Melody Generation from Classical Music with LSTMs

Leveraging neural networks to create novel musical sequences

Siddharth Shah · Follow
8 min read · Dec 11, 2023

By: Siddharth Shah and Ian Pompliano

Introduction and Motivation:

In recent years, the fusion of AI and art has emerged as a captivating frontier. We have seen generative models produce impressive work, obscuring the boundary between humans and machines in artistic creation. As musicians and computer scientists, we wanted to investigate music generation via machine learning with the goal of producing meaningful melodies that could be used to enhance the creative process of song/score writing.

To accomplish this, we aimed to train a recurrent neural network model architecture called LSTM on sequential musical notes. LSTMs possess the ability to selectively store, read, and remove information from their memory cell to capture dependencies over longer sequences, making them more effective for tasks that require understanding context over time. By feeding in sequences of real musical data and training the model to predict the subsequent note following the given sequence, we can capture a greater degree of musical structure and tones.

Data Collection and Processing:

In order to meaningfully train our machine learning model, we needed to first collect solid music data. We chose to get this data from http://www.piano-midi.de/, a site containing the compositions (335 in total) of famous classical piano composers.

Music data for each composition on the site is stored in a MIDI file under the composer's respective sub-page. In order to scrape each of these MIDI files for processing, we used the Beautiful Soup Python library to find instances of links that led to our desired file type. Once we found these links, we downloaded the MIDI files to a work directory.

```
baseURL = 'http://www.piano-midi.de/'

# create a directory to save downloaded MIDI files
if not os.path.exists('midiFiles'):
    os.makedirs('midiFiles')

for composer in composers:
    # create URL using respective composer
    url = baseURL + composer + '.htm'
    response = requests.get(url)

    soup = BeautifulSoup(response.content, 'html.parser')

    # Find MIDI links on the composer's page
    midiLinks = soup.find_all('a', href=True)

    for link in midiLinks:
        # removes duplicate MIDI files (special format0)
        if link['href'].endswith('.mid') and '_format0' not in link['href']:
            midiURL = baseURL + link['href']
            fileName = f"midiFiles/{composer}_{link['href'].split('/')[-1]}"

            # download MIDI file and save it to directory
            with open(fileName, 'wb') as midiFile:
                midiResponse = requests.get(midiURL)
                midiFile.write(midiResponse.content)
```

BeautifulSoup to scrape MIDI files from the web

Now that we had collected all of our MIDI files, we were ready to begin processing note data for our model. Because our focus was on generating melodies, we collected single note data using the music21 toolkit, which is capable of representing notes in MIDI files as string values (e.g. “G#4, F7”).

One issue we ran into while collecting our data was that there are often chords (multiple notes played at the same time) in classical piano music. To solve this, we elected to take the highest pitched note in the chord, which typically belongs to the melody. Although this is not a perfect solution, as there are occasionally melodies played with the internal (or even low) notes of chords, we believed this would yield the best results.

```python
# helper function to get the notes in a given MIDI file
def getNotes(file):
    # initalize list of notes to return
    notes = []
    pick = file.recurse()

    for element in pick:
        # if element is note, add to list of notes
        if isinstance(element, note.Note):
            notes.append(str(element.pitch))
        # if element is chord, add highest pitch (generally belongs to melody) to list of notes.
        elif isinstance(element, chord.Chord):
            highestPitch = max(element.pitches)
            notes.append(str(highestPitch))
    return notes
```

```python
# initialize list that will hold sub-lists of notes for each song
allNotes = []

# retrieve paths of MIDI files
midiFiles = [os.path.join('midiFiles', file) for file in os.listdir('midiFiles') if file.endswith('.mid')]

# append allNotes with note data for each song
for path in midiFiles:
    midi = converter.parse(path)
    notes = getNotes(midi)
    allNotes.append(notes)
```

Parsing through saved MIDI files for note sequences

At this point, we had collected the note data for over 300 MIDI files into a giant list of sublists, where each sublist contained all of the note data for one composition. Our final step was to map each of the individual notes to integers, which we could sequence and feed into the model for training.

```python
def mapNote(note):

    noteMapping = {
        'C2': 0, 'C#2': 1, 'D-2': 1, 'D2': 2, 'D#2': 3, 'E-2': 3, 'E2': 4, 'F2': 5, 'F#2': 6,
        'G-2': 6, 'G2': 7, 'G#2': 8, 'A-2': 8, 'A2': 9, 'A#2': 10, 'B-2': 10, 'B2': 11,
        'C3': 12, 'C#3': 13, 'D-3': 13, 'D3': 14, 'D#3': 15, 'E-3': 15, 'E3': 16, 'F3': 17, 'F#3': 18,
        'G-3': 18, 'G3': 19, 'G#3': 20, 'A-3': 20, 'A3': 21, 'A#3': 22, 'B-3': 22, 'B3': 23,
        'C4': 24, 'C#4': 25, 'D-4': 25, 'D4': 26, 'D#4': 27, 'E-4': 27, 'E4': 28, 'F4': 29, 'F#4': 30,
        'G-4': 30, 'G4': 31, 'G#4': 32, 'A-4': 32, 'A4': 33, 'A#4': 34, 'B-4': 34, 'B4': 35,
        'C5': 36, 'C#5': 37, 'D-5': 37, 'D5': 38, 'D#5': 39, 'E-5': 39, 'E5': 40, 'F5': 41, 'F#5': 42,
        'G-5': 42, 'G5': 43, 'G#5': 44, 'A-5': 44, 'A5': 45, 'A#5': 46, 'B-5': 46, 'B5': 47,
        'C6': 48, 'C#6': 49, 'D-6': 49, 'D6': 50, 'D#6': 51, 'E-6': 51, 'E6': 52, 'F6': 53, 'F#6': 54,
        'G-6': 54, 'G6': 55, 'G#6': 56, 'A-6': 56, 'A6': 57, 'A#6': 58, 'B-6': 58, 'B6': 59,
        'C7': 60, 'C#7': 61, 'D-7': 61, 'D7': 62, 'D#7': 63, 'E-7': 63, 'E7': 64, 'F7': 65, 'F#7': 66,
        'G-7': 66, 'G7': 67, 'G#7': 68, 'A-7': 68, 'A7': 69, 'A#7': 70, 'B-7': 70, 'B7': 71,
        'C8': 72, 'C#8': 73, 'D-8': 73, 'D8': 74, 'D#8': 75, 'E-8': 75, 'E8': 76, 'F8': 77, 'F#8': 78,
        'G-8': 78, 'G8': 79, 'G#8': 80, 'A-8': 80, 'A8': 81, 'A#8': 82, 'B-8': 82, 'B8': 83,
        'B-1': 84, 'A0': 85, 'A#0': 86, 'B0': 87, 'C1': 88, 'C#1': 89, 'D1': 90, 'D#1': 91, 'E1': 92,
        'E-1': 93, 'F1': 94, 'F#1': 95, 'G1': 96, 'G#1': 97,
        'A1': 98, 'B-0': 99, 'B1': 100,
    }

    return noteMapping.get(note, note)
```

Dictionary mapping music notes to integer representations

Splitting Dataset into Train/Seed:

After completing our processing on the raw musical data, we split our corpus into input sequences and target notes as output from our recurrent neural networks. We selected an arbitrary sequence length of 40 notes to capture musical complexity without reducing model training speed. After iterating through all processed songs, we arrived at 519,046 input-output pairs for model training. The number of classes is reflective of the note-to-integer mapping earlier which converts musical notes into an integer representation between 0 and 100 inclusive.

```python
# mapped_songs -- array of songs representing lists of notes mapped to arbitrary integers
all_notes = mapped_songs

# split corpus into labels and targets
seq_length = 40 # arbitrarily choose input sequences of length 40
features = []
targets = []

for song in all_notes:
    for i in range(0, len(song) - length, 1):
        features.append(song[i:i + length])
        targets.append([song[i + length]]*seq_length)
```

```python
# display summary statistics
num_sequences = len(targets)
num_classes = 101 # obtained from mapping

# dimensions of X input tensor
print(f"num sequences: {num_sequences}")
print(f"sequence length: {seq_length}")
print(f"num classes: {num_classes}")
```

```
num sequences: 519046
sequence length: 40
num classes: 101
```

Compute dimensions for the model's input and output tensors

After building our features and target arrays for inputs and outputs respectively, we one-hot encoded our musical notes and reshaped the arrays into 3D numpy column vectors. The purpose of one-hot encoding our note integer representations was to create uniform input magnitudes of 0s and 1s (binary) which does not overweight any one class while training. Furthermore, one-hot encoding removes the possibilities of linear relationships between proximal numbers (ie 44 and 46) which represent completely independent classes of notes. The reshaping of the second dimension y output tensor was performed to mirror the resultant X input tensor while training. Finally, we split the X and y tensors into training and seed datasets based on an 80:20 split for sufficient data when training the model and later predicting new musical melodies.

```python
# reshape and one hot encode both X sequences and Y target output
X = np.reshape(features, (num_sequences, seq_length, 1))
X = tf.keras.utils.to_categorical(X, num_classes=num_classes)

y = np.reshape(targets, (num_sequences, seq_length, 1))
y = tf.keras.utils.to_categorical(y, num_classes=num_classes)
```

```python
# confirm the shape of the input tensors
print(f"X shape: {X.shape}")
print(f"y shape: {y.shape}")
```

```
X shape: (519046, 40, 101)
y shape: (519046, 40, 101)
```

```python
# split labels and targets into training and seed data
X_train, X_seed, y_train, y_seed = train_test_split(X, y, test_size=0.2, random_state=42)
```

Reshape the features and targets arrays before splitting the dataset 80:20

Training the Model:

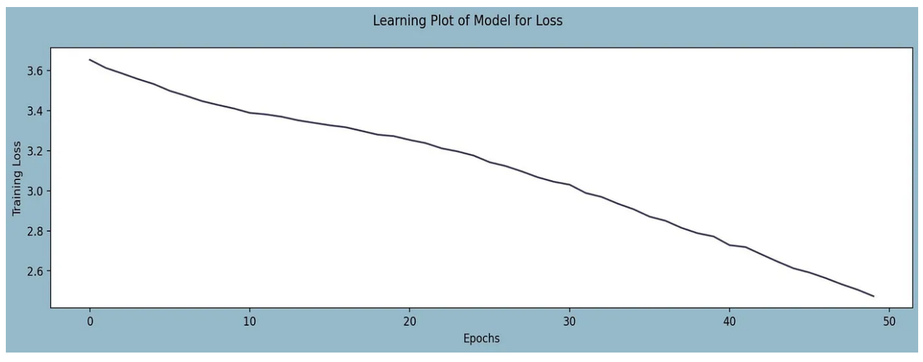
```
# display model summary
```

click to expand output; double click to hide output

```
Model: "sequential_6"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 lstm_10 (LSTM)              (None, 40, 512)           1257472

 dropout_9 (Dropout)         (None, 40, 512)           0

 lstm_11 (LSTM)              (None, 40, 256)           787456

 dropout_10 (Dropout)        (None, 40, 256)           0

 dense_8 (Dense)             (None, 40, 128)           32896

 dense_9 (Dense)             (None, 40, 101)           13029

=================================================================
Total params: 2090853 (7.98 MB)
Trainable params: 2090853 (7.98 MB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

Model summary for LSTM architecture

We initialized an LSTM model consisting of two LSTM layers, dropout layers to prevent overfitting, and finally dense layers with a softmax activation function to generate a probability distribution for the target note following the input sequence of musical notes. We selected the AdamX optimizer for minimizing the categorical cross entropy across independent classes for our loss function. We trained the model for 50 epochs and observed the following learning plot.



Plot of LSTM minimizing training loss across 50 epochs

Finalizing the Output

After finishing our model training, we were able to perform predictions based on seed input sequences to generate new musical melodies. The LSTM never encountered the note sequences within seed dataset while training, making for some uncertainty in the structure and diversity of music generated. We created helper functions to transform the model predictions into integer representations via argmax and musical note strings via reverse mapping. The predicted notes were appended to the end of the seed input sequences to predict the following N subsequent notes. The resultant new sequences we appended to a container notes_generated array from which we could generate musical notes.

```
def reverseMap(note):

    reverseNoteMapping = {
        0: 'C2', 1: 'C#2', 2: 'D2', 3: 'D#2', 4: 'E2', 5: 'F2', 6: 'F#2',
        7: 'G2', 8: 'G#2', 9: 'A2', 10: 'A#2', 11: 'B2', 12: 'C3', 13: 'C#3',
        14: 'D3', 15: 'D#3', 16: 'E3', 17: 'F3', 18: 'F#3', 19: 'G3', 20: 'G#3',
        21: 'A3', 22: 'A#3', 23: 'B3', 24: 'C4', 25: 'C#4', 26: 'D4', 27: 'D#4',
        28: 'E4', 29: 'F4', 30: 'F#4', 31: 'G4', 32: 'G#4', 33: 'A4', 34: 'A#4',
        35: 'B4', 36: 'C5', 37: 'C#5', 38: 'D5', 39: 'D#5', 40: 'E5', 41: 'F5',
        42: 'F#5', 43: 'G5', 44: 'G#5', 45: 'A5', 46: 'A#5', 47: 'B5', 48: 'C6',
        49: 'C#6', 50: 'D6', 51: 'D#6', 52: 'E6', 53: 'F6', 54: 'F#6', 55: 'G6',
        56: 'G#6', 57: 'A6', 58: 'A#6', 59: 'B6', 60: 'C7', 61: 'C#7', 62: 'D7',
        63: 'D#7', 64: 'E7', 65: 'F7', 66: 'F#7', 67: 'G7', 68: 'G#7', 69: 'A7',
        70: 'A#7', 71: 'B7', 72: 'C8', 73: 'C#8', 74: 'D8', 75: 'D#8', 76: 'E8',
        77: 'F8', 78: 'F#8', 79: 'G8', 80: 'G#8', 81: 'A8', 82: 'A#8', 83: 'B8',
        84: 'B-1', 85: 'A0', 86: 'A#0', 87: 'B0', 88: 'C1', 89: 'C#1', 90: 'D1',
        91: 'D#1', 92: 'E1', 93: 'E-1', 94: 'F1', 95: 'F#1', 96: 'G1', 97: 'G#1',
        98: 'A1', 99: 'B-0', 100: 'B1',
    }

    return reverseNoteMapping.get(note, note)
```

Dictionary mapping integer representations back to musical notes

```
# helper function to sample an index from a probability array
# input parameters are preds (1D probability distribution) and temperature (diversity)
# pulled from week6_exercises with RNNs
def sample(preds, temperature=1.0):
    preds = np.asarray(preds).astype('float64')
    preds = np.exp(np.log(preds) / temperature)  # softmax
    preds = preds / np.sum(preds)                #
    probas = np.random.multinomial(1, preds, 1)  # sample index
    return np.argmax(probas)
```

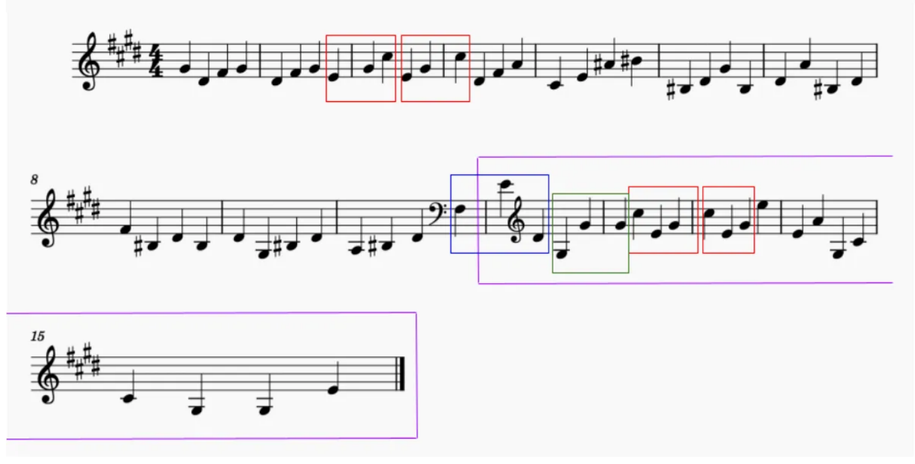
```
# helper function employing model predictions to generate new melodies
# input params are seed of shape (seq_length, num_classes), number of notes, and diversity value
# inspired in part from week
def generate_music(seed, num_notes, diversity):
    notes_generated = []

    # add seed notes at beginning
    for note_enc in seed:
        notes_generated.append(reverseMap(np.argmax(note_enc)))

    for i in range(num_notes):
        # model prediction
        X_pred = seed.reshape(1, seq_length, num_classes)
        y_pred = model.predict(X_pred, verbose=0)

        # helper function to sample the next note_int
        next_note_int = sample(y_pred[0, -1], diversity)

        # reverse map next_note_int and append
        next_note = reverseMap(next_note_int)
        notes_generated.append(next_note)

        # one-hot encode next_note_int
        next_note_encoded = np.zeros(num_classes)
        next_note_encoded[next_note_int] = 1

        # update seed with prediction
        seed = np.append(seed, next_note_encoded.reshape(1, num_classes), axis=0)
        seed = seed[1:,:] # remove first note from sequence

    return notes_generated
```

Script to generate N additional notes to form new musical melody

Once generating a new prediction of 20 notes, we appended it to the previous 40-note sequence. This allowed us to interpret the quality of our model's output in a musical context. Finally, we built MIDI files for each of our generated samples (500 total) and randomly selected a few to analyze.

Results:

In our analysis of the generated scores, we found some interesting results:



Sheet music display of generated music sample (boxed in purple)

Listen to audio file here

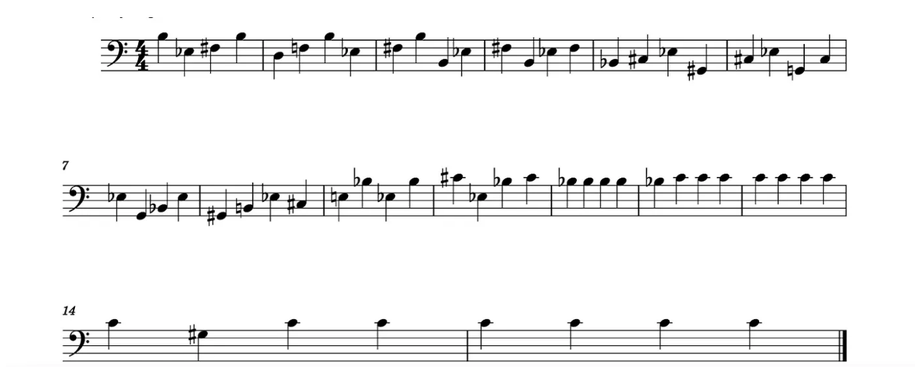
The first 40 notes of this score are part of the melody of Beethoven's famous "Moonlight Sonata," while the last 20 were generated by our model. The song is in the key of C# minor, with a chordal melody outlying specific chords that create tension and release in this key.

Looking at the notes within the red squares, we found that our generated melody (purple) not only learned to play notes in the correct key, it even followed the same melodic structure of the original piece — outlining chords.

The most amazing thing however, is that the melody produced by our model implies an understanding of chord progressions, an advanced subset of music theory. Looking at the first note in the blue square, which is also the last note from the original melody, we can see it is an F#. If you play the audio file, and follow along with the sheet music, you can hear that this note creates tension. This is because it is not a note in our "home" chord of C# minor (C#, E, G#).

Using this F# as a pivotal point in the melody, our model generated notes consistent with a 4–5–1 progression, commonly found in classical music. In the rest of the blue box, the notes outline an F# (the 4 of C# minor) diminished chord. In the green box, the notes are all G#, acting as the 5 chord. Finally, the melody resolves back to C# minor in the red boxes, our home.

This occurrence was, by far, the most impressive feat we saw from our model. Essentially, it recognized a point of tension in the music, and produced a melody to resolve it while adhering to classical music theory principles. Furthermore, due to the numerous conditions required for this result, we find it very unlikely this was generated at random. All of the notes produced make sense in the context of our key, follow the appropriate melody structure that outlines chords in groups of three, and follow a popular chord progression found in classical music.



Sheet music display of generated music sample lacking clear structure

Listen to audio file here

While our model succeeded in generating many pleasing melodies, it's important to acknowledge instances where the output fell short of expectations. There were a few scores (like the one above) that exhibited repetitive note patterns and lacked a clear musical structure. These outcomes highlighted a common challenge we had while exploring basic music generation — consistent creativity and musical coherence.

Overall, our generative model produced promising results with occasional inconsistencies. It demonstrated a solid understanding of musical theory fundamentals, playing in the correct key most of the time. Importantly, it also understood the concept of playing in context, often following the existing melodic structures of the previous sequence.

Conclusion:

We were very happy with the overall results of this project. We achieved our goal of creating meaningful melodies that can be used to assist the creative musical process and even exceeded expectations with some of the more complex scores that were generated.

To improve the project, we could append additional data regarding more complex musical concepts such as chromaticism or modal scales into the X input tensor when training the model. The additional feature data could provide additional insights into capturing greater melody complexity and

improve training accuracy. Nevertheless, we were impressed with the performance of the model to generate musical melodies with chordal structure and a Bb 'home' presence.

All code for our project can be found here: https://github.com/ianpompliano/NeuralNetworksFinalProject/

Sources:

[1] Piano MIDI, MIDI data (2023), http://www.piano-midi.de/

[2] Beautiful Soup Documentation, (2023), https://tedboy.github.io/bs4_doc/

[3] Keras Documentation, (2023), https://keras.io/about/

[4] music21 Documentation, (2023), https://web.mit.edu/music21/

[5] J. Brownlee, A Gentle Introduction to Long Short-Term Memory Networks (2021), https://machinelearningmastery.com/gentle-introduction-long-short-term-memory-networks-experts/

Neural Networks | Lstm | Ai Music Generator | Midi

Written by Siddharth Shah

1 Follower · 3 Following

Data science enthusiast explore the intersections between data and the real world

Follow

No responses yet