RNN Chopin Music Generation

Yushuo (Shawn) Han

June 2020

1 Introduction

With the advance of deep learning, computers can now compose music using trained models based on the sample pieces. This not only allows artists that are lack of the domain knowledge to accomplish their ideas, but also provides possible creative styles and patterns that may foster the composer's creative process.

In the project, a Recurrent Neural Network (RNN) model is built using LSTM and dense layers. The model is then trained on 10 of Frédéric Chopin's Valse and Waltz pieces in MIDI format downloaded from *kunstderfuge.com*. The MIDI files are converted to numeric, sequential data and fed into the RNN for training. The trained RNN reads a pattern of notes, then generates new notes based on the sequence.

Moreover, to explore different styles of piano pieces, a dataset of 8 Nocturne pieces by Frédéric Chopin downloaded from *8notes.com* are used to train a complete seperate RNN model. The generated sample pieces from the Valse model and the Nocturne model can be found here.

2 Data Preprocessing

Import necessary libraries and collect data files.

[1]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
%matplotlib inline

from music21 import converter, instrument, note, chord, stream
from timidity import Parser, play_notes

import pickle
import glob

from keras.utils import to_categorical
from keras.models import Sequential
from keras.layers import Activation, Dense, LSTM, Dropout, Flatten, Embedding
from keras.callbacks import ModelCheckpoint
```

```
Using TensorFlow backend.
```

[2]:
```
valseMusic = glob.glob('Nocturnes Dataset/*.mid')
valseMusic
```

[2]:
```
['Nocturnes Dataset/chopin_Nocturne_b49.mid',
 'Nocturnes Dataset/chopin_op55_1.mid',
 'Nocturnes Dataset/Nocturne_Opus_72_No_1_in_E_Minor.mid',
 'Nocturnes Dataset/chopin_nocturneop9nr1.mid',
 'Nocturnes Dataset/chopin_op55_2.mid',
 'Nocturnes Dataset/chop32n2.mid',
 'Nocturnes Dataset/chopin_nocturne_9_2.mid',
 'Nocturnes Dataset/chopin_cminor_nocturne_b108_PNO.mid']
```

Using the *music21* library, the MIDI files are first parsed into single instrument track of notes and chords, then chords are further broken down into groups of notes. The MIDI files are concatenated and are eventually converted to a sequence of notes.

[3]:
```
processedNotes = []

for valse in valseMusic:

    # parse Midi
    valse = converter.parse(valse)

    # Handle cases for multi-instruments
    try:
        parts = instrument.partitionByInstrument(valse)
        notes = parts.parts[0].recurse()
    except:
        notes = valse.flat.notes

    # Handle chords
    for singleNote in notes:
        if isinstance(singleNote, chord.Chord):
            processedNotes.append('.'.join(str(norm) for norm in singleNote.
 ↪normalOrder))
        elif isinstance(singleNote, note.Note):
            processedNotes.append(str(singleNote.pitch))

# Save a notes library
with open('nocturnesNotesLibrary', 'wb') as file:
    pickle.dump(processedNotes, file)
```

Furthermore, each unique note/pitch is assigned with a unique number. The sequence of notes is then divided into subsequences, each containing the numeric representation of 100 notes. The training data that is fed into the network is then sequences of 100 numbers, each representing a corresponding note. The label/output of each input sequence is the next note that is after the 100-note sequence, also represented by a number.

[4]:
```
SEQSIZE = 100

alphabetSize = len(set(processedNotes))

# Find unique pitches
pitches = sorted(set(processedNote for processedNote in processedNotes))

# a dict to convert each unique note to a unique number.
dictRef = dict((pitch, num) for num, pitch in enumerate(pitches))

inputNotes = []
outputNotes = []

# Divide sequence into subsequences of 100 notes, then convert notes to numbers
for i in range(0, len(processedNotes) - SEQSIZE):
    batchInput = processedNotes[i: i+SEQSIZE]
    batchOutput = processedNotes[i+SEQSIZE]
    inputNotes.append([dictRef[n] for n in batchInput])
    outputNotes.append([dictRef[batchOutput]])

# reshape the input and normalize, convert output to categorical.
batchNum = len(inputNotes)
inputNotes = np.reshape(inputNotes, (batchNum, SEQSIZE, 1)) / float(alphabetSize)
outputNotes = to_categorical(outputNotes)
```

3 RNN Model Training

The design of the RNN model is traditional and is structured as follows. Two LSTM-and-Dropout groups are followed by two densely-connected layers, also accompanied by Dropout layers. The output activation function is softmax. Since this is a single-label classification task, categorical crossentropy loss is used.

[5]:
```
def createRNNModel(inputNotes, outputNum):

    rnnModel = Sequential([
        LSTM(128,
             input_shape=inputNotes.shape[1:],
             return_sequences=True,
             recurrent_initializer='glorot_uniform',
             recurrent_activation='sigmoid'),
        Dropout(0.2),

        LSTM(128,
             return_sequences=True,
             recurrent_initializer='glorot_uniform',
             recurrent_activation='sigmoid'),
```

```
        Dropout(0.2),

        Flatten(),
        Dropout(0.2),
        Dense(256),
        Dropout(0.2),

        Dense(outputNum, activation='softmax')
    ])

    rnnModel.compile(loss='categorical_crossentropy', optimizer='adam')

    return rnnModel

rnnModel = createRNNModel(inputNotes, alphabetSize)
```

To ensure the style is mapped, the model is trained with 500 epochs. It is worth mentioning that since there are such great varieties in the labels, validation and early stopping mechanisms are not used.

[6]:
```
checkpointCallback = ModelCheckpoint('ChopinNocturnesLSTMx2.hdf5',␣
 ↪monitor='loss',
                              verbose=2, save_best_only=True)
rnnModel.fit(inputNotes, outputNotes, epochs=500, batch_size=32,␣
 ↪callbacks=[checkpointCallback])
```

```
Epoch 1/500
7164/7164 [==============================] - 36s 5ms/step - loss: 4.4920

Epoch 00001: loss improved from inf to 4.49198, saving model to
ChopinNocturnesLSTMx2.hdf5
Epoch 2/500
7164/7164 [==============================] - 33s 5ms/step - loss: 4.4054

Epoch 00002: loss improved from 4.49198 to 4.40539, saving model to
ChopinNocturnesLSTMx2.hdf5
Epoch 3/500
7164/7164 [==============================] - 31s 4ms/step - loss: 4.3492

Epoch 00003: loss improved from 4.40539 to 4.34921, saving model to
ChopinNocturnesLSTMx2.hdf5
Epoch 4/500
7164/7164 [==============================] - 31s 4ms/step - loss: 4.2998

Epoch 00004: loss improved from 4.34921 to 4.29977, saving model to
ChopinNocturnesLSTMx2.hdf5
Epoch 5/500
7164/7164 [==============================] - 31s 4ms/step - loss: 4.2629
```

```
Epoch 00005: loss improved from 4.29977 to 4.26293, saving model to
ChopinNocturnesLSTMx2.hdf5
Epoch 496/500
7164/7164 [==============================] - 36s 5ms/step - loss: 0.0660

Epoch 00496: loss did not improve from 0.02900
Epoch 497/500
7164/7164 [==============================] - 36s 5ms/step - loss: 0.0531

Epoch 00497: loss did not improve from 0.02900
Epoch 498/500
7164/7164 [==============================] - 36s 5ms/step - loss: 0.0354

Epoch 00498: loss did not improve from 0.02900
Epoch 499/500
7164/7164 [==============================] - 36s 5ms/step - loss: 0.0421

Epoch 00499: loss did not improve from 0.02900
Epoch 500/500
7164/7164 [==============================] - 36s 5ms/step - loss: 0.0587

Epoch 00500: loss did not improve from 0.02900
```

[6]:
```
<keras.callbacks.callbacks.History at 0xa41775950>
```

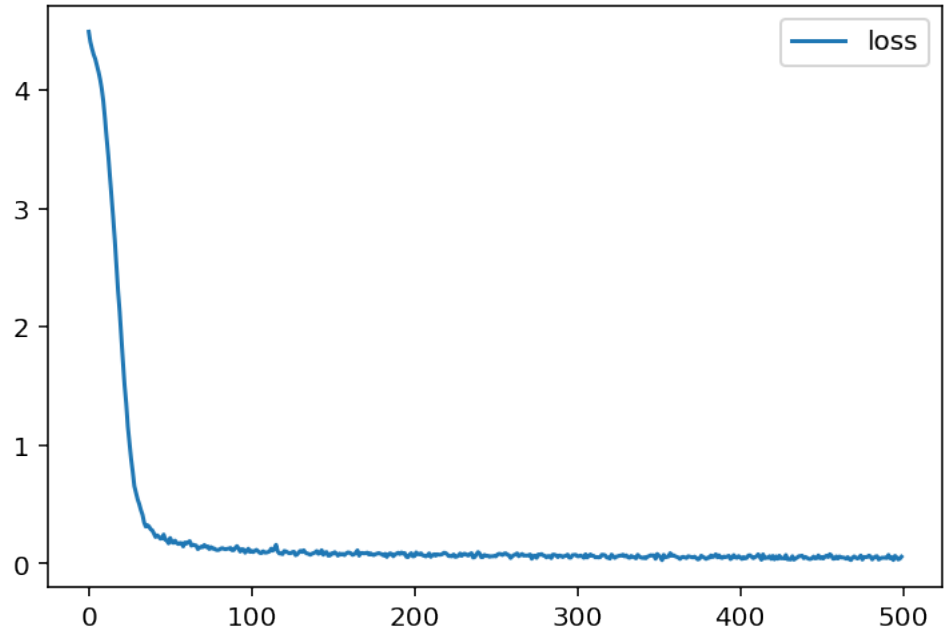
[7]:
```
rnnModel.summary()
```

```
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
lstm_1 (LSTM)                (None, 100, 128)          66560
_________________________________________________________________
dropout_1 (Dropout)          (None, 100, 128)          0
_________________________________________________________________
lstm_2 (LSTM)                (None, 100, 128)          131584
_________________________________________________________________
dropout_2 (Dropout)          (None, 100, 128)          0
_________________________________________________________________
flatten_1 (Flatten)          (None, 12800)             0
_________________________________________________________________
dropout_3 (Dropout)          (None, 12800)             0
_________________________________________________________________
dense_1 (Dense)              (None, 256)               3277056
_________________________________________________________________
dropout_4 (Dropout)          (None, 256)               0
_________________________________________________________________
```

```
dense_2 (Dense)              (None, 200)               51400
=================================================================
Total params: 3,526,600
Trainable params: 3,526,600
Non-trainable params: 0
_________________________________________________________________
```

[8]:
```
modelLoss = pd.DataFrame(rnnModel.history.history)
modelLoss.plot()
```

[8]:
```
<matplotlib.axes._subplots.AxesSubplot at 0xa415be990>
```



4 Music Generation

After the weights are re-loaded into the model, five sequences of 100 numbers, each representing a 100-note sequence, are randomly drawn to be the input data for model prediction.

[9]:
```
SEQSIZE = 100

# Set up prediction input by mapping notes to integers.
inputNotes = []

for i in range(0, len(notes) - SEQSIZE):
    batchInput = processedNotes[i: i+SEQSIZE]
    inputNotes.append([dictRef[n] for n in batchInput])
```

```
# reshape input and normalize
batchNum = len(inputNotes)
inputNotes = np.reshape(inputNotes, (batchNum, SEQSIZE, 1)) / float(alphabetSize)

# create model using weight.
rnnModel2 = createRNNModel(inputNotes, alphabetSize)
rnnModel2.load_weights('ChopinNocturnesLSTMx2.hdf5')

# inputs: 5 random sequences
patterns = []
for i in range(0, 5):
    startNote = np.random.randint(0, len(inputNotes)-1)
    patterns.append(list(inputNotes[startNote]))

dictRefReverse = dict((num, pitch) for num, pitch in enumerate(pitches))
```

5 predictions are made based on the 5 randomly selected input sequences. Each input is first reshaped and normalized before fed into the model. The prediction output is then converted to a note, and is used to generate the next note as part of the input sequence.

[10]:
```
from copy import deepcopy
songs = [[], [], [], [], []]

# predict, generate output notes.
# Generate 5 songs, each song has 800 notes.
for j in range(5):
    pat = patterns[j]
    for i in range(800):
        # reshape input and predict
        predIn = np.reshape(pat, (1, len(pat), 1)) / float(alphabetSize)
        prediction = rnnModel2.predict(predIn, verbose=0)

        # Convert prediction back to notes
        index = np.argmax(prediction)
        prediction = dictRefReverse[index]

        # append to output, and use the new note for further generation.
        songs[j].append(prediction)
        pat.append(index)
        pat = pat[1: len(pat)]
    print('{} song created.'.format(j+1))
```

```
1 song created.
2 song created.
3 song created.
4 song created.
5 song created.
```

The prediction outputs, each from a sequence of notes, are converted back to a stream of MIDI data.

[11]:
```
def generateMidi(outputNotes, fileName):
    offset = 0
    midiOutput = []

    for notePat in outputNotes:

        # It is a chord.
        if ('.' in notePat) or notePat.isdigit():

                # Split to notes
                notes_in_chord = notePat.split('.')
                notes = []

                # chord as notes
                for eachNote in notes_in_chord:
                    new_note = note.Note(int(eachNote))
                    new_note.storedInstrument = instrument.Piano()
                    notes.append(new_note)

                # Convert to midi chord and store
                new_chord = chord.Chord(notes)
                new_chord.offset = offset
                midiOutput.append(new_chord)

        # pattern is a note
        else:
            # Convert to midi note and store
            new_note = note.Note(notePat)
            new_note.offset = offset
            new_note.storedInstrument = instrument.Piano()
            midiOutput.append(new_note)

        # increase offset to make sure notes do not stack
        offset += 0.5

    # Convert to midi stream
    midiOutput = stream.Stream(midiOutput)
    midiOutput.write('midi', fp=fileName)

for num, song in enumerate(songs):
    generateMidi(song, 'ChopinNocturnesGenerated_{}.mid'.format(num))
```

[12]:
```
# Listen to the midi music, uncomment the file wish to listen.
# ps = Parser("ChopinValseGenerated_0.mid")
# ps = Parser("ChopinValseGenerated_1.mid")
```

```
# ps = Parser("ChopinValseGenerated_2.mid")
# ps = Parser("ChopinValseGenerated_3.mid")
# ps = Parser("ChopinValseGenerated_4.mid")
# ps = Parser("ChopinNocturnesGenerated_0.mid")
# ps = Parser("ChopinNocturnesGenerated_1.mid")
# ps = Parser("ChopinNocturnesGenerated_2.mid")
# ps = Parser("ChopinNocturnesGenerated_3.mid")
# ps = Parser("ChopinNocturnesGenerated_4.mid")

# play_notes(*ps.parse(), np.sin)
```

5 Conclusion

Music generation is a new emerging field with recent advance in deep learning. As a crossroad of technology and art, it extends the ability of artists and brings new elements of style and tempo into music creation. In this music generation project, an LSTM-based RNN model is designed, implemented and trained with a set of 10 Chopin's Valse, and separately, a set of 8 Chopin's Nocturnes. While the generated music pieces are definitely of no match to Frédéric Chopin's original pieces, it provides an insight to the ability of deep learning in art generation, and has proven that music pieces generated by deep learning models are non-trivial and possess a considerable degree of artistic values.

6 Acknowledgement

This notebook is partially based on the instructions given by Shubham Gupta. His effort and guidance is greatly appreciated.