Music Generation using Deep Neural Network

Individual Report

for the course

DATS 6303 Deep Learning

Submitted by: Tushar Sharma

GWID: G24951989

Index

Introduction

Preprocessing modifications

Stacked bidirectional LSTM architecture building and training

Discussion of results

I. Introduction

The intricate patterns within music make the task of automated melody generation a fascinating challenge in the realm of artificial intelligence. In this project, I explored the potential of deep learning models to generate folk-inspired melodies. My contributions centered around initial modifications to the preprocessing pipeline suitable for a large dataset of melodies. I then implemented a stacked, bidirectional LSTM model using the robust PyTorch framework. Addressing limitations found in the original codebase, I implemented dynamic vocabulary handling to ensure the model could adapt to diverse datasets. Currently, my focus is on integrating multi-head attention mechanisms. This has the potential to enhance the model's ability to learn complex long-range dependencies within musical sequences.

II. Preprocessing modifications

- **Dataset:** I decided to use a subset of the total dataset for the project, aiming to use approximately 1700 songs (still yielding over 360,000 training sequences) contained in the folder called ‘erk’. This ambition was aimed to check the model’s (which was to be built) performance.

- **Flexible Directory Handling:** I replaced the original hardcoded directory path with logic that dynamically checks for the existence of a "dataset_erk" folder. If the folder doesn't exist, it is created automatically. This improvement makes the code more adaptable and user-friendly in different environments. The relevant code snippet for this logic is shown in Fig. 1.

```
# THIS SCRIPT IS DESIGNED TO LOAD A TOTAL OF 1700 SONGS ('ERK' FOLDER INSIDE MAIN DATA FOLDER) ONLY

# Constants
KERN_DATASET_PATH = "/home/ubuntu/generating-melodies-with-rnn-lstm/Preprocessing-dataset-for-melody-generation-pt-1/code/deutschl/erk/"

folder_name = "dataset_erk"
if not os.path.exists(folder_name):
    os.makedirs(folder_name)
    print(f"Folder '{folder_name}' created.")
else:
    print(f"Folder '{folder_name}' already exists, loading songs from it.")

SAVE_DIR = "dataset_erk"
SINGLE_FILE_DATASET = "file_dataset"
MAPPING_PATH = "mapping.json"
SEQUENCE_LENGTH = 64
```

Fig.1. Code snippet showing the flexible directory handling.

- **Key-aware Filtering:** I made a fundamental change in how the key of melodies was handled. Instead of transposing all songs into a single key, I implemented logic to identify and preserve the original song's major or minor tonality. Since major and minor are the most prevalent melodic structures in Western music, this filtering aimed to reduce noise during training and potentially improve the model's focus on core musical patterns. Songs not in major or minor keys were skipped during the preprocessing stage. The code snippet for the function is shown in Fig. 2. However, as our team progressed through the project, it was decided to instead create a function to transpose into all they 24 keys instead of not transposing at all and only checking for tonality.

```python
# MODIFIED 'TRANSPOSE' FUNCTION TO 'NOT' TRANSPOSE ALL THE MELODIES TO A SINGLE KEY
# SUCH THAT THE MODEL COULD BE EXPOSED TO A WIDER RANGE OF KEYS TO LEARN FROM.
# THUS, IN THIS FUNCTION, WE JUST CHECK IF THE MELODY HAS A MAJOR OR MINOR TONALITY OR NOT.

def bypass_transpose(song):
  """Analyzes the song key and keeps it within 'major' or 'minor' tonality.

  :param song (m21 stream): Piece to analyze
  :return song (m21 stream): Original song (not transposed)
  """

  # Analyze key using music21
  key = song.analyze("key")

  # Check if key is not already major or minor
  if not (key.mode == "major" or key.mode == "minor"):
    print(f"Song {song} has a mode that is not major or minor. Skipping.")
    return None

  # Keep the song in its original key
  return song
```

Fig.2. Code snippet showing the modified transposition function.

III. Stacked bidirectional LSTM architecture building and training

LSTM background

- **Recurrent Neural Networks (RNNs):** LSTMs are a specialized type of RNN designed to handle sequential data like text or music. Vanilla (basic) RNNs suffer from the "vanishing gradients" problem, making it difficult for them to learn long-term dependencies.

- **LSTM Solution:** LSTMs solve the vanishing gradients problem by introducing memory cells and gates that regulate information flow. This allows them to selectively retain or forget information over longer stretches of a sequence, improving their ability to handle temporal patterns. The LSTM cell is shown in Fig. 3.

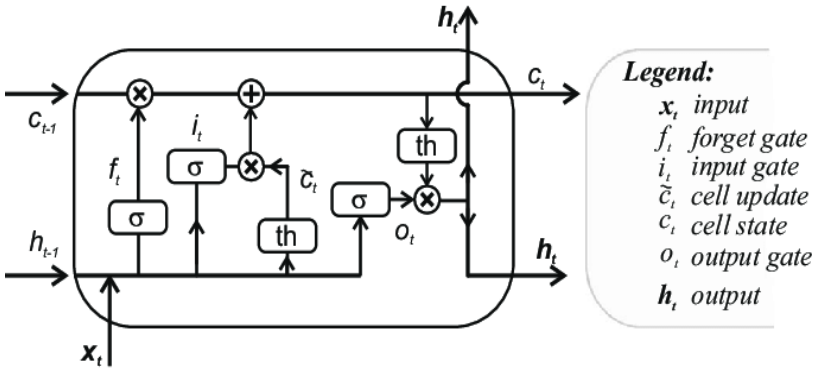

Fig. 3. Diagram showing the LSTM cell.

Framework Shift: The most fundamental change is the transition from TensorFlow (Keras) to PyTorch. This brings a different approach to defining model architecture, handling data, optimization, etc.

Bidirectional LSTMs: The core model architecture now employs bidirectional LSTMs, which process the input sequence both in the forward and reverse directions. This potentially allows the model to better capture long-range dependencies in musical sequences.

Stacked Layers: The model has been made deeper with a stacked LSTM architecture. Two LSTM layers are used, with the output of the first layer feeding into the second. This can help the model learn more complex hierarchical representations of the music.

1. Setup

- **Constants:** Defines parameters like hidden layer sizes (`NUM_UNITS`), loss function (nn.CrossEntropyLoss), learning rate, epochs, batch size, and the model save path.
- **Device Check:** Checks if a GPU (CUDA) is available and uses it for training if so. Otherwise, defaults to CPU.

2. Data Loading & Preprocessing

- **Loading Vocabulary:** Loads the `mapping.json` file (likely created during preprocessing) containing the unique musical symbols found in the dataset. The `OUTPUT_UNITS` is dynamically set based on the vocabulary size.

Potential Bug Fix: Dynamic Vocabulary Handling

- **Original Issue:** The original code hardcoded the `OUTPUT_UNITS` variable to a specific value (38), assuming a fixed dataset vocabulary. This would lead to errors if the dataset changed, as the number of output neurons needs to match the number of unique symbols.
- **Modification:** The modified code addresses this by loading the `mapping.json` file generated during preprocessing. It calculates the number of unique symbols from this file and dynamically sets the `OUTPUT_UNITS` value accordingly.

The relevant code snippet of this handling is shown in Fig. 4.

```
# Load the json file contaning the unique symbols
with open('mapping.json', 'r') as file:
    data = json.load(file)
    size = len(data)

# flexible output size based on unique symbols found in train data
OUTPUT_UNITS = size
```

Fig. 4. Code snippet showing the bug fix.

- **Assumptions:** It's assumed that the preprocessing function `generate_training_sequences` now creates PyTorch-compatible data (Tensors) instead of the structures originally used with TensorFlow.

3. Dataset Class

- **`MusicDataset`:** A PyTorch `Dataset` class is defined to handle the input and target sequences. It provides methods for getting the length (`__len__`) and retrieving individual samples (`__getitem__`). The snippet for the MusicDataset class is shown in Fig. 5.

```
# Class for dataset
class MusicDataset(Dataset):
    def __init__(self, inputs, targets):
        self.inputs = inputs
        self.targets = targets

    def __len__(self):
        return len(self.inputs)

    def __getitem__(self, index):
        return self.inputs[index], self.targets[index]
```

Fig. 5. Code snippet showing the MusicDataset class definition.

4. Model Architecture (`LSTMModel`)

- **Bidirectionality:** Each LSTM layer explicitly sets the `bidirectional=True` parameter.

- **Stacking:** Two consecutive `nn.LSTM` layers are used, creating the stacked architecture. The output of the first layer is passed as input to the second.
- **Dropouts:** Dropout layers (`nn.Dropout`) are added after each LSTM layer to help prevent overfitting.
- **Output Layer:** A linear layer (`nn.Linear`) maps the final hidden state to the output units, corresponding to the size of the music vocabulary.

The relevant code snippet showing the LSTM model is shown in Fig. 5.

```
# Bidirectional stacked LSTM Model architecture
class LSTMModel(nn.Module):
  def __init__(self, output_units, num_units):
    super(LSTMModel, self).__init__()
    self.lstm1 = nn.LSTM(output_units, num_units[0], batch_first=True, bidirectional=True)
    self.dropout1 = nn.Dropout(0.2)

    self.lstm2 = nn.LSTM(num_units[0] * 2, num_units[1], batch_first=True, bidirectional=True)  # Second LSTM layer
    self.dropout2 = nn.Dropout(0.2)

    self.linear = nn.Linear(num_units[1] * 2, output_units)  # Output layer

  # Forward pass
  def forward(self, x):
    output, _ = self.lstm1(x)
    output = self.dropout1(output)

    output, _ = self.lstm2(output)  # Pass output of first LSTM through the second
    output = self.dropout2(output)

    output = output[:, -1, :]  # Take the last timestep output
    output = self.linear(output)
    return output
```

Fig. 6.. Code snippet showing the LSTMModel class definition.

5. Model Building and Training Loop

- **`build_model`:** Helper function to instantiate the `LSTMModel`, define the loss function (`nn.CrossEntropyLoss`) and optimizer (`torch.optim.Adam`). The model is moved to the appropriate device (GPU in this case).
- **`train` Function:**
    - Loads input and target sequences (assumed to already be PyTorch Tensors).
    - Creates the `MusicDataset` and a `DataLoader` to handle mini-batches.
    - Calls `build_model` to get the model, its loss function, and the optimizer.
    - Iterates for the specified number of epochs:
        - Processes mini-batches from the `DataLoader`.
        - Moves data to the correct device.

        - Zeros gradients, calculates output, loss, and performs backpropagation.
        - Updates weights with the optimizer.
        - Prints the loss for monitoring.
    - Saves the trained model state to `SAVE_MODEL_PATH`.

The relevant code snippet showing the build_model function is shown in Fig. 7.

```python
# Build model
def build_model(output_units, num_units, loss_function, learning_rate):
    model = LSTMModel(output_units, num_units)
    model = model.to(device)
    loss_fn = loss_function
    optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)
    return model, loss_fn, optimizer

# Training loop
def train(output_units=OUTPUT_UNITS, num_units=NUM_UNITS, loss_function=LOSS_FUNCTION, learning_rate=LEARNING_RATE):

    inputs, targets = generate_training_sequences(SEQUENCE_LENGTH)
    inputs = inputs.clone().detach().requires_grad_(False)
    targets = targets.clone().detach().requires_grad_(False)

    dataset = MusicDataset(inputs, targets)
    dataloader = DataLoader(dataset, batch_size=BATCH_SIZE, shuffle=True)

    model, loss_fn, optimizer = build_model(output_units, num_units, loss_function, learning_rate)

    for epoch in range(EPOCHS):
        for batch_x, batch_y in dataloader:
            batch_x = batch_x.to(device).float()
            batch_y = batch_y.to(device)
            optimizer.zero_grad()
            output = model(batch_x)
            loss = loss_fn(output, batch_y)
            loss.backward()
            optimizer.step()

        print(f"Epoch {epoch+1}, Loss: {loss.item()}")

    torch.save(model.state_dict(), SAVE_MODEL_PATH)
```

Fig. 7. Code snippet showing the build model function definition.

IV. Melody Generation with Modified Model and Parameters

The `MelodyGenerator` class and its associated functions have been adapted to work with the custom trained bidirectional LSTM model and to experiment with different generation parameters:

- **Loading the New Model:** The `__init__` method of the `MelodyGenerator` class now loads your trained model from "model_bidirectional_LSTM.pt", replacing the original model loading mechanism.

- **Input Seed Customization:** In the `__main__` section, the `seed` input has been changed to a B minor scale sequence. This demonstrates how you can control the starting point for the melody generation process.
- **Temperature Adjustment:** The `generate_melody` takes a `temperature` argument explicitly set to 0.5. Recall that temperature controls the randomness of the generation: lower temperatures make the model more deterministic, while higher values increase unpredictability.

The relevant code snippet for the change in melody generation file is show in Fig. 8.

```
def __init__(self, model_path = "model_bidirectional_LSTM.pt"):
    """Constructor that initialises PyTorch model"""

    self.model_path = model_path
    self.model = LSTMModel(OUTPUT_UNITS, NUM_UNITS)
    self.model.load_state_dict(torch.load(model_path, map_location = torch.device('cpu')))
    self.model.eval()

    with open(MAPPING_PATH, "r") as fp:
        self._mappings = json.load(fp)

    self._start_symbols = ["/"] * SEQUENCE_LENGTH
```

```
if __name__ == "__main__":
    mg = MelodyGenerator()
    seed = "59 _ 61 _ 62 _ _ 66 67 _ 69 _ _"
    seed2 = "67 _ _ _ _ _ 65 _ 64 _ 62 _ 60 _ _ _"
    melody = mg.generate_melody(seed, 500, SEQUENCE_LENGTH, 0.5)   # temperature set to 0.5 here
    print('Extrapolated melody sequence:', melody)
    mg.save_melody(melody)
```

Fig. 8. Code snippet showing the model call and seed input modification.

V. Discussion of results

Based on the B minor scale input seed to the model, the melody generated by the model is shown in Fig. 9.

- ❖ The melody was able to capture some of the more complex musical ideas in the training data, but was not able to correctly stay in key and changed key from B minor to C

major, and then G major towards the end. This could be due to the nature of preprocessing (especially the bypassing of transposition) and model architecture.

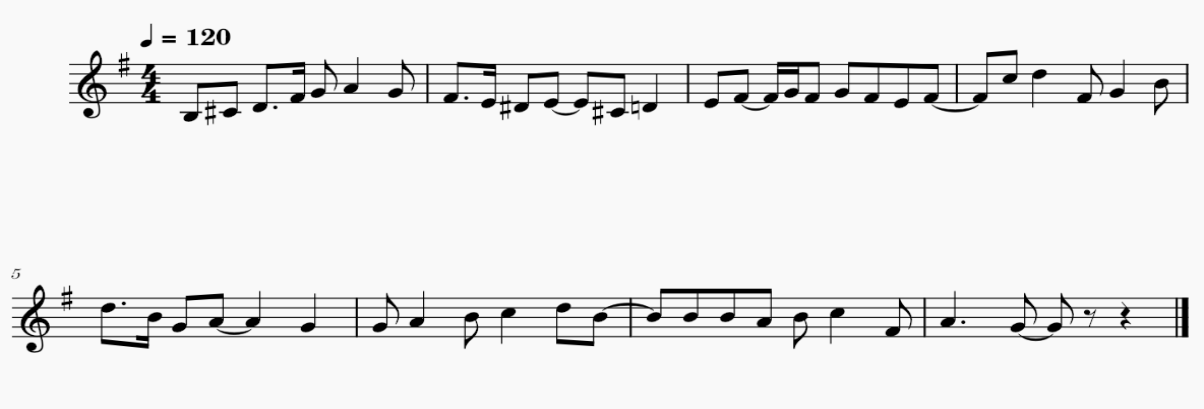

Fig. 9. The final melody generated after training the stacked bidirectional LSTM model.

While the initial stacked, bidirectional LSTM model provided decent results, the decision was made to explore a more complex architecture incorporating multi-head attention mechanisms to potentially improve performance.

Percentage of code taken from the internet:

preprocess_erk.py

Lines copied from the internet: 106
Lines modified: 12
Lines added: 6
Index calculation: {(106 – 12) / (106 + 6)}*100 = 83.9%

train_bidirectional_lstm.py

Lines copied from the internet: 55
Lines modified: 25
Lines added: 57
Index calculation: {(55 – 25) / (55 + 57)} * 100 = 26.8%

melodygenerator_bidirectional_lstm.py

Lines copied from the internet: 92

Lines modified: 14

Lines added: 5

Index calculation: {(92 – 14) / (92 + 5)}* 100 = 80.4%

References

S. Hochreiter and J. Schmidhuber, "Long Short-Term Memory," in Neural Computation, vol. 9, no. 8, pp. 1735-1780, 15 Nov. 1997, doi: 10.1162/neco.1997.9.8.1735.