AI Music Generation: How to Make Music When Tone Deaf

Chris Howard, JP Reilly, Sam Turchetta

State-Based Approach

Overview: One of our approaches for tackling the melody generation task was modeling the problem as a state-based search problem. We thought this problem naturally lent itself to being modeled as a state-based search problem because intuitively generating a melody consists of sequentially adding notes of different tones and different durations. Here is how we defined the state and action space for this particular model:

- **State**: list of (note, duration) tuples
- **Action**: Append a new (note, duration) tuple to the existing state

Assumptions: By modeling the problem as a state-based search problem, we made assumptions that constrained our output melody in two key ways.

1. By assuming the sequential nature of generating melodies, we limited our melodies to be songs with no overlapping notes.
2. For the sake of simplicity, we limited our output MIDI file to have a single channel, in particular, a channel corresponding to a piano. Thus, we limited our output to play the melody on a single instrument, the piano.

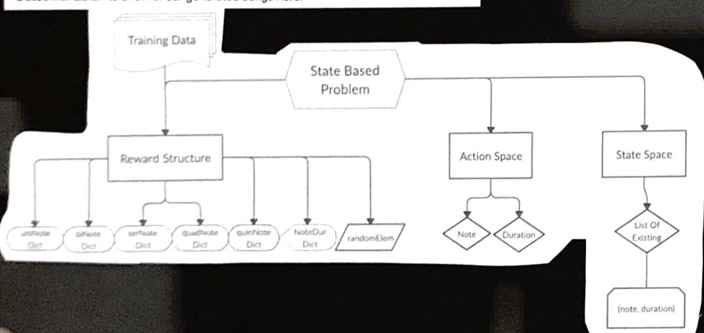
Reward Structure: The crux of the state-based search problem is the reward structure built around different actions. Our model builds a song by maximizing the reward associated with adding 150 notes in sequential order. We employ a weighted reward structure composed of the following:

- **UniNote Dictionary**: a frequency dictionary with a normalized frequency value for each of the notes of a 88 note piano. For any particular (note, duration) tuple considered, one component of the reward associated with this action is the value in this uniNote dictionary associated with the note used as the key.
- **BiNote Dictionary**: a frequency dictionary with a normalized frequency value for a given pair of notes. For any particular (note, duration) action, the value associated with the (last note in current state, note in action) key in the biNote dictionary is a component of the reward.
- **TerNote Dictionary**: Same as the biNote dictionary, but now we consider the frequency associated with the prior *two* notes in the current state and the note in a proposed action.
- **QuadNote Dictionary**: Same as terNote dictionary, but now we consider the frequency associated with the prior *three* notes in the current state and the note in a proposed action.
- **QuinNote Dictionary**: Same as quadNote dictionary, but now we consider the frequency associated with the prior *four* notes in the current state and the note in a proposed action.
- **NoteDur Dictionary**: a dictionary with a key for every single note of an 88 note piano and an associated value that is a dictionary. The value dictionary is a frequency dictionary which contains the frequency associated with 5 bucketed durations of a particular note. For example, the value associated with NoteDur[25][1] would be the normalized frequency that note 25 on a piano appears in the training set for a duration ranging from 0.0 - 0.2 seconds. The 5 duration buckets are (0.0 - 0.2), (0.2 - 0.4), (0.4 - 0.6), (0.6 - 0.8), (0.8, +). We chose these durations because we found it produced the maximum distribution across the different buckets.
- **Randomized Element**: the final component of the reward for a particular action is a small randomized element that promotes non-convergence and pushes the algorithm to explore some notes that maybe do not appear in the training set particularly often.

Difficulties: We've encountered a few difficulties associated with this particular model:

1. There is a tendency for the songs to converge around a single note that appears most often in the training set. The randomized element of the reward structure helps alleviate this issue, but it is a band-aid fix that wouldn't be necessary in other models.
2. To begin our song, we must start with 5 randomized notes so that all the frequency dictionaries work on the first action. This results in a chaotic start to the song that does not sound particularly good and oftentimes derails the entire song since the foundational notes are not melodious.

Outcome: Listen to a few of our generated songs here!

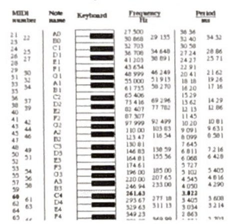


Data Preprocessing

MIDI FILES contain arrays of instruments within each song
Each instrument contains an array of notes and note is comprised as follows:
(Start time (**ms**), end time (**ms**), Pitch (**21 - 108**), Velocity (**1 - 127**))

1. First approach
   1.1. MIDI FILES ⇒ ~150 **classical** piano songs from http://www.piano-midi.de/midi_files.htm
   1.2. Transformed the note array of the three piano instruments into ***Piano Rolls***
   - Array of pitches contained in each interval of **0.2 seconds**
   - **'e'** slotted into the array if there are no notes at a given interval
   - Left and Right piano along with the base

   1.3. Created 5 different dictionaries that are updated after each time interval and written to csv files for the algorithms to build songs from the rewards
   - N-gram from 1 → 5 notes (similar to assignment 3)
   - Dictionary key ⇒ tuples of pitches (1 to 5 pitches in sequential order)
   - Dictionary value ⇒ amount of times the tuple is seen (normalized)



Here is the piano roll taken from pretty_midi
- Pretty_midi allows us to manipulate data from Midi files in more creative ways

   1.4. Problems and Possible feature changes:
   - Due to songs being split up into buckets, the same note will often play in back to back intervals which leads to convergence on one note being over values in the dictionaries with multiple notes ((58, 58) weighted higher than (58, any other note))
   - Add note variation values to ensure there is always different types of notes
   - Differentiate in the piano roll when a note is the same note

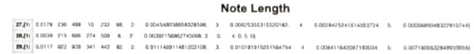
2. Second approach
   2.1. MIDI FILES ⇒ ~20 **popular** piano songs from http://www.MIDI.com with music from Queen, Billy Joel, John Legend, and more
   - Building up this data set trying to mimic identified best songs

   2.2. Sort the notes by their **start times** as this is the value we are most concerned about
   2.3. Using the note array for the piano, we built 9 different values
   - 5 different dictionaries from the first attempt → n-gram dictionaries
   - Note length dictionary Pitch → 5 buckets of how long a note is (**0.2 second** split)
   - Overlapping dictionary where keys are the pitch of a song → values are tuples of possible notes a pitch overlaps with and the frequency that those notes overlap
   - Typical amount of notes within a total song (**average**) - value
   - Typical amount of notes distribution within a slotted amount of time (**1 second**) - value

   2.4. Goal is to build songs up in a similar format as the raw MIDI FILES so we can replicate the rolls as much as possible

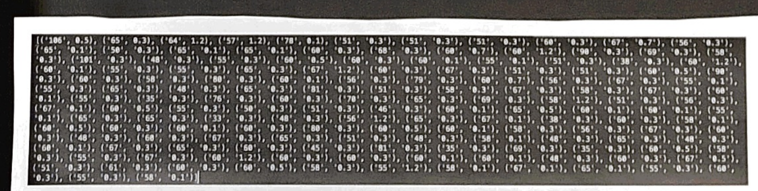
Unigram		Bigram		Ternary	
28	0.0078431372549	**-57** 50)	0.00264200792602	**-75** 67 75)	0.000520833333333
29	0.048724815939	**-39** 70)	0.00227401932916	**-74** 81 81)	0.0058651026393
30	0.0274509803922	**-65** 50)	0.00884233349823	**-59** 71 68)	0.00438614827202

Note Length



   2.5. Problems and possible features available:
   - Difficulty adding the feature of multiple simultaneous notes in our output→ real songs oftentimes have overlapping notes
   - Move away from discretized time intervals towards a more continuous preprocessing model in order to better replicate real songs

Sample State-based Song

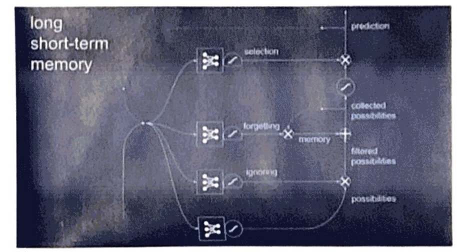


RNN and LTSM

Definitions:
- RNN: Recurrent Neural Network
- LSTM: Long Short-Term Memory

Pros	Cons
• Can process input of any length • Model size doesn't increase with size of input • Computation takes into account historical information • Weights are shared across time	• Computation takes a long time • It can be difficult to access information from a significant time ago • Cannot consider any future input for the current state

https://stanford.edu/~shervine/teaching/cs-230/cheatsheet-recurrent-neural-networks



https://www.youtube.com/watch?v=WCUNPb-5EYI

Overview and Process

Why RNN and LSTM?
- RNN is effective for sequential patterns embedded in time, such as text, speech, and **music**
- LSTM solves the issue that RNN has with vanishing gradients. The derivative of the activation function is always less than or equal to 1, thus the gradient tends to approach 0 resulting in little to no training error occuring. LSTM creates a connection between the forget gate activations and the gradient computations which creates a path for information that the LSTM should not forget

Processing and Generation
- Parsed songs into 2 categories: Notes and Chords. Chords consist of a pair or grouping of notes played at the same time.
- Ran input using LSTM using Keras through 140 epochs, loss started at 5.1349 and ended at 0.4305. Each epoch took ~15 minutes.
- Saved weights file at every checkpoint (after every epoch). Then, we used our final weights to produce a MIDI file.

Training Process

Results
- The songs generated were significantly diverse in note structure, with little repetition. Melody's closely modeled the songs in the data set which we trained upon, but that was expected and welcomed. See an example note structure below for a portion of a generated song.