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

Individual Final Report

Smit Pancholi (G31443926)

Introduction:

This project explores the integration of artificial intelligence with music generation, harnessing the synergy between deep learning and creative processes to autonomously produce melodies. Advanced deep learning strategies are implemented within a time series prediction framework, employing the music21 Python library for detailed musical analysis and PyTorch for the development of sophisticated neural networks. These networks include a GRU with an attention layer, a Variational Autoencoder, and a multi-head attention LSTM, specifically designed to address the unique demands of musical data and enhance the understanding of musical structures. MuseScore, which provides graphical displays of music scores, and a Streamlit-based web interface that facilitates real-time interactions with the models augment visualization and interactive engagement. This setup not only highlights the capabilities of deep learning in artistic domains but also broadens its accessibility, encouraging widespread participation in the process of music creation.

Outline of the shared work:

1. Inputs in Preprocessing.
   - Batch Processing
2. Model Building
   - Background Information
     - Explanation of the Variational Autoencoder and its suitability for melody generation through deep learning.
     - Network Architecture.

- Encoding-Decoding Framework
    - Description of how the VAE compresses input data into a latent space and reconstructs it back into the original data space.
- Encoder Layers
    - Details on how the encoder transforms the input into latent space representations (mean and log variance), defining a Gaussian distribution for sampling latent variables.
- Reparameterization Trick.
- Decoder layers.
- Benefits for Time Series Prediction
    - Discussion of the VAE's capability to generate data that adheres to the sequential dependencies and patterns found in music, making it ideal for time series prediction.

3. Model Training

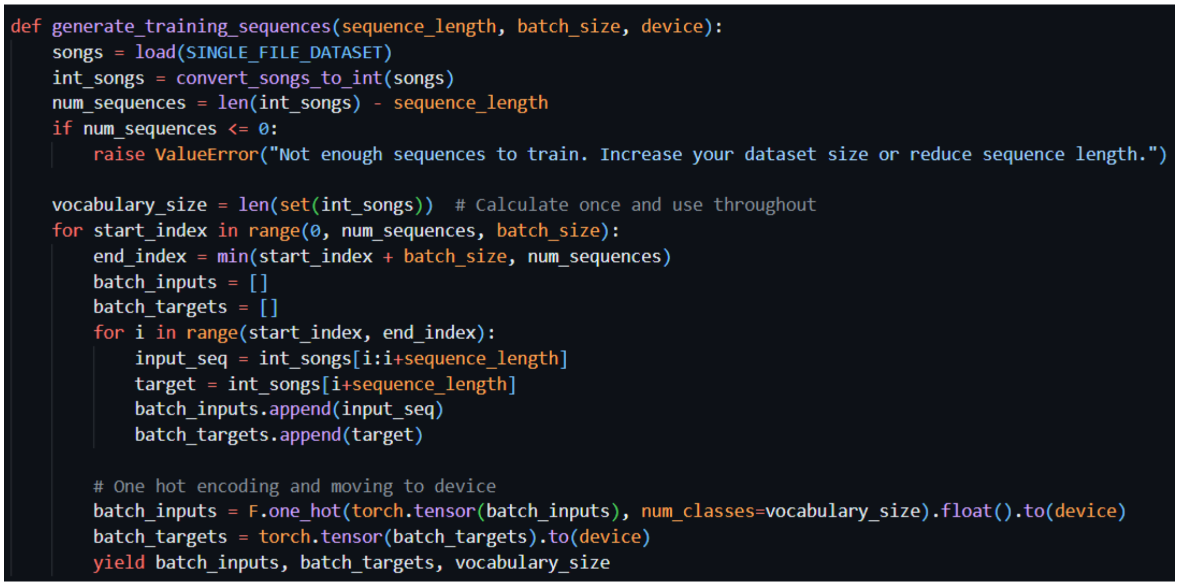
- Training Process Overview
    - Overview of the structured and detailed training process using preprocessed musical sequences.
- Implementation Details
    - Use of the PyTorch framework for efficient handling of dynamic computational graphs and implementation of stochastic gradient descent.
- Initialization and Learning Parameters
    - Initial setting of network weights using the Xavier uniform method.
    - Determination and adjustment of learning rate and batch size based on preliminary experiments and benchmarks.
- Loss Function
    - Description of the loss function combining binary cross-entropy loss and Kullback-Leibler divergence.
- Optimizer setup.
- Training loop.
- Performance monitoring.

Description of Contribution:

1. Inputs in Preprocessing.

Updated the generate_training_sequences function from the preprocess file for VAE.

Batch Handling: The inclusion of batch_size and device parameters facilitates data handling in batches and allows for the utilization of different computing devices. Batch processing is critical in neural network training for managing memory usage effectively, which is particularly important for large datasets. It can also potentially accelerate the convergence of the training process by enabling more frequent updates to the model's weights.



```python
def generate_training_sequences(sequence_length, batch_size, device):
    songs = load(SINGLE_FILE_DATASET)
    int_songs = convert_songs_to_int(songs)
    num_sequences = len(int_songs) - sequence_length
    if num_sequences <= 0:
        raise ValueError("Not enough sequences to train. Increase your dataset size or reduce sequence length.")

    vocabulary_size = len(set(int_songs))  # Calculate once and use throughout
    for start_index in range(0, num_sequences, batch_size):
        end_index = min(start_index + batch_size, num_sequences)
        batch_inputs = []
        batch_targets = []
        for i in range(start_index, end_index):
            input_seq = int_songs[i:i+sequence_length]
            target = int_songs[i+sequence_length]
            batch_inputs.append(input_seq)
            batch_targets.append(target)

        # One hot encoding and moving to device
        batch_inputs = F.one_hot(torch.tensor(batch_inputs), num_classes=vocabulary_size).float().to(device)
        batch_targets = torch.tensor(batch_targets).to(device)
        yield batch_inputs, batch_targets, vocabulary_size
```

Batch-wise Loop: The function iterates from 0 to num_sequences, advancing by batch_size. This approach ensures that the data is processed in manageable chunks, allowing for efficient memory usage and avoiding overloading the compute device.

Dynamic End Index: By calculating the end index with min(start_index + batch_size, num_sequences), the function ensures the correct sizing of the last batch. This is essential for maintaining consistent batch processing even when the total number of sequences does not perfectly align with the batch size. It prevents potential indexing errors and ensures all data is used during training.

2. Model building:

I implemented a Variational Autoencoder (VAE) model to tackle the challenge of generating melodies through deep learning. The VAE is particularly suited for this task because it excels at learning latent representations of input data , crucial for time series prediction in music. This model uses an encoding-decoding framework that efficiently compresses the input data into a latent space and reconstructs it back into the original data space. The encoder transforms the input into latent space representations—mean (μ) and log variance ($\log(\sigma^2)$)—which define a Gaussian distribution for sampling latent variables. This approach is beneficial for time series prediction because it allows the model to generate data that follows the sequential dependencies and patterns found in music, making it ideal for predicting linear time series data like melodies.

Background Information:

The Variational Autoencoder (VAE) employed in this project is a sophisticated deep learning model designed specifically for generating high-fidelity musical sequences. The VAE architecture excels in learning latent representations of input data, which are crucial for capturing the underlying musical structures necessary for time series prediction in melody generation.

Network Architecture:

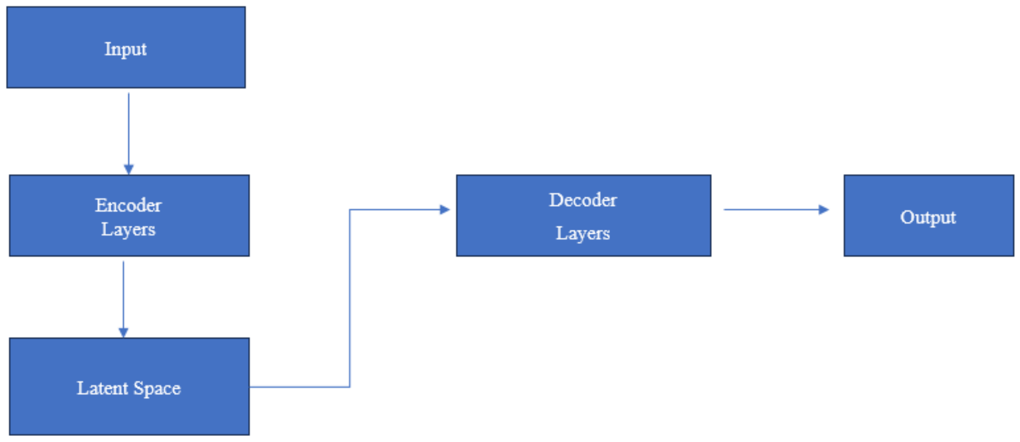


Figure 1. Network Architecture of VAE.

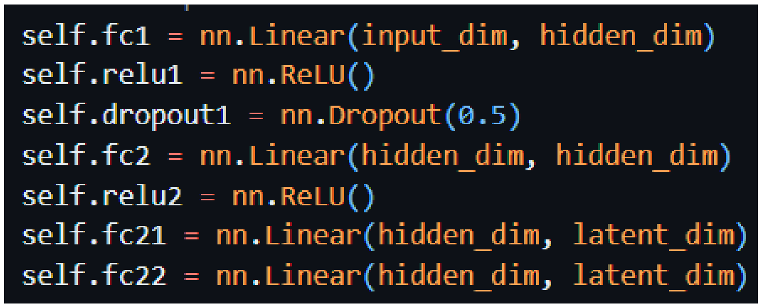
Encoding-Decoding Framework

At the core of the VAE's functionality is its encoding-decoding framework. This framework takes input data and compresses it into a latent space, a process facilitated by the encoder. The latent space holds encoded representations of the input data which capture its essential aspects in a more condensed form. Subsequently, these representations are used by the decoder to reconstruct the input data as closely as possible to its original form. This process is fundamental to the VAE's ability to learn and generate accurate reproductions of complex inputs such as melodies.

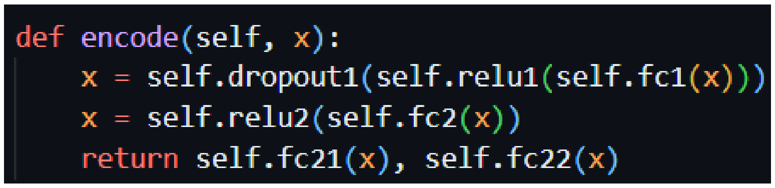
Encoder Layers:

Purpose: The encoder is critical for transforming the input data into a compact latent space. This transformation involves learning the mean (μ) and log variance ($\log(\sigma^2)$) of the data's distribution, from which latent variables are sampled.

Architecture Detail: The encoder consists of two main linear layers with ReLU activation, aimed at compressing the input data while maintaining essential information. This is followed by two additional linear layers that specifically output the parameters for the latent distribution: one for the mean and one for the log variance.



```
self.fc1 = nn.Linear(input_dim, hidden_dim)
self.relu1 = nn.ReLU()
self.dropout1 = nn.Dropout(0.5)
self.fc2 = nn.Linear(hidden_dim, hidden_dim)
self.relu2 = nn.ReLU()
self.fc21 = nn.Linear(hidden_dim, latent_dim)
self.fc22 = nn.Linear(hidden_dim, latent_dim)
```



```
def encode(self, x):
    x = self.dropout1(self.relu1(self.fc1(x)))
    x = self.relu2(self.fc2(x))
    return self.fc21(x), self.fc22(x)
```

Figure 2. Code Snippet of Encoder.

Equation: The output of the encoder is derived from the input x through a series of transformations

$\mu, \log(\sigma^2) = f(x)$, where f represents the network's operation.

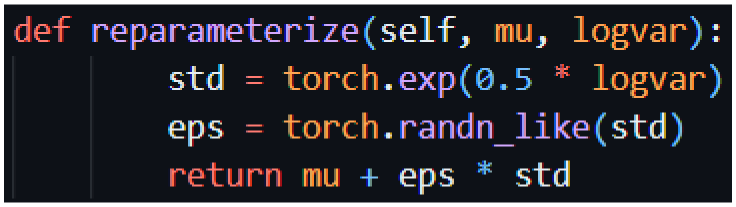
Reparameterization Trick:

Purpose: To allow backpropagation through random operations, providing a pathway for gradients from the decoder back to the encoder.

Architecture Detail: This involves sampling from the latent distribution using the parameters provided by the encoder, combined with a random element to maintain stochasticity.

Equation:

$z = \mu + \sigma \cdot \epsilon,$ where, ϵ is a random variable sampled from a standard normal distribution.



```python
def reparameterize(self, mu, logvar):
        std = torch.exp(0.5 * logvar)
        eps = torch.randn_like(std)
        return mu + eps * std
```

Figure 3. Reparameterize Trick.

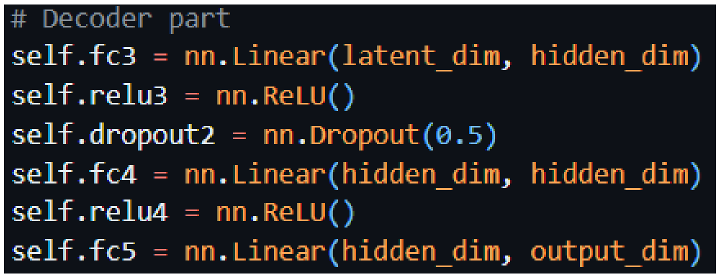
Decoder Layers:

Purpose: To reconstruct the input data from the compressed latent representation, allowing the network to learn a generative model of the data.

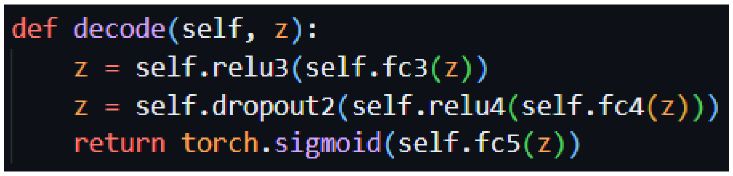
Architecture Detail: Mirroring the encoder, the decoder also consists of several linear layers with ReLU activations, culminating in a final layer with a sigmoid activation to produce output matching the original data scale.

Equation: The output of the decoder is calculated as,

$\hat{x} = g(z)$, where g denotes the decoder operations converting latent variables back to data space.



```python
# Decoder part
self.fc3 = nn.Linear(latent_dim, hidden_dim)
self.relu3 = nn.ReLU()
self.dropout2 = nn.Dropout(0.5)
self.fc4 = nn.Linear(hidden_dim, hidden_dim)
self.relu4 = nn.ReLU()
self.fc5 = nn.Linear(hidden_dim, output_dim)
```



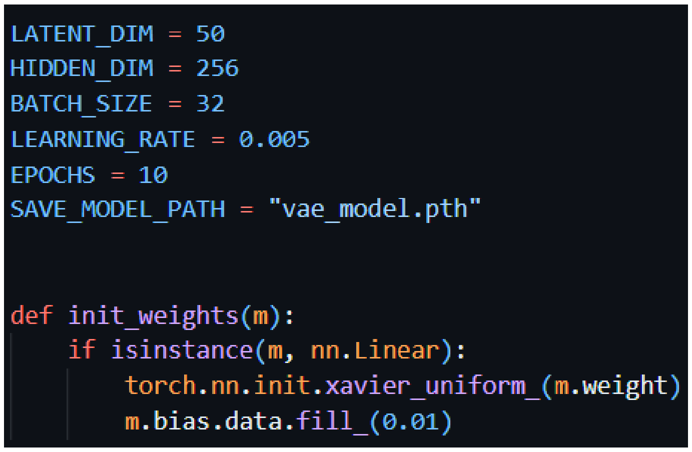
```python
def decode(self, z):
    z = self.relu3(self.fc3(z))
    z = self.dropout2(self.relu4(self.fc4(z)))
    return torch.sigmoid(self.fc5(z))
```

Figure 4. Code Snippet of Decoder.

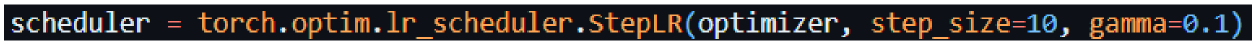
3. Model training:

The training of the Variational Autoencoder (VAE) for melody generation is a structured and detailed process, utilizing preprocessed musical sequences encoded into numerical formats representing various musical attributes such as pitch and rhythm. The VAE's architecture, implemented using the PyTorch framework, includes distinct layers for encoding the input into a latent space and decoding it to reconstruct the original input. This setup leverages PyTorch's ability to handle dynamic computational graphs efficiently, which is crucial for the stochastic gradient descent method used during training.

Initialization and Learning Parameters: Initially, the network weights are set using the Xavier uniform method to promote faster and more stable convergence. Key training parameters, like the learning rate and batch size, are determined based on initial testing and established benchmarks. The learning rate starts at 0.005 and is adjusted by a learning rate scheduler that decreases this rate by 0.1 every ten epochs, aiding in finer adjustments and stabilization as training progresses.



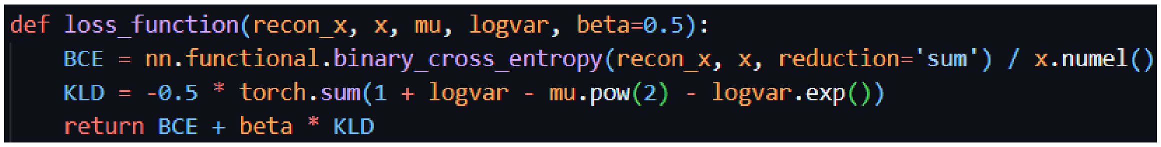
```python
LATENT_DIM = 50
HIDDEN_DIM = 256
BATCH_SIZE = 32
LEARNING_RATE = 0.005
EPOCHS = 10
SAVE_MODEL_PATH = "vae_model.pth"


def init_weights(m):
    if isinstance(m, nn.Linear):
        torch.nn.init.xavier_uniform_(m.weight)
        m.bias.data.fill_(0.01)
```



```python
scheduler = torch.optim.lr_scheduler.StepLR(optimizer, step_size=10, gamma=0.1)
```

Figure 5. Learning Parameters.

Loss Function: The model's performance is primarily evaluated by the loss incurred during training, which consists of binary cross-entropy loss and the Kullback-Leibler divergence. The binary cross-entropy loss assesses the reconstruction accuracy, comparing the original data with the reconstructed output, while the Kullback-Leibler divergence monitors the appropriateness of the latent variable distribution, ensuring it aligns closely with a standard Gaussian distribution.



```python
def loss_function(recon_x, x, mu, logvar, beta=0.5):
    BCE = nn.functional.binary_cross_entropy(recon_x, x, reduction='sum') / x.numel()
    KLD = -0.5 * torch.sum(1 + logvar - mu.pow(2) - logvar.exp())
    return BCE + beta * KLD
```

Figure 6. Loss Function.

Optimizer Setup: The Adam optimizer is used for its adaptive learning rate capabilities, which helps in converging faster and more effectively. A learning rate of 0.005 and a slight weight decay are set to prevent overfitting.

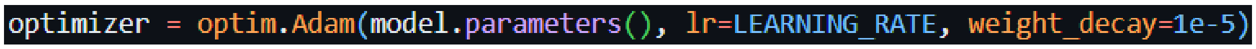

```
optimizer = optim.Adam(model.parameters(), lr=LEARNING_RATE, weight_decay=1e-5)
```

Figure 7. Optimizer setup.

Training loop:

- Batch Processing: The model is trained over multiple epochs, where each epoch iterates over the entire dataset in batches. This is facilitated by a data loader that generates training sequences batch by batch, ensuring that each batch is loaded onto the appropriate device (GPU or CPU).

- Forward Pass: For each batch, the model computes the reconstructed output along with the mean (μ) and log variance (logvar) of the latent variables using the forward pass of the network.

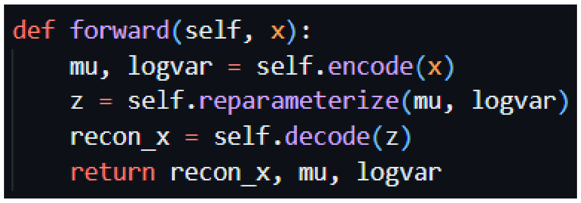

```
def forward(self, x):
    mu, logvar = self.encode(x)
    z = self.reparameterize(mu, logvar)
    recon_x = self.decode(z)
    return recon_x, mu, logvar
```

Figure 8. Forward pass.

- Backward Pass and Optimization: The gradients of the loss function are computed through backpropagation. The optimizer then updates the model parameters based on these gradients. After the optimizer step, the learning rate scheduler adjusts the learning rate according to its schedule.

- Loss Tracking and Output: Throughout the training, the loss for each batch is accumulated to calculate the average loss for the epoch. This average loss is monitored to track the improvement of the model over time. The best model (with the lowest loss) is saved for later use, ensuring that the most optimal parameters are retained.

Performance Monitoring:

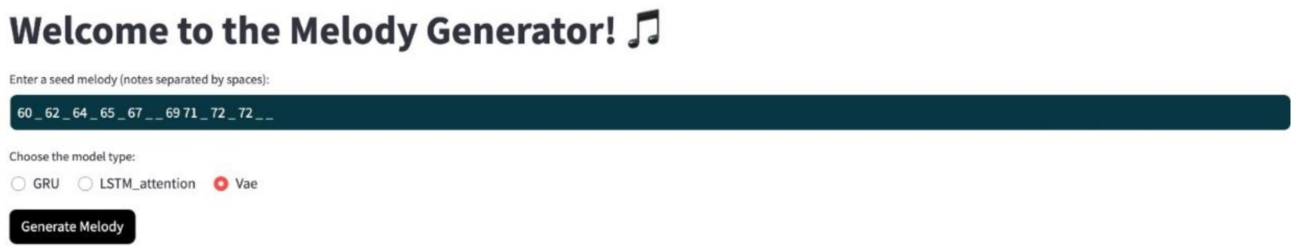
- Progress Tracking: A progress bar is displayed for each epoch, providing real-time feedback on the loss per batch, which helps in monitoring the model's training performance dynamically.

- Model Saving: If the model achieves a lower average loss than any previously recorded, it is saved to disk. This checkpointing strategy ensures that the best-performing model configuration is preserved.

Results:

Seed Input

The seed provided for the VAE model consists of the series of MIDI note numbers: "60, 62, 64, 65, 67, 69, 71, 72,...". These numbers represent specific pitches in the MIDI standard, where each number corresponds to a particular musical note (e.g., 60 = Middle C, 62 = D, 64 = E, etc.). The sequence starts from 60 and moves through various notes up to 72, showcasing a range of one full octave plus a few additional notes. This gives a diverse and rich base from which the AI can develop further musical ideas. This sequence of notes appears to outline a simple ascending scale, serving as the thematic basis upon which the VAE model generates additional notes.


Welcome to the Melody Generator! 🎵

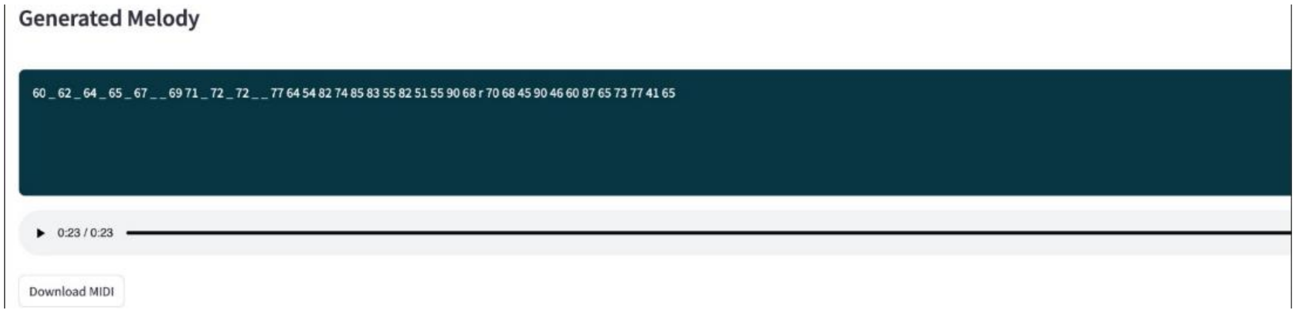
Enter a seed melody (notes separated by spaces):

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

Choose the model type:

GRU LSTM_attention [✓ Vae]

Figure 9. Seed for the VAE.


Generated Melody

```
60 _ 62 _ 64 _ 65 _ 67 _ _ 69 71 _ 72 _ 72 _ _ 77 64 54 82 74 85 83 55 82 51 55 90 68 r 70 68 45 90 46 60 87 65 73 77 41 65
```

Figure 10. Generated melody by VAE.

Generation Process:

The generated melody, as depicted in the musical sheet, illustrates a sophisticated interplay of musical components, enabled by the use of a Variational Autoencoder (VAE) model. The initial seed melody provided forms the base, which is then extensively elaborated upon, demonstrating the system's ability to expand and enrich simple musical inputs into more comprehensive compositions.

Music Sheet

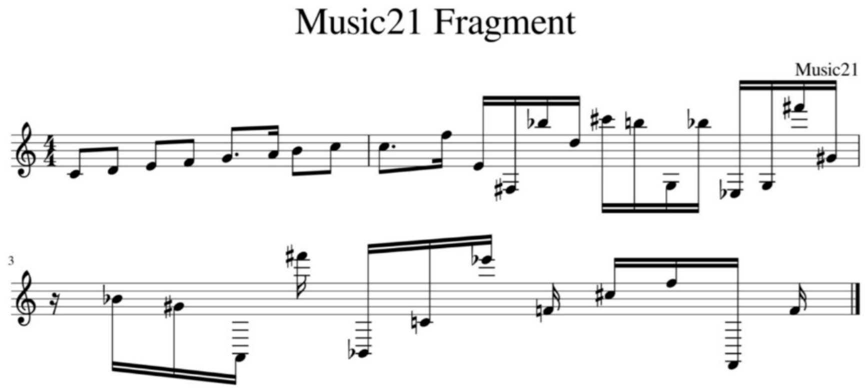


Figure 11. Music Sheet.

Upon receiving the seed melody, the VAE model processes it by initially converting the MIDI numbers into their corresponding integers based on the provided mapping. This sequence of integers is then fed into the VAE, which has been trained to understand and predict musical patterns based on its latent space representation of the input data.

The VAE model generates a continuation of the melody by predicting the next probable note in the sequence at each step. This is achieved through the model's decoder, which reconstructs the possible continuation of the melody from the latent representations. The output sequence from the VAE is then converted back into MIDI numbers using the inverse of the mapping used at the input stage.

Analysis of the Generated Melody

- Rhythmic Development: The system starts with basic rhythmic patterns, mainly using quarter notes, and gradually integrates more intricate rhythmic figures like eighth and sixteenth notes. This progression introduces a dynamic and lively quality to the melody, enhancing its engagement and variability.

- Pitch Variation and Movement: Throughout the composition, there is significant variation in pitch, with the notes moving up and down the scale. This includes both typical melodic ascents and descents as well as unexpected jumps, contributing to a melodically diverse and unpredictable piece.

- Harmonic Complexity: The incorporation of accidentals (sharps and flats) signals a flexible approach to harmony. The melody transitions across various tonal centers rather than sticking to a single key. This is evidenced in the music sheet through changes in key signatures and the use of accidentals, indicating a nuanced approach to tonality that may involve modal interchanges or chromatic elements.

- Structural Coherence: Despite the complexity and diversity in the elements, the melody maintains a coherent structural integrity. The system skillfully blends rhythm, pitch, and harmony into a seamless flow that remains musically coherent. The presence of recurring motifs or thematic elements could also point to the system’s ability to apply compositional techniques typically utilized by human composers.

- Emotional and Aesthetic Impact: The resulting melody, with its dynamic range and expressive rhythmic and harmonic features, is likely to evoke a spectrum of emotions. Variations in key and rhythm invite listeners on a journey through different emotional terrains, making the piece compelling and thought-provoking.

Limitations of the Melody Generator Using a VAE Model

While the Melody Generator leveraging a Variational Autoencoder (VAE) showcases impressive capabilities in musical creativity and complexity, several limitations persist which could affect its application and output quality:

Dependence on Seed Quality: The quality of the generated melody heavily depends on the seed melody provided. A simplistic or monotonous seed might not inspire diverse or complex melodies, limiting the generator's ability to create varied and interesting outputs.

Model Generalization: VAEs, while effective in many scenarios, might struggle with generalizing beyond the types of data they were trained on. If the training data is not sufficiently diverse or large enough, the model might produce outputs that are either too conservative or overly repetitive.

Musical Coherence Over Long Durations: While short snippets might appear coherent, longer sequences generated by VAEs might lose structural and harmonic coherence over time. This could lead to musically illogical or aesthetically displeasing progressions, particularly in complex musical forms or compositions requiring long-term development.

Understanding of Complex Musical Elements: The Variational Autoencoder primarily processes music on a note-by-note basis, which may limit its ability to grasp more intricate musical structures like phrasing, overall form, and thematic development. These elements are crucial for crafting more nuanced and sophisticated musical compositions.

Creativity vs. Novelty: There's a fine line between creativity and randomness. While the model can generate novel sequences, these might not always be musically creative or appealing. Ensuring that the output is both innovative and musically engaging remains a challenge.

User Control and Customization: The current setup offers limited control to users over the specifics of the generation process, such as influencing the style, genre, or harmonic direction of the generated music. This can make it challenging for users with specific creative goals to fully utilize the tool.

Summary and Conclusion:

The Melody Generator, which utilizes a Variational Autoencoder (VAE), marks a notable advancement in computational creativity and music generation. This sophisticated system underscores the potent ability of neural networks to transform basic data inputs into complex and structured musical pieces. The success of the VAE in producing musically cohesive and expressive compositions showcases the dynamic relationship between modern technology and the art of music composition.

Future improvements could focus on enhancing training methods, developing more advanced model architectures, and creating more interactive and intuitive user interfaces. These enhancements would provide users with greater control over the creative process, enabling more personalized and impactful musical expressions. Moreover, incorporating adaptive systems that can modify outputs based on user feedback would likely align the produced music more closely with human musical preferences.

In terms of what I've learned from studying the Variational Autoencoder model, the experience has been deeply enriching. Understanding how VAEs process and recreate data to generate new content has significantly broadened my insight into the effective application of complex models for creative purposes. This learning process has not only highlighted the practical uses of these models in solving real-world challenges but also opened up potential avenues for future developments in creative fields. This investigation into VAE technology underlines the importance of continuous learning and adaptation in the rapidly evolving sphere of computational technologies, reinforcing the notion that these tools are best used as enhancements to human creativity, expanding and enriching our creative capabilities.

Code utilization:

Train File:

Original Lines from the internet = 80

Modified Lines = 20

Added Lines = 15

For the file Train File, the calculation is as follows:
(80 – 20) / (80 + 15) * 100 = 63.15%.

Preprocess File:

Original Lines from the internet = 50

Modified Lines = 15

Added Lines = 10

For the file Preprocess File, the calculation is as follows:
(50 – 15) / (50 + 10) * 100 = 58.33%.

Melody generator File:

Original Lines from the internet = 110

Modified Lines = 25

Added Lines = 10

For the file Preprocess File, the calculation is as follows:
(110 – 15) / (110 + 10) * 100 = 70.83%.

References:

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

[2] Cai, B., Yang, S., Gao, L., & Xiang, Y. (2023). Hybrid Variational Autoencoder for Time Series Forecasting. *Knowledge-Based Systems, 281*, 111079. https://doi.org/10.1016/j.knosys.2023.111079

[3] Kingma, D. P., & Welling, M. (2019). An Introduction to Variational Autoencoders. Foundations and Trends in Machine Learning, 12(4), 307-392. https://doi.org/10.1561/2200000056

[4] Yu, T., & Zhu, H. (2020). Hyper-Parameter Optimization: A Review of Algorithms and Applications. https://doi.org/10.48550/arXiv.2003.05689

[5] Odaibo, S. (2019). Tutorial: Deriving the Standard Variational Autoencoder (VAE) Loss Function. https://doi.org/10.48550/arXiv.1907.08956

[6] ESAC Folk Song Dataset

[7] Bank, D., Koenigstein, N., & Giryes, R. (2021). Autoencoders. https://doi.org/10.48550/arXiv.2003.05991

[8] Wu, J., Hu, C., Wang, Y., Hu, X., & Zhu, J. (2019). A Hierarchical Recurrent Neural Network for Symbolic Melody Generation. IEEE Transactions on Cybernetics, 50(12), 3751-3761. https://doi.org/10.1109/TCYB.2019.2953194

[9] Lu, P., Tan, X., Yu, B., Qin, T., Zhao, S., & Liu, T.-Y. (2022). MeloForm: Generating Melody with Musical Form based on Expert Systems and Neural Networks. https://doi.org/10.48550/arXiv.2208.14345