Task 2:

1. Exploratory analysis, data collection, pre-processing, and discussion:

Context: Where does your dataset come from?

The dataset used in this project is the **Nin Video Game MIDIs** collection. It contains approximately 4,700 MIDI files that were scraped in early 2024 from ninsheetmusic.org, a community-driven website that provides sheet music and MIDI transcriptions of video game soundtracks—primarily from Nintendo titles. The MIDI files are all solo piano transcriptions, representing a wide range of musical styles from different Nintendo franchises. The dataset and its metadata (including transcriber credits) are available on Kaggle.

The dataset was chosen because it aligned well with our goal of **generating melodies in the style of Nintendo video game music**. Specifically, we were curious to explore what kinds of melodies a model could generate when conditioned on iconic chord progressions like those from the **Super Mario Bros.** theme.

Discussion: Data processing steps

We processed the data entirely within our pipeline, using Python and the music21 library. All 3,896 MIDI files were loaded and parsed programmatically. For each file:

- The **melody** was extracted from the first part of the score and converted into a sequence of MIDI pitch integers.

- The **chord progression** was extracted by applying .chordify() to the full score and labeling each chord with its root and quality (e.g., "C_major"). These labels were then converted into integer tokens.

- A **vocabulary** was built for both chords and melody notes by assigning each unique token (chord label or note pitch) a corresponding index.

- We truncated or padded sequences to a fixed maximum length of 64 tokens to ensure uniformity.

- Only sequences where both the chord and melody tracks were exactly 64 tokens long were included in the final dataset.

This process yielded a cleaned and aligned dataset of **chord-melody pairs**, which was then used to train a sequence-to-sequence LSTM model to generate melodies conditioned on chord sequences.

2. Modeling

Modeling: Symbolic Conditioned Melody Generation

Context: Formulation as an ML Problem

Our task is formulated as a sequence-to-sequence learning problem where the goal is to generate a melody (output sequence) conditioned on a given chord progression (input sequence).

- **Inputs:** The input to our model is a sequence of chord labels. These labels, such as "C_major" or "A_minor", are first mapped to a numerical index using a vocabulary (`chord_vocab`) created from the training data. Each input sequence has a fixed length of 64 (`seq_len`).
- **Outputs:** The desired output is a sequence of MIDI note numbers representing the generated melody. These numerical IDs are obtained from a `melody_vocab` built from the notes in the training data. The output sequence also has a fixed length of 64, corresponding to the input chord sequence length.
- **Optimization:** The model is trained to predict the next melody note at each time step given the history of the chord progression. This is treated as a multi-class classification problem over the possible MIDI note values in the `melody_vocab`. The model's parameters are optimized by minimizing the **Cross-Entropy Loss** between the predicted probability distribution over the melody vocabulary and the actual melody note in the training data. The Adam optimizer (`optim.Adam`) is used to update the model's weights during training. The underlying assumption is that by accurately predicting the next note conditioned on the chords and the learned temporal dependencies, the model will generate musically plausible melodies that harmonize with the given chord progression.

For this sequence-to-sequence task, Recurrent Neural Networks (RNNs) are appropriate models due to their ability to process sequential data and maintain an internal state to capture temporal dependencies. Specifically, Long Short-Term Memory (LSTM) networks are well-suited for this task as they address the vanishing/exploding gradient problems of vanilla RNNs, allowing them to learn longer-range dependencies crucial for musical structure. Furthermore, we have incorporated a **bidirectional LSTM** and a **self-attention mechanism** to enhance the model's ability to capture contextual information.

Discussion: Advantages and Disadvantages of Different Modeling Approaches

We initially considered several modeling approaches for this task:

- **Recurrent Neural Networks (RNNs):** Basic RNNs are conceptually simple for sequence modeling. However, they often struggle with learning long-range dependencies in the data due to the vanishing or exploding gradient problem. This limitation could hinder the generation of melodies that maintain coherence over longer musical phrases.

- **Long Short-Term Memory Networks (LSTMs):** LSTMs, with their gating mechanisms, are designed to handle long-range dependencies more effectively than basic RNNs. This makes them well-suited for music generation where the relationship between chords and melody notes can span several time steps.

    - **Advantages:** Effective at capturing temporal dependencies, relatively well-established and understood, and generally less computationally expensive than more complex models like Transformers (though our updated model incorporates more complexity). The addition of bidirectionality allows the LSTM to consider both past and future context within the input sequence at each time step, potentially leading to better informed melody generation.
    - **Disadvantages:** Can still struggle with very long sequences compared to purely attention-based mechanisms.
- **Transformers:** Transformer networks, leveraging self-attention mechanisms, have shown remarkable success in various sequence-to-sequence tasks, including music generation. They excel at capturing both short-range and long-range dependencies in parallel.

    - **Advantages:** Excellent at capturing long-range dependencies, can be highly parallelizable during training.
    - **Disadvantages:** Significantly more computationally expensive in terms of parameters and memory, especially for longer sequences (due to the quadratic complexity of self-attention with respect to sequence length). This high computational cost, particularly the need for powerful GPUs, made their initial implementation challenging given our available resources. However, we have integrated a **multi-head self-attention layer** into our LSTM-based architecture to leverage some of these benefits while still building upon the strengths of the LSTM.
- **Rule-Based Systems and Statistical Models (e.g., Markov Chains):** These approaches could also be considered but often lack the flexibility and ability to learn complex, nuanced relationships present in musical data compared to deep learning models.

Given our computational constraints (specifically the initial lack of a high-end GPU) and the need for a model capable of capturing meaningful temporal dependencies, we initially prioritized the LSTM architecture. We have since enhanced this architecture by incorporating a **bidirectional LSTM** to consider context from both directions in the chord sequence and a **multi-head self-attention layer** to allow the model to weigh the importance of different parts of the input sequence when generating each melody note. This hybrid approach aims to balance computational efficiency with the ability to capture complex musical relationships.

Code: Architectural Choices and Implementation Details

The core of our melody generation model is the `ChordToMelodyLSTM` class, which inherits from `nn.Module`:

- **`__init__(self, chord_vocab, melody_vocab, hidden_dim=128)`**:
    - We initialize an **embedding layer (`nn.Embedding`)** for the input chord sequences. This layer maps each discrete chord index to a continuous vector representation of `hidden_dim` size. This allows the model to learn semantic relationships between different chords. The size of the embedding layer's vocabulary is determined by the `chord_vocab` size.
    - The core of the model is a **two-layer bidirectional LSTM (`nn.LSTM`)**. It takes the embedded chord sequences as input. The `hidden_dim` parameter (set to 128) defines the size of the hidden state vectors in each direction, resulting in a total hidden state size of 2×hidden_dim. `num_layers=2` indicates that we are stacking two bidirectional LSTM layers. `batch_first=True` ensures that the input and output tensors have the batch dimension as the first dimension. A dropout layer (`dropout=0.3`) is included to prevent overfitting. The `bidirectional=True` argument enables the LSTM to process the input sequence in both forward and backward directions, allowing it to capture context from both the past and the future.
    - We have added a **multi-head self-attention layer (`nn.MultiheadAttention`)**. This layer takes the output of the bidirectional LSTM as input (query, key, and value) and allows the model to attend to different parts of the chord sequence when generating each melody note. `embed_dim=hidden_dim*2` reflects the output dimension of the bidirectional LSTM, and `num_heads=2` specifies the number of parallel attention heads. `batch_first=True` ensures the batch dimension is first.
    - The output layer is a **sequential layer (`nn.Sequential`)**. It consists of:
        - A **linear layer (`nn.Linear`)** that maps the output of the attention layer (which has the same dimension as the LSTM output, hidden_dim∗2) to another intermediate representation of size hidden_dim∗2.
        - A **ReLU activation function (`nn.ReLU()`)** to introduce non-linearity.

        - A **dropout layer (`nn.Dropout(0.2)`)** for further regularization.
        - A final **linear layer (`nn.Linear`)** that maps the intermediate representation to a vector with a size equal to the `melody_vocab` size. The values in this output vector (logits) represent the unnormalized probabilities for each possible melody note at that time step.
- **`forward(self, chords)`:**
    - The `forward` method defines the flow of data through the network.
    - The input `chords` (a batch of chord index sequences) is first passed through the `self.chord_embed` layer to obtain their vector embeddings.
    - These embeddings are then fed into the bidirectional LSTM (`self.lstm`), which outputs the hidden state sequence `h` (with a shape of `[batch_size, seq_len, hidden_dim*2]`) and the final hidden and cell states (`_`). We are interested in the sequence of hidden states at each time step, which incorporates information from both forward and backward passes.
    - The output of the LSTM `h` is then passed through the **multi-head self-attention layer (`self.attn`)**. The attention layer takes the LSTM output as query, key, and value, allowing each time step in the sequence to attend to all other time steps. The `attn_output` is the weighted sum of the value vectors, where the weights are determined by the attention mechanism.
    - Finally, the `attn_output` is passed through the sequential output layer `self.out` to produce the `logits`, which represent the model's predictions for the melody notes at each time step.

The training process and the `generate_from_model` and `generate_midi` functions operate on the outputs of the updated model architecture. The training aims to minimize the `CrossEntropyLoss` between the logits produced by the final linear layer and the target melody sequences. The generation process involves feeding a chord sequence through the updated model and sampling from the resulting probability distributions to create a new melody.

This architecture, incorporating a bidirectional LSTM and a multi-head self-attention mechanism followed by a sequential output layer, aims to improve the model's ability to learn complex relationships between the chord progression and the generated melody by considering context from both directions and weighing the importance of different parts of the input sequence.