THE GEORGE WASHINGTON UNIVERSITY
WASHINGTON, DC

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

Music Generation with Deep Neural Networks
Individual Project Report (team 7)

for the course

DATS 6303 'Deep Learning'

Spring 2024

Name: Aneri Patel

1. Introduction:

1.1 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.

1.2 Outline of Shared Work:

1.2.1.Model Building: Development of GRU Model with Attention Layers -

- I was responsible for designing and implementing the GRU (Gated Recurrent Unit) model, which is a type of neural network used for time sequence prediction. This model is particularly suited to tasks like music generation where understanding the sequence of events (notes) is crucial.
- Enhanced the GRU model with attention layers. The addition of attention mechanisms helped the model focus on significant parts of the input data, improving the accuracy and relevance of the music generation process.

1.2.2 Model Training:

- Developed the training functions necessary to train the GRU model. This involved setting up the training loops, defining the loss function, and selecting the appropriate optimizer for efficient learning.
- Ensured the training process was robust by implementing techniques to monitor and mitigate overfitting, thus enhancing the model's ability to generalize to new, unseen musical data.

1.2.3 Application Development: Building the Web Application Using Streamlit:

- Led the development of a user-friendly web application using Streamlit, which allows users to interact with the music generation model in real-time.
- Designed the application interface to be intuitive and responsive, enabling users to input seed of melody and receive generated music instantly. This aspect of the project made the model accessible to a broader audience and facilitated real-world testing and feedback.

2. Model Building:

2.1 Background Information on Algorithm Development:

The development of the MusicModel algorithm was driven by the objective to enhance the process of automated music generation through deep learning techniques. Given the sequential nature of music, where each note or chord is dependent on previous elements, employing recurrent neural networks (RNNs) was a natural choice. However, standard RNNs often struggle with long-range dependencies and can be computationally expensive or difficult to train due to issues like vanishing gradients.

To address these challenges, the Gated Recurrent Unit (GRU) was selected as a more efficient variant of the standard RNN. GRUs simplify the model architecture by combining the forget and input gates into a single update gate, reducing the risk of vanishing gradients and improving the model's ability to capture information from earlier in the sequence. This is particularly important in music, where motifs or themes recur with variations over time.

Enhancing the GRU with an attention mechanism allowed the model to focus selectively on parts of the input sequence that are most relevant for predicting the next note. This approach is inspired by the human cognitive process of focusing more on certain musical phrases than on others when composing or performing music. The attention mechanism dynamically weights the significance of different parts of the input data, which helps in improving the accuracy and relevance of the predictions.

2.2 Equations:

2.2.1 GRU Layer :

$$r_t = \sigma(W_{ir}x_t + b_{ir} + W_{hr}h_{(t-1)} + b_{hr})$$

$$z_t = \sigma(W_{iz}x_t + b_{iz} + W_{hz}h_{(t-1)} + b_{hz})$$

$$n_t = \tanh(W_{in}x_t + b_{in} + r_t * (W_{hn}h_{(t-1)} + b_{hn}))$$

$$h_t = (1 - z_t) * n_t + z_t * h_{(t-1)}$$

Where xt is the input step, ht is the hidden state at time t , rt, zt are the reset and update gates, and W,b are parameters of the model.

2.2.2 Attention Mechanism :

- **Attention Scores and Weights Calculation:**

$$e_t = \tanh(W_a h_t + b_a)$$

$$\alpha_t = \frac{\exp(e_t)}{\sum_{t'} \exp(e_{t'})}$$

- **Weighted Sum:**

$$c = \sum_t \alpha_t h_t$$

Where *et* represents the energy or attention score at time t, calculated as a function of the hidden state ht, and αt are the attention weights derived by applying softmax to the scores.

2.3 Model Architecture:

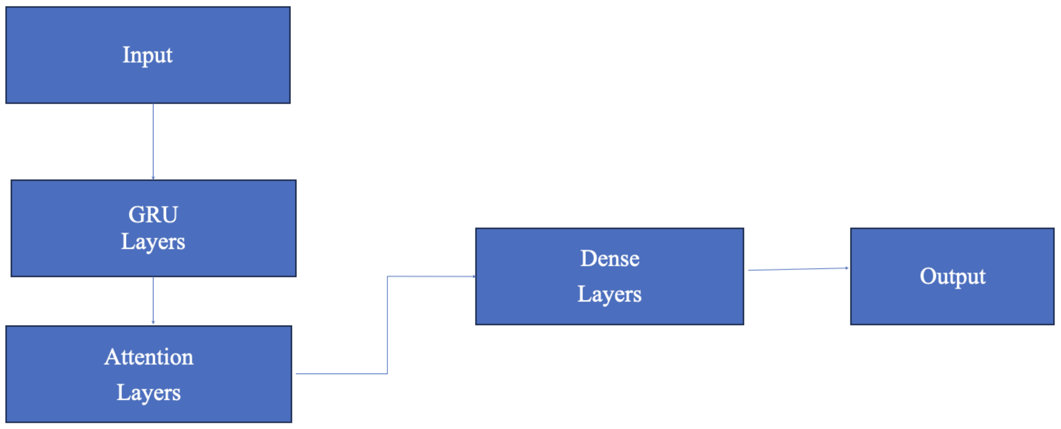


Figure 01. Model Architecture Layout.

2.3.1 Bidirectional GRU Layer

Purpose: The GRU is utilized for its efficiency in modeling sequences where current output is dependent on previous information. The bidirectional aspect of the GRU allows it to capture information from both past (backward) and future (forward) directions, providing a more comprehensive context for prediction.

Configuration:

Input Size:

The dimensionality of the input feature space is determined by the length of our mapping JSON file, representing the total number of unique musical symbols in our dataset.

Hidden Size:

Set at 128, this value defines the dimensionality of the output space for each GRU unit, allowing the model to maintain substantial information about the input sequences.

Number of Layers:

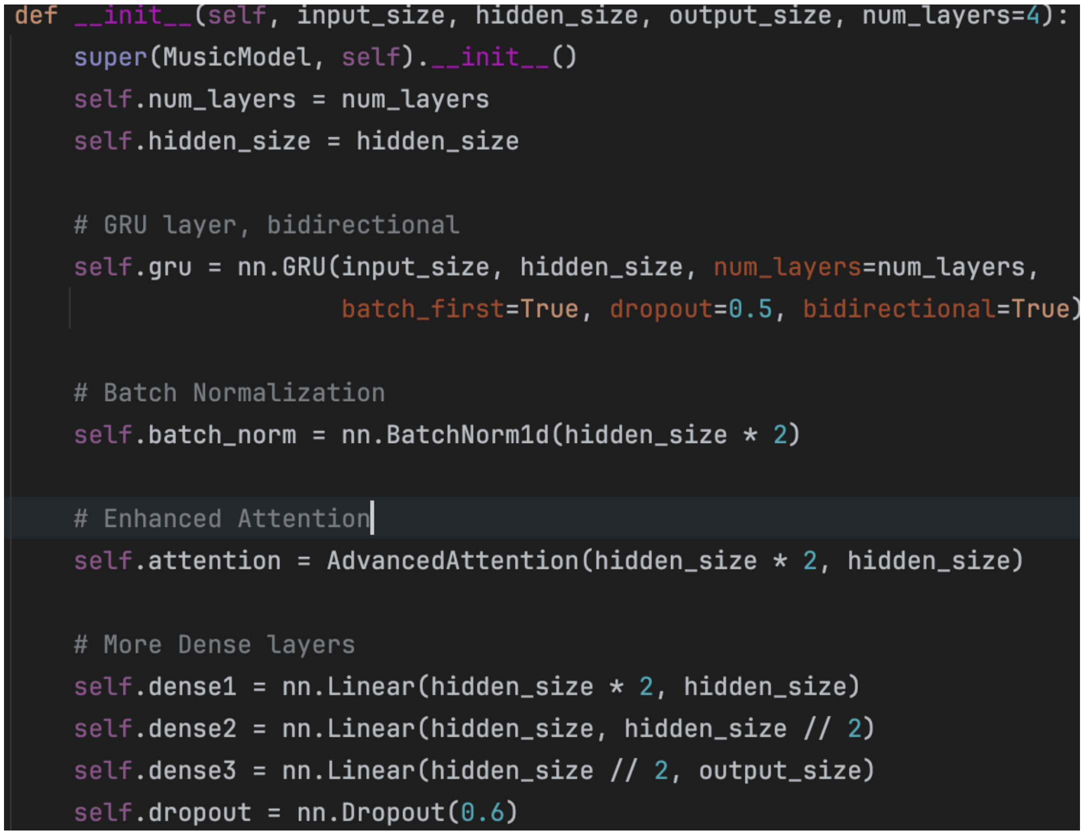
The model includes four GRU layers. This multilayer setup enables the network to learn increasingly complex patterns and dependencies within the music data.

Dropout:

A dropout rate of 0.5 is implemented between GRU layers. This setting helps prevent overfitting by randomly omitting half of the units in each update during training, promoting a more generalizable model.

Mathematical Operation:

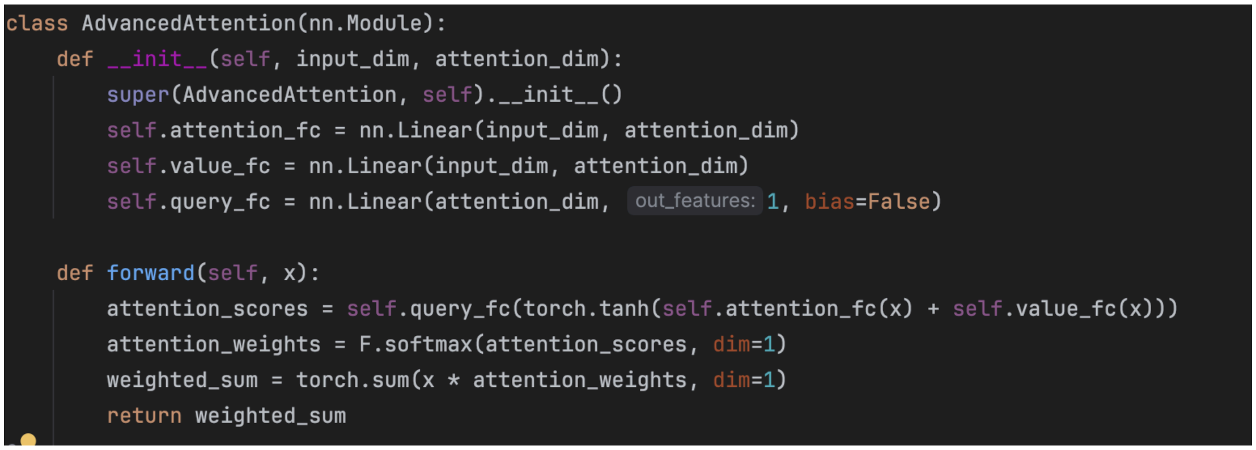
The GRU modifies the hidden state per timestep, factoring in new input and the previous timestep's hidden state, guided by its reset and update gates which help in controlling the flow of information.



```python
def __init__(self, input_size, hidden_size, output_size, num_layers=4):
    super(MusicModel, self).__init__()
    self.num_layers = num_layers
    self.hidden_size = hidden_size

    # GRU layer, bidirectional
    self.gru = nn.GRU(input_size, hidden_size, num_layers=num_layers,
                      batch_first=True, dropout=0.5, bidirectional=True)

    # Batch Normalization
    self.batch_norm = nn.BatchNorm1d(hidden_size * 2)

    # Enhanced Attention
    self.attention = AdvancedAttention(hidden_size * 2, hidden_size)

    # More Dense layers
    self.dense1 = nn.Linear(hidden_size * 2, hidden_size)
    self.dense2 = nn.Linear(hidden_size, hidden_size // 2)
    self.dense3 = nn.Linear(hidden_size // 2, output_size)
    self.dropout = nn.Dropout(0.6)
```

2.3.2 Advanced Attention Mechanism

Purpose: To enhance the model's focus on significant portions of the input sequence that are crucial for predicting subsequent notes or sequences.



```python
class AdvancedAttention(nn.Module):
    def __init__(self, input_dim, attention_dim):
        super(AdvancedAttention, self).__init__()
        self.attention_fc = nn.Linear(input_dim, attention_dim)
        self.value_fc = nn.Linear(input_dim, attention_dim)
        self.query_fc = nn.Linear(attention_dim, out_features: 1, bias=False)

    def forward(self, x):
        attention_scores = self.query_fc(torch.tanh(self.attention_fc(x) + self.value_fc(x)))
        attention_weights = F.softmax(attention_scores, dim=1)
        weighted_sum = torch.sum(x * attention_weights, dim=1)
        return weighted_sum
```

Components:

Attention FC and Value FC: Two fully connected layers that transform the input into an intermediate space where attention scores can be effectively computed.

Query FC: A linear layer that maps the attention values to a single score per input element without bias, focusing the model on important features.

Operation:

The output from the GRU layers is passed through the attention mechanism where it computes an attention score for each element in the sequence using a combination of tanh activations and linear transformations. The softmax function is then applied to these scores to generate a probability distribution (attention weights). The final output is a weighted sum of the input features, scaled by these attention weights.

2.3.3 Batch Normalization

Purpose: To normalize the inputs of each layer within the network to improve the speed, performance, and stability of the neural network.

Operation: Applied right after the attention mechanism's weighted output, ensuring that the input to the subsequent dense layers has a consistent distribution that aids in faster and more stable training.

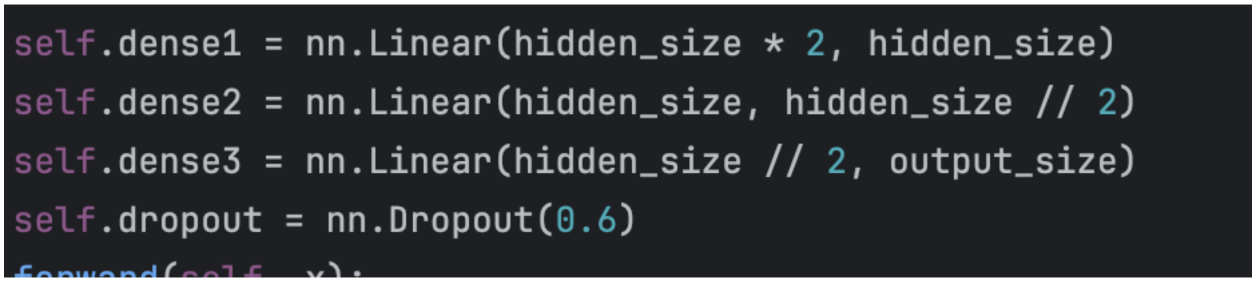
2.3.4. Dense Layers and Dropout

Configuration:

Dense1: Transforms the batch-normalized features into a higher dimension space, allowing for non-linear transformations via the ReLU activation function.

Dropout: Follows the first dense layer to again prevent overfitting during training by randomly dropping units and their connections during training.

Dense2 and Dense3: Further compress the representation to the desired output size, with each layer followed by ReLU activations to introduce non-linearity.

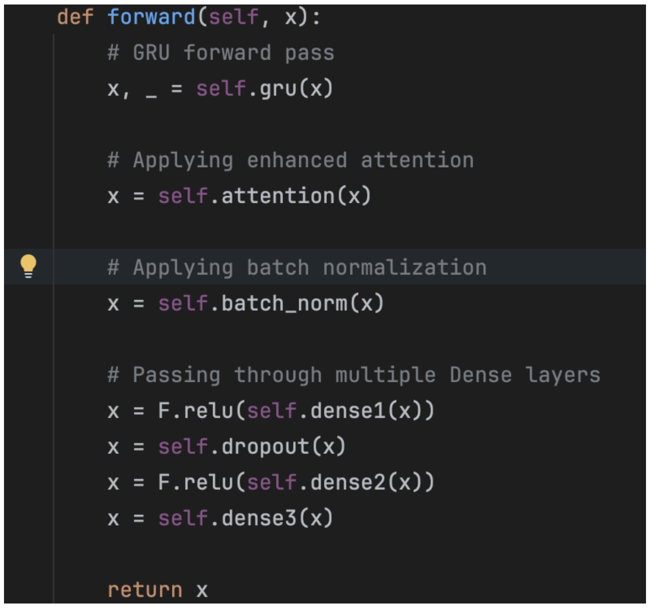


```
self.dense1 = nn.Linear(hidden_size * 2, hidden_size)
self.dense2 = nn.Linear(hidden_size, hidden_size // 2)
self.dense3 = nn.Linear(hidden_size // 2, output_size)
self.dropout = nn.Dropout(0.6)
```

Purpose: These layers are crucial for mapping the learned high-level, abstract features from the GRU and attention outputs to the final output shape that represents the generated music sequence.

2.3.5 Forward Pass

During the forward pass, input data first traverses the bidirectional GRU, then the attention mechanism, followed by batch normalization, and finally through the series of dense layers with intermittent dropout and ReLU activations before producing the final output.



```python
def forward(self, x):
    # GRU forward pass
    x, _ = self.gru(x)

    # Applying enhanced attention
    x = self.attention(x)

    # Applying batch normalization
    x = self.batch_norm(x)

    # Passing through multiple Dense layers
    x = F.relu(self.dense1(x))
    x = self.dropout(x)
    x = F.relu(self.dense2(x))
    x = self.dense3(x)

    return x
```

3. Model Training:

The model training process is a crucial phase in the deployment of the `MusicModel`. It involves setting up and executing a series of steps to optimize the model's parameters using a dataset of musical sequences. This section describes the training process implemented in the `train_model` function, detailing the configuration, execution, and monitoring aspects involved.

3.1 Model Training Setup and Execution

3.1.1 Device Configuration: The model is set up to train on a GPU when available, enhancing the speed and efficiency needed for handling large datasets common in deep learning projects. If a GPU isn't available, the model defaults to using the CPU.

3.1.2 Model Initialization: We start by creating an instance of the `MusicModel` with specific output dimensions and then move this model to our chosen device (GPU or CPU). This ensures that all calculations are consistently performed on the same device throughout the training process.

3.1.3 Optimizer Setup: For optimizing the model, I used the Adam optimizer. This optimizer is preferred for its ability to handle varying data and learning rates effectively, helping the model learn faster and more efficiently than traditional methods.

3.2 Training Process

3.2.1 Batch Processing: We process the data in batches, which helps manage memory better and speeds up the training by updating model weights batch by batch instead of data point by data point.

3.2.2 Loss and Backpropagation: In each batch, the model makes predictions which are then used to calculate the loss (Cross Entropy loss function is used)—the difference between the predicted and actual values. We then use this loss to perform backpropagation, adjusting the model's parameters (like weights) to improve predictions in the next batch.

3.2.3 Monitoring Performance: After each batch, we update and review the model's loss and accuracy. These metrics are vital for keeping an eye on how well the model is learning and deciding if any adjustments to the training process are needed.

3.2.4 Epoch Review: At the end of each epoch, we calculate and note the average loss and accuracy. This gives us a summary of how the model performed throughout that entire epoch, helping to gauge overall progress and effectiveness.

Model Saving Once all epochs are completed, we save the model's parameters to a file. This step is essential as it allows us to use the trained model later without needing to retrain from scratch. This saved model can be used for further evaluation, continued training, or deployment.

4. Application Development:

4.1 Custom Theme Configuration:

For the "Melody Generator" application, a custom dark theme was developed to enhance the user interface's visual appeal and ensure a comfortable viewing experience.

4.1.1. Page Configuration:

The application's page configuration is set with a wide layout to maximize the use of screen space, accommodating more content and interactive elements on the screen simultaneously. This setting is particularly useful for applications involving complex interactions and data presentations like a melody generator.

4.1.2 CSS Customization:

The dark theme is implemented using custom CSS, defining a set of color and style variables to ensure consistency across the application. Key elements of the theme include:

Primary Color: Set to black (`#000000`), providing a deep dark background that reduces strain on the eyes.

Background Color: A slightly lighter shade of dark blue (`#002b36`), which helps elements like buttons and input fields to stand out without stark contrast.

Text Colors: Text is uniformly white (`#ffffff`), ensuring high readability against the darker backgrounds.

Accent Color: A darker shade of blue (`#073642`) is used for interactive elements and highlights, maintaining the overall dark theme while making the interface dynamic.

4.1.3 Style Adjustments:

Various components of the Streamlit framework are customized to align with the dark theme:

Padding and margins are adjusted to reduce unnecessary space and focus the user’s attention on the content

4.1.4 Advanced CSS Features:

The CSS defines detailed adjustments for specific components like buttons, text inputs, text areas, and markdown containers, ensuring that all parts of the UI consistently reflect the dark theme.

4.2 MelodyGenerator Based on Selected Model:

The `MelodyGenerator` class in final_melodygenerator.py is designed to interface with different neural network models for generating music, allowing users to select the model dynamically through radio buttons. Here's how each function within the `MelodyGenerator` operates depending on the selected model:

4.2.1 Initialization (`__init__` method):

The constructor initializes the `MelodyGenerator` with essential parameters including the model type, input size, hidden size, and output size.

Based on the `model_type` provided (`model1`, `model2`, or `model3`), it loads the corresponding model (either `MusicModel`(GRU), `MusicModel2`(LSTM_attention), or `MusicModel3`(VAE)) and sets the path to the trained model file.

The model is loaded into the specified computing device (GPU or CPU) and set to evaluation mode for generating melodies without altering the model weights.

4.2.2 Model Loading:

The specific model file is loaded using `torch.load` with error handling to ensure the file exists. If the model file is not found, an error message is displayed and the application halts, prompting the user to check the model file path.

4.2.3 Load Mapping:

Independent of the model type, the mappings from symbols to integers (used in one-hot encoding during training) are loaded from a JSON file. These mappings are crucial for converting input seeds into formats understandable by the model.

4.2.4 Melody Generation (`generate_melody` method):

For `model1` and `model2`, which are typically GRU and LSTM models respectively:

- The input seed is split into symbols, extended with start symbols, and converted into integers using mappings.
- These integers are then transformed into one-hot encoded tensors and passed through the model to generate the next notes in the sequence.
- The process iterates, appending each new note to the melody until the sequence is completed or a stop symbol ("/") is generated.

For `model3`, which is a VAE model:

- The process similarly starts with converting the seed into encoded formats, but involves additional steps such as encoding into a latent space, sampling from this space, and then decoding to generate output.
- This model employs a different approach to handle the continuity and diversity of the generated music, leveraging the properties of VAEs.

4.2.5 Saving Melody (save_melody method)

The save_melody method in the MelodyGenerator class is crucial for exporting the generated melody into a MIDI format, which will be used for playback or further musical processing. The approach varies based on the neural network model used for generating the melody:

For Models 1 and 2 (GRU and LSTM):

- **Direct MIDI Creation:** The melody, represented as a sequence of musical symbols (notes and rests), is iteratively processed. Each symbol is converted into a corresponding music21 object:
- **Notes:** The numeric values are translated into pitches using the music21.pitch.Pitch class, which are then used to create music21.note.Note objects with specified durations.
- **Rests:** Represented by "r" in the sequence, these are turned into music21.note.Rest objects with appropriate durations.
- **Stream Assembly:** These objects are sequentially added to a music21.stream.Stream, effectively building the musical composition in MIDI format.
- **File Saving:** The stream is then saved as a MIDI file, preserving the generated melody for external use.

For Model 3 (VAE):

- Model 3 generates more complex or continuous sequences, which require careful handling to ensure the MIDI output accurately represents the generated music.
- **Sequence Processing:** The method formats each element in the melody, possibly adjusting durations and note transitions to accommodate the model's output characteristics, ensuring that timing and rhythmic integrity are maintained in the transition from model output to MIDI format.
- **MIDI Creation:** The processed musical elements are converted into music21 objects and compiled into a MIDI file.

4.2.6 Sampling Method (_sample_with_temperature method)

This method plays a pivotal role in determining the variability and creativity of the music generation process by adjusting the probability distribution used for selecting the next note in the sequence:

Temperature Control:

Softmax Temperature: This parameter modulates how concentrated or dispersed the probabilities are. A low temperature sharpens the distribution, making the selection process more deterministic and favoring the most probable notes. A high temperature flattens the distribution, increasing the randomness and likelihood of selecting less probable notes.

Probability Adjustment:

- Models 1 and 2: Probabilities are logarithmically scaled by the temperature, normalized, and then used to perform categorical sampling. This method allows fine control over the melody's predictability and diversity.
- Model 3: The method involves more complex probability adjustments, reflecting the sophisticated nature of the model. It could include transformations that accommodate the specific characteristics of outputs from models like VAEs, ensuring that the diversity is maintained without losing coherence in the musical output.

Sampling Execution:

Categorical Sampling: Based on the adjusted probabilities, the method uses numpy functions to randomly select a note index, which is then used to continue building the melody. This step is critical for infusing creativity and ensuring that each generated melody is unique and engaging.

4.3 Application Functionality:

4.3.1 Model Selection and Melody Generation

Purpose: It will allow users to generate music by selecting a model type through a radio button interface. This function is central to linking the user interface with the backend processing of music generation.

Implementation

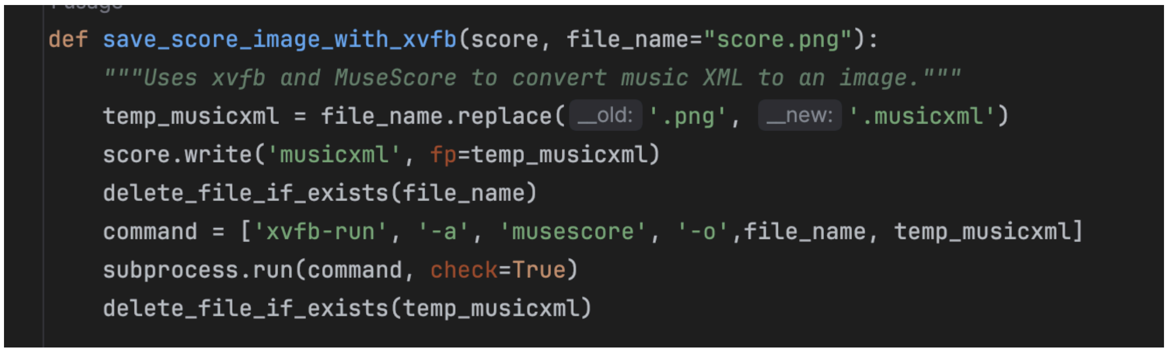
- **Model Type Determination**: Based on the user's selection from the radio buttons, the function assigns a model_type. This is done through conditional statements that map the user's choice to a specific model identifier ('model1' for GRU, 'model2' for LSTM with attention, and 'model3' for VAE).
- **Model Initialization and Melody Generation**:
    - The function initializes the MelodyGenerator class with parameters tailored to the selected model type. It sets attributes like input_size, hidden_size, and output_size appropriate for the chosen model.
    - It then invokes the generate_melody method on the instance of MelodyGenerator. This method takes the seed provided by the user and processes it through the specified neural network model to generate a musical sequence.

4.3.2.Displaying Generated Melody:

Purpose: To provide a visual and textual representation of the generated melody.

Implementation:

- **Text Display:** The generated melody is displayed in a Streamlit text area, allowing users to see the sequence of notes and rests as text.
- **Music Sheet Generation:** The `text_to_score` function converts the text-based melody into a `music21` score object, which represents the musical notation.
- **Score Visualization:** The `save_score_image_with_xvfb` function converts the music score into an image using MuseScore, facilitated by X virtual framebuffer (Xvfb) to render the score without a display server. This image is then displayed in the application, giving users a graphical view of the musical notation.



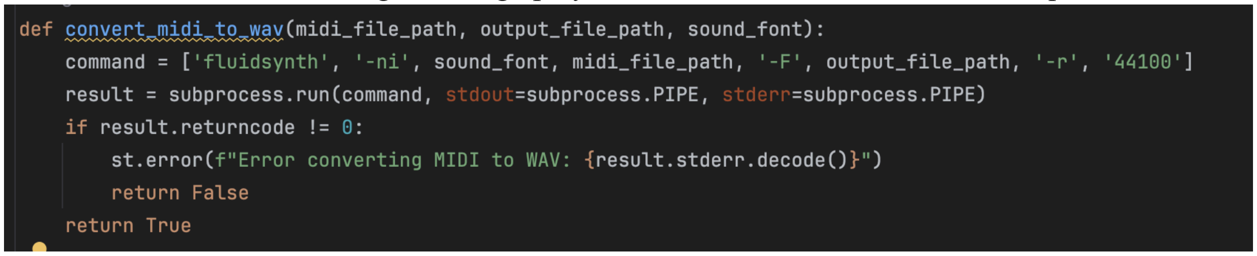
```python
def save_score_image_with_xvfb(score, file_name="score.png"):
    """Uses xvfb and MuseScore to convert music XML to an image."""
    temp_musicxml = file_name.replace( __old: '.png', __new: '.musicxml')
    score.write('musicxml', fp=temp_musicxml)
    delete_file_if_exists(file_name)
    command = ['xvfb-run', '-a', 'musescore', '-o',file_name, temp_musicxml]
    subprocess.run(command, check=True)
    delete_file_if_exists(temp_musicxml)
```

4.3.3 Audio Playback:

Purpose: To enable users to hear the generated melody.

Implementation:

- **MIDI to WAV Conversion:** The `convert_midi_to_wav` function takes the MIDI file produced by `save_melody` and converts it into a WAV file using FluidSynth. This conversion is crucial for generating playable audio files from MIDI sequences.



```python
def convert_midi_to_wav(midi_file_path, output_file_path, sound_font):
    command = ['fluidsynth', '-ni', sound_font, midi_file_path, '-F', output_file_path, '-r', '44100']
    result = subprocess.run(command, stdout=subprocess.PIPE, stderr=subprocess.PIPE)
    if result.returncode != 0:
        st.error(f"Error converting MIDI to WAV: {result.stderr.decode()}")
        return False
    return True
```

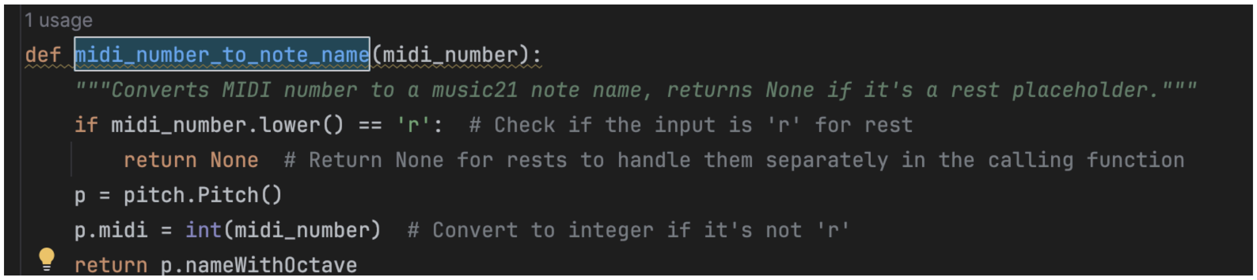
- **Playback in App:** The generated WAV file is loaded into Streamlit's audio widget, which embeds a simple audio player in the application interface. This allows users to play back the generated melody directly within the app.

4.3.4.Utility Functions:

Purpose: These functions support the main functionalities by handling specific tasks like note conversion, score preparation, and file management.

Implementation:

- **Note Conversion** (`midi_number_to_note_name`): Converts MIDI number to musical note names, facilitating the translation of numerical note representations into human-readable formats.

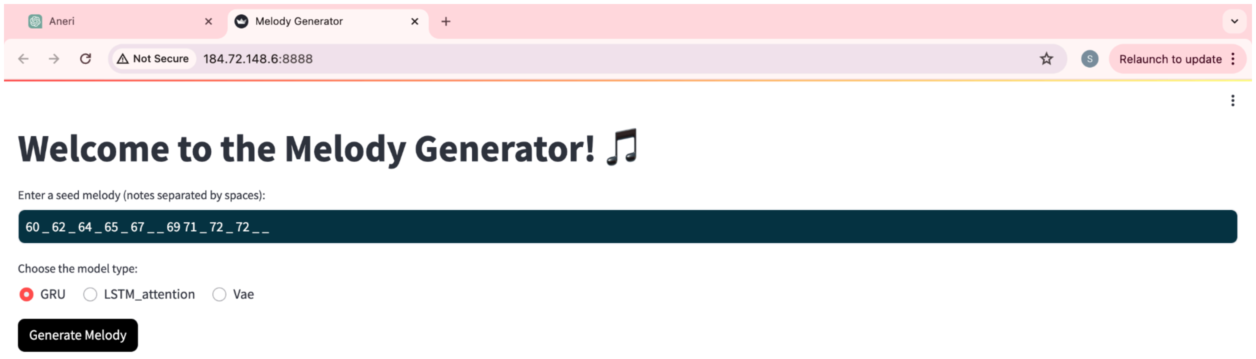


```
1 usage
def midi_number_to_note_name(midi_number):
    """Converts MIDI number to a music21 note name, returns None if it's a rest placeholder."""
    if midi_number.lower() == 'r':  # Check if the input is 'r' for rest
        return None  # Return None for rests to handle them separately in the calling function
    p = pitch.Pitch()
    p.midi = int(midi_number)  # Convert to integer if it's not 'r'
    return p.nameWithOctave
```

- **Score Preparation** (`text_to_score`): Parses the melody text and organizes it into a structured music21 score, setting the groundwork for both MIDI creation and score visualization.
- **File Management** (`delete_file_if_exists`): Ensures that the system does not encounter file conflicts by removing existing files before creating new ones, maintaining the integrity and cleanliness of the file system.

5. Results:

This section presents the outcomes of deploying the Melody Generator application, which was developed to explore the capabilities of machine learning models in generating musically coherent melodies. Utilizing a user-provided seed as the baseline input, the application employs a selected model (GRU) to extend this seed into a full melody.



Welcome to the Melody Generator! 🎵

Enter a seed melody (notes separated by spaces):

60 _ 62 _ 64 _ 65 _ 67 _ _ 69 71 _ 72 _ 72 _ _

Choose the model type:

[✓ GRU] LSTM_attention Vae

Generate Melody

Explanation of the Seed Melody:

MIDI Note Numbers: The numbers represent MIDI note values, which correspond to specific pitches:

60: Middle C (C4)

62: D4

64: E4

65: F4

67: G4

69: A4

71: B4

72: C5 (C one octave above Middle C)

Underscores (_): In this context, underscores are used to denote either a continuation of the previous note (sustain) or a rest (silence), depending on how the melody generator is programmed to interpret these symbols. Given the typical use in similar systems:

Single underscores typically indicate the continuation of the previous note, suggesting that the note should be held through the duration represented by the underscore.

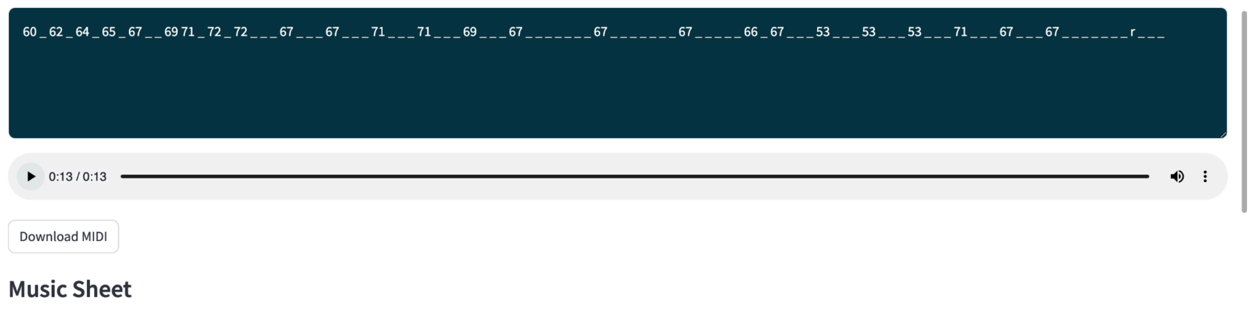
Consecutive underscores could represent longer sustains or rests, extending the duration of silence or the sustain of the last played note before them.

Musical Interpretation:

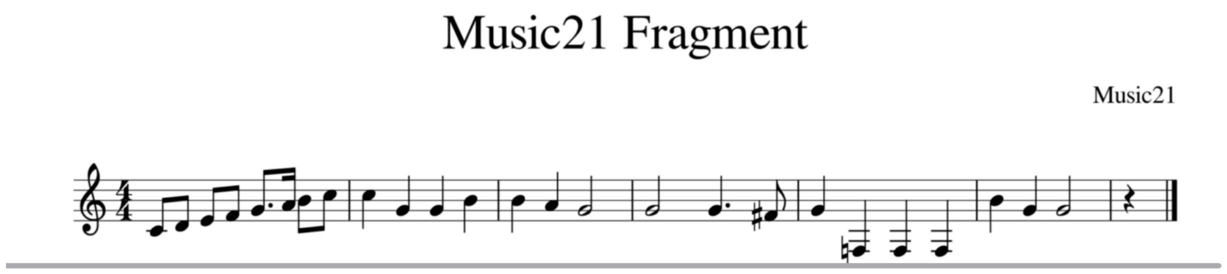
Ascending Pattern: The sequence starts at Middle C and ascends through a standard scale pattern up to C5. This pattern is common in both practice exercises and many musical compositions, providing a melodic line that is harmonically and melodically straightforward.

Repetition and Rhythm: The sequence ends with repeated notes at C5, underscored by multiple sustains or rests, which could imply emphasis or a pause in the melody, adding a rhythmic element to the sequence.

Use in Melody Generation: Such a seed is particularly useful for a melody generator as it provides a clear tonal foundation and rhythmic variety, which can help the AI model in predicting and generating a continuation that is both melodically and rhythmically coherent.



```
60 _ 62 _ 64 _ 65 _ 67 _ _ 69 71 _ 72 _ 72 _ _ _ 67 _ _ _ 67 _ _ _ 71 _ _ _ 71 _ _ _ 69 _ _ _ 67 _ _ _ _ _ _ _ 67 _ _ _ _ _ _ _ 67 _ _ _ _ _ 66 _ 67 _ _ _ 53 _ _ _ 53 _ _ _ 53 _ _ _ 71 _ _ _ 67 _ _ _ 67 _ _ _ _ _ _ _ r _ _ _
```



Generated Melody Overview:

This sequence is musically rendered, allowing us to see both the notes and their rhythmic values.

Musical Characteristics:

Scale Ascension and Descent: The sequence starts with an ascending scale from Middle C (60) up to C5 (72), which is a straightforward, scale-wise progression. Following this, the melody explores variations and returns, focusing on notes such as G4 (67), and even dipping down to F3 (53), showing a broader range.

Repetition and Variation: There is a notable repetition of certain notes, particularly G4 (67) and F3 (53), which suggests a thematic element or motif being developed by the model. This kind of repetition can lend a sense of unity to the piece, though it may also reflect a limitation in the model's creative variability.

Music Sheet Interpretation:

Visual Representation: The music sheet shows a good distribution of note values, indicating dynamic rhythm changes and giving a clearer sense of how the melody might be perceived audibly.

Cohesion: The piece starts and ends on notes within the same octave, providing a frame for the melody. The excursion down to F3 and back to notes around G4 and B4 (71) suggests an attempt to create a narrative or journey within the melody.

6. Limitations and Considerations

6.1 Predictability and Complexity:

The melody heavily relies on scalar runs and basic leaps within a familiar diatonic range. This can make the melody somewhat predictable and less interesting for more advanced musical contexts.

6.2 Musical Depth:

Dynamics, articulation, and other expressive qualities are not specified in MIDI directly and are not visible in the music sheet. The absence of these elements can result in a performance that feels flat or mechanically rigid.

6.3 Repetitiveness:

The repeated use of certain pitches and rhythmic figures might indicate a limitation in the model's ability to generate more varied and complex sequences. While repetition is a common musical device, excessive use without significant variation can make the music seem uninspired.

6.4 Structural Development:

The melody does not show a clear structural development such as distinct musical phrases or sections that you might expect in more formal compositions. It resembles more of an exercise or an etude rather than a complete musical piece.

7. Conclusion

The generated melody demonstrates the capability of the model to create musically coherent sequences that adhere to basic tonal and rhythmic rules. However, it also highlights common challenges in computational music generation, such as achieving expressive depth and structural complexity. Future improvements might include integrating more sophisticated models that can better understand and reproduce the complex human composition and performance, possibly through deeper learning architectures like Transformers or by incorporating a wider array of training data encompassing different styles and complexities.

8. Code Utilization:

Train file:

Original Lines from the internet: 50

Modified Lines: 10

Added line :8

The percentage of the original code from the internet, after modifying and adding lines, is approximately 68.97%.

Melody Generator:

Original Lines from the internet: 47

Modified Lines: 9

Added line :5

The percentage of the original code from the internet, after modifying and adding lines, is approximately 73.08%.

Streamlit:

Original Lines from the internet: 150

Modified Lines: 50

Added line :20

The percentage of the original code that remained unchanged after modifications and additions is approximately 58.82%.

9. References:

MuseScore. (n.d.). Retrieved April 26, 2024, from https://musescore.com/

Dey, R., & Salem, F. M. (Year). Gate-Variants of Gated Recurrent Unit (GRU) Neural Networks. Circuits, Systems, and Neural Networks (CSANN) LAB Department of Electrical and Computer Engineering, Michigan State University. East Lansing, MI 48824-1226, USA.

Zhou, Y. (n.d.). *Music Generation Based on Bidirectional GRU Model*. Department of Statistics, University of Michigan.

Jozefowicz, R., Zaremba, W., & Sutskever, I. (2015). An Empirical Exploration of Recurrent Network Architectures. *Proceedings of the 32nd International Conference on Machine Learning (ICML-15)*, 2342-2350.