Project Milestone LaTeX Sheet

Cameron Brenner

5/15/2019

1 Abstract

Machine learning algorithms are used for training data to fit the trained model to reproduce results that fit the intended use. These techniques have been applied to many fields, such as statistics, economics, and even classifying objects in images. The paper's goal is to show the various methods used by machine learning and data science enthusiasts, such as Recurrent Neural Networks, to generate music given training data.

2 Introduction

Music is one of the oldest of the arts known to man, and has most likely existed long before the dawn of man thanks to birds and other singing animals. Humans have been generating music for at least 18,000 years, as some of the earliest known songs come from religious hymns found in the Hindu Vedas. In the late 1940's, AT&T engineers were able to create the transistor, a component that would be integral in almost all modern electronics. Fast forward to today, and humans are able to train algorithms on data and have them reproduce the results in the data. Machine learning techniques can be applied to a wide variety of fields, not just computer and data science. One of these fields, one of humanity's oldest friends, is music. Machine learning techniques can be applied to the theory of music, and given training data, can reproduce, and even generate original music of its own. It's amazing that computing technology has advanced enough where we can program and teach algorithms to generate tones and melodies that we, ourselves, have been creating for thousands of years. If machine learning algorithms can generate music, pictures, art, etc, how long will it be until machine learning techniques will be able to replicate the human thought processes? This project intends to explore the machine learning techniques in conjunction with music theory principles used to generate music.

3 Background and other related work

A popular tool that machine learning specialists have used to generate music is the Music21 Python library. It's used for its ability to render musical notation given a MIDI file. This is useful for processing sound files to be used as training data. Recurrent Neural Networks are unique in the way that hidden neurons will take it's output and feed it back into itself for additional input. This property is useful to create time-invariant music, or, each note is produced in an iterative manner so it follows some time guide. The iterative property of RNNs is also good for producing notes, which in a song are produced to follow patterns. Some machine learning folk that use RNNs are the following,

`https://towardsdatascience.com/how-to-generate-music-using-a-lstm-neural-network-in-keras-68786834d4`

`http://www.hexahedria.com/2015/08/03/composing-music-with-recurrent-neural-networks/`

In another source that I have found,"Making Music: When Simple Probabilities Outperform Deep Learning", Haebichan Jung uses Music21 and the "Markov Process" to calculate the statistical probability in note relationships. He uses this to select a random note, then based on the probabilities of notes, selects the next note and repeats this process. This particular example is interesting, because Jung tries to keep in mind that melody and harmony need to be connected, something that the previous two machine learning scientists disregarded.

`https://towardsdatascience.com/making-music-when-simple-probabilities-outperform-deep-learning-75f4e`

These examples are all very useful, as they give diagrams of the models they use, and describe the theory behind the machine learning techniques such that even a layperson would understand. They also give examples of the outputs of their generated music, and it is very interesting to hear the differences in their outputs. The input data they all utilize is various piano music selections.

4 Methodology

For this project, I want to examine the different methods in which these scientists use to develop their machine learning generated music. They all have the Githubs to their projects, making it available to others.I would like to use input data of my own. Using different genres of music and experimenting with input data will hopefully produce interesting results.

5 Experiments

Here is the code for Haebichan's model

```
# -*- coding: utf-8 -*-
"""haebichanmusic.ipynb

Automatically generated by Colaboratory.

Original file is located at
    https://colab.research.google.com/drive/1cLu3jjFm5SGCyuAprzYMHAxlZ_lcyCRe

# Haebichan Jung's algorithm
"""

import pandas as pd
import numpy as np
from music21 import converter, chord, note, instrument, stream
from collections import Counter
import random

pd.set_option('display.max_columns', 999)


def main(input):

    midi_list = input

    def get_midi(midi_list):
        all_midis = []
        all_parts = []

        for song in midi_list:
            midi = converter.parse('chpn_op66.mid' + song)
            for i in midi.parts:
                i.insert(0, instrument.Piano())
            parts = instrument.partitionByInstrument(midi)

            all_midis.append(midi)
            all_parts.append(parts)
```

```
    return all_midis, all_parts


all_midis, all_parts = get_midi(midi_list)

def get_notes_offset_durations(all_parts):
    notes = []
    notes_offset = []
    durations = []

    for parts in all_parts:
        for i in parts.recurse():
            if isinstance(i, note.Note):
                notes.append(str(i.pitch))
                notes_offset.append(float(i.offset))
                durations.append(float(i.duration.quarterLength))

            elif isinstance(i, chord.Chord):
                notes_offset.append(float(i.offset))
                durations.append(float(i.duration.quarterLength))

                i = str(i).replace('>', '')
                chords = '|'.join(i.split()[1:])
                notes.append(chords)


    return notes, notes_offset, durations

notes, notes_offset, durations = get_notes_offset_durations(all_parts)


def get_harmony(midi_list):
    harmony = []
    harmony_duration = []

    for song in midi_list:
        midi = converter.parse('/Users/Haebichan/Desktop/Final_Project_Galvanize/C_Maj

        for i in midi[1].recurse():
            if isinstance(i, note.Note):
                harmony.append(str(i.pitch))
                harmony_duration.append(i.duration.quarterLength)
            elif isinstance(i, chord.Chord):
                harmony_duration.append(i.duration.quarterLength)

                i = str(i).replace('>', '')
                chords = '|'.join(i.split()[1:])
                harmony.append(chords)


    return harmony, harmony_duration

harmony, harmony_duration = get_harmony(midi_list)
allnotes = set(note for note in notes)
```

```
keys = list(set(key for key in harmony))
harmony_duration = [4.0 for i in harmony_duration]


def create_vertical_dependency_dictionary(keys, notes, notes_offset):
    dic = {}

    for key in keys:
        dic[key] = []

    for i in range(len(notes)):
        if notes[i] not in keys:
            continue

        j = i + 1
        while j < len(notes) and np.abs(notes_offset[i]-notes_offset[j]) <= 4.0:
            dic[notes[i]].append(notes[j])
            j+=1

    return dic


vertical_dependency_dictionary = create_vertical_dependency_dictionary(keys, notes, no

def create_dependency_map(dic, notes):
    keys_notes_map = {}

    for key in keys:
        keyMap = {}
        for note in set(notes):
            keyMap[note] = 0.0
        keys_notes_map[key] = keyMap

    for key in keys:
        my_notes = dic[key]
        note_count = Counter(my_notes)
        for my_note_count in note_count.keys():
            keys_notes_map[key][my_note_count] = note_count[my_note_count]/len(my_notes

    return keys_notes_map


vert_keys_notes_map = create_dependency_map(vertical_dependency_dictionary, notes)

vert_dep_matrix = pd.DataFrame(vert_keys_notes_map).T


def get_melody_offset_durations(instrument_parts):
    melody = []
    melody_offset = []
    melody_durations = []

    for parts in instrument_parts:
        for i in parts.recurse():
```

```
            if isinstance(i, note.Note):
                melody.append(str(i.pitch))
                melody_offset.append(float(i.offset))
                melody_durations.append(float(i.duration.quarterLength))

            elif isinstance(i, chord.Chord):
                melody_offset.append(float(i.offset))
                melody_durations.append(float(i.duration.quarterLength))

                i = str(i).replace('>', '')
                chords = '|'.join(i.split()[1:])
                melody.append(chords)

    return melody, melody, melody_durations


melody, melody_offset, melody_durations = get_melody_offset_durations(all_parts)

all_melody_notes = set(note for note in melody)



def create_horizontal_dependency_dictionary(all_melody_notes, melody, melody_durations
    hori_dic = {}
    count = 0

    for melody_note in all_melody_notes:
        hori_dic[melody_note] = []

    for note, next_note, duration in zip(list(melody), list(melody)[1:], melody_duratio
        if count <= 4:
            hori_dic[note].append(next_note)
            count += duration
        count = 0

    return hori_dic


hori_dic = create_horizontal_dependency_dictionary(all_melody_notes, melody, melody_du


def create_horizontal_dependency_map(hori_dic):
    horizontal_dependency_map = {}

    for y_axis_key in hori_dic.keys():
        hori_map = {}
        for x_axis_key in hori_dic.keys():
            hori_map[x_axis_key] = 0.0
        horizontal_dependency_map[y_axis_key] = hori_map

    for y_axis_key in hori_dic.keys():
        new_notes = hori_dic[y_axis_key]
        new_note_count = Counter(new_notes)
        for my_note_count in new_note_count.keys():
```

```
            horizontal_dependency_map[y_axis_key][my_note_count] = new_note_count[my_n

    return horizontal_dependency_map


horizontal_dependency_map = create_horizontal_dependency_map(hori_dic)
hori_dep_matrix = pd.DataFrame(horizontal_dependency_map).T

hori_dep_matrix = hori_dep_matrix.drop(hori_dep_matrix[hori_dep_matrix.sum(1).values =



all_harmony_notes = set(note for note in harmony)
harmony_hori_dic = create_horizontal_dependency_dictionary(all_harmony_notes, harmony,

harmony_horizontal_dependency_map = create_horizontal_dependency_map(harmony_hori_dic)

harmony_hori_dep_matrix = pd.DataFrame(harmony_horizontal_dependency_map).T

harmony_hori_dep_matrix = harmony_hori_dep_matrix.drop(
    harmony_hori_dep_matrix[harmony_hori_dep_matrix.sum(1).values == 0.0].index)



def create_vert_duration_dictionary(matrix, notes):
    vert_duration_dictionary = {}

    for single_note in list(matrix.columns):
        vert_duration_dictionary[single_note] = []

    for note, duration in zip(notes, durations):
        vert_duration_dictionary[note].append(duration)

    return vert_duration_dictionary

def create_hori_duration_dictionary(matrix, melody, melody_durations):
    hori_duration_dictionary = {}

    for single_note in list(matrix.columns):
        hori_duration_dictionary[single_note] = []

    for note, duration in zip(melody, melody_durations):
        hori_duration_dictionary[note].append(duration)

    return hori_duration_dictionary

vert_duration_dictionary = create_vert_duration_dictionary(vert_dep_matrix, notes)


hori_duration_dictionary = create_hori_duration_dictionary(hori_dep_matrix, melody, me


harmony_hori_duration_dictionary = create_hori_duration_dictionary(harmony_hori_dep_ma
```

```
def create_vert_duration_map(allnotes, durations, duration_dictionary):
    duration_notes_map = {}

    for single_note in allnotes:
        durationMap = {}
        for each_duration in durations:
            durationMap[each_duration] = 0.0
        duration_notes_map[single_note] = durationMap

    for single_note in allnotes:
        note_duration = duration_dictionary[single_note]
        note_duration_count = Counter(note_duration)
        for single_note_duration in note_duration_count.keys():
            duration_notes_map[single_note][single_note_duration] = note_duration_count

    return duration_notes_map

vert_duration_notes_map = create_vert_duration_map(allnotes, durations, vert_duration_d


vert_duration_matrix = pd.DataFrame(vert_duration_notes_map).T

def create_hori_duration_map(all_melody_notes, melody_durations, hori_duration_dictiona
    duration_notes_map = {}

    for single_note in all_melody_notes:
        durationMap = {}
        for each_duration in melody_durations:
            durationMap[each_duration] = 0.0
        duration_notes_map[single_note] = durationMap


    for single_note in all_melody_notes:
        note_duration = hori_duration_dictionary[single_note]
        note_duration_count = Counter(note_duration)
        for single_note_duration in note_duration_count.keys():
            duration_notes_map[single_note][single_note_duration] = note_duration_count

    return duration_notes_map


hori_duration_notes_map = create_hori_duration_map(all_melody_notes, melody_durations,

hori_duration_matrix = pd.DataFrame(hori_duration_notes_map).T

harmony_hori_duration_map = create_hori_duration_map(all_harmony_notes, harmony_duratio

harmony_hori_duration_matrix = pd.DataFrame(harmony_hori_duration_map).T




harmony_hori_duration_matrix = harmony_hori_duration_matrix.rename(columns={'1/3': '0.3
```

```
every_note = ['C1', 'D-1', 'D1', 'E-1', 'E1', 'F-1', 'F1', 'G1', 'A-1', 'A1', 'B-1', 'I
              'E-2', 'E2', 'F-2', 'F2', 'G-2', 'G2', 'A-2', 'A2', 'B-2', 'B2', 'C-3',
              'E3', 'F-3', 'F3', 'G-3', 'G3', 'A-3', 'A3', 'B-3', 'B3', 'C-4', 'C4', 'I
              'F-4', 'F4', 'G-5', 'G4', 'A-4', 'A4', 'B-4', 'B4', 'C-5', 'C5', 'D-5',
              'F5', 'G-5', 'G5', 'A-5', 'A5', 'B-5', 'B5', 'C-6', 'C6', 'D-6', 'D6', 'I
              'G-6', 'G6', 'A-6', 'A6', 'B-6', 'B6', 'C-7', 'C7', 'D-7', 'D7', 'E-7',
              'G7']
every_note_number = [i for i in range(len(every_note))]

every_note_dic = {}

for i, j in zip(every_note_number, every_note):
    every_note_dic[j] = i

for i in hori_dep_matrix.index:
    if len(i) > 3:
        split_note = i.split('|')
        every_note_dic[i] = every_note_dic[split_note[0]]

def get_harmony(midi_list):
    harmony = []
    harmony_duration = []

    for song in midi_list:
        midi = converter.parse('/Users/Haebichan/Desktop/Final_Project_Galvanize/C_Maj

        for i in midi[1].recurse():
            if isinstance(i, note.Note):
                harmony.append(str(i.pitch))
                harmony_duration.append(i.duration)
            elif isinstance(i, chord.Chord):
                harmony.append('|'.join(i.pitchNames))
                harmony_duration.append(i.duration)

    return harmony, harmony_duration

harmony, harmony_duration = get_harmony(midi_list)

harmony_duration = [4.0 for i in harmony_duration]

def create_harmony_list(harmony_hori_dep_matrix, harmony_duration, harmony_offset_count
    harmony_list = []
    harmony_duration_list = []

    harmony_offset_count = 0

    harmony_note = random.choice(list(harmony_hori_dep_matrix.index))
    harmony_duration = float(np.random.choice(harmony_hori_duration_matrix.loc[harmony
                                              p=harmony_hori_duration_matrix.loc[harmo

    # If harmony duration is 2, then 64 means that the harmony will repeat after 8 harm
    while harmony_offset_count <= harmony_offset_count_number and (
            harmony_duration + harmony_offset_count) < harmony_offset_count_number:
```

```
        harmony_list.append(harmony_note)
        harmony_duration_list.append(harmony_duration)

        harmony_offset_count += harmony_duration

        harmony_note = np.random.choice(harmony_hori_dep_matrix.loc[harmony_note].index
                                        p=harmony_hori_dep_matrix.loc[harmony_note].va
        harmony_duration = float(np.random.choice(harmony_hori_duration_matrix.loc[harm
                                                  p=harmony_hori_duration_matrix.loc[ha

    return harmony_list, harmony_duration_list

harmony_list, harmony_duration_list = create_harmony_list(harmony_hori_dep_matrix, harm

harmony_offset_count_number = 64

harmony_duration_list[-1] = (harmony_offset_count_number - sum(harmony_duration_list[:-

harmony_duration_list[-1] = harmony_duration[0]

highest_harmony_number = []
for harmony_note in harmony_list:

    if len(harmony_note) > 2:

        split_harmony = harmony_note.split('|')
        highest_harmony_number.append(split_harmony[1])


    else:
        highest_harmony_number.append(harmony_note)

highest_harmony_number = [every_note_dic[i] for i in highest_harmony_number]

highest_harmony_number = max(highest_harmony_number)


def create_song_cluster_1(vert_dep_matrix, vert_duration_matrix, hori_dep_matrix, hori_
    song_generation = []
    song_generation_duration = []


    segment_count = 2

    repetition_count = 0

    while repetition_count <= segment_count:

        for harmony_note, harmony_duration in zip(harmony_list[::2], harmony_duration_
            song_generation.append(harmony_note)
            song_generation_duration.append(harmony_duration)

            melody_note = np.random.choice(vert_dep_matrix.loc[harmony_note].index, p =
            melody_duration = float(np.random.choice(vert_duration_matrix.loc[harmony_
```

```
            temporary_gap_width = gap_width
            while every_note_dic[melody_note] < (highest_harmony_number + gap_width):
                melody_note = np.random.choice(vert_dep_matrix.loc[harmony_note].index
                melody_duration = float(np.random.choice(vert_duration_matrix.loc[harm
                if temporary_gap_width != 0:
                    temporary_gap_width -= 1


            melody_dic = {}
            count = 0

            while count <= 8.0 and (melody_duration + count < 8.0):

                melody_dic[melody_note] = melody_duration

                count+= melody_duration


                melody_note = np.random.choice(hori_dep_matrix.loc[melody_note].index,
                melody_duration = float(np.random.choice(hori_duration_matrix.loc[melo

                temporary_gap_width = gap_width
                while every_note_dic[melody_note] < (highest_harmony_number + gap_widt
                    melody_note = np.random.choice(hori_dep_matrix.loc[melody_note].inc
                    melody_duration = float(np.random.choice(hori_duration_matrix.loc[
                    if temporary_gap_width != 0:
                        temporary_gap_width -= 1

                melody_dic[melody_note] = melody_duration
                count+= melody_duration



            song_generation.append(melody_dic)

        repetition_count += 1
    segment_count +=1


    return song_generation, song_generation_duration


def create_song_cluster_2(vert_dep_matrix, vert_duration_matrix, hori_dep_matrix, hori_

    song_generation2 = []
    song_generation_duration2 = []


    segment_count = 2

    repetition_count = 0

    while repetition_count <= segment_count:
```

```
        for harmony_note, harmony_duration in zip(harmony_list[::2], harmony_duration_
            song_generation2.append(harmony_note)
            song_generation_duration2.append(harmony_duration)

            melody_note = np.random.choice(vert_dep_matrix.loc[harmony_note].index, p =
            melody_duration = float(np.random.choice(vert_duration_matrix.loc[harmony_

            temporary_gap_width = gap_width2
            while every_note_dic[melody_note] < (highest_harmony_number + gap_width2):
                melody_note = np.random.choice(vert_dep_matrix.loc[harmony_note].index
                melody_duration = float(np.random.choice(vert_duration_matrix.loc[harm
                if temporary_gap_width != 0:
                    temporary_gap_width -= 1


            melody_dic = {}
            count = 0


            while count <= 8.0 and (melody_duration + count < 8.0):

                melody_dic[melody_note] = melody_duration

                count+= melody_duration



                melody_note = np.random.choice(hori_dep_matrix.loc[melody_note].index,
                melody_duration = float(np.random.choice(hori_duration_matrix.loc[melo

                temporary_gap_width = gap_width2
                while every_note_dic[melody_note] < (highest_harmony_number + gap_widt
                    melody_note = np.random.choice(hori_dep_matrix.loc[melody_note].inc
                    melody_duration = float(np.random.choice(hori_duration_matrix.loc[
                    if temporary_gap_width != 0:
                        temporary_gap_width -= 1

                melody_dic[melody_note] = melody_duration
                count+= melody_duration



            song_generation2.append(melody_dic)

        repetition_count += 1
    segment_count +=1

    return song_generation2, song_generation_duration2


def create_song_cluster_3(vert_dep_matrix, vert_duration_matrix, hori_dep_matrix, hori_

    song_generation3 = []
```

```
    song_generation_duration3 = []


    segment_count = 3

    repetition_count = 0

    while repetition_count <= segment_count:

        for harmony_note, harmony_duration in zip(harmony_list[::2], harmony_duration_
            song_generation3.append(harmony_note)
            song_generation_duration3.append(harmony_duration)

            melody_note = np.random.choice(vert_dep_matrix.loc[harmony_note].index, p =
            melody_duration = float(np.random.choice(vert_duration_matrix.loc[harmony_

            temporary_gap_width = gap_width3
            while every_note_dic[melody_note] < (highest_harmony_number + gap_width3):
                melody_note = np.random.choice(vert_dep_matrix.loc[harmony_note].index
                melody_duration = float(np.random.choice(vert_duration_matrix.loc[harm
                if temporary_gap_width != 0:
                    temporary_gap_width -= 1


            melody_dic = {}
            count = 0


            while count <= 8.0 and (melody_duration + count < 8.0):

                melody_dic[melody_note] = melody_duration

                count+= melody_duration



                melody_note = np.random.choice(hori_dep_matrix.loc[melody_note].index,
                melody_duration = float(np.random.choice(hori_duration_matrix.loc[melo

                temporary_gap_width = gap_width3
                while every_note_dic[melody_note] < (highest_harmony_number + gap_widt
                    melody_note = np.random.choice(hori_dep_matrix.loc[melody_note].inc
                    melody_duration = float(np.random.choice(hori_duration_matrix.loc[
                    if temporary_gap_width != 0:
                        temporary_gap_width -= 1

                melody_dic[melody_note] = melody_duration
                count+= melody_duration



            song_generation3.append(melody_dic)

        repetition_count += 1
```

```
    segment_count +=1

    return song_generation3, song_generation_duration3



song, song_duration = create_song_cluster_1(vert_dep_matrix, vert_duration_matrix, hor
song2, song_duration2 = create_song_cluster_2(vert_dep_matrix, vert_duration_matrix, h
song3, song_duration3 = create_song_cluster_3(vert_dep_matrix, vert_duration_matrix, h

song = song + song2 + song3 + song2 + song3
song_duration = song_duration + song_duration2 + song_duration3 + song_duration2 + son


score = stream.Score()
p1 = stream.Part()
p1.id = 'harmony'

p2 = stream.Part()
p2.id = 'melody'

for harmony_note, h_duration, melody_note in zip(song[::2], song_duration, song[1::2])

    if len(harmony_note) > 2:
        harmony_chord = chord.Chord(harmony_note.split('|'))
        harmony_chord.quarterLength = h_duration
        p1.append(harmony_chord)
    else:
        n = note.Note(harmony_note)
        n.quarterLength = h_duration
        p1.append(n)

    if bool(melody_note) == False:
        rest = note.Rest()
        rest.quarterLength = h_duration
        p2.append(rest)

    else:
        for individual_note, individual_note_duration in melody_note.items():

            if "|" not in individual_note:
                ind_note = note.Note(individual_note)
                ind_note.quarterLength = individual_note_duration
                p2.append(ind_note)
            else:
                split_notes = individual_note.split("|")
                chords = chord.Chord(split_notes)
                chords.quarterLength = individual_note_duration

                p2.append(chords)

score.insert(0, p2)
score.insert(0, p1)
```

```
    for i in score.parts:
        i.insert(0, instrument.Piano())

    score.write('midi', fp='your_song.midi')
```

Here is the code for Skuli's model

```
# -*- coding: utf-8 -*-
"""skuligenerate.ipynb

Automatically generated by Colaboratory.

Original file is located at
    https://colab.research.google.com/drive/1KQ3_cffBXEhPric517I5Vdsu7NZG4RIV

#Skuli Code

Run this to train the algorithm
"""

""" This module prepares midi file data and feeds it to the neural
    network for training """
import glob
import pickle
import numpy
from music21 import converter, instrument, note, chord
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import Dropout
from keras.layers import LSTM
from keras.layers import Activation
from keras.utils import np_utils
from keras.callbacks import ModelCheckpoint

def train_network():
    """ Train a Neural Network to generate music """
    notes = get_notes()

    # get amount of pitch names
    n_vocab = len(set(notes))

    network_input, network_output = prepare_sequences(notes, n_vocab)

    model = create_network(network_input, n_vocab)

    train(model, network_input, network_output)

def get_notes():
    """ Get all the notes and chords from the midi files in the ./midi_songs directory """
    notes = []

    for file in glob.glob("midi_songs/*.mid"):
        midi = converter.parse(file)

        print("Parsing %s" % file)
```

```python
        notes_to_parse = None

        try: # file has instrument parts
            s2 = instrument.partitionByInstrument(midi)
            notes_to_parse = s2.parts[0].recurse()
        except: # file has notes in a flat structure
            notes_to_parse = midi.flat.notes

        for element in notes_to_parse:
            if isinstance(element, note.Note):
                notes.append(str(element.pitch))
            elif isinstance(element, chord.Chord):
                notes.append('.'.join(str(n) for n in element.normalOrder))

    with open('data/notes', 'wb') as filepath:
        pickle.dump(notes, filepath)

    return notes

def prepare_sequences(notes, n_vocab):
    """ Prepare the sequences used by the Neural Network """
    sequence_length = 100

    # get all pitch names
    pitchnames = sorted(set(item for item in notes))

     # create a dictionary to map pitches to integers
    note_to_int = dict((note, number) for number, note in enumerate(pitchnames))

    network_input = []
    network_output = []

    # create input sequences and the corresponding outputs
    for i in range(0, len(notes) - sequence_length, 1):
        sequence_in = notes[i:i + sequence_length]
        sequence_out = notes[i + sequence_length]
        network_input.append([note_to_int[char] for char in sequence_in])
        network_output.append(note_to_int[sequence_out])

    n_patterns = len(network_input)

    # reshape the input into a format compatible with LSTM layers
    network_input = numpy.reshape(network_input, (n_patterns, sequence_length, 1))
    # normalize input
    network_input = network_input / float(n_vocab)

    network_output = np_utils.to_categorical(network_output)

    return (network_input, network_output)

def create_network(network_input, n_vocab):
    """ create the structure of the neural network """
    model = Sequential()
```

```
    model.add(LSTM(
        512,
        input_shape=(network_input.shape[1], network_input.shape[2]),
        return_sequences=True
    ))
    model.add(Dropout(0.3))
    model.add(LSTM(512, return_sequences=True))
    model.add(Dropout(0.3))
    model.add(LSTM(512))
    model.add(Dense(256))
    model.add(Dropout(0.3))
    model.add(Dense(n_vocab))
    model.add(Activation('softmax'))
    model.compile(loss='categorical_crossentropy', optimizer='rmsprop')

    return model

def train(model, network_input, network_output):
    """ train the neural network """
    filepath = "weights-improvement-{epoch:02d}-{loss:.4f}-bigger.hdf5"
    checkpoint = ModelCheckpoint(
        filepath,
        monitor='loss',
        verbose=0,
        save_best_only=True,
        mode='min'
    )
    callbacks_list = [checkpoint]

    model.fit(network_input, network_output, epochs=200, batch_size=64, callbacks=callback

if __name__ == '__main__':
    train_network()

"""Then run this to generate the music"""

""" This module generates notes for a midi file using the
    trained neural network """
import pickle
import numpy
from music21 import instrument, note, stream, chord
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import Dropout
from keras.layers import LSTM
from keras.layers import Activation

def generate():
    """ Generate a piano midi file """
    #load the notes used to train the model
    with open('data/notes', 'rb') as filepath:
        notes = pickle.load(filepath)

    # Get all pitch names
```

```
    pitchnames = sorted(set(item for item in notes))
    # Get all pitch names
    n_vocab = len(set(notes))

    network_input, normalized_input = prepare_sequences(notes, pitchnames, n_vocab)
    model = create_network(normalized_input, n_vocab)
    prediction_output = generate_notes(model, network_input, pitchnames, n_vocab)
    create_midi(prediction_output)

def prepare_sequences(notes, pitchnames, n_vocab):
    """ Prepare the sequences used by the Neural Network """
    # map between notes and integers and back
    note_to_int = dict((note, number) for number, note in enumerate(pitchnames))

    sequence_length = 100
    network_input = []
    output = []
    for i in range(0, len(notes) - sequence_length, 1):
        sequence_in = notes[i:i + sequence_length]
        sequence_out = notes[i + sequence_length]
        network_input.append([note_to_int[char] for char in sequence_in])
        output.append(note_to_int[sequence_out])

    n_patterns = len(network_input)

    # reshape the input into a format compatible with LSTM layers
    normalized_input = numpy.reshape(network_input, (n_patterns, sequence_length, 1))
    # normalize input
    normalized_input = normalized_input / float(n_vocab)

    return (network_input, normalized_input)

def create_network(network_input, n_vocab):
    """ create the structure of the neural network """
    model = Sequential()
    model.add(LSTM(
        512,
        input_shape=(network_input.shape[1], network_input.shape[2]),
        return_sequences=True
    ))
    model.add(Dropout(0.3))
    model.add(LSTM(512, return_sequences=True))
    model.add(Dropout(0.3))
    model.add(LSTM(512))
    model.add(Dense(256))
    model.add(Dropout(0.3))
    model.add(Dense(n_vocab))
    model.add(Activation('softmax'))
    model.compile(loss='categorical_crossentropy', optimizer='rmsprop')

    # Load the weights to each node
    model.load_weights('weights.hdf5')

    return model
```

```
def generate_notes(model, network_input, pitchnames, n_vocab):
    """ Generate notes from the neural network based on a sequence of notes """
    # pick a random sequence from the input as a starting point for the prediction
    start = numpy.random.randint(0, len(network_input)-1)

    int_to_note = dict((number, note) for number, note in enumerate(pitchnames))

    pattern = network_input[start]
    prediction_output = []

    # generate 500 notes
    for note_index in range(500):
        prediction_input = numpy.reshape(pattern, (1, len(pattern), 1))
        prediction_input = prediction_input / float(n_vocab)

        prediction = model.predict(prediction_input, verbose=0)

        index = numpy.argmax(prediction)
        result = int_to_note[index]
        prediction_output.append(result)

        pattern.append(index)
        pattern = pattern[1:len(pattern)]

    return prediction_output

def create_midi(prediction_output):
    """ convert the output from the prediction to notes and create a midi file
        from the notes """
    offset = 0
    output_notes = []

    # create note and chord objects based on the values generated by the model
    for pattern in prediction_output:
        # pattern is a chord
        if ('.' in pattern) or pattern.isdigit():
            notes_in_chord = pattern.split('.')
            notes = []
            for current_note in notes_in_chord:
                new_note = note.Note(int(current_note))
                new_note.storedInstrument = instrument.Piano()
                notes.append(new_note)
            new_chord = chord.Chord(notes)
            new_chord.offset = offset
            output_notes.append(new_chord)
        # pattern is a note
        else:
            new_note = note.Note(pattern)
            new_note.offset = offset
            new_note.storedInstrument = instrument.Piano()
            output_notes.append(new_note)

        # increase offset each iteration so that notes do not stack
```

```
        offset += 0.5

    midi_stream = stream.Stream(output_notes)

    midi_stream.write('midi', fp='test_output.mid')

if __name__ == '__main__':
    generate()
```

Here is the code for Daniel Johnson's model

```
# -*- coding: utf-8 -*-
"""DanielJohnsonGenerate.ipynb

Automatically generated by Colaboratory.

Original file is located at
    https://colab.research.google.com/drive/11v-10ezYjTMS5EyHqIsyNdGvacQY7w4p

#Daniel Johnson's model

training
"""

import os, random
from midi_to_statematrix import *
from data import *
import cPickle as pickle

import signal

batch_width = 10 # number of sequences in a batch
batch_len = 16*8 # length of each sequence
division_len = 16 # interval between possible start locations

def loadPieces(dirpath):

    pieces = {}

    for fname in os.listdir(dirpath):
        if fname[-4:] not in ('.mid','.MID'):
            continue

        name = fname[:-4]

        outMatrix = midiToNoteStateMatrix(os.path.join(dirpath, fname))
        if len(outMatrix) < batch_len:
            continue

        pieces[name] = outMatrix
        print "Loaded {}".format(name)

    return pieces

def getPieceSegment(pieces):
```

```
    piece_output = random.choice(pieces.values())
    start = random.randrange(0,len(piece_output)-batch_len,division_len)
    # print "Range is {} {} {} -> {}".format(0,len(piece_output)-batch_len,division_len, s

    seg_out = piece_output[start:start+batch_len]
    seg_in = noteStateMatrixToInputForm(seg_out)

    return seg_in, seg_out

def getPieceBatch(pieces):
    i,o = zip(*[getPieceSegment(pieces) for _ in range(batch_width)])
    return numpy.array(i), numpy.array(o)

def trainPiece(model,pieces,epochs,start=0):
    stopflag = [False]
    def signal_handler(signame, sf):
        stopflag[0] = True
    old_handler = signal.signal(signal.SIGINT, signal_handler)
    for i in range(start,start+epochs):
        if stopflag[0]:
            break
        error = model.update_fun(*getPieceBatch(pieces))
        if i % 100 == 0:
            print "epoch {}, error={}".format(i,error)
        if i % 500 == 0 or (i % 100 == 0 and i < 1000):
            xIpt, xOpt = map(numpy.array, getPieceSegment(pieces))
            noteStateMatrixToMidi(numpy.concatenate((numpy.expand_dims(xOpt[0], 0), model.
            pickle.dump(model.learned_config,open('output/params{}.p'.format(i), 'wb'))
    signal.signal(signal.SIGINT, old_handler)

"""model"""

import theano, theano.tensor as T
import numpy as np
import theano_lstm

from out_to_in_op import OutputFormToInputFormOp

from theano_lstm import Embedding, LSTM, RNN, StackedCells, Layer, create_optimization_upda

def has_hidden(layer):
    """
    Whether a layer has a trainable
    initial hidden state.
    """
    return hasattr(layer, 'initial_hidden_state')

def matrixify(vector, n):
    # Cast n to int32 if necessary to prevent error on 32 bit systems
    return T.repeat(T.shape_padleft(vector),
                    n if (theano.configdefaults.local_bitwidth() == 64) else T.cast(n,'int3
                    axis=0)

def initial_state(layer, dimensions = None):
```

```
    """
    Initalizes the recurrence relation with an initial hidden state
    if needed, else replaces with a "None" to tell Theano that
    the network **will** return something, but it does not need
    to send it to the next step of the recurrence
    """
    if dimensions is None:
        return layer.initial_hidden_state if has_hidden(layer) else None
    else:
        return matrixify(layer.initial_hidden_state, dimensions) if has_hidden(layer) else

def initial_state_with_taps(layer, dimensions = None):
    """Optionally wrap tensor variable into a dict with taps=[-1]"""
    state = initial_state(layer, dimensions)
    if state is not None:
        return dict(initial=state, taps=[-1])
    else:
        return None

class PassthroughLayer(Layer):
    """
    Empty "layer" used to get the final output of the LSTM
    """

    def __init__(self):
        self.is_recursive = False

    def create_variables(self):
        pass

    def activate(self, x):
        return x

    @property
    def params(self):
        return []

    @params.setter
    def params(self, param_list):
        pass


def get_last_layer(result):
    if isinstance(result, list):
        return result[-1]
    else:
        return result

def ensure_list(result):
    if isinstance(result, list):
        return result
    else:
        return [result]
```

```
class Model(object):

    def __init__(self, t_layer_sizes, p_layer_sizes, dropout=0):

        self.t_layer_sizes = t_layer_sizes
        self.p_layer_sizes = p_layer_sizes

        # From our architecture definition, size of the notewise input
        self.t_input_size = 80

        # time network maps from notewise input size to various hidden sizes
        self.time_model = StackedCells( self.t_input_size, celltype=LSTM, layers = t_layer
        self.time_model.layers.append(PassthroughLayer())

        # pitch network takes last layer of time model and state of last note, moving upwa
        # and eventually ends with a two-element sigmoid layer
        p_input_size = t_layer_sizes[-1] + 2
        self.pitch_model = StackedCells( p_input_size, celltype=LSTM, layers = p_layer_size
        self.pitch_model.layers.append(Layer(p_layer_sizes[-1], 2, activation = T.nnet.sigm

        self.dropout = dropout

        self.conservativity = T.fscalar()
        self.srng = T.shared_randomstreams.RandomStreams(np.random.randint(0, 1024))

        self.setup_train()
        self.setup_predict()
        self.setup_slow_walk()

    @property
    def params(self):
        return self.time_model.params + self.pitch_model.params

    @params.setter
    def params(self, param_list):
        ntimeparams = len(self.time_model.params)
        self.time_model.params = param_list[:ntimeparams]
        self.pitch_model.params = param_list[ntimeparams:]

    @property
    def learned_config(self):
        return [self.time_model.params, self.pitch_model.params, [l.initial_hidden_state fo

    @learned_config.setter
    def learned_config(self, learned_list):
        self.time_model.params = learned_list[0]
        self.pitch_model.params = learned_list[1]
        for l, val in zip((l for mod in (self.time_model, self.pitch_model) for l in mod.la
            l.initial_hidden_state.set_value(val.get_value())

    def setup_train(self):

        # dimensions: (batch, time, notes, input_data) with input_data as in architecture
```

```
self.input_mat = T.btensor4()
# dimensions: (batch, time, notes, onOrArtic) with 0:on, 1:artic
self.output_mat = T.btensor4()

self.epsilon = np.spacing(np.float32(1.0))

def step_time(in_data, *other):
    other = list(other)
    split = -len(self.t_layer_sizes) if self.dropout else len(other)
    hiddens = other[:split]
    masks = [None] + other[split:] if self.dropout else []
    new_states = self.time_model.forward(in_data, prev_hiddens=hiddens, dropout=ma
    return new_states

def step_note(in_data, *other):
    other = list(other)
    split = -len(self.p_layer_sizes) if self.dropout else len(other)
    hiddens = other[:split]
    masks = [None] + other[split:] if self.dropout else []
    new_states = self.pitch_model.forward(in_data, prev_hiddens=hiddens, dropout=m
    return new_states

# We generate an output for each input, so it doesn't make sense to use the last o
# Note that we assume the sentinel start value is already present
# TEMP CHANGE: NO SENTINEL
input_slice = self.input_mat[:,0:-1]
n_batch, n_time, n_note, n_ipn = input_slice.shape

# time_inputs is a matrix (time, batch/note, input_per_note)
time_inputs = input_slice.transpose((1,0,2,3)).reshape((n_time,n_batch*n_note,n_ip
num_time_parallel = time_inputs.shape[1]

# apply dropout
if self.dropout > 0:
    time_masks = theano_lstm.MultiDropout( [(num_time_parallel, shape) for shape in
else:
    time_masks = []

time_outputs_info = [initial_state_with_taps(layer, num_time_parallel) for layer in
time_result, _ = theano.scan(fn=step_time, sequences=[time_inputs], non_sequences=

self.time_thoughts = time_result

# Now time_result is a list of matrix [layer](time, batch/note, hidden_states) for
# the hidden state of the last layer.
# Transpose to be (note, batch/time, hidden_states)
last_layer = get_last_layer(time_result)
n_hidden = last_layer.shape[2]
time_final = get_last_layer(time_result).reshape((n_time,n_batch,n_note,n_hidden))

# note_choices_inputs represents the last chosen note. Starts with [0,0], doesn't
# In (note, batch/time, 2) format
# Shape of start is thus (1, N, 2), concatenated with all but last element of outp
start_note_values = T.alloc(np.array(0,dtype=np.int8), 1, time_final.shape[1], 2 )
```

```
        correct_choices = self.output_mat[:,1:,0:-1,:].transpose((2,0,1,3)).reshape((n_note
        note_choices_inputs = T.concatenate([start_note_values, correct_choices], axis=0)

        # Together, this and the output from the last LSTM goes to the new LSTM, but rotate
        # one direction are the steps in the other, and vice versa.
        note_inputs = T.concatenate( [time_final, note_choices_inputs], axis=2 )
        num_timebatch = note_inputs.shape[1]

        # apply dropout
        if self.dropout > 0:
            pitch_masks = theano_lstm.MultiDropout( [(num_timebatch, shape) for shape in s
        else:
            pitch_masks = []

        note_outputs_info = [initial_state_with_taps(layer, num_timebatch) for layer in se
        note_result, _ = theano.scan(fn=step_note, sequences=[note_inputs], non_sequences=

        self.note_thoughts = note_result

        # Now note_result is a list of matrix [layer](note, batch/time, onOrArticProb) for
        # the hidden state of the last layer.
        # Transpose to be (batch, time, note, onOrArticProb)
        note_final = get_last_layer(note_result).reshape((n_note,n_batch,n_time,2)).transpo

        # The cost of the entire procedure is the negative log likelihood of the events al
        # For the purposes of training, if the ouputted probability is P, then the likeliho
        # the likelihood of seeing 0 is (1-P). So the likelihood is (1-P)(1-x) + Px = 2Px -
        # Since they are all binary decisions, and are all probabilities given all previou
        # multiply the likelihoods, or, since we are logging them, add the logs.

        # Note that we mask out the articulations for those notes that aren't played, beca
        # whether or not those are articulated.
        # The padright is there because self.output_mat[:,:,:,0] -> 3D matrix with (b,x,y)
        # (b,x,y,1) instead
        active_notes = T.shape_padright(self.output_mat[:,1:,:,0])
        mask = T.concatenate([T.ones_like(active_notes),active_notes], axis=3)

        loglikelihoods = mask * T.log( 2*note_final*self.output_mat[:,1:] - note_final - s
        self.cost = T.neg(T.sum(loglikelihoods))

        updates, _, _, _, _ = create_optimization_updates(self.cost, self.params, method="
        self.update_fun = theano.function(
            inputs=[self.input_mat, self.output_mat],
            outputs=self.cost,
            updates=updates,
            allow_input_downcast=True)

        self.update_thought_fun = theano.function(
            inputs=[self.input_mat, self.output_mat],
            outputs= ensure_list(self.time_thoughts) + ensure_list(self.note_thoughts) + [
            allow_input_downcast=True)

    def _predict_step_note(self, in_data_from_time, *states):
        # States is [ *hiddens, last_note_choice ]
```

```
        hiddens = list(states[:-1])
        in_data_from_prev = states[-1]
        in_data = T.concatenate([in_data_from_time, in_data_from_prev])

        # correct for dropout
        if self.dropout > 0:
            masks = [1 - self.dropout for layer in self.pitch_model.layers]
            masks[0] = None
        else:
            masks = []

        new_states = self.pitch_model.forward(in_data, prev_hiddens=hiddens, dropout=masks)

        # Now new_states is a per-layer set of activations.
        probabilities = get_last_layer(new_states)

        # Thus, probabilities is a vector of two probabilities, P(play), and P(artic | pla

        shouldPlay = self.srng.uniform() < (probabilities[0] ** self.conservativity)
        shouldArtic = shouldPlay * (self.srng.uniform() < probabilities[1])

        chosen = T.cast(T.stack(shouldPlay, shouldArtic), "int8")

        return ensure_list(new_states) + [chosen]

    def setup_predict(self):
        # In prediction mode, note steps are contained in the time steps. So the passing g

        self.predict_seed = T.bmatrix()
        self.steps_to_simulate = T.iscalar()

        def step_time(*states):
            # States is [ *hiddens, prev_result, time]
            hiddens = list(states[:-2])
            in_data = states[-2]
            time = states[-1]

            # correct for dropout
            if self.dropout > 0:
                masks = [1 - self.dropout for layer in self.time_model.layers]
                masks[0] = None
            else:
                masks = []

            new_states = self.time_model.forward(in_data, prev_hiddens=hiddens, dropout=ma

            # Now new_states is a list of matrix [layer](notes, hidden_states) for each la
            time_final = get_last_layer(new_states)

            start_note_values = theano.tensor.alloc(np.array(0,dtype=np.int8), 2)

            # This gets a little bit complicated. In the training case, we can pass in a c
            # time net's activations with the known choices. But in the prediction case, th
            # exist yet. So instead of iterating over the combination, we iterate over onl
```

```
        # and then combine in the previous outputs in the step. And then since we are
        # previous inputs, we need an additional outputs_info for the initial "previou
        note_outputs_info = ([ initial_state_with_taps(layer) for layer in self.pitch_m
                                [ dict(initial=start_note_values, taps=[-1]) ])

        notes_result, updates = theano.scan(fn=self._predict_step_note, sequences=[tim

        # Now notes_result is a list of matrix [layer/output](notes, onOrArtic)
        output = get_last_layer(notes_result)

        next_input = OutputFormToInputFormOp()(output, time + 1) # TODO: Fix time
        #next_input = T.cast(T.alloc(0, 3, 4),'int64')

        return (ensure_list(new_states) + [ next_input, time + 1, output ]), updates

    # start_sentinel = startSentinel()
    num_notes = self.predict_seed.shape[0]

    time_outputs_info = ([ initial_state_with_taps(layer, num_notes) for layer in self
                        [ dict(initial=self.predict_seed, taps=[-1]),
                          dict(initial=0, taps=[-1]),
                          None ])

    time_result, updates = theano.scan( fn=step_time,
                                        outputs_info=time_outputs_info,
                                        n_steps=self.steps_to_simulate )

    self.predict_thoughts = time_result

    self.predicted_output = time_result[-1]

    self.predict_fun = theano.function(
        inputs=[self.steps_to_simulate, self.conservativity, self.predict_seed],
        outputs=self.predicted_output,
        updates=updates,
        allow_input_downcast=True)

    self.predict_thought_fun = theano.function(
        inputs=[self.steps_to_simulate, self.conservativity, self.predict_seed],
        outputs=ensure_list(self.predict_thoughts),
        updates=updates,
        allow_input_downcast=True)

def setup_slow_walk(self):

    self.walk_input = theano.shared(np.ones((2,2), dtype='int8'))
    self.walk_time = theano.shared(np.array(0, dtype='int64'))
    self.walk_hiddens = [theano.shared(np.ones((2,2), dtype=theano.config.floatX)) for

    # correct for dropout
    if self.dropout > 0:
        masks = [1 - self.dropout for layer in self.time_model.layers]
        masks[0] = None
    else:
```

```
            masks = []

        new_states = self.time_model.forward(self.walk_input, prev_hiddens=self.walk_hidden

        # Now new_states is a list of matrix [layer](notes, hidden_states) for each layer
        time_final = get_last_layer(new_states)

        start_note_values = theano.tensor.alloc(np.array(0,dtype=np.int8), 2)
        note_outputs_info = ([ initial_state_with_taps(layer) for layer in self.pitch_mode
                             [ dict(initial=start_note_values, taps=[-1]) ])

        notes_result, updates = theano.scan(fn=self._predict_step_note, sequences=[time_fi

        # Now notes_result is a list of matrix [layer/output](notes, onOrArtic)
        output = get_last_layer(notes_result)

        next_input = OutputFormToInputFormOp()(output, self.walk_time + 1) # TODO: Fix tim
        #next_input = T.cast(T.alloc(0, 3, 4),'int64')

        slow_walk_results = (new_states[:-1] + notes_result[:-1] + [ next_input, output ])

        updates.update({
                self.walk_time: self.walk_time+1,
                self.walk_input: next_input
            })

        updates.update({hidden:newstate for hidden, newstate, layer in zip(self.walk_hidden

        self.slow_walk_fun = theano.function(
            inputs=[self.conservativity],
            outputs=slow_walk_results,
            updates=updates,
            allow_input_downcast=True)

    def start_slow_walk(self, seed):
        seed = np.array(seed)
        num_notes = seed.shape[0]

        self.walk_time.set_value(0)
        self.walk_input.set_value(seed)
        for layer, hidden in zip((l for l in self.time_model.layers if has_hidden(l)),self
            hidden.set_value(np.repeat(np.reshape(layer.initial_hidden_state.get_value(),

"""main"""

import cPickle as pickle
import gzip
import numpy
from midi_to_statematrix import *

import multi_training
import model

def gen_adaptive(m,pcs,times,keep_thoughts=False,name="final"):
```

```
        xIpt, xOpt = map(lambda x: numpy.array(x, dtype='int8'), multi_training.getPieceSeg
        all_outputs = [xOpt[0]]
        if keep_thoughts:
                all_thoughts = []
        m.start_slow_walk(xIpt[0])
        cons = 1
        for time in range(multi_training.batch_len*times):
                resdata = m.slow_walk_fun( cons )
                nnotes = numpy.sum(resdata[-1][:,0])
                if nnotes < 2:
                        if cons > 1:
                                cons = 1
                        cons -= 0.02
                else:
                        cons += (1 - cons)*0.3
                all_outputs.append(resdata[-1])
                if keep_thoughts:
                        all_thoughts.append(resdata)
        noteStateMatrixToMidi(numpy.array(all_outputs),'output/'+name)
        if keep_thoughts:
                pickle.dump(all_thoughts, open('output/'+name+'.p','wb'))

def fetch_train_thoughts(m,pcs,batches,name="trainthoughts"):
        all_thoughts = []
        for i in range(batches):
                ipt, opt = multi_training.getPieceBatch(pcs)
                thoughts = m.update_thought_fun(ipt,opt)
                all_thoughts.append((ipt,opt,thoughts))
        pickle.dump(all_thoughts, open('output/'+name+'.p','wb'))

if __name__ == '__main__':

        pcs = multi_training.loadPieces("music")

        m = model.Model([300,300],[100,50], dropout=0.5)

        multi_training.trainPiece(m, pcs, 10000)

        pickle.dump( m.learned_config, open( "output/final_learned_config.p", "wb" ) )
```

6 Sources

http://www.hexahedria.com/2015/08/03/composing-music-with-recurrent-neural-networks/
https://towardsdatascience.com/how-to-generate-music-using-a-lstm-neural-network-in-keras-68786834d4c5
https://towardsdatascience.com/making-music-when-simple-probabilities-outperform-deep-learning-75f4ee1b