HARMONIQ: MUSIC GENERATION WITH DYNAMIC GENRE SHIFT

TATHAGATA GHOSH, RITABRATA CHAKRABORTY, SANDEEP CHAURASIA

TIMELINE– 26TH OCTOBER–16TH NOVEMBER (TILL MODEL TRAINING AND EVAL)

## **Objective**

The goal of this assignment is to design and implement a deep learning model for piano music generation with dynamic genre control using RNNs, GRUs, and LSTMs. The model will be capable of composing piano music based on a specific genre or mood and will offer genre transitions throughout the generated piece. In this assignment, we will:

* Build a custom model using recurrent neural networks (RNNs, GRUs, and LSTMs).
* Implement a mechanism for genre/mood control.
* Explore ways to introduce smooth genre transitions during music generation.
* Investigate how attention mechanisms can improve sequence generation.
* Discuss future improvements using transformer-based models.

By the end of this assignment, the system will be capable of generating piano compositions and transitioning smoothly between different genres based on user input.

## **Datasets**

To train a music generation model, we need high-quality MIDI data for piano music. The following datasets are suitable for this task:

### **1. MAESTRO Dataset**

* **Description**: A large dataset of classical piano performances.
* **Genre Use**: Can approximate genres by filtering based on composers (e.g., Baroque, Romantic, Classical).
* **URL**: MAESTRO Dataset

### **2. Emopia Dataset**

* **Description**: Contains MIDI files labeled with emotional attributes such as happy, sad, angry, or calm, providing an emotional tone for mood control.
* **URL**: [Emopia Dataset](https://github.com/annahung31/EMOPIA)

### **3. GiantMIDI-Piano**

* **Description**: A dataset of classical piano performances, which can be filtered by composer to emulate genres.
* **URL**: [GiantMIDI-Piano](https://github.com/bytedance/GiantMIDI-Piano)

## **Data Preprocessing**

### **Steps to Preprocess the MIDI Data:**

1. **MIDI Parsing**:
   * Use libraries like pretty\_midi or mido to parse MIDI files.
   * Extract musical features such as pitch, velocity, and timing information.
2. **Sequence Representation**:
   * Convert MIDI events into sequences of note representations (e.g., a 128-dimensional vector for each note, where 1 represents a played note, and 0 represents silence).
   * Standardize the time step across sequences (e.g., 1/16 or 1/32 notes) to make the sequences uniform in time.
3. **Genre/Mood Labeling**:
   * MAESTRO: Use composer metadata to label genres.
   * Emopia: Utilize the mood labels provided by the dataset.
4. **Padding and Truncation**:
   * Pad sequences to a fixed length (e.g., 100 notes per sequence) or truncate longer sequences.
5. **Normalization**:
   * Normalize note velocities (dynamics) between 0 and 1.
6. **Train/Test Split**:
   * Split the dataset into training (80%), validation (10%), and test (10%) sets for proper model evaluation.

## **Model Architecture**

The model will be based on recurrent neural networks (RNNs) and their advanced variants like GRUs and LSTMs. The architecture will also include a mechanism for genre control and future support for attention mechanisms.

### **1. Baseline RNN/LSTM Model**

#### **Components:**

1. **Input Layer**:
   * Takes in sequences of notes (e.g., a one-hot encoded representation of each note) and genre/mood embeddings.
2. **Genre/Mood Embedding**:
   * Encodes the genre/mood information into a dense vector representation and concatenates it with the note sequence input.
3. **Bidirectional LSTM Layers**:
   * Two LSTM layers (or GRUs) with bidirectional connections to capture both past and future dependencies in the note sequence.
4. **Attention Mechanism**:
   * Adds a context vector that weighs each part of the sequence based on its relevance to the current note being predicted.
5. **Dense Output Layer**:
   * Outputs the next note in the sequence, represented as a probability distribution over the set of possible notes.

#### **Model Code Example:**

import torch

import torch.nn as nn

class PianoComposer(nn.Module):

def \_\_init\_\_(self, note\_input\_size, genre\_input\_size, embedding\_size, hidden\_size, output\_size, num\_layers=2):

super(PianoComposer, self).\_\_init\_\_()

# Embedding layer for genre/mood input

self.genre\_embedding = nn.Embedding(genre\_input\_size, embedding\_size)

# Bidirectional LSTM layers for note sequence processing

self.lstm = nn.LSTM(input\_size=note\_input\_size + embedding\_size, hidden\_size=hidden\_size,

num\_layers=num\_layers, batch\_first=True, bidirectional=True)

# Attention mechanism

self.attention = Attention(hidden\_size \* 2) # Bidirectional LSTM doubles hidden size

# Fully connected output layers

self.fc = nn.Linear(hidden\_size \* 2, output\_size)

def forward(self, note\_seq, genre\_label):

genre\_embedded = self.genre\_embedding(genre\_label)

genre\_embedded = genre\_embedded.unsqueeze(1).repeat(1, note\_seq.size(1), 1)

lstm\_input = torch.cat((note\_seq, genre\_embedded), dim=-1)

lstm\_out, \_ = self.lstm(lstm\_input)

context, \_ = self.attention(lstm\_out, None)

output = self.fc(context)

return output

### **2. Attention Mechanisms for Improved Learning**

#### **Options for Attention:**

1. **Basic Attention**:
   * Compute attention scores based on the importance of different notes in the sequence.
2. **Self-Attention**:
   * Uses each note's relationship with all other notes in the sequence to determine relevance. This can improve long-range dependencies and thematic consistency in music.
3. **Multi-Head Attention**:
   * Multiple attention mechanisms running in parallel can focus on different parts of the sequence at the same time, further enhancing complexity in music generation.

### **3. Future Model Improvements Using Transformers**

For future iterations, transformer-based models like the **Transformer Encoder** or **GPT-style models** could be explored. These models are particularly good at capturing long-range dependencies and producing coherent sequences over longer timeframes.

class TransformerMusicComposer(nn.Module):

def \_\_init\_\_(self, input\_size, genre\_size, embedding\_size, nhead, hidden\_size, num\_layers):

super(TransformerMusicComposer, self).\_\_init\_\_()

self.note\_embedding = nn.Linear(input\_size, hidden\_size)

self.genre\_embedding = nn.Embedding(genre\_size, embedding\_size)

self.transformer = nn.TransformerEncoder(

nn.TransformerEncoderLayer(d\_model=hidden\_size + embedding\_size, nhead=nhead),

num\_layers=num\_layers)

self.fc = nn.Linear(hidden\_size, input\_size)

def forward(self, note\_seq, genre\_label):

note\_embedded = self.note\_embedding(note\_seq)

genre\_embedded = self.genre\_embedding(genre\_label).unsqueeze(1).repeat(1, note\_seq.size(1), 1)

combined = torch.cat((note\_embedded, genre\_embedded), dim=-1)

transformer\_out = self.transformer(combined)

output = self.fc(transformer\_out)

return output

## **Training Loss Functions and Optimizers**

1. **Loss Function**:
   * **Categorical Cross-Entropy**: For predicting the next note in the sequence, treating it as a multi-class classification problem.
   * **Mean Squared Error (MSE)**: If including velocity (intensity) or duration prediction.
2. **Optimizers**:
   * **Adam Optimizer**: A robust choice for recurrent networks, offering adaptive learning rates.
   * **RMSProp**: Works well for training RNNs with temporal dependencies.

## **Training Process**

1. **Training Loop**:
   * Feed a sequence of notes and genre/mood labels into the model.
   * Use the model to predict the next note in the sequence.
   * Compute the loss between the predicted and actual next note.
   * Backpropagate the error and update model weights.

criterion = nn.CrossEntropyLoss()

optimizer = torch.optim.Adam(model.parameters(), lr=0.001)

for epoch in range(epochs):

for i, (sequence, genre, target) in enumerate(train\_loader):

optimizer.zero\_grad()

output = model(sequence, genre)

loss = criterion(output.view(-1, output\_size), target.view(-1))

loss.backward()

optimizer.step()

print(f"Epoch [{epoch+1}/{epochs}], Loss: {loss.item()}")

1. **Early Stopping**:
   * Implement early stopping to avoid overfitting if the validation loss stops improving.
2. **Learning Rate Scheduler**:
   * Use a scheduler to reduce the learning rate when the model plateaus.

### **Music Generation and Genre Transition (continued)**

1. **Generating Music**:
   * Start with a **seed sequence** (a few notes or a small melody) and a **genre/mood label** (either fixed or transitioning).
   * The model will use the seed sequence to predict the next note in the sequence. After each prediction, the generated note is appended to the sequence and fed back into the model to generate the subsequent note, forming an autoregressive loop.

#### **Example Code for Music Generation:**

def generate\_sequence(model, seed\_sequence, genre\_label, sequence\_length):

generated\_sequence = seed\_sequence

for \_ in range(sequence\_length):

# Get the next predicted note

output = model(generated\_sequence, genre\_label)

next\_note = torch.argmax(output[:, -1, :], dim=-1) # Get the note with highest probability

# Append the predicted note to the sequence

generated\_sequence = torch.cat((generated\_sequence, next\_note.unsqueeze(1)), dim=1)

return generated\_sequence

1. **Genre Transition**:
   * To smoothly transition between genres, you can **interpolate** between the starting genre and the target genre over the duration of the sequence. This gradual change in genre embedding will allow the model to blend musical styles seamlessly.

#### **Example Code for Genre Transition:**

def genre\_transition(model, seed\_sequence, start\_genre, end\_genre, transition\_steps, sequence\_length):

generated\_sequence = seed\_sequence

for i in range(sequence\_length):

# Interpolate between start and end genre embeddings over transition\_steps

genre\_embedding = start\_genre + (end\_genre - start\_genre) \* min(i / transition\_steps, 1.0)

# Predict the next note

output = model(generated\_sequence, genre\_embedding)

next\_note = torch.argmax(output[:, -1, :], dim=-1)

# Append the next note to the sequence

generated\_sequence = torch.cat((generated\_sequence, next\_note.unsqueeze(1)), dim=1)

return generated\_sequence

In this code, the genre embedding is interpolated at each step to provide a smooth transition between two musical styles or moods.

## **Post-Processing**

Once the music sequence has been generated, you'll need to convert it back into a playable format, such as MIDI or audio:

1. **Convert Generated Sequences to MIDI**:
   * Use libraries like pretty\_midi or mido to convert the generated note sequence into a MIDI file. This will allow you to play back the generated music on a piano or a digital audio workstation (DAW).

#### **Example Code for MIDI Conversion:**

python

Copy code

import pretty\_midi

def sequence\_to\_midi(sequence, filename):

midi = pretty\_midi.PrettyMIDI()

piano = pretty\_midi.Instrument(program=pretty\_midi.instrument\_name\_to\_program('Acoustic Grand Piano'))

# Convert each note in the sequence to a MIDI note

for time, note in enumerate(sequence):

midi\_note = pretty\_midi.Note(velocity=100, pitch=note.item(), start=time \* 0.5, end=(time + 1) \* 0.5)

piano.notes.append(midi\_note)

midi.instruments.append(piano)

midi.write(filename)

1. **Audio Rendering**:
   * You can use software synthesizers like **FluidSynth** to render the MIDI file into an audio format (WAV or MP3).

## **Evaluation**

1. **Qualitative Evaluation**:
   * Listen to the generated music to evaluate how well the model follows the genre style and how smooth the genre transitions are.
   * Evaluate melody coherence, harmony, and rhythmic structure. This may involve subjective evaluation from musicians or listeners.
2. **Quantitative Evaluation**:
   * **Perplexity**: Evaluate the model’s perplexity to measure how well it predicts the next note in the sequence.
   * **Genre Classification Accuracy**: Use a genre classification model to assess how well the generated music adheres to the target genre.
   * **Diversity Metrics**: Measure the diversity of the generated sequences (e.g., using entropy or note sequence diversity).
3. **A/B Testing**:
   * Present users with music generated by different versions of the model (e.g., with and without genre transitions, different genres) and collect feedback on user preferences.

## **Deployment and User Interface (UI)**

Once the model is trained and evaluated, the next step is to deploy it with an interactive user interface. This allows users to input a genre or mood and listen to the generated music.

### **Deployment on Huggingface Spaces or Gradio**

1. **Deployment Platform**:
   * Use **Huggingface Spaces** or **Gradio** to create an interactive web app for users to generate music based on their genre preferences.
   * The UI will allow users to:
     + Select a **starting genre** and **ending genre**.
     + Choose the **length** of the sequence.
     + Listen to the generated music or download it in MIDI format.

#### **Example Code for Gradio UI:**

import gradio as gr

# Function to generate and convert the music sequence into an audio file

def generate\_music\_gradio(start\_genre, end\_genre, transition\_steps, sequence\_length):

seed\_sequence = torch.randn(1, 1, note\_input\_size) # Initialize with random seed

start\_genre\_embedding = torch.tensor([start\_genre])

end\_genre\_embedding = torch.tensor([end\_genre])

# Generate music sequence with genre transition

generated\_sequence = genre\_transition(model, seed\_sequence, start\_genre\_embedding, end\_genre\_embedding,

transition\_steps, sequence\_length)

# Convert the generated sequence into a MIDI file

sequence\_to\_midi(generated\_sequence, "generated\_music.mid")

# Return the generated MIDI file

return "generated\_music.mid"

# Gradio Interface

gr.Interface(fn=generate\_music\_gradio,

inputs=["dropdown", "dropdown", "slider", "slider"],

outputs="file").launch()

1. **User Interface Features**:
   * **Dropdowns**: Allow users to select the start and end genres (e.g., Classical, Jazz, Rock, etc.).
   * **Sliders**: Adjust parameters such as sequence length and the smoothness of the genre transition.
   * **Play and Download**: Provide a way for users to listen to the generated music and download the MIDI file.

### **Future Improvements for UI:**

* Add more controls like adjusting **tempo** or **note velocity** to make the output more dynamic.
* Integrate a **genre classifier** to show real-time feedback on how closely the generated sequence adheres to the target genre.