Name: Soon Chye Lim
Title: Generate Music Using LSTM Network

I. Introduction:

I used the LSTM Network to build a classical piece for my project. Both Music and Deep Learning knowledge are required when building this project. I never thought one could use two very different subjects to create a masterpiece. However, there are many limitations or quality of the music, which I will discuss later.

Classical music has been existing for a long time. And there are many periods of classical music. Below are the most recent periods:

Baroque (1600-1750)
Classical (1750-1820)
Romantic (1800-1915)
20th Century (1900-2000)
N
Each period of classical music has its distinct characteristics. Therefore, I will use the most sophisticated period (20th Century music) as my input to feed into my neural network. And hopefully, the network will generate a piece of 20th-century music.

II. Problem and Data Description:

Two problems arise when working on this project.

1. What inputs do I use to generate music?

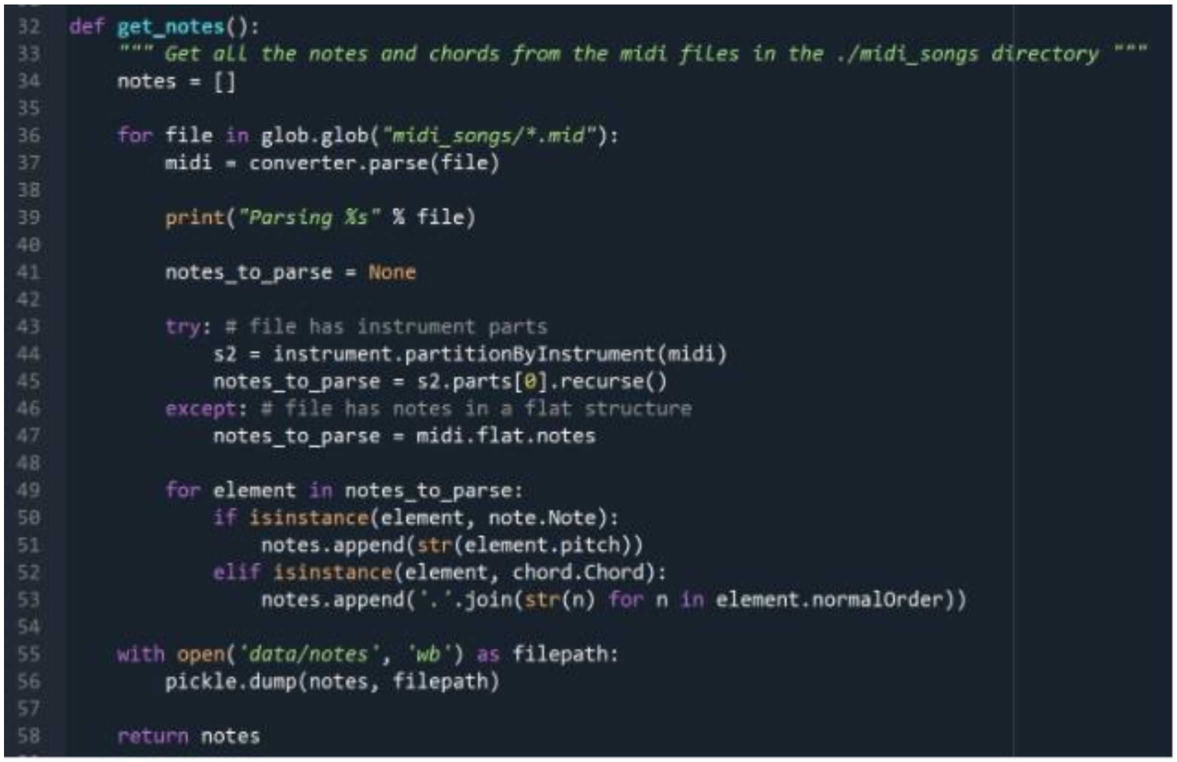
I downloaded 39 midi files from the link below:
https://www.midiworld.com/classic.htm/beethoven.htm
I selected music composed by 20th-century composers like Debussy, Ravel, Stravinsky, Prokofiev, and Satie.

2. How do I build the model to generate music?

My model consists of two parts.
Part 1 – Training (lstm.py):
This template will learn the 39 midi files.
Part 2 – Prediction (predict.py):
This template will generate a piece of music based on the output of the training model.

III. Data Preparation:

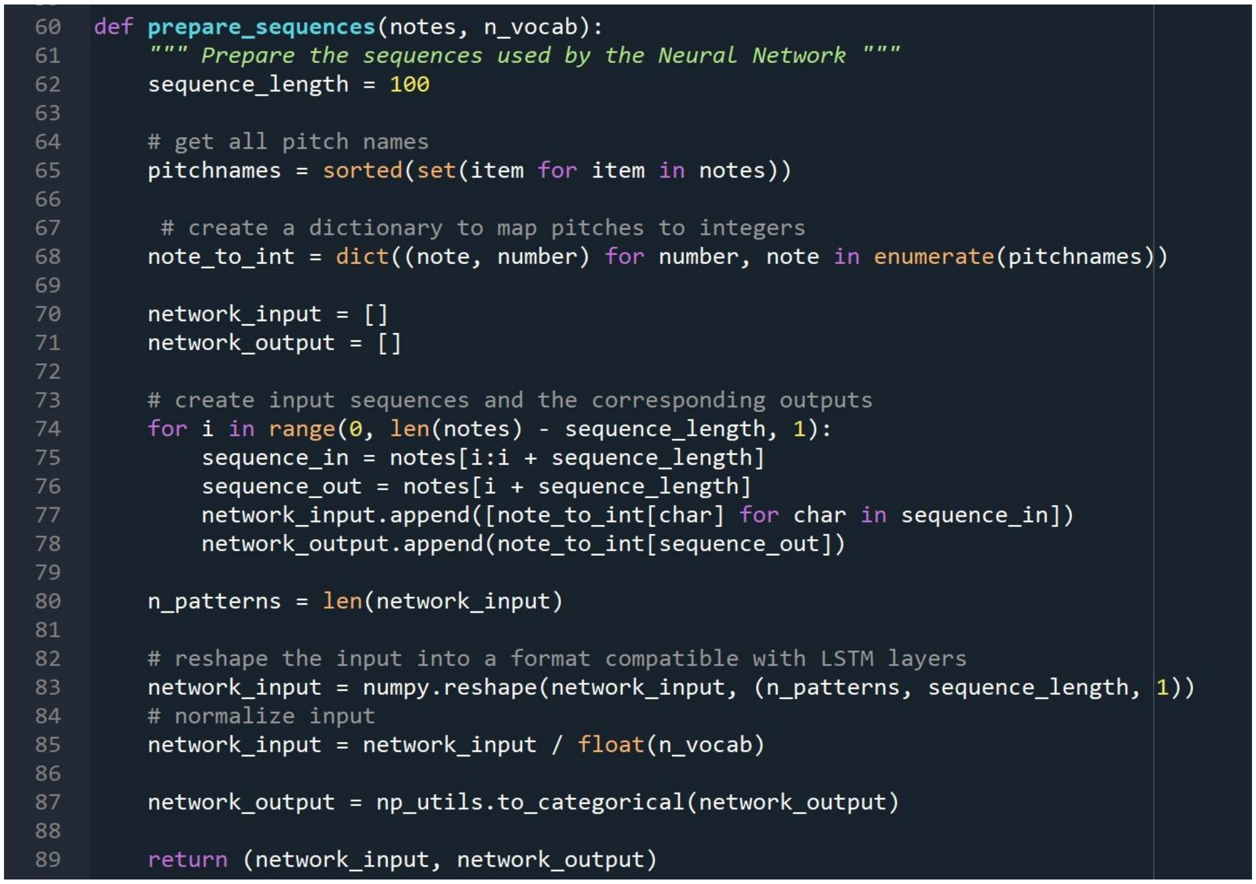
Accessing audio files became more manageable when I used the Music21 Python package. Music21 extracts the notes from the music and takes the neural network's output, translating it to musical notation. This package can only read midi files. Therefore I make sure all my inputs are in midi format.



```python
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)

        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
```

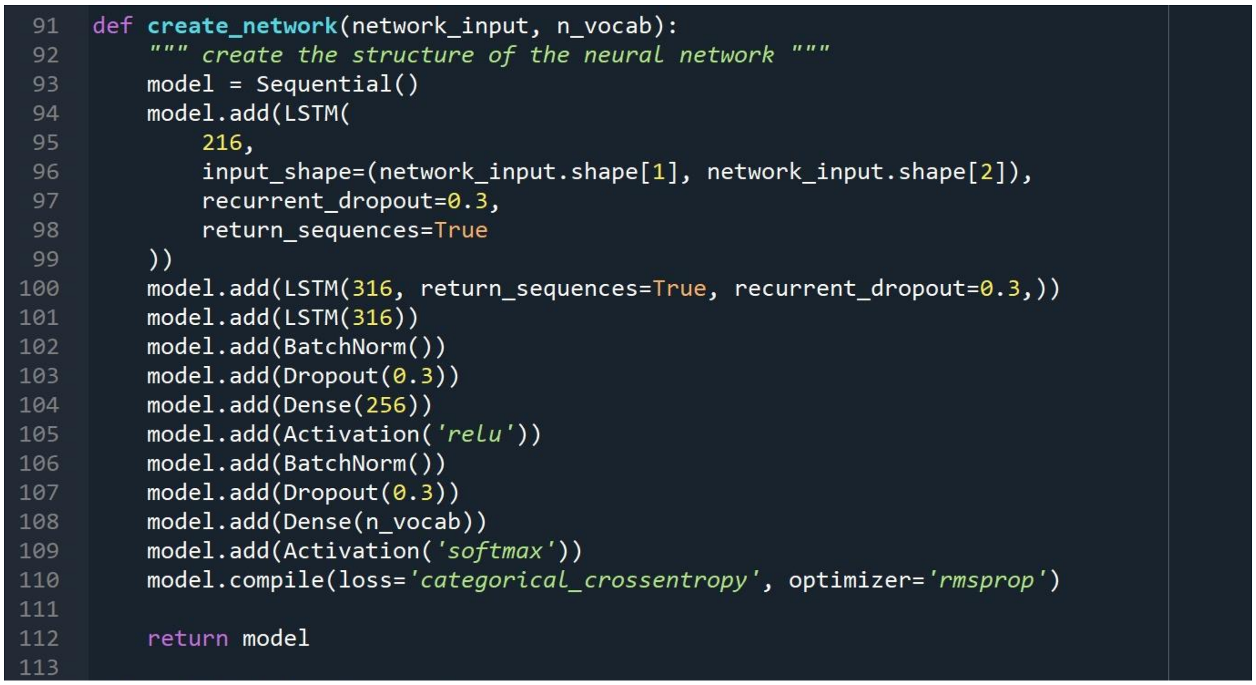
The above codes show how I parsed each note from the midi file to a list of notes.

IV. Model Description:



```python
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)
```

Music is the perfect example of the LSTM model because it flows with time. And based on my input and output, I found out that this model is a sequence to sequence model. Thus, I started to prepare sequences for each midi file, as shown in the codes above. For this model, I set my sequence length to 100 notes. The network will learn 100 notes per sequence and then move on to the next 100 notes.



```python
def create_network(network_input, n_vocab):
    """ create the structure of the neural network """
    model = Sequential()
    model.add(LSTM(
        216,
        input_shape=(network_input.shape[1], network_input.shape[2]),
        recurrent_dropout=0.3,
        return_sequences=True
    ))
    model.add(LSTM(316, return_sequences=True, recurrent_dropout=0.3,))
    model.add(LSTM(316))
    model.add(BatchNorm())
    model.add(Dropout(0.3))
    model.add(Dense(256))
    model.add(Activation('relu'))
    model.add(BatchNorm())
    model.add(Dropout(0.3))
    model.add(Dense(n_vocab))
    model.add(Activation('softmax'))
    model.compile(loss='categorical_crossentropy', optimizer='rmsprop')

    return model

```

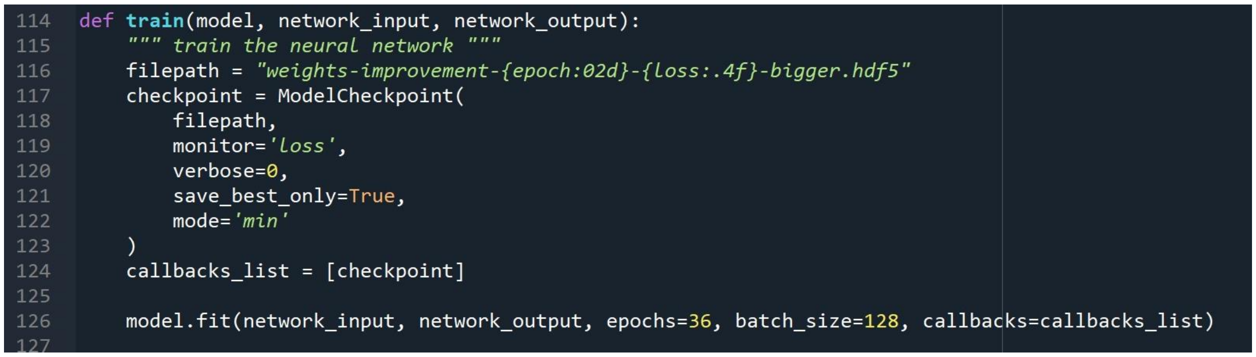
I built the first three layers of LSTM in my model and added some batch normalization to speed up my learning rate. Also, I added both recurrent dropout and dropout layers as my regularization technique to reduce overfitting.

So what does overfitting mean in the music context? I want to make sure the output doesn't sound like one of the songs in my input files. Yet, it must have characteristics of 20th-Century music.

My last layer is a dense layer of n_vocab, a variable set by me. Every piece of music has a set of keys, which is known as the key signature. And this is what n_vocab means; it stores a set of keys for each midi file. Later in the prediction phase, the model will take the keys from n_vocab and construct a new piece.

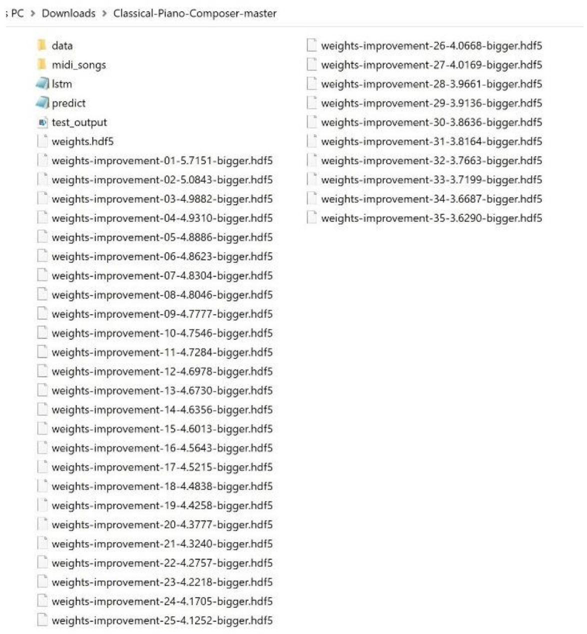
My activation function is a softmax because music consists of 12 keys per octave. For example, there are 88 keys in the piano, and each key is like a category of its own. My loss model is categorical crossentropy, and I used rmsprop as my optimizer.

V. Training:



```python
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=36, batch_size=128, callbacks=callbacks_list)
```

I make sure to save every epoch during my training phase. And I ran 36 epochs with a batch size of 128. Below is the result of each epoch with its loss measure.



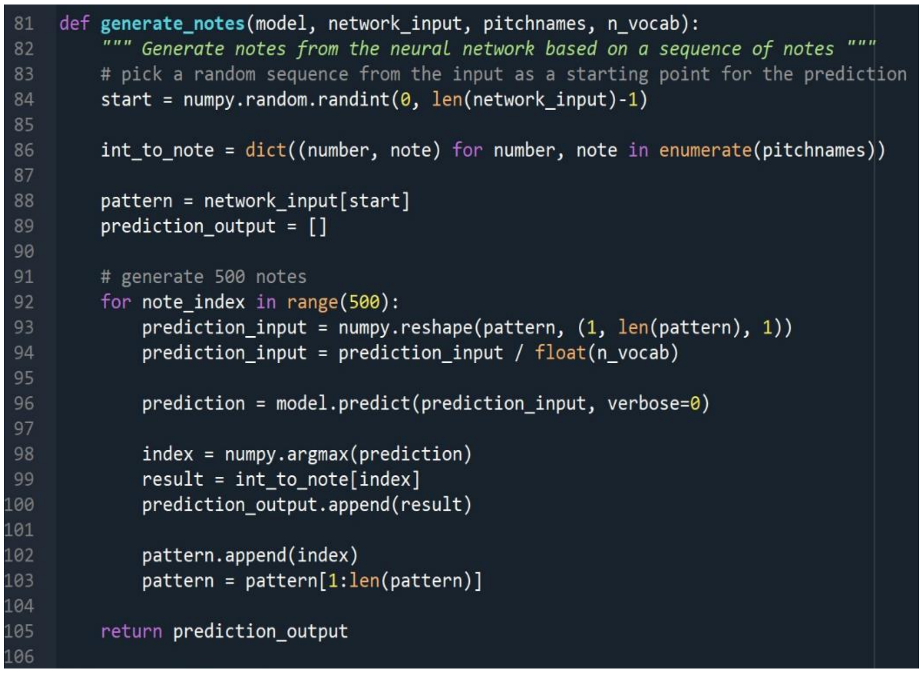
```
; PC > Downloads > Classical-Piano-Composer-master

data
midi_songs
lstm
predict
test_output
weights.hdf5
weights-improvement-01-5.7151-bigger.hdf5
weights-improvement-02-5.0843-bigger.hdf5
weights-improvement-03-4.9882-bigger.hdf5
weights-improvement-04-4.9310-bigger.hdf5
weights-improvement-05-4.8886-bigger.hdf5
weights-improvement-06-4.8623-bigger.hdf5
weights-improvement-07-4.8304-bigger.hdf5
weights-improvement-08-4.8046-bigger.hdf5
weights-improvement-09-4.7777-bigger.hdf5
weights-improvement-10-4.7546-bigger.hdf5
weights-improvement-11-4.7284-bigger.hdf5
weights-improvement-12-4.6978-bigger.hdf5
weights-improvement-13-4.6730-bigger.hdf5
weights-improvement-14-4.6356-bigger.hdf5
weights-improvement-15-4.6013-bigger.hdf5
weights-improvement-16-4.5643-bigger.hdf5
weights-improvement-17-4.5215-bigger.hdf5
weights-improvement-18-4.4838-bigger.hdf5
weights-improvement-19-4.4258-bigger.hdf5
weights-improvement-20-4.3777-bigger.hdf5
weights-improvement-21-4.3240-bigger.hdf5
weights-improvement-22-4.2757-bigger.hdf5
weights-improvement-23-4.2218-bigger.hdf5
weights-improvement-24-4.1705-bigger.hdf5
weights-improvement-25-4.1252-bigger.hdf5
weights-improvement-26-4.0668-bigger.hdf5
weights-improvement-27-4.0169-bigger.hdf5
weights-improvement-28-3.9661-bigger.hdf5
weights-improvement-29-3.9136-bigger.hdf5
weights-improvement-30-3.8636-bigger.hdf5
weights-improvement-31-3.8164-bigger.hdf5
weights-improvement-32-3.7663-bigger.hdf5
weights-improvement-33-3.7199-bigger.hdf5
weights-improvement-34-3.6687-bigger.hdf5
weights-improvement-35-3.6290-bigger.hdf5
```

I picked the lowest loss, which is less than 3.6290, and then changed the name to weights.hdf5. This file will be used during the prediction phase.

VI. Prediction:

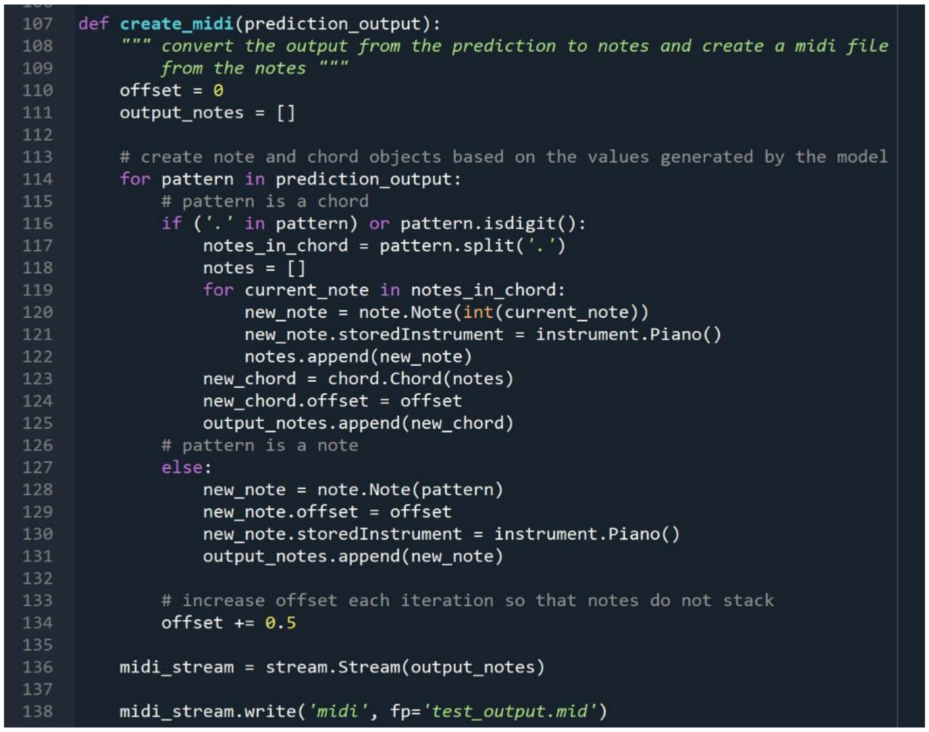
In my prediction phase, I reused codes from the training section to prepare the data and build the network model similar to the training phase. However, instead of training the network, I load the weights that I saved (weights.hdf5) during the training section into the model.

In addition to the copied codes from the training section, I added two definitions as follows:



```python
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

```

I will generate 500 notes from the n_vocab list, as I mentioned before.

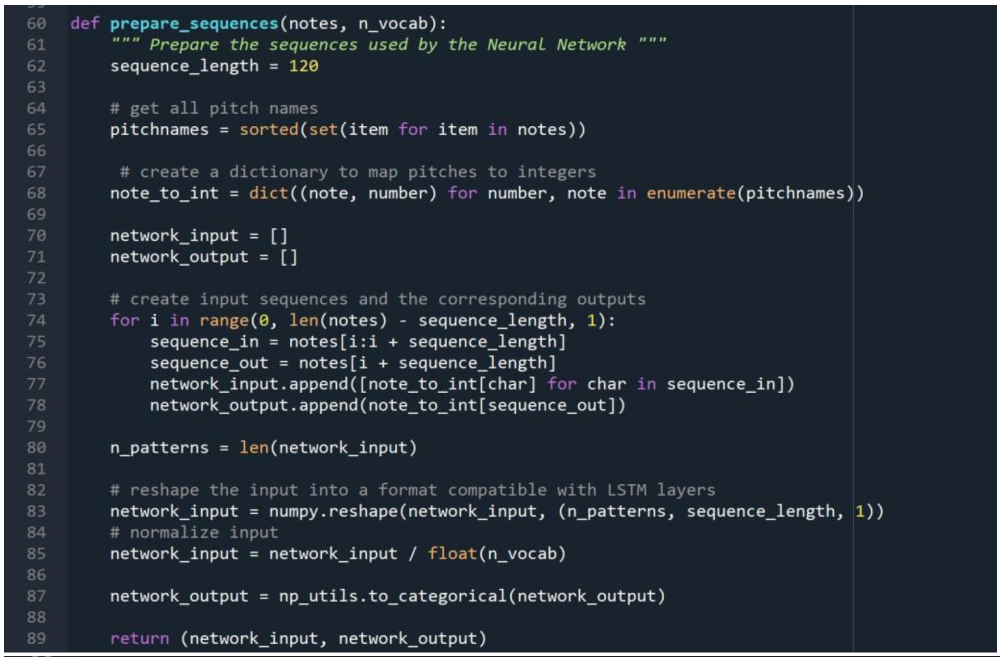


```python
def create_midi(prediction_output):
    """ convert the output from the prediction to notes and create a midi file
        from the notes """
    offset = 0
    output_notes = []

    # 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')
```

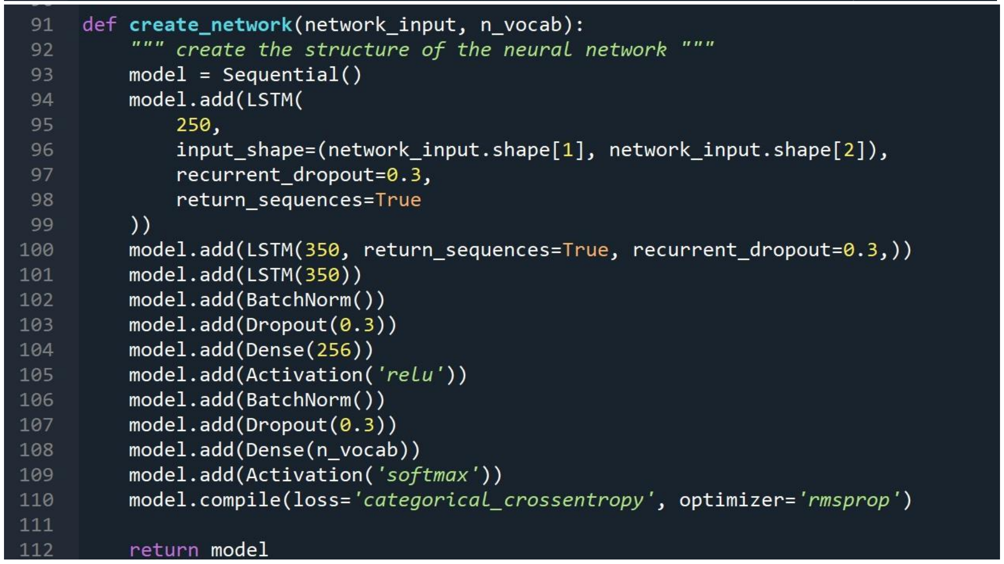
When creating the mid file, I set my instrument sound to piano and offset between notes to 0.5. Offset in this context means the rhythm in music. The offset between 0.5 seconds means the notes will be played at every 0.5 seconds. This resulted in a limitation of the music pieces I produce (the rhythm is very rigid and does not change).

VII. Alternative Model:

I set sequence from 100 to 120, first three lstm neurons from to (216,316,316) to (250,350,350).

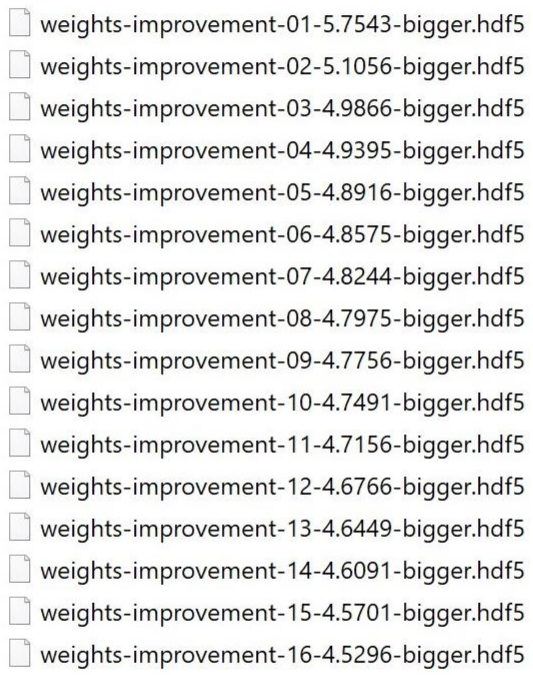


```python
def prepare_sequences(notes, n_vocab):
    """ Prepare the sequences used by the Neural Network """
    sequence_length = 120

    # 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)
```



```python
def create_network(network_input, n_vocab):
    """ create the structure of the neural network """
    model = Sequential()
    model.add(LSTM(
        250,
        input_shape=(network_input.shape[1], network_input.shape[2]),
        recurrent_dropout=0.3,
        return_sequences=True
    ))
    model.add(LSTM(350, return_sequences=True, recurrent_dropout=0.3,))
    model.add(LSTM(350))
    model.add(BatchNorm())
    model.add(Dropout(0.3))
    model.add(Dense(256))
    model.add(Activation('relu'))
    model.add(BatchNorm())
    model.add(Dropout(0.3))
    model.add(Dense(n_vocab))
    model.add(Activation('softmax'))
    model.compile(loss='categorical_crossentropy', optimizer='rmsprop')

    return model
```

And I ran 16 epochs as shown in the diagram below.



```
weights-improvement-01-5.7543-bigger.hdf5
weights-improvement-02-5.1056-bigger.hdf5
weights-improvement-03-4.9866-bigger.hdf5
weights-improvement-04-4.9395-bigger.hdf5
weights-improvement-05-4.8916-bigger.hdf5
weights-improvement-06-4.8575-bigger.hdf5
weights-improvement-07-4.8244-bigger.hdf5
weights-improvement-08-4.7975-bigger.hdf5
weights-improvement-09-4.7756-bigger.hdf5
weights-improvement-10-4.7491-bigger.hdf5
weights-improvement-11-4.7156-bigger.hdf5
weights-improvement-12-4.6766-bigger.hdf5
weights-improvement-13-4.6449-bigger.hdf5
weights-improvement-14-4.6091-bigger.hdf5
weights-improvement-15-4.5701-bigger.hdf5
weights-improvement-16-4.5296-bigger.hdf5
```

It looks like the first model has a lower loss. So I will stick to the first model.

VIII. Challenges:

The epoch is running very slowly because the model had to process big chunks of data. The model had to remember all sequences, resulting in reaching only a limited number of epochs.

Besides, other music elements, like the rhythm, dynamics, texture, harmony, and timbre, are not shown in the neural network's music. The expression is lacking when we listen to midi files compared to a professional pianist's live performance.

IX. Improvements:

I can add more PCs and use Hadoop System to run the job so that the epoch is running faster. Besides, if there is a way to transcribe a midi file to sheet music, someone could play it without sounding too "robotic."

Sometimes it is best for the human to do their job rather than relying on computers entirely. The AI does not understand the complex expression humans have been experiencing in their lives. Maybe in the future, someone could create an AI that would pass the Turing Test.