Music Generation using an LSTM model

By Vasudeva H N(221175)
Prof. Sandip Tiwari

November 25, 2024

1 Objective

This project focuses on generating melodies using a Recurrent Neural Network (RNN) with Long Short-Term Memory (LSTM) architecture. By treating melody creation as a time-series prediction problem, the model captures long-term dependencies in musical patterns. Leveraging tools like Keras, Music21, and MuseScore, the project involves training the network on symbolic music data, such as traditional folk melodies, to produce coherent and structured sequences. Some key objectives are:

1. Load the songs in the dataset and filter out the songs that have non acceptable durations. After the filter, transpose the songs to either C major to A minor just to ensure easy learning for the model. Encode the songs in music time series representation and save the song to text file.

2. Create a single file dataset of the encode songs as it would be easier to manipulate to create the sequences for the training process ,and create a lookup table to map the symbols that are in the encoded form to integers to feed in the neural networks.

3. Encode the songs in the file created and making it a string ,add delimiters to distinguish the songs for generating the sequences for the neural networks and save the string.

4. Create a mapping to integers for the identification of the symbols in the dataset string and save the vocabulary in the json file for loading in the training process. Load the mapping table and cast the songs to a list. Map the symbols in the list to integers using the table and convert the songs to integer list.

5. Generate the input training sequences with a fixed sequence length and one hot-encode the sequences created as it's the easiest way to deal with a categorical data with neural networks and create a target array which contains the target elements for each sequence for model training.

6. Build the RNN-LSTM network by creating the model architecture and compiling the model. We add a dropout layer to avoid overfitting. Train the model and save the network to proceed for further testing of the model.

7. Make predictions on the possible output values for the input,and sample the output symbols with temperature from the probability distribution that comes as the output to ensure a flexible prediction of the output. Map the output integer to the MIDI symbol using the mapping table.

8. Create a music21 stream for the conversion of the melodies we obtained as the result in time series representation to the MIDI objects. Parse all the symbols in the melodies and create note/rest objects. Write the midi stream to a midi file and save the file. Using Musescore we can listen to the melody and analyse the notes sequence.

2 Approach

2.1 Data Preprocessing for Melody Generation Using RNN-LSTM

The goal of this task is to preprocess symbolic music data in Kern format to prepare it for training an RNN-LSTM model for melody generation. Preprocessing involves loading songs, ensuring acceptable

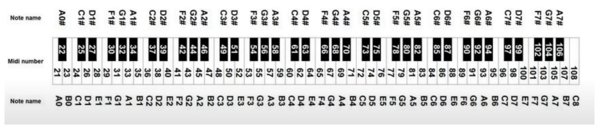
Figure 1: Midi notes

note durations, transposing them to a common key (C major/A minor), and encoding them for further processing.

2.1.1 Loading Songs from Dataset

The function `load_songs_in_kern()` is used to load all Kern files (`.krn`) from the specified dataset directory using the *music21* library. This function iterates through the dataset folder structure, parses each Kern file, and stores the songs as *music21 stream objects.* The total number of songs successfully loaded is printed for verification.

2.1.2 Filtering Songs with Acceptable Durations

The function `has_acceptable_durations()` filters out songs containing notes or rests with durations that are not in a predefined list of acceptable durations (e.g., quarter notes, half notes). This ensures consistency in note durations during model training. Songs with unacceptable note durations are excluded from further processing.

Transposing Songs to C Major/A Minor To standardize the dataset, each song is transposed to either C major or A minor. This is achieved using the `transpose()` function, which analyzes the key of the song and calculates the appropriate interval for transposition:

- Analyze the key of the song (major or minor).
- Calculate the transposition interval (e.g., B major to C major).
- Transpose the song by the calculated interval using *music21's transpose method.*

Songs are transposed to a standardized key, facilitating consistent training data.

2.2 Encoding Songs for Time-Series Representation

To convert symbolic music data into a time-series-like representation that can be used for training the RNN-LSTM model. This involves encoding notes and rests into symbols that represent musical events at specific time steps.

2.2.1 Encoding Songs

The `encode_song()` function processes each note and rest in the input song to create a sequence that represents time-series data. Each time step is expressed in terms of a quarter-length duration, with the following symbols:

- **MIDI Numbers:** Used for representing musical notes (e.g., 60 for Middle C).
- **'r':** Represents a rest.
- **"_":** Denotes the continuation of a note or rest across multiple time steps.

For example, for $p = 63$ and $d = 1$ (a quarter note), the representation is a list `[63, ‘‘_’’, ‘‘_’’, ‘‘_’’]`.

2.2.2 Integrating the Encoding Process

Each song in the preprocessed dataset is passed through the `encode_song()` function to generate its time-series representation.

2.2.3 Saving the Encoded Songs

Encoded songs are stored as text files, one for each song, in a specified directory (`SAVE_DIR`). This makes it easier to load and use the encoded data for training later. Each file is named using an index to maintain order. Each encoded song represents a melody in a structured, time-series format. The songs are saved in individual text files as strings for subsequent processing during training.

2.3 Combining and Mapping Encoded Songs for Model Training

To create a unified dataset from individual encoded song files, introduce delimiters for separating songs, and map symbols (notes, rests, etc.) to integers for efficient processing during training.

2.3.1 Creating a Single File Dataset

The `create_single_file_dataset()` function consolidates all encoded songs into a single file while adding a delimiter to indicate the end of each song. This simplifies data handling for sequence-based models.

- Load each encoded song from the dataset folder.
- Append a delimiter ("/ " repeated for `sequence_length`) after each song.
- Save the consolidated string to a file for future use.

A single file containing all encoded songs with delimiters to separate pieces.

2.3.2 Creating a Mapping of Symbols to Integers

The `create_mapping()` function maps the symbols in the consolidated song dataset (e.g., MIDI notes, "r", "_") to unique integers. This mapping is essential for converting symbolic data into numerical form, a prerequisite for training neural networks.

- Split the songs string to extract all unique symbols.
- Assign a unique integer to each symbol.
- Save the mapping as a JSON file for reuse during training and inference.

Key points:

- Unified Dataset : Combining all encoded songs simplifies data handling for sequence-based models.
- Symbol-to-Integer Mapping: Essential for converting symbolic music representation into numerical input for neural networks.
- Sequence Delimiters: Introduce structure in the dataset, helping the model differentiate between melodies during training.

2.4 Converting and Preparing Songs for Training

The aim is to transform the preprocessed symbolic music dataset into numerical sequences and prepare the inputs and targets necessary for training an RNN-LSTM model.

2.4.1 Converting Songs to Integer Representation

The function `convert_songs_to_int()` transforms the symbolic representation of songs into integer sequences using the pre-generated symbol-to-integer mapping.

1. Load the mapping from the JSON file.
2. Split the consolidated song string into individual symbols.
3. Replace each symbol with its corresponding integer based on the mapping.

A list of integers representing the entire song dataset.

2.4.2 Generating Training Sequences

The function `generate_training_sequences()` creates sequences for training the RNN-LSTM model. Each sequence consists of an input (a fixed-length sequence of integers) and a target (the next symbol to predict).

1. Load the consolidated dataset from the single file.
2. Convert the symbolic representation to integers using `convert_songs_to_int()`.
3. Generate sequences of a fixed length (`sequence_length`) from the integer dataset.
4. One-hot encode the input sequences for compatibility with the model.
5. Store the inputs and corresponding targets for training.

Now we have a 3D array of shape (number of sequences, sequence length, vocabulary size) containing one-hot encoded training sequences as the inputs and a 1D array containing the corresponding next symbol for each input sequence as the target. The resulting inputs and targets are ready for training the RNN-LSTM model, enabling it to learn patterns and generate musically coherent sequences.

Key points:

- Symbolic music data is mapped to integers for efficient processing by neural networks.
- Fixed-length input sequences and corresponding targets are generated for supervised learning.
- Inputs are one-hot encoded to ensure compatibility with the categorical output format of the RNN-LSTM model.

2.5 Building and Training the Melody Generation Model

Construct and train an RNN-LSTM-based neural network capable of generating melodies from symbolic music data. The model leverages TensorFlow/Keras for implementation and is designed to predict the next musical event based on a sequence of inputs.

2.5.1 Model Architecture

The `build_model()` function creates the RNN-LSTM model.

– Input Layer: Accepts sequences of variable length, with features equal to the vocabulary size (number of unique symbols).

– LSTM Layer: Processes sequential data and captures temporal dependencies with 256 units (modifiable).

– Dropout Layer: Regularization technique to prevent overfitting by randomly setting 20

– Dense Output Layer: Outputs probabilities for each possible musical event using a softmax activation function.

2.5.2 Training the Model

1. Use `generate_training_sequences()` to create input-output pairs from the preprocessed dataset.
2. One-hot encoded sequences of length SEQUENCE_LENGTH as the input.
3. The next symbol in the sequence as the target.

4. Invoke `build_model()` with the specified parameters to create the RNN-LSTM architecture.
5. Use the fit() method to train the model on the generated data. The model is trained for 50 epochs with a batch size of 64.
6. Save the Model: Save the trained model to a file (model.h5) for reuse during inference.

This implementation establishes a robust pipeline for training an RNN-LSTM model for melody generation. The model learns to predict the next musical event in a sequence, enabling it to generate musically coherent compositions. With further fine-tuning and experimentation, the system can be enhanced to produce melodies that align with specific styles or patterns.

2.6 Generating music using the LSTM model

Generate melodies using a trained LSTM model, integrating advanced mathematical techniques to dynamically adjust randomness and symbol biases for enhanced control over melody structure and diversity.

2.6.1 Symbol weight initialisation

Defines weights for each musical symbol. For example, rests can be penalized slightly to reduce excessive gaps in the melody.

– Rests (r) are weighted lower (0.9) to make them less frequent.

– Notes are given a default weight of 1.0.

2.6.2 Melody Generation

– **Parse the seed:** Convert the seed into a sequence (`seed_sequence`).

– **Pad with start symbols:** Add `self._start_symbols` to ensure the seed has the correct length.

– **Encode the seed:** Map the seed symbols to integers using the predefined mapping (`self._mappings`).

2.6.3 Iterative Melody Generation

Loop for `num_steps`, generating one note or rest at each step:

– Limit the sequence to the last `max_sequence_length` symbols.

– One-hot encode the sequence for compatibility with the model.

– Predict the next symbol probabilities using the trained model.

– Adjust probabilities dynamically using entropy-based temperature scaling and symbol biasing.

– Sample the next symbol and update the sequence with the generated symbol.

2.6.4 Iterative Generation Logic

– **Sliding Window:** Keeps only the last `max_sequence_length` symbols to use as the input to the model.

– **One-Hot Encoding:** Converts the integer sequence to a binary matrix, where each row corresponds to a symbol. The matrix shape is:

$$(1, \texttt{max\_sequence\_length}, \texttt{vocabulary\_size}).$$

2.6.5 Prediction and temperature adjustments

- **Prediction:** The model predicts probabilities for the next symbol based on the input sequence.
- **Entropy Calculation:** Shannon entropy is calculated as:

  $$H(p) = -\sum_i p_i \log(p_i)$$

  which measures the diversity of the probabilities.

  * **Low entropy** (values close to 0): The model is confident; increase randomness.
  * **High entropy** (values greater than 2.5): The model is uncertain; decrease randomness.
- **Temperature Adjustment:** Dynamically modifies the randomness of the predictions using the formula:

  $$p'_i = \frac{\exp\left(\frac{\log p_i}{T}\right)}{\sum_j \exp\left(\frac{\log p_j}{T}\right)}$$

  where T is the temperature parameter.
- **Bias Adjustment and Sampling:** Each probability is multiplied by the corresponding symbol weight w_i:

  $$p'_i = \frac{p'_i \cdot w_i}{\sum_j p'_j \cdot w_j}$$

  This ensures that less desirable symbols (e.g., rests) are less likely to be chosen. A symbol is then selected based on the adjusted probabilities.

2.6.6 Updating the melody

- The chosen index is mapped back to its corresponding symbol.
- If the end-of-melody symbol (/) is reached, generation stops.
- The generated symbol is appended to the melody.

Combine temperature scaling and symbol weighting for enhanced control over predictions. This integrates advanced mathematical techniques to create musically rich and diverse outputs. By incorporating entropy-based dynamic temperature adjustment and symbol biasing, the model balances predictability and creativity, producing melodies that are both structured and varied.

2.7 Convert the generated melodies to MIDI

This code extends the melody generator to include functionality for saving generated melodies as MIDI files using the `save_melody()` method. The method parses the generated melody sequence, converting symbols (notes, rests, and prolongation markers) into music21 note or rest objects with calculated durations. These objects are assembled into a music21.stream.Stream, which is then exported as a MIDI file. The main driver initializes the enhanced melody generator, generates melodies using seeds with dynamic temperature adjustments and bias corrections, and saves them as MIDI files. This enables the generation and export of musically coherent, AI-driven melodies.

3 Results

The results demonstrate the effectiveness of an LSTM-based architecture enhanced with dynamic mathematical techniques like Shannon entropy-driven temperature adjustments and symbol biasing.

- The model generated musically diverse and structured melodies from symbolic seeds, showcasing its ability to learn long-term dependencies and maintain coherence over extended sequences.
- Dynamic temperature adjustments ensured a balance between randomness and structure, reducing repetitive patterns and avoiding chaotic outputs. Weighted adjustments in symbol probabilities minimized excessive rests, maintaining musical flow and rhythmic appeal.
- The generated melodies were converted into MIDI files using music21, allowing playback and further analysis in music editing software. These files demonstrated coherent timing, accurate note durations, and logical progression based on the seed input.
- The LSTM model consistently achieved high accuracy in predicting the next symbols during training, translating to high-quality melodies during inference.

4 Failures and learning

During the project, several approaches were explored but did not yield the desired results, prompting adjustments:

1. Fixed Temperature for Sampling: Initially, a constant temperature was used for sampling probabilities during melody generation. This led to either overly repetitive melodies (low temperature) or chaotic outputs (high temperature). So, I introduced entropy-driven dynamic temperature scaling, allowing the model to adaptively balance randomness and structure based on the prediction uncertainty.
2. Uniform Symbol Probabilities: Assigning equal probabilities to all symbols in the vocabulary during sampling caused excessive rests and disrupted rhythmic flow. A biasing mechanism was implemented, weighting symbols like rests less, ensuring musically appealing patterns.
3. Short Sequence Lengths: Short training sequences limited the model's ability to capture long-term dependencies in melodies. So, I increased sequence length, ensuring the model could learn broader contextual patterns.

These iterative improvements refined the model's ability to generate coherent and diverse melodies.

5 Value of Research

This research bridges artificial intelligence and music composition, showcasing how neural networks can generate coherent and creative melodies. By integrating mathematical techniques the model balances predictability and creativity, mimicking human-like composition. The ability to train models on diverse datasets enables applications in personalized music generation, therapeutic soundscapes, and dynamic audio systems for games and films. This work contributes to advancing AI in the arts, offering new tools for composers and highlighting the potential for machine learning to enhance cultural and creative industries. It underscores the synergy between mathematics, machine learning and art.

6 References

– Lecture slides of EE798Z to learn how LSTM works and its algorithm.

– R. Pascanu, T. Mikolov, and Y. Bengio, "On the difficulty of training recurrent neural networks," 30th Int. Conf. Mach. Learn. ICML 2013, no. PART 3, pp. 2347–2355, 2013.