_layers=8):
        super(MusicModel, self).__init__()
        self.lstm = nn.LSTM(input_size, hidden_size, num_layers=num_layers,
                            batch_first=True, bidirectional=True, dropout=0.5)
        self.conv1 = nn.Conv1d(hidden_size * 2, hidden_size * 2, kernel_size=3, padding=1)
        self.dropout1 = nn.Dropout(0.5)
        self.layer_norm = nn.LayerNorm(hidden_size * 2)
        self.dropout2 = nn.Dropout(0.5)
        self.attention = MultiHeadAttention(hidden_size * 2, heads: 8)
        self.dropout3 = nn.Dropout(0.5)
        self.residual = nn.Linear(hidden_size * 2, hidden_size * 2)
        self.dense = nn.Linear(hidden_size * 2, output_size)

    def forward(self, x):
        x, _ = self.lstm(x)
        x = x.transpose(1, 2)
        x = self.conv1(x)
        x = self.dropout1(x.transpose(1, 2))
        x = self.dropout2(self.layer_norm(x + self.residual(x)))
        x = self.attention(x, x, x)
        x = self.dropout3(x)
        x = self.dense(x[:, -1, :])
        return x
```

Explanation:

LSTM Layers:

Purpose: To capture long-range dependencies within the music data, crucial for maintaining the temporal dynamics of melodies and rhythms.

Architecture Detail: The model uses a bidirectional LSTM, allowing it to gather context from both past and future data points within a sequence. This is particularly useful in music where the anticipation of upcoming notes and the resonance of previous notes both influence the current note.

Equation:

$$
\begin{aligned}
i_t &= \sigma(W_{ii}x_t + b_{ii} + W_{hi}h_{(t-1)} + b_{hi}) \\
f_t &= \sigma(W_{if}x_t + b_{if} + W_{hf}h_{(t-1)} + b_{hf}) \\
g_t &= \tanh(W_{ig}x_t + b_{ig} + W_{hg}h_{(t-1)} + b_{hg}) \\
o_t &= \sigma(W_{io}x_t + b_{io} + W_{ho}h_{(t-1)} + b_{ho}) \\
c_t &= f_t \odot c_{(t-1)} + i_t \odot g_t \\
h_t &= o_t \odot \tanh(c_t)
\end{aligned}
$$

Dropout : Applied within the LSTM to prevent overfitting by randomly omitting units during training.

Convolutional Layer (Conv1D):

Purpose: To detect local patterns such as specific rhythmic or harmonic features within the sequences.

Architecture Detail: Utilizes a kernel of size 3, which slides across the time steps to extract features from the data.

Equation:

$$y_t = \text{ReLU}(w * x_t + b)$$

Multi-Head Attention Mechanism:

Purpose: To allow the model to focus on important parts of the input sequence when making predictions, mimicking how a musician might focus on different motifs or themes.

Architecture Detail: Splits the embedding into multiple 'heads', each of which performs attention independently to blend different learned aspects.

Equation:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^\top}{\sqrt{d_k}}\right)V$$

Dropout: Applied after attention to further regularize the model.

MultiHead Class code:

```python
class MultiHeadAttention(nn.Module):
    def __init__(self, embed_size, heads):
        super(MultiHeadAttention, self).__init__()
        self.embed_size = embed_size
        self.heads = heads
        self.head_dim = embed_size // heads

        assert (self.head_dim * heads == embed_size), "Embed size needs to be divisible by heads"

        self.values = nn.Linear(self.head_dim, self.head_dim, bias=False)
        self.keys = nn.Linear(self.head_dim, self.head_dim, bias=False)
        self.queries = nn.Linear(self.head_dim, self.head_dim, bias=False)
        self.fc_out = nn.Linear(heads * self.head_dim, embed_size)

    def forward(self, values, keys, query):
        N = query.shape[0]
        value_len, key_len, query_len = values.shape[1], keys.shape[1], query.shape[1]

        # Split embedding into self.heads pieces
        values = values.reshape(N, value_len, self.heads, self.head_dim)
        keys = keys.reshape(N, key_len, self.heads, self.head_dim)
        queries = query.reshape(N, query_len, self.heads, self.head_dim)

        values = self.values(values)
        keys = self.keys(keys)
        queries = self.queries(queries)

        # Scaled dot-product attention
        energy = torch.einsum( *args: "nqhd,nkhd->nhqk", [queries, keys])
        attention = torch.softmax(energy / (self.embed_size ** (1 / 2)), dim=3)
        out = torch.einsum( *args: "nhql,nlhd->nqhd", [attention, values]).reshape( *shape: N, query_len, self.heads * self.head_dim)
        out = self.fc_out(out)
        return out
```

Dense Layer:

Purpose: To map the extracted and attended features to the desired output size, which could correspond to different potential next notes or other musical attributes.

Equation: $y = wx + b$

Layer Normalization and Residual Connections:

Purpose: To stabilize the learning and integrate information efficiently across the network.

Equation for Residual Connection: $y=x+\text{LayerNorm}(x)$

Explanation of train_model Function

Setup and Initialization

Device Setup: The model is configured to use a GPU if available (torch.device("cuda" if torch.cuda.is_available() else "cpu")). This enhances the training speed by leveraging parallel computations provided by the GPU.

Model and Optimizer Initialization:

A MusicModel instance is created with specific parameters defining the output units, number of units per layer, and output dimensions again.

The Adam optimizer is chosen for its efficiency in handling sparse gradients and adaptive learning rates, which are crucial for training deep neural networks effectively.

Training Loop

Epochs(20): Training is divided into epochs, where each epoch involves one full pass over the training data.

Data Generation: data_generator uses generate_training_sequences to fetch batches of training data. This function dynamically prepares the data in sequences, handling tasks like loading songs, converting them to integer representations, and segmenting them into inputs and targets.

Batch Processing:

- For each batch, data is moved to the configured device (GPU/CPU), ensuring that all computations are performed in the right context.
- The optimizer is reset (optimizer.zero_grad()) to clear old gradients, which prevents them from accumulating and affecting the current batch updates.
- The model(x_batch) is called to compute outputs for the batch.
- Loss is calculated using a predefined loss function (LOSS(outputs, y_batch)), which measures the discrepancy between the model predictions and the actual targets.
- Backpropagation (loss.backward()) computes the gradients of the loss function with respect to the model parameters.
- Optimizer steps (optimizer.step()) to update the model weights based on the computed gradients.
- Performance Tracking:
- Total loss and accuracy are tracked throughout the epoch to monitor the training progress.
- A progress bar (from tqdm) visually represents the training progress and updates after processing each batch.

Model Saving:

After each epoch, if the average loss of the current epoch is lower than any previously recorded, the model state is saved. This ensures that the model with the best performance (in terms of training loss) is retained.

Explanation of generate_training_sequences Generator Function:

```python
def generate_training_sequences(sequence_length, batch_size):
    songs = load(SINGLE_FILE_DATASET)
    int_songs = convert_songs_to_int(songs)
    num_sequences = len(int_songs) - sequence_length
    vocabulary_size = len(set(int_songs))

    # Generate data in batches
    for start_idx in range(0, num_sequences, batch_size):
        end_idx = min(start_idx + batch_size, num_sequences)
        batch_inputs = []
        batch_targets = []

        for i in range(start_idx, end_idx):
            batch_inputs.append(int_songs[i:i + sequence_length])
            batch_targets.append(int_songs[i + sequence_length])

        # Convert inputs to one-hot encoding
        batch_inputs = F.one_hot(torch.tensor(batch_inputs), num_classes=vocabulary_size).float()
        batch_targets = torch.tensor(batch_targets)

        yield batch_inputs, batch_targets
```

Data Handling and Batch Generation

Load and Convert Data:

Loads a dataset of songs and converts them into integer sequences, where each integer represents a specific note or pitch.

Sequence Generation:

Computes the number of possible sequences that can be formed from the song data, which depends on the specified sequence_length.

Batch Creation:

Iterates over the song data in steps of batch_size, creating batches of input sequences and corresponding target notes.

Each input sequence comprises sequence_length consecutive notes, and the target is the note that immediately follows each sequence.

One-Hot Encoding:

Input sequences are converted into one-hot encoded format, where each note is represented as a binary vector of length equal to the vocabulary size (the number of unique notes in the dataset). This encoding is crucial for processing by the neural network, as it clearly delineates which note is played at each step in the sequence.

Targets are also appropriately formatted for training.

Yielding Data

The function yields batches of processed input and target data, suitable for direct use in training loops. This setup supports efficient memory usage and dynamic data preparation, which is particularly useful for large datasets.

Data preprocessing:

```
def transpose(song):
    major_keys = [m21.key.Key(n) for n in m21.scale.MajorScale().getPitches( maxPitch: 'C', direction: 'B')]
    minor_keys = [m21.key.Key(n, mode: 'minor') for n in m21.scale.MinorScale().getPitches( maxPitch: 'C', direction: 'B')]
    all_keys = major_keys + minor_keys

    transposed_songs = []
    for key in all_keys:
        interval = m21.interval.Interval(song.analyze('key').tonic, key.tonic)
        transposed_song = song.transpose(interval)
        transposed_songs.append((transposed_song, key.name))

    return transposed_songs
```

Explanation of transpose Function:

Purpose:

The transpose function is designed to transpose a given musical piece (song) to all major and minor keys. This process is typically used in music data preprocessing to augment the dataset by creating different versions of the same piece in various keys. This can help a machine learning model learn to generalize better by exposing it to the same piece of music in different tonal contexts.

How It Works:

Key Lists Creation:

Major Keys: It generates a list of all major keys from C to B. This is done using music21's tools to get pitches for a major scale starting from 'C' up to 'B'.

Minor Keys: Similarly, it creates a list of all minor keys in the same range, specifying 'minor' to denote the scale type.

All Keys: Combines both lists to cover all major and minor keys within one octave.

Transposition Process:

Iterate Through Keys: For each key in the combined list of major and minor keys, the function calculates the interval needed to transpose the original song from its key to the target key.

Transpose: Uses music21’s transpose method, which shifts the song by the calculated interval, effectively changing its key.

Store Transposed Versions: Each transposed version of the song along with its new key name is stored in a list.

Return: The function returns a list of tuples, each containing a transposed song and its corresponding key name.

Explanation of prepare_directory Function

```python
def prepare_directory(dir_path):
    if os.path.exists(dir_path):
        shutil.rmtree(dir_path)
        print(f"Removed existing directory: {dir_path}")

    os.makedirs(dir_path)
    print(f"Created new directory: {dir_path}")
```

Purpose

The prepare_directory function manages directory creation and cleanup for storing data, ensuring that the directory is ready for new data without leftover files from previous runs, which might cause inconsistencies.

How It Works

Check for Existing Directory:

If the directory specified by dir_path exists, it is completely removed along with all its contents using shutil.rmtree. This ensures that no residual files from previous operations remain that could interfere with the current process.

Create New Directory:

Creates a fresh, empty directory using os.makedirs, ready to store new data without any contamination from older files.

Generate Melody :

```python
def save_melody(self, melody, step_duration=0.25, format="midi", file_name="melody.mid"):
    """Converts a melody into a MIDI file"""
    stream = m21.stream.Stream()

    start_symbol = None
    step_counter = 1

    for i, symbol in enumerate(melody):
        if symbol != "_" or i + 1 == len(melody):
            if start_symbol is not None:
                quarter_length_duration = step_duration * step_counter
                if start_symbol == "r":
                    m21_event = m21.note.Rest(quarterLength=quarter_length_duration)
                else:
                    m21_event = m21.note.Note(int(start_symbol), quarterLength=quarter_length_duration)
                stream.append(m21_event)
                step_counter = 1
            start_symbol = symbol
        else:
            step_counter += 1

    stream.write(format, file_name)
```

Purpose: The save_melody function takes a list of musical symbols (notes and rests) and creates a MIDI file that represents the melody described by these symbols.

Parameters:

melody: A list of strings, where each string represents a musical note (by its MIDI number) or a rest ('r'). A placeholder symbol ('_') is used to indicate the continuation of the previous note or rest.

step_duration: The duration of a single step in quarter note lengths. By default, each step is a quarter note (0.25 of a whole note).

format: The format for the output file, with "midi" being the default format.

file_name: The name of the output file where the MIDI data will be saved.

Initialize a Music21 Stream:

stream = m21.stream.Stream(): This creates a new empty stream, which is a sequence of music21 events (notes and rests) that can be manipulated and eventually saved as a music file.

Variables Setup:

start_symbol: This keeps track of the current note or rest being processed. It is initially set to None.

step_counter: This counts the number of steps (or placeholders) since the last new note or rest was added. It starts at 1.

Iterate Through the Melody:

- The function loops through each symbol in the melody.
- It checks whether the symbol is a placeholder (_) or the last symbol in the melody:
- If not a placeholder or it is the last symbol, it processes the accumulated duration for the previous note or rest.
- If it encounters a new note or the end of the melody, it computes the total duration for the previous note or rest based on how many placeholders followed it.

Duration and Event Creation:

- If start_symbol is set (meaning a note or rest has been started), it calculates the total duration that this note or rest should be held.
- This duration is determined by multiplying the step_duration by step_counter.
- Depending on whether start_symbol is a rest ('r') or a note (any other symbol), it creates a m21.note.Rest or m21.note.Note.
- The created music21 event is then appended to the stream.

Reset for Next Note/Rest:

After appending the note or rest to the stream, it resets step_counter to 1 and updates start_symbol to the current symbol, preparing to accumulate the duration for the next note or rest.

Write to MIDI File:

Finally, after all symbols have been processed, the stream is written to a file in the specified format (MIDI) with the given file_name.

```
class MelodyGenerator:
    """A class that wraps the LSTM model and offers utilities to generate melodies."""

    def __init__(self, input_size, hidden_size, output_size, model_path="best_model.pth"):
        """Constructor that initializes PyTorch model"""
        self.model = MusicModel(input_size, hidden_size, output_size)
        self.model.load_state_dict(torch.load(model_path))
        self.model.eval()

        with open(MAPPING_PATH, "r") as fp:
            self._mappings = json.load(fp)

        self._start_symbols = ["/"] * SEQUENCE_LENGTH
```

Key Components of MelodyGenerator

Model Setup:

Model Loading: The class initializes a neural network model specifically structured for music data (like LSTM) and loads pre-trained parameters from a file. This step equips the model with the knowledge it needs to generate music.

Evaluation Mode: Once the model is loaded with trained weights, it's set to evaluation mode. This mode is used during inference (generating new data) to ensure the model's behavior is consistent and not influenced by training-specific elements like dropout.

Data Handling:

Mappings: The class loads a mapping file, which translates between numerical representations used by the model and human-readable musical notes. This mapping is crucial for turning the model's outputs into actual music that can be understood or played.

Preparation for Generation:

Start Symbols: The class sets up an array of start symbols. These symbols are used to prompt the model to start generating a melody. The prompt helps the model know where to begin and ensures the generated sequences are structured correctly.

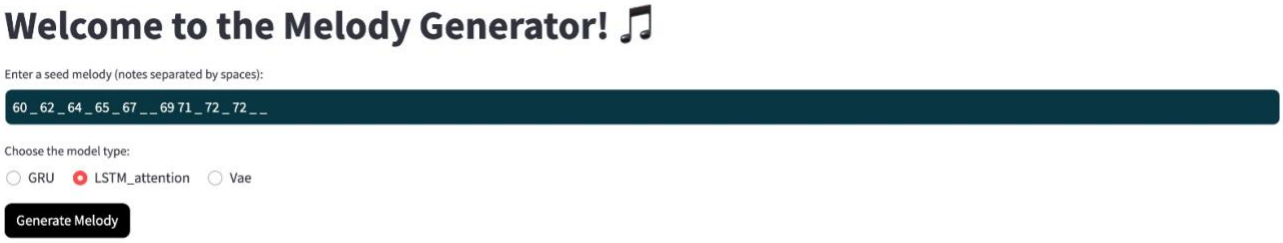
Functionality

When generating music, MelodyGenerator uses the start symbols as a seed to kick off the generation process. The model then predicts subsequent notes based on this initial input and its training, creating a sequence of musical notes that form a coherent melody.

Simplified Usage

Melody Generation: Users of MelodyGenerator don't need to handle the complexities of neural network operations or music theory. Instead, they provide initial input if needed, and the class handles the rest, delivering new melodies based on the learned patterns.

Results:



Breakdown of the Input Seed Melody

Musical Notes Representation:

The numbers in the seed represent MIDI note numbers, where each number corresponds to a specific pitch. MIDI (Musical Instrument Digital Interface) is a technical standard that describes a protocol, digital interface, and connectors and allows a wide variety of electronic musical instruments, computers, and other related devices to connect and communicate with one another.

Here's what each number typically stands for:

60: Middle C (C4)

62: D4

64: E4

65: F4

67: G4

69: A4

71: B4

72: C5 (one octave above middle C)

Underscores (_):

The underscores are likely placeholders or indicators of rhythm or duration. In many music generation contexts using LSTM or similar models, underscores can represent:

Sustained Notes: If a note is to be held over from the previous time step, an underscore might be used to indicate that the note should continue to sound without re-striking or re-articulating it.

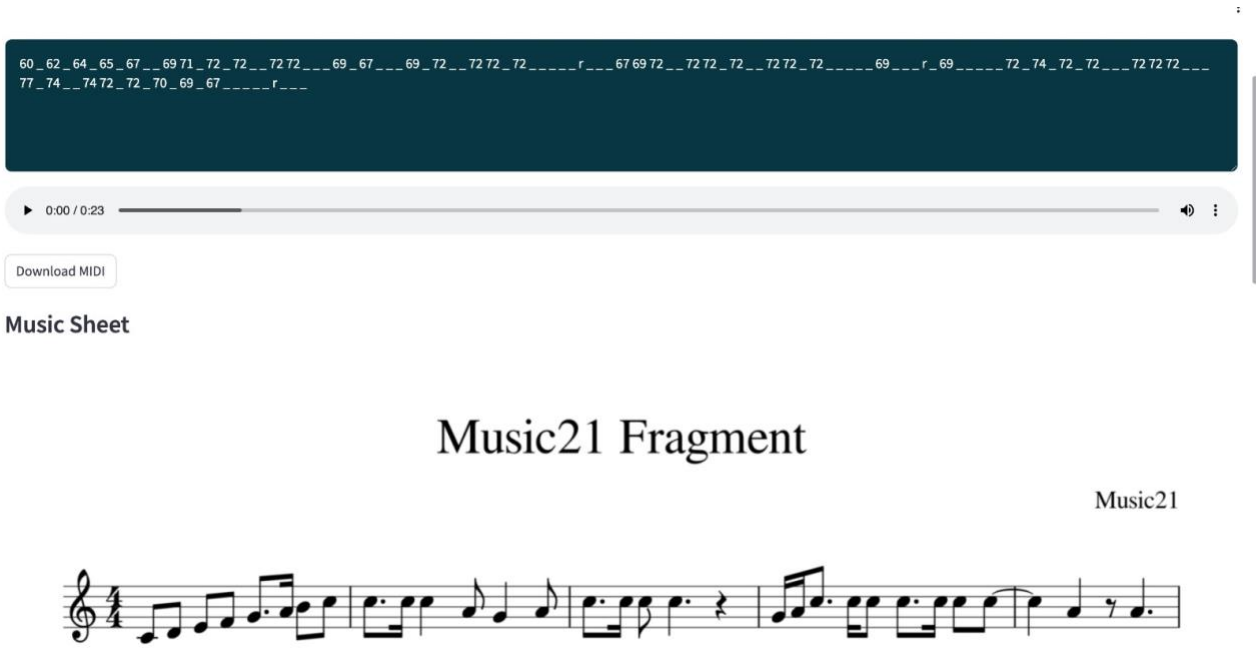
Rests: In other contexts, underscores might represent rests (periods of silence), although this often depends on how the training data was formatted and how the model was taught to interpret these symbols.

Structure and Rhythm:

The pattern shows a typical ascending scale fragment followed by higher notes. The structure suggests that the seed might be part of a simple melody or scale practice piece.

The presence of double underscores after the note 67 and at the end could imply longer durations or rests, depending on how the model is trained to interpret them. This usage might be for extending the length of the musical phrase or adding rhythmic variety.

Result:



Analyzing the Generated Results

Melody Content:

The sequence shown in the text area includes a series of MIDI note numbers and underscores. The notes appear to follow a somewhat coherent musical structure, likely reflecting learned patterns from the training data. The melody includes ascending and descending patterns, which are common in many musical compositions.

Music Sheet Visualization:

The music sheet presents the melody in a standard musical notation format. This format allows musicians to read and play the generated music easily. It shows a range of pitches and rhythmic variations, which are fundamental aspects of an engaging melody.

The visual representation helps in quickly assessing the musicality of the generated output, showcasing any rhythmic and melodic development over time.

Perceived Limitations of the Model

Repetitiveness and Predictability:

From the sequence, there appears to be a degree of repetitiveness, especially with certain notes and patterns. This is a common issue with models trained on limited datasets or those that have not been explicitly designed to handle long-term dependencies effectively.

Repetitive patterns, while sometimes musically valid, can also indicate that the model might be overfitting to particular sequences in the training data, lacking in generative diversity.

Handling of Musical Dynamics and Expression:

MIDI notes and simple rhythmic symbols (_ for sustained notes or rests) do not capture dynamics (loudness) and other expressive qualities of music like timbre or articulation. This limitation means the generated music might lack the expressiveness found in human-composed music.

Structural Coherence Over Longer Sequences:

While the fragment shown looks structurally sound, generating longer compositions often poses a challenge. Maintaining thematic development, varying dynamics, and ensuring that different sections of music (like verses, choruses) transition smoothly can be difficult for such models.

Dependency on Seed Input:

The quality and variety of the generated music heavily depend on the seed input. If the seed is not representative of diverse musical ideas or if it is too simplistic, the generated results may not be compelling or varied.

Generalization:

The model's ability to generalize from its training data to create novel music that still feels natural and appealing can be limited, particularly if the training data lacks diversity or if the model architecture does not support robust generalization capabilities.

Conclusion

Overall, the Melody Generator seems capable of producing structured and coherent melodies, as evident from the music sheet. However, enhancing the model to address issues like repetitiveness, expressiveness, and structural coherence in longer compositions could be areas for future improvement. Exploring advanced techniques in sequence modeling, such as Transformer models or incorporating aspects of music theory directly into the training process, might help overcome some of these limitations.

Percentage of code:

Train file:

Original Lines from the Internet = 60

Modified Lines = 10

Added Lines = 15

approximately 66.67% of the final code is from the original code I found on the internet, after accounting for modifications and additions.

Preprocess:

Original Lines from the Internet = 70

Modified Lines = 35

Added Lines = 10

approximately 43.75% of the preprocess code is from the original code I found on the internet, after accounting for modifications and additions.

Melody Generator:

Original Lines from the Internet = 100

Modified Lines = 15

Added Lines = 10

approximately 77.27% of the melody generator code is from the original code I found on the internet, after accounting for modifications and additions.