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.

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.
    - **Disadvantages:** Can still struggle with very long sequences compared to attention-based mechanisms, and their capacity might be limited compared to deeper or wider architectures.
- **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 implementation challenging given our available resources.
- **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 lack of a high-end GPU) and the need for a model capable of capturing meaningful temporal dependencies, we prioritized the LSTM architecture. While Transformers offer potentially higher modeling capacity, the practical limitations of implementation and training efficiency made the LSTM a more feasible and ultimately successful choice for this project.

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 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 within the LSTM. num_layers=2 indicates that we are stacking two LSTM layers, allowing the model to learn hierarchical representations of the input. 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.
    - A **linear output layer (nn.Linear)** maps the output of the LSTM (the hidden state at each time step) 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 LSTM (self.lstm), which outputs the hidden state sequence (h) and the final hidden and cell states (_). We are primarily interested in the sequence of hidden states at each time step.
    - Finally, the sequence of hidden states h is passed through the linear output layer self.out to produce the logits, which represent the model's predictions for the melody notes at each time step.

The ChordMelodyDataset class is a PyTorch Dataset that handles loading and preparing the chord and melody sequences for training. It takes lists of chord and melody sequences and a seq_len as input and returns tensors of corresponding chord and melody sequences of the specified length.

The training process involves iterating through the DataLoader, feeding batches of chord sequences to the model, calculating the CrossEntropyLoss between the model's output logits and the target melody sequences, and using backpropagation and the Adam optimizer to update the model's weights.

The generate_from_model function implements the inference process. It takes a chord sequence as input, feeds it through the trained model, and then samples from the output

probability distribution at each time step (using a temperature parameter to control the randomness of the sampling) to generate a sequence of melody note IDs.

The generate_midi function then converts this sequence of MIDI note IDs back into a playable MIDI file using the pretty_midi library.

This architecture, combining an embedding layer for the conditional input, an LSTM for capturing temporal dependencies, and a linear layer for predicting the output melody notes as a classification task, provides a framework for generating melodies conditioned on chord progressions.