Generating Melodies with LSTM Networks: A Comprehensive Guide

Project Overview

This project focuses on developing a system that utilizes Recurrent Neural Networks (RNNs) with Long Short-Term Memory (LSTM) units to generate folk melodies. The goal is to create novel and unique musical sequences that capture the essence of traditional folk music. This system has potential applications in music composition, background scores for media, and interactive music experiences. This document provides a step-by-step guide through the project, encompassing music theory fundamentals, data preprocessing, model training, and the melody generation process.

Scope and Key Outcomes

The project specifically targets the generation of melodies inspired by folk music traditions. Key outcomes include:

- A trained LSTM model capable of generating diverse and musically coherent folk melodies.
- A comprehensive understanding of the music theory concepts relevant to melody generation.
- A robust data preprocessing pipeline for preparing folk music data for LSTM training.
- A flexible melody generation interface that allows for control over various parameters, such as melody length and creativity.

Understanding Melodies

A melody, at its core, is a sequence of notes and rests. In Western music notation, melodies are represented on a staff, where pitch is represented on the y-axis and time flows from left to right on the x-axis. Each symbol on the staff represents a musical note, which carries information about both pitch and duration.

- **Pitch:** Indicates how high or low a note is and is directly related to frequency. The higher the frequency, the higher the pitch.
- **Duration:** Indicates how long a note is held during a performance.

Scientific Pitch Notation and MIDI

To unambiguously identify notes, the scientific pitch notation (SPN) is used. SPN combines the note name (e.g., C, D#) with an octave number (e.g., C4, D#5). MIDI (Musical Instrument Digital Interface) is a protocol used for recording, editing, and playing digital music. MIDI assigns integer values to notes, with middle C (C4) being 60.

Note Values and Time Signatures

- **Note Values:** These symbols (e.g., whole notes, half notes, quarter notes) express the relative durations of notes.
- **Time Signatures:** These symbols (e.g., 4/4, 3/4) define the number of beats per measure (top number) and the type of note that receives one beat (bottom number).
- **Beats:** The basic units of time in music, often represented by the pulse or tapping of a foot.

Keys and Transposition

- **Keys:** Groups of pitches (scales) that form the core elements of a musical piece. A key is defined by its tonic (the central pitch) and mode (major or minor).
- **Transposition:** The process of shifting a piece of music up or down by a consistent interval, changing the key while preserving the relationships between pitches.

Music Representation for Machine Learning

Traditional music notation is not suitable for direct input into neural networks. Two possible representations for machine learning are:

1. **Sequence of Tuples:** Each tuple contains a note name and duration.
2. **Time Series Representation:** Each time step corresponds to a sixteenth note, and notes are represented by their MIDI values. Special symbols are used for rests and held notes.

This project will use the time series representation, as it is more conducive to the sequential nature of LSTM networks.

Data Preprocessing with music21

The music21 library is a powerful tool for manipulating and analyzing symbolic music data. It allows for loading, converting, and saving music files in various formats (e.g., Kern, MIDI, MusicXML). It also provides methods for analyzing music, such as estimating the key of a piece.

In this project, music21 is used to:

- **Load Songs:** The load_songs_in_kern function reads Kern files from the dataset and parses them into music21 stream objects.
- **Check for Acceptable Durations:** The has_acceptable_duration function filters out songs with note durations that are not in a predefined list of acceptable durations.
- **Transpose Songs:** The transpose function transposes songs to C major or A minor, depending on their original key.

Preprocessing Pipeline

The preprocess function orchestrates the entire preprocessing pipeline:

1. Loads the folk songs from the dataset.
2. Filters out songs with unacceptable durations.
3. Transposes the remaining songs to C major or A minor.
4. Encodes the songs into the music time series representation.
5. Saves each encoded song to a separate text file.

Explanation and Details

- **Checking Acceptable Durations:** The has_acceptable_duration function iterates through all notes and rests in a song, ensuring their durations are within the allowed values.
- **Transposing Songs:** The transpose function analyzes the song's key and transposes it to C major or A minor to simplify the learning process for the model.
- **Loading Songs:** The load_songs_in_kern function uses music21 to load .krn files from the dataset.
- **Preprocessing Pipeline:** The preprocess function combines all the steps, loading, filtering, transposing, and preparing the songs for further analysis or model training.

Project Structure and Dataset

The project folder structure includes the Python file for preprocessing (preprocess.py) and the dataset folder (deutschl) containing .krn files (a symbolic music representation format). The dataset is obtained from the Essen Folk Song Database.

Loading Songs with music21

The core of this section is the implementation of a function called load_songs_in_kern(dataset_path). This function:

1. **Imports necessary libraries:** os for file system navigation and music21 (imported as m21) for music data manipulation.
2. **Iterates through files:** Uses os.walk to recursively traverse the dataset directory.

3. **Filters .krn files:** Selects only files with the .krn extension.
4. **Loads songs:** Employs m21.converter.parse to load each .krn file into a music21 stream object, representing the musical score.
5. **Appends to list:** Adds each loaded song to a list called songs.
6. **Returns list:** Finally, returns the songs list containing all loaded music21 scores.

Filtering Songs by Acceptable Durations

To simplify the learning process for the model, the code filters out songs containing note durations that are not in a predefined list of acceptable durations. This is achieved through the has_acceptable_duration(song, acceptable_durations) function:

1. **Iterates through notes and rests:** Uses song.flat.notesAndRests to access all note and rest objects in the song.
2. **Checks duration:** For each note/rest, compares its quarterLength (duration in quarter notes) against the acceptable_durations list.
3. **Returns False if unacceptable:** If any note/rest has an unacceptable duration, the function immediately returns False.
4. **Returns True if all acceptable:** If the loop completes without finding any unacceptable durations, the function returns True.

Transposing Songs to C Major or A Minor

To further simplify the dataset, the code transposes all songs to either C major (for major key songs) or A minor (for minor key songs). This is done using the transpose(song) function:

1. **Gets the key:** Attempts to extract the key directly from the song's metadata. If not found, estimates the key using song.analyze("key").
2. **Calculates interval:** Determines the interval needed to transpose the song to C major or A minor based on the original key's mode (major or minor).
3. **Transposes song:** Applies the song.transpose(interval) method to shift all notes in the song by the calculated interval.
4. **Returns transposed song:** Returns the modified song in the new key.

Integrating into Preprocessing Pipeline

The preprocess(dataset_path) function orchestrates the entire preprocessing pipeline:

1. **Loads songs:** Calls load_songs_in_kern(dataset_path).
2. **Filters by duration:** Iterates through the loaded songs, keeping only those that pass the has_acceptable_duration check.
3. **Transposes songs:** Applies the transpose function to each remaining song.

4. **(Not shown in video):** The next steps would involve encoding the songs into a music time series representation and saving them to text files, which will be covered in subsequent videos.

Key Improvements

The code presented in the video transcript is a refined version of the initial preprocessing attempt shown in "preprocess_1st_iteration.pdf". The improvements include:

- **Handling of key extraction:** The code now correctly extracts the key from the fourth element ([4]) of the first measure ([0]) of the first part ([0]) of the song.
- **Key estimation:** If the key is not explicitly notated in the song, the code now uses song.analyze("key") to estimate it.
- **Transposition logic:** The transposition logic has been corrected to accurately calculate the interval needed to shift the song to C major or A minor.

Encoding Songs:

The encode_song function converts music21 song objects into the music time series representation. This function takes a music21 stream object as input and returns a string representing the encoded song.

```
Python
def encode_song(song, time_step=0.25):
    """Converts a score into a time-series-like music representation. Each item in the encoded list represents 'min_duration'
    quarter lengths. The symbols used at each step are: integers for MIDI notes, 'r' for representing a rest, and '_'
    for representing notes/rests that are carried over into a new time step. Here's a sample encoding:

        ["r", "_", "60", "_", "_", "_", "72" "_"]

    :param song (m21 stream): Piece to encode
    :param time_step (float): Duration of each time step in quarter length
    :return:
    """

    encoded_song = []

    for event in song.flat.notesAndRests:
```

```
        # handle notes
        if isinstance(event, m21.note.Note):
            symbol = event.pitch.midi # 60
        # handle rests
        elif isinstance(event, m21.note.Rest):
            symbol = "r"

        # convert the note/rest into time series notation
        steps = int(event.duration.quarterLength / time_step)
        for step in range(steps):

            # if it's the first time we see a note/rest, let's encode it. Otherwise, it means we're carrying the same
            # symbol in a new time step
            if step == 0:
                encoded_song.append(symbol)
            else:
                encoded_song.append("_")

    # cast encoded song to str
    encoded_song = " ".join(map(str, encoded_song))

    return encoded_song
```

1. **Iterate through events:** The function iterates through all notesAndRests (notes and rests) in the flattened representation of the song.
2. **Handle notes and rests:**
    - If the event is a Note, the symbol is the MIDI pitch of the note.
    - If the event is a Rest, the symbol is "r".
3. **Convert to time-series notation:**
    - The duration of the event (in quarter lengths) is divided by the time_step (defaulting to 0.25, representing a 16th note) to determine the number of time steps the event spans.
    - A loop iterates for the calculated number of steps:
        - If it's the first step, the symbol is appended to the encoded_song list.
        - For subsequent steps, "_" is appended to indicate the continuation of the note/rest.

4. **Cast to string:** The encoded_song list is converted into a string, with spaces separating the symbols.
5. **Return encoded song:** The function returns the encoded song as a string.

Creating a Single File Dataset:

The create_single_file_dataset function consolidates all encoded songs into a single file, adding delimiters between songs.

```
Python
def create_single_file_dataset(dataset_path, file_dataset_path, sequence_length):
    """Generates a file collating all the encoded songs and adding new piece delimiters.

    :param dataset_path (str): Path to folder containing the encoded songs
    :param file_dataset_path (str): Path to file for saving songs in single file
    :param sequence_length (int): # of time steps to be considered for training
    :return songs (str): String containing all songs in dataset + delimiters
    """

    new_song_delimiter = "/ " * sequence_length
    songs = ""

    # load encoded songs and add delimiters
    for path, _, files in os.walk(dataset_path):
        for file in files:
            file_path = os.path.join(path, file)
            song = load(file_path)
            songs = songs + song + " " + new_song_delimiter

    # remove empty space from last character of string
    songs = songs[:-1]

    # save string that contains all the dataset
    with open(file_dataset_path, "w") as fp:
        fp.write(songs)

    return songs
```

1. **Delimiter:** A delimiter string is created, consisting of "/" characters repeated sequence_length times. This delimiter will be used to separate songs in the combined file.
2. **Load and combine songs:** The function iterates through the files in the dataset directory, loads each encoded song, and appends it to the songs string along with the delimiter.
3. **Remove trailing space:** The last character (an extra space) is removed from the songs string.
4. **Save to file:** The combined songs string is saved to the specified file path.
5. **Return combined songs:** The function returns the songs string.

Creating a Mapping

The create_mapping function creates a dictionary that maps each unique symbol in the dataset (MIDI note numbers, 'R', '_') to a unique integer. This mapping is essential because the LSTM network can only process numerical data. The mapping is saved as a JSON file for later use during model training and melody generation.

```
Python
def create_mapping(songs, mapping_path):
    """Creates a json file that maps the symbols in the song dataset onto integers

    :param songs (str): String with all songs
    :param mapping_path (str): Path where to save mapping
    :return:
    """
    mappings = {}

    # identify the vocabulary
    songs = songs.split()
    vocabulary = list(set(songs))

    # create mappings
    for i, symbol in enumerate(vocabulary):
        mappings[symbol] = i

    # save voabulary to a json file
    with open(mapping_path, "w") as fp:
        json.dump(mappings, fp, indent=4)
```

1. **Identify Vocabulary:** The function first splits the songs string into a list of individual symbols. It then creates a set from this list to get unique symbols, eliminating duplicates. Finally, it converts the set back into a list called vocabulary.
2. **Create Mappings:** The function iterates through the vocabulary list and assigns a unique integer to each symbol. The integer is the index of the symbol in the vocabulary list.
3. **Save Vocabulary:** The resulting mappings dictionary, which maps symbols to integers, is saved as a JSON file at the specified mapping_path.

Converting Songs to Integers

The convert_songs_to_int function converts a string of encoded songs into a list of integers using the mapping created by create_mapping.

```
Python
def convert_songs_to_int(songs):
    int_songs = []

    # load mappings
    with open(MAPPING_PATH, "r") as fp:
        mappings = json.load(fp)

    # transform songs string to list
    songs = songs.split()

    # map songs to int
    for symbol in songs:
        int_songs.append(mappings[symbol])

    return int_songs
```

1. **Load Mappings:** The function loads the mappings dictionary from the JSON file created earlier.
2. **Split Songs:** The input songs string (containing all encoded songs) is split into a list of individual symbols.

3. **Map to Integers:** The function iterates through the symbols and looks up their corresponding integer values in the mappings dictionary. These integers are appended to the int_songs list.
4. **Return Integer Representation:** The function returns the int_songs list, which is the integer representation of the encoded songs.

Generating Training Sequences

The generate_training_sequences function is a crucial step in preparing the data for training the LSTM model. It creates input and output pairs for the model to learn from.

```
Python
def generate_training_sequences(sequence_length):
    """Create input and output data samples for training. Each sample is a sequence.

    :param sequence_length (int): Length of each sequence. With a quantisation at 16th notes,
64 notes equates to 4 bars

    :return inputs (ndarray): Training inputs
    :return targets (ndarray): Training targets
    """

    # load songs and map them to int
    songs = load(SINGLE_FILE_DATASET)
    int_songs = convert_songs_to_int(songs)

    inputs = []
    targets = []

    # generate the training sequences
    num_sequences = len(int_songs) - sequence_length
    for i in range(num_sequences):
        inputs.append(int_songs[i:i+sequence_length])
        targets.append(int_songs[i+sequence_length])

    # one-hot encode the sequences
    vocabulary_size = len(set(int_songs))
    # inputs size: (# of sequences, sequence length, vocabulary size)
    inputs = keras.utils.to_categorical(inputs, num_classes=vocabulary_size)
    targets = np.array(targets)
```

```
    print(f"There are {len(inputs)} sequences.")

    return inputs, targets
```

1. **Load and Convert Songs:** The function loads the encoded songs from the single file dataset and converts them into a list of integers using the convert_songs_to_int function.
2. **Generate Sequences:**
    - It calculates the number of sequences that can be generated based on the length of the int_songs list and the desired sequence_length.
    - It iterates through the int_songs list, creating sequences of length sequence_length and their corresponding targets (the next integer in the list).
3. **One-Hot Encode Sequences:**
    - The sequences are one-hot encoded using keras.utils.to_categorical. This converts each integer in the sequence into a binary vector where only one element is 1 (representing the integer) and all others are 0.
    - The vocabulary_size is calculated as the number of unique integers in the int_songs list.
4. **Return Inputs and Targets:** The function returns two NumPy arrays: inputs (the one-hot encoded sequences) and targets (the corresponding target integers).

Building the LSTM Network

The build_model function is responsible for constructing the architecture of the LSTM neural network using Keras.

```
Python
def build_model(output_units, num_units, loss, learning_rate):
    """Builds and compiles model

    :param output_units (int): Num output units
    :param num_units (list of int): Num of units in hidden layers
    :param loss (str): Type of loss function to use
    :param learning_rate (float): Learning rate to apply

    :return model (tf model): Where the magic happens :D
```

```
"""

# create the model architecture
input = keras.layers.Input(shape=(None, output_units))
x = keras.layers.LSTM(num_units[0])(input)
x = keras.layers.Dropout(0.2)(x)

output = keras.layers.Dense(output_units, activation="softmax")(x)

model = keras.Model(input, output)

# compile model
model.compile(loss=loss,
              optimizer=keras.optimizers.Adam(learning_rate=learning_rate),
              metrics=["accuracy"])

model.summary()

return model
```

1. **Input Layer:**
    - keras.layers.Input(shape=(None, output_units)): This line creates the input layer of the model.
    - The shape argument specifies the shape of the input data. In this case, it's a 3D tensor where:
        - The first dimension is None, meaning the model can accept sequences of any length.
        - The second dimension is also None, allowing for variable-length sequences during inference.
        - The third dimension is output_units, which is the size of the vocabulary (number of unique symbols in the dataset). Each time step in the input sequence will be a one-hot encoded vector of this size.
2. **LSTM Layer:**
    - x = keras.layers.LSTM(num_units[0])(input): This line adds an LSTM layer to the model.

    - LSTMs are a type of RNN designed to capture long-term dependencies in sequential data, making them well-suited for music generation.
    - num_units[0] specifies the number of units (neurons) in the LSTM layer. This determines the model's capacity to learn complex patterns.
3. **Dropout Layer:**
    - x = keras.layers.Dropout(0.2)(x): This line adds a dropout layer with a rate of 0.2.
    - Dropout is a regularization technique that helps prevent overfitting by randomly dropping out (setting to zero) a fraction of the units during training.
4. **Output Layer:**
    - output = keras.layers.Dense(output_units, activation="softmax")(x): This line creates the output layer of the model.
    - It's a dense (fully connected) layer with output_units neurons (same as the vocabulary size).
    - The softmax activation function is used to ensure that the output is a probability distribution over the vocabulary. Each neuron in the output layer represents a symbol in the vocabulary, and the softmax activation ensures that the output values sum to 1, representing the probabilities of each symbol being the next in the sequence.
5. **Model Creation and Compilation:**
    - model = keras.Model(input, output): This line creates the Keras model, specifying the input and output layers.
    - model.compile(...): This line compiles the model, configuring it for training.
        - loss: Specifies the loss function to use during training. In this case, it's "sparse_categorical_crossentropy," which is suitable for integer targets.
        - optimizer: Specifies the optimization algorithm to use. Adam is a popular choice for deep learning models.
        - metrics: Specifies the metrics to track during training. Here, it's "accuracy," which measures the percentage of correct predictions.
6. **Model Summary:**
    - model.summary(): This line prints a summary of the model architecture, including the number of parameters in each layer. This is helpful for understanding the model's complexity and potential for overfitting.
7. **Return Model:** The function returns the compiled Keras model, ready for training.

Training the Model

The train function is the main function that orchestrates the training process. It generates the training sequences, builds the model, trains it, and saves the trained model.

```
Python
```

```
def train(output_units=OUTPUT_UNITS, num_units=NUM_UNITS, loss=LOSS,
learning_rate=LEARNING_RATE):
    """Train and save TF model.

    :param output_units (int): Num output units
    :param num_units (list of int): Num of units in hidden layers
    :param loss (str): Type of loss function to use
    :param learning_rate (float): Learning rate to apply
    """

    # generate the training sequences
    inputs, targets = generate_training_sequences(SEQUENCE_LENGTH)

    # build the network
    model = build_model(output_units, num_units, loss, learning_rate)

    # train the model
    model.fit(inputs, targets, epochs=EPOCHS, batch_size=BATCH_SIZE)

    # save the model
    model.save(SAVE_MODEL_PATH)
```

1. **Generate Training Sequences:**
    - Calls the generate_training_sequences function to get the input sequences (inputs) and their corresponding target values (targets).
    - The inputs are sequences of 64 integers representing musical events, and the targets are the integers representing the next event in the sequence.
2. **Build Model:**
    - Calls the build_model function to create the LSTM model architecture.
    - The model is built with the specified hyperparameters:
        - output_units: The number of unique symbols in the dataset (vocabulary size).
        - num_units: A list containing the number of units in each LSTM layer (in this case, only one layer with 256 units).
        - loss: The loss function used for training (sparse categorical crossentropy).
        - learning_rate: The learning rate for the Adam optimizer.

3. **Train Model:**
    - The fit method is used to train the model.
    - It takes the inputs and targets as training data.
    - epochs specifies the number of times the model will iterate over the entire dataset (50 in this case).
    - batch_size specifies the number of samples processed before the model updates its weights (64 in this case).
4. **Save Model:**
    - After training, the save method is used to save the model to the file specified by SAVE_MODEL_PATH ("model.h5").
    - This allows the trained model to be loaded and used for melody generation later.

GPU Utilization

The code also includes a check for GPU availability. If a GPU is found, it enables memory growth, allowing the model to utilize the GPU's resources for faster training. If no GPU is available, the training will be performed on the CPU.

```
Python
if __name__ == "__main__":
    # Check for GPU availability
    physical_devices = tf.config.experimental.list_physical_devices('GPU')
    if len(physical_devices) > 0:
        tf.config.experimental.set_memory_growth(physical_devices[0], True)
        print("GPU is available and will be used for training.")
    else:
        print("No GPU found. Training will be performed on the CPU.")

    train()
```

Key Points and Considerations

- **Data Preparation:** The quality and quantity of the training data significantly impact the model's performance. The preprocessing steps ensure that the data is clean, consistent, and in a suitable format for the LSTM network.

- **Model Architecture:** The choice of LSTM architecture (number of layers, units per layer) and hyperparameters (loss function, optimizer, learning rate, epochs, batch size) can affect the model's ability to learn and generate melodies.
- **Training Time:** Training deep learning models can be computationally expensive and time-consuming, especially with large datasets. Using a GPU can significantly speed up the training process.
- **Evaluation:** While the code doesn't explicitly include an evaluation step, it's crucial to evaluate the model's performance on a separate test set to assess its ability to generalize to unseen data.

This detailed explanation of the code and the underlying concepts should provide a comprehensive understanding of how to train an LSTM network for melody generation.

Melody Generation

The final phase involves using the trained model to generate new melodies. This is achieved through the MelodyGenerator class, which provides a convenient interface for interacting with the model.

1. **MelodyGenerator Class:**
    - The MelodyGenerator class encapsulates the melody generation functionality.
    - It loads the trained Keras model and initializes attributes for mappings, start symbols, and sequence length.

```
Python
class MelodyGenerator:
    """A class that wraps the LSTM model and offers utilities to generate melodies."""

    def __init__(self, model_path="model.h5"):
        """Constructor that initializes TensorFlow model"""
        self.model_path = model_path
        self.model = keras.models.load_model(model_path)
        with open(MAPPING_PATH, "r") as fp:
            self._mappings = json.load(fp)
        self._start_symbols = ["/"] * SEQUENCE_LENGTH
```

2. **generate_melody Method:**
    - This method is the core of the melody generation process.

- It takes a seed melody (a short sequence of notes), the number of steps to generate, the maximum sequence length, and a temperature parameter as input.
- The seed melody serves as the starting point for the generated melody.
- The number of steps determines the length of the generated melody.
- The maximum sequence length limits the amount of context the model considers when generating each note.
- The temperature parameter controls the randomness of the generated melody. Higher temperatures result in more creative and unpredictable melodies, while lower temperatures produce more conservative and repetitive ones.

```
Python
  def generate_melody(self, seed, num_steps, max_sequence_length, temperature):
      """Generates a melody using the DL model and returns a midi file."""
      seed = seed.split()
      melody = seed
      seed = self._start_symbols + seed
      seed = [self._mappings[symbol] for symbol in seed]
      for _ in range(num_steps):
          seed = seed[-max_sequence_length:]
          onehot_seed = keras.utils.to_categorical(seed, num_classes=len(self._mappings))
          onehot_seed = onehot_seed[np.newaxis, ...]
          probabilities = self.model.predict(onehot_seed)[0]
          output_int = self._sample_with_temperature(probabilities, temperature)
          seed.append(output_int)
          output_symbol = [k for k, v in self._mappings.items() if v == output_int][0]
          if output_symbol == "/":
              break
          melody.append(output_symbol)
      return melody
```

- **Split Seed:** The seed melody is split into a list of symbols.
- **Initialize Melody:** The melody list is initialized with the seed.
- **Prepend Start Symbols:** The _start_symbols (a sequence of '/') are added to the beginning of the seed.
- **Map to Integers:** The symbols in the seed are mapped to their corresponding integer representations using the _mappings dictionary.

- **Iterative Generation:** The function enters a loop that iterates for the specified num_steps. In each iteration:
    - The seed is trimmed to the max_sequence_length.
    - The seed is one-hot encoded.
    - The model predicts the probability distribution for the next note.
    - A new note is sampled from the probability distribution using _sample_with_temperature.
    - The sampled note is appended to both the seed and melody lists.
    - If the sampled note is the end-of-song delimiter ('/'), the loop breaks.

3. **Temperature Sampling (**_sample_with_temperature**)**:
    - This method implements temperature sampling, a technique that allows for controlling the randomness of the generated melody.
    - Higher temperatures lead to more diverse and unpredictable melodies, while lower temperatures result in more conservative and repetitive ones.

```
Python
def _sample_with_temperature(self, probabilities, temperature):
    """Samples an index from a probability array reapplying softmax using temperature"""
    predictions = np.log(probabilities) / temperature
    probabilities = np.exp(predictions) / np.sum(np.exp(predictions))
    choices = range(len(probabilities))
    index = np.random.choice(choices, p=probabilities)
    return index
```

- **Logarithmic Transformation:** The probabilities are transformed using a logarithm and divided by the temperature.
- **Softmax Reapplication:** The softmax function is reapplied to the transformed values to obtain a new probability distribution.
- **Sampling:** An index is sampled from the modified distribution using np.random.choice, where the probabilities of each index are given by the modified distribution.
- **Saving Melody (**save_melody**)**:
    - Takes a melody (list of symbols), step_duration (duration of each time step in quarter lengths), format (the desired output format, defaulting to MIDI), and file_name (the name of the output file) as input.
    - Creates an empty music21 stream.
    - Iterates through the melody:
        - If the current symbol is not "_" (hold) or it's the last symbol in the melody:

        - Calculates the quarter length duration of the previous note/rest.
        - Creates a music21 Rest object if the previous symbol was "r", or a Note object otherwise.
        - Appends the m21_event to the stream.
        - Resets the step_counter.
    - If the current symbol is "_", increments the step_counter.
- Writes the stream to a file in the specified format and file name.

```
Python
def save_melody(self, melody, step_duration=0.25, format="midi", file_name="mel.mid"):
    """Converts a melody into a MIDI file"""
    stream = m21.stream.Stream()
    start_symbol = None
    step_counter = 1

    for i, symbol in enumerate(melody):
        # Handle case in which we have a note/rest
        if symbol != "_" or i + 1 == len(melody):
            # Ensure we're beyond the first symbol
            if start_symbol is not None:
                quarter_length_duration = step_duration * step_counter  # Calculate duration
                # If it's a rest, create a Rest object
                if start_symbol == "r":
                    m21_event = m21.note.Rest(quarterLength=quarter_length_duration)
                # If it's not a rest, create a Note object
                else:
                    m21_event = m21.note.Note(int(start_symbol),
quarterLength=quarter_length_duration)
                stream.append(m21_event)  # Append the event to the stream
                step_counter = 1  # Reset the step counter
            start_symbol = symbol  # Update the start symbol
        # Handle case in which we have a prolongation sign "_"
        else:
            step_counter += 1  # Increase step counter

    stream.write(format, file_name)  # Write the stream to a MIDI file
```

1. **Create Stream:** An empty music21 stream object is created to hold the musical events.

2. **Initialize Variables:**
    - start_symbol: Keeps track of the current note or rest being processed.
    - step_counter: Counts the number of time steps for the current note/rest.
3. **Iterate Through Melody:**
    - The function iterates through each symbol in the melody list.
    - If the symbol is not a hold ('_') or it's the last symbol:
        - If start_symbol is not None (meaning a previous note/rest is being held), the duration of that note/rest is calculated based on the step_counter and step_duration.
        - A music21 Rest or Note object is created based on the start_symbol.
        - The created event is appended to the stream.
        - The step_counter is reset to 1.
    - If the symbol is a hold ('_'), the step_counter is incremented.
4. **Write to MIDI File:**
    - Finally, the stream.write method is called to write the musical events in the stream to a MIDI file.
    - The format argument specifies the file format (MIDI in this case), and file_name specifies the name of the output file.

Conclusion

This project successfully demonstrates the potential of LSTM networks for generating folk melodies that capture the essence of traditional music. The generated melodies exhibit both musical coherence and creative flair, adhering to fundamental music theory concepts while introducing variations and novel combinations of musical ideas. The use of temperature sampling further enhances the model's flexibility, allowing for control over the degree of randomness and innovation in the output.

Through a comprehensive approach encompassing music theory fundamentals, data preprocessing, model architecture design, and training procedures, this project offers valuable insights into the application of deep learning techniques in music generation. The resulting LSTM model not only produces musically pleasing melodies but also provides a foundation for future research and development in this exciting field.

Key Contributions and Future Directions

- Robust Methodology: The project establishes a robust and adaptable methodology for training LSTM networks on symbolic music data, which can be extended to other musical styles or genres.
- Creative Potential: The generated melodies demonstrate the creative potential of AI in music composition, offering new possibilities for musicians, composers, and music enthusiasts.
- Interdisciplinary Collaboration: The project highlights the benefits of interdisciplinary collaboration between music theory, computer science, and artificial intelligence.
- Future Research: Potential future work includes exploring more sophisticated model architectures, incorporating additional musical features (harmony, rhythm), and developing interactive systems that allow users to guide the melody generation process.

By pushing the boundaries of AI-driven music generation, this project contributes to the ongoing dialogue between human creativity and artificial intelligence, opening up new avenues for musical expression and exploration.