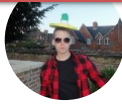


Images haven't loaded yet. Please exit printing, wait for images to load, and try to print again.



Sigurdur Skuli [Follow](#)

Data Science and Machine Learning Enthusiast

Dec 7, 2017 · 14 min read

How to Generate Music using a LSTM Neural Network in Keras



Introduction

Neural networks are being used to improve all aspects of our lives. They provide us with recommendations for items we want to purchase, generate text based on the style of an author and can even be used to change the art style of an image. In recent years, there have been a number of tutorials on how to generate text using neural networks but a lack of tutorials on how to create music. In this article we will go through how to create music using a recurrent neural network in Python using the Keras library.

For the impatient, there is a link to the Github repository at the end of the tutorial.

Background

Before we go into the details of the implementation there is some terminology that we must clarify.

Recurrent Neural Networks (RNN)

A recurrent neural network is a class of artificial neural networks that make use of sequential information. They are called recurrent because they perform the same function for every single element of a sequence, with the result being dependent on previous computations. Whereas outputs are independent of previous computations in traditional neural networks.

In this tutorial we will use a **Long Short-Term Memory (LSTM)** network. They are a type of Recurrent Neural Network that can efficiently learn via gradient descent. Using a gating mechanism, LSTMs are able to recognise and encode long-term patterns. LSTMs are extremely useful to solve problems where the network has to remember information for a long period of time as is the case in music and text generation.

Music21

Music21 is a Python toolkit used for computer-aided musicology. It allows us to teach the fundamentals of music theory, generate music examples and study music. The toolkit provides a simple interface to acquire the musical notation of MIDI files. Additionally, it allows us to create Note and Chord objects so that we can make our own MIDI files easily.

In this tutorial we will use Music21 to extract the contents of our dataset and to take the output of the neural network and translate it to musical notation.

Keras

Keras is a high-level neural networks API that simplifies interactions with Tensorflow. It was developed with a focus on enabling fast experimentation.

In this tutorial we will use the Keras library to create and train the LSTM model. Once the model is trained we will use it to generate the musical notation for our music.

Training

In this section we will cover how we gathered data for our model, how we prepared it so that it could be used in a LSTM model and the architecture of our model.

Data

In our [Github repository](#) we used piano music, mostly consisting of music from Final Fantasy soundtracks. We picked Final Fantasy music due to the very distinct and beautiful melodies that the majority of the pieces have and the sheer amount of pieces that exist. But any set of MIDI files consisting of a single instrument would work for our purposes.

The first step to implementing the neural network is to examine the data we will be working with.

Below we can see an excerpt from a midi file that has been read using Music21:

```
...  
<music21.note.Note F>  
<music21.chord.Chord A2 E3>  
<music21.chord.Chord A2 E3>  
<music21.note.Note E>  
<music21.chord.Chord B-2 F3>  
<music21.note.Note F>  
<music21.note.Note G>  
<music21.note.Note D>  
<music21.chord.Chord B-2 F3>
```

```
<music21.note.Note F>  
<music21.chord.Chord B-2 F3>  
<music21.note.Note E>  
<music21.chord.Chord B-2 F3>  
<music21.note.Note D>  
<music21.chord.Chord B-2 F3>  
<music21.note.Note E>  
<music21.chord.Chord A2 E3>  
...
```

The data splits into two object types: Notes and Chords. Note objects contain information about the **pitch**, **octave**, and **offset** of the Note.

- **Pitch** refers to the frequency of the sound, or how high or low it is and is represented with the letters [A, B, C, D, E, F, G], with A being the highest and G being the lowest.
- **Octave** refers to which set of pitches you use on a piano.
- **Offset** refers to where the note is located in the piece.

And Chord objects are essentially a container for a set of notes that are played at the same time.

Now we can see that to generate music accurately our neural network will have to be able to predict which note or chord is next. That means that our prediction array will have to contain every note and chord object that we encounter in our training set. In the training set on the Github page the total number of different notes and chords was 352. That seems like a lot of possible output predictions for the network to handle, but a LSTM network can easily handle it.

Next we have to worry about where we want to put the notes. As most people that have listened to music have noticed, notes usually have varying intervals between them. You can have many notes in quick succession and then followed by a rest period where no note is played for a short while.

Below we have another excerpt from a midi file that has been read using Music21, only this time we have added the offset of the object behind it. This allows us to see the interval between each note and chord.

```
...
<music21.note.Note B> 72.0
<music21.chord.Chord E3 A3> 72.0
<music21.note.Note A> 72.5
<music21.chord.Chord E3 A3> 72.5
<music21.note.Note E> 73.0
<music21.chord.Chord E3 A3> 73.0
<music21.chord.Chord E3 A3> 73.5
<music21.note.Note E-> 74.0
<music21.chord.Chord F3 A3> 74.0
<music21.chord.Chord F3 A3> 74.5
<music21.chord.Chord F3 A3> 75.0
<music21.chord.Chord F3 A3> 75.5
<music21.chord.Chord E3 A3> 76.0
<music21.chord.Chord E3 A3> 76.5
<music21.chord.Chord E3 A3> 77.0
<music21.chord.Chord E3 A3> 77.5
<music21.chord.Chord F3 A3> 78.0
<music21.chord.Chord F3 A3> 78.5
<music21.chord.Chord F3 A3> 79.0
...
```

As can be seen from this excerpt and most of the dataset, the most common interval between notes in the midi files is 0.5. Therefore, we can simplify the data and model by disregarding the varying offsets in the list of possible outputs. It will not affect the melodies of the music generated by the network too severely. So we will ignore the offset in this tutorial and keep our list of possible outputs at 352.

Preparing the Data

Now that we have examined the data and determined that the features that we want to use are the notes and chords as the input and output of our LSTM network it is time to prepare the data for the network.

First, we will load the data into an array as can be seen in the code snippet below:

```
from music21 import converter, instrument, note, chord

notes = []

for file in glob.glob("midi_songs/*.mid"):
    midi = converter.parse(file)
    notes_to_parse = None

    parts = instrument.partitionByInstrument(midi)

    if parts: # file has instrument parts
        notes_to_parse = parts.parts[0].recurse()
    else: # 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))
```

. . .

We start by loading each file into a Music21 stream object using the *converter.parse(file)* function. Using that stream object we get a list of all the notes and chords in the file. We append the pitch of every note object using its string notation since the most significant parts of the note can be recreated using the string notation of the pitch. And we append every chord by encoding the id of every note in the chord together into a single string, with each note being separated by a dot. These encodings allows us to easily decode the output generated by the network into the correct notes and chords.

Now that we have put all the notes and chords into a sequential list we can create the sequences that will serve as the input of our network.

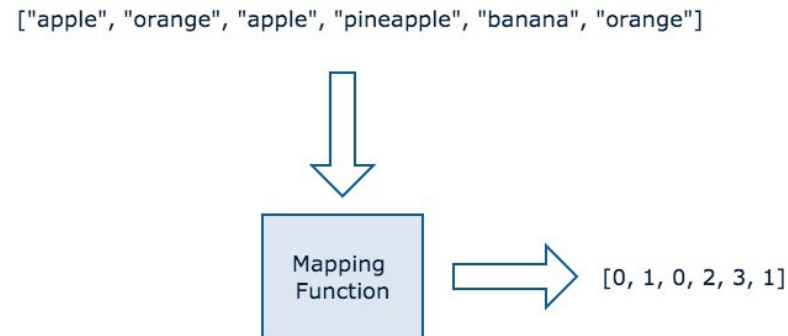


Figure 1: When converting from categorical to numerical data the data is converted to integer indexes representing where the category is positioned in the set of distinct values. E.g. apple is the first distinct value so it maps to 0, orange is the second so it maps to 1, pineapple is the third so it maps to 2, and so forth.

First, we will create a mapping function to map from string-based categorical data to integer-based numerical data. This is done because neural network perform much better with integer-based numerical data than string-based categorical data. An example of a categorical to numerical transformation can be seen in Figure 1.

Next, we have to create input sequences for the network and their respective outputs. The output for each input sequence will be the first note or chord that comes after the sequence of notes in the input sequence in our list of notes.

```
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)
```

In our code example, we have put the length of each sequence to be 100 notes/chords. This means that to predict the next note in the sequence the network has the previous 100 notes to help make the prediction. I highly recommend training the network using different sequence lengths to see the impact different sequence lengths can have on the music generated by the network.

The final step in preparing the data for the network is to normalise the input and one-hot encode the output.

Model

Finally we get to designing the model architecture. In our model we use four different types of layers:

LSTM layers is a Recurrent Neural Net layer that takes a sequence as an input and can return either sequences (`return_sequences=True`) or a matrix.

Dropout layers are a regularisation technique that consists of setting a fraction of input units to 0 at each update during the training to prevent overfitting. The fraction is determined by the parameter used with the layer.

Dense layers or **fully connected layers** is a fully connected neural network layer where each input node is connected to each output node.

The Activation layer determines what activation function our neural network will use to calculate the output of a node.

```
model = Sequential()
model.add(LSTM(
    256,
    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(256))
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')
```

Now that we have some information about the different layers we will be using it is time to add them to the network model.

For each LSTM, Dense, and Activation layer the first parameter is how many nodes the layer should have. For the Dropout layer the first parameter is the fraction of input units that should be dropped during training.

For the first layer we have to provide a unique parameter called *input_shape*. The purpose of the parameter is to inform the network of the shape of the data it will be training.

The last layer should always contain the same amount of nodes as the number different outputs our system has. This assures that the output of the network will map directly to our classes.

For this tutorial we will use a simple network consisting of three LSTM layers, three Dropout layers, two Dense layers and one activation layer. I would recommend playing around with the structure of the network to see if you can improve the quality of the predictions.

To calculate the loss for each iteration of the training we will be using categorical cross entropy since each of our outputs only belongs to a single class and we have more than two classes to work with. And to optimise our network we will use a RMSprop optimizer as it is usually a very good choice for recurrent neural networks.

```
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=callbacks_list)
```

Once we have determined the architecture of our network the time has come to start the training. The *model.fit()* function in Keras is used to train the network. The first parameter is the list of input sequences that we prepared earlier and the second is a list of their respective outputs. In our tutorial we are going to train the network for 200 epochs (iterations), with each batch that is propagated through the network containing 64 samples.

To make sure that we can stop the training at any point in time without losing all of our hard work, we will use model checkpoints. Model checkpoints provide us with a way to save the weights of the network nodes to a file after every epoch. This allows us to stop running the neural network once we are satisfied with the loss value without having to worry about losing the weights. Otherwise we would have to wait until the network has finished going through all 200 epochs before we could get the chance to save the weights to a file.

Generating Music

Now that we have finished training the network it is time to have some fun with the network we have spent hours training.

To be able to use the neural network to generate music you will have to put it into the same state as before. For simplicity we will reuse code from the training section to prepare the data and set up the network model in the same way as before. Except, that instead of training the network we load the weights that we saved during the training section into the model.

```
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')
```

Now we can use the trained model to start generating notes.

Since we have a full list of note sequences at our disposal we will pick a random index in the list as our starting point, this allows us to rerun the generation code without changing anything and get different results every time. However, If you wish to control the starting point simply replace the random function with a command line argument.

Here we also need to create a mapping function to decode the output of the network. This function will map from numerical data to categorical data (from integers to notes).

```
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)]
```

We chose to generate 500 notes using the network since that is roughly two minutes of music and gives the network plenty of space to create a melody. For each note that we want to generate we have to submit a sequence to the network. The first sequence we submit is the sequence of notes at the starting index. For every subsequent sequence that we use as input, we will remove the first note of the sequence and insert the output of the previous iteration at the end of the sequence as can be seen in Figure 2.

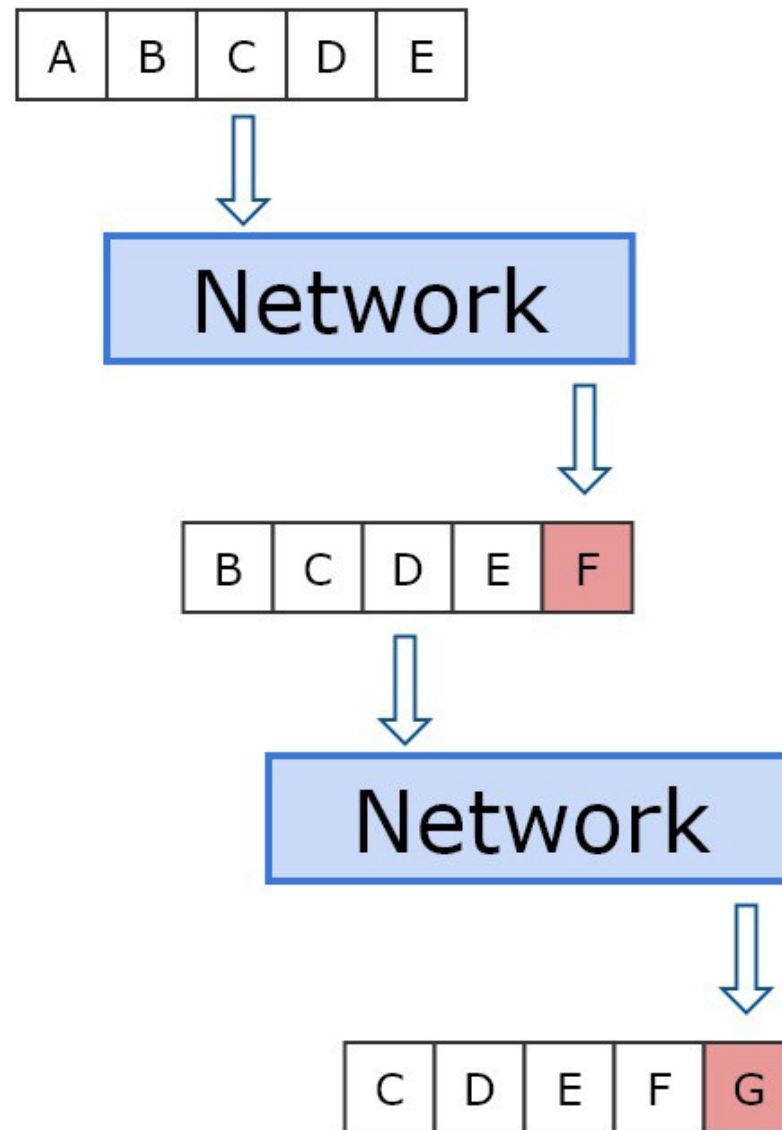


Figure 2: The first input sequence is ABCDE. The output we get from feeding that to the network is F. For the next iteration we remove A from the sequence and append F to it. Then we repeat

the process.

To determine the most likely prediction from the output from the network, we extract the index of the highest value. The value at index X in the output array correspond to the probability that X is the next note. Figure 3 helps explain this.

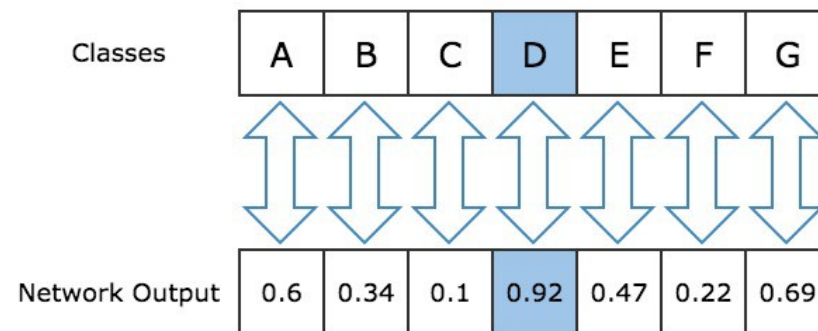


Figure 3: Here we see the mapping between the an output prediction from the network and classes. As we can see the highest probability is that the next value should be D, so we choose D as the most probable class.

Then we collect all the outputs from the network into a single array.

Now that we have all the encoded representations of the notes and chords in an array we can start decoding them and creating an array of Note and Chord objects.

First we have to determine whether the output we are decoding is a Note or a Chord.

If the pattern is a **Chord**, we have to split the string up into an array of notes. Then we loop through the string representation of each note and create a Note object for each of them. Then we can create a Chord object containing each of these notes.

If the pattern is a **Note**, we create a Note object using the string representation of the pitch contained in the pattern.

At the end of each iteration we increase the offset by 0.5 (as we decided in a previous section) and append the Note/Chord object created to a list.

```
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
```

Now that we have a list of Notes and Chords generated by the network we can create a Music21 Stream object using the list as a parameter. Then finally to create the MIDI file to contain the music generated by the network we use the *write* function in the Music21 toolkit to write the stream to a file.

```
midi_stream = stream.Stream(output_notes)

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

Results

Now it is time to marvel at the results. Figure 4 contains sheet music representation of music that was generated using the LSTM network. At a quick glance we can see that there is some structure to it. This is especially obvious in the third to last line on the second page.

People that are knowledgeable about music and can read musical notation can see that there are some weird notes strewn about the sheet. This is a result of the neural network not being able to create perfect melodies. With our current implementation there will always be some false notes and to be able to achieve better results we will need a bigger network.



Figure 4: An example of sheet music generated by the LSTM network

The results from this relatively shallow network are still really impressive as can be heard from the example music in Embed 1. For those interested, the sheet music in Figure 4 represents the musical notation of NeuralNet Music 5.



Embed 1: Examples generated by the network

Future Work

We have achieved remarkable results and beautiful melodies by using a simple LSTM network and 352 classes. However, there are areas that can be improved.

First, the implementation we have at the moment does not support varying duration of notes and different offsets between notes. To achieve that we could add more classes for each different duration and add rest classes that represent the rest period between notes.

To achieve satisfying results with more classes added we would also have to increase the depth of the LSTM network, which would require a significantly more powerful computer. It took the laptop I use at home approximately twenty hours to train the network as it is now.

Second, add beginnings and endings to pieces. As the network is now there is no distinction between pieces, that is to say the network does not know where one piece ends and another one begins. This would allow the network to generate a piece from start to finish instead of ending the generated piece abruptly as it does now.

Third, add a method to handle unknown notes. As it is now the network would enter a fail state if it encounters a note that it does not know. A possible method to solve that issue would be to find the note or chord that is most similar to the unknown note.

Finally, adding more instruments to the dataset. As it is now, the network only supports pieces that only have a single instrument. It would be interesting to see if it could be expanded to support a whole orchestra.

Conclusion

During this tutorial we have shown how to create a LSTM neural network to generate music. While the results may not be perfect, they are pretty impressive nonetheless and shows us that neural networks can create music and could potentially be used to help create more complex musical pieces.

[Check out the Github repository for the tutorial here](#)

