# イントロダクション

ニューラルネットワークは様々な面で私達の生活の向上に慣れています。それらは購入したい商品を推薦します。著者の癖をもとに文章を作成 することが出来たり、 [画像の芸術様式を変える](https://arxiv.org/pdf/1508.06576.pdf)ことさえ慣れています。最近ではかなりの数のニューラルネットワークで文章を作成する方法のチュートリアルがあります。しかし、ニューラルネットワークで作曲する方法のチュートリアルは全然ありません。この記事ではPythonのKerasライブラリを使って、リカレントニューラルネットワークで作曲する方法を練習しましょう。

せっかちな人のために, チュートリアルの最後にGithub リポジトリのリンクがあります。

背景

実装について詳しく入る前に、いくつかの用語を解説しましょう。

リカレントニューラルネットワーク (RNN)

リカレントニューラルネットワークは一連の情報を使って作るニューラルネットワークの一種です。これらがリカレント（再発）と呼ばれている理由は同じ単一関数の要素を順次実行するからです。 結果は前の計算に依存します。 一方で出力は伝統的なニューラルネットワークの前の計算と独立します。

このチュートリアルでは[**長短期記憶 (LSTM)**](http://colah.github.io/posts/2015-08-Understanding-LSTMs/) を使います。これはRNNの一種で、最急降下法経由で効率的に学習することが出来ます。ゲート機構を使います。LSTMは認識と長期的なパターンのエンコードが可能です。LSTM はネットワークが長期的に覚えないといけない情報がある問題を解決するのに非常に使いやすいです。そのまま作曲や文章作成と言ったケースに使えます。

Music21

[Music21](http://web.mit.edu/music21/) はコンピュータを使った音楽学に使うPythonのツールキットです。基本的な音楽理論を教えるのと、 作曲例と音楽を勉強する事が可能になります。このツールキットはmidiファイルの楽譜を取得することをシンプルなインターフェイスで提供します。さらに, 音符とコードのオブジェクトを作成し簡単にmidiファイルを作ることを可能に します。

このチュートリアルでは Music21を使用して、コンテンツを展開してデータセットにしたり、ニューラルネットワークの出力を楽譜に直します。

Keras

[Keras](https://keras.io/)は[Tensorflow](https://www.tensorflow.org/" \t "_blank)を簡略化した相互作用のある高度なニューラルネットワークAPI です。迅速な実験に重点を置き、開発されました。

このチュートリアルではKerasライブラリを使って、LSTMモデルを作ったり、訓練させたりします。一度モデルを訓練させた後、それを使用して楽譜を生成します。

トレーニング

この段落では、データの集め方、LSTMとモデルのアーキテクチャの準備について取り扱います。

データ

私達の[Githubリポジトリ](https://github.com/Skuldur/Classical-Piano-Composer" \t "_blank) では、ピアノ音楽を使っています、 多くの場合、ファイナルファンタジーのサウンドトラックの音楽を使っています。 ファイナルファンタジーの音楽を選んだのは、とても明確で美しいメロディーなのと、ファイルが多数存在するためです。しかしこれらのMIDIは私達の目的のために一つの楽器からなっています。

ニューラルネットワークを実装する最初のステップは、動いてるデータを見ることです。

下の抜粋はMusic21を用いて、midiファイルを読み込んだものです：

...  
<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>  
...

データは [Note](http://web.mit.edu/music21/doc/moduleReference/moduleNote.html#note)と[Chord](http://web.mit.edu/music21/doc/moduleReference/moduleChord.html" \t "_blank)という２つのオブジェクトタイプに分割されます。 Noteオブジェクトは音符に関する **pitch**, **octave**, そして **offset** という情報が入っています。

* **Pitch** は音の周波数、もしくは音の高低を参照します。 [A, B, C, D, E, F, G]の文字で表します。Aが最も高くG が最も低いです。
* [**Octave**](http://web.mst.edu/~kosbar/test/ff/fourier/notes_pitchnames.html)**はピアノで使うピッチを参照します**。
* **Offsetは曲の音符の位置を参照します**。

そしてChord オブジェクトは本質的に同時に演奏された音符のコンテナです。

今、作曲には次の音符やコードを正確に予測できるニューラルネットワークが必要だと言うことが分かりました。それが意味することは予測には遭遇する全ての音符やコードオブジェクトが配列になっているトレーニングセットが必要ということです。Githubのトレーニングセットを見ると、違う音符とコードの合計は352個です。それは沢山の可能な出力を処理するネットワークが必要そうに見えますが、LSTMネットワークは簡単に処理できます。

次に、音符をどこに置くかについて心配しないといけない事があり、音楽を聞いたことがある沢山の人々は気づいたように普通、音符の間隔は変化します。沢山の音符を短い時間で並べた後、 音符がない時間が続きます。

下の抜粋はMusic21を用いてmidi を読んだ図ですが、今回はオブジェクトの末尾に間隔の時間を追加しました。 これは音符とコードそれぞれの間隔を確認することが出来ます。

...  
<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  
...

今、見た抜粋と大抵のデータセットはmidiに0.5という共通の音符の間隔を持っています。したがって、出力可能なオフセットを無視することで、データとモデルを簡略化することが出来ます。ネットワークが作曲したメロディーにひどく影響を及ぼすことはないでしょう。 なので、このチュートリアルでは出力可能なリストは352のままで、間隔は無視します。

データの準備

今データを調べ、使いたい機能がLSTMに入出力する音符とコードと決定したので、次はネットワークのデータの準備をしましょう。

最初に下のコードにあるように、データをロードして、配列にします。

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))

色々なファイルをロードするために*converter.parse(file)*関数を使い、Music21ストリームオブジェクトにします。ストリームオブジェクトを使い、ファイル全ての音符とコードをリストにして取得します。とても重要な一部の音符は再生成して音程の文字列を使い、全ての音符オブジェクトの音程を文字列にして追加します。そして、全ての音符のidを1つの文字列でエンコードして、それをドットで区切って、全てのコードを追加します。これらのエンコードはネットワークが生成した正しい音符とコード出力時のデコードを簡単にすることが可能です。

全ての音符とコードを一連のリストに入れたので、ネットワークは入力を全て受けるシーケンスを組み立てることができます。

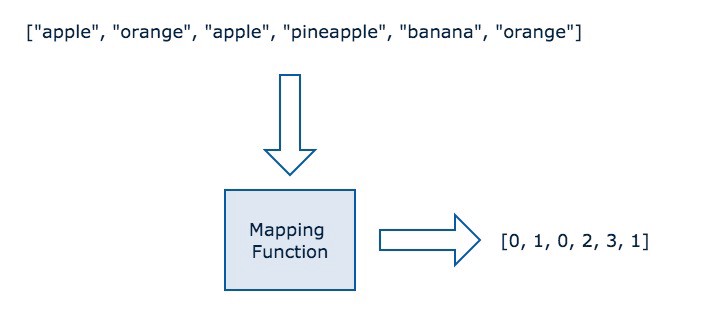


図1: カテゴリーデータから数値データに変更するとき、カテゴリーの位置を明確な値の整数インデックスに変換されます。例えば、 apple は最初の明確な値で0を代入し、orange は二番目なので1を代入、 pineapple は3番目なので2を代入、そして4番目と続きます。

最初に、文字列ベースのカテゴリーデータを整数ベースの数字データに変換する関数を作ります。これを行う理由は、ニューラルネットワークはカテゴリーデータより数字データのほうがずっと実行しやすいからです。例としてカテゴリーデータから数字データに変換できることは、図１で説明しています。

次に、それぞれの出力と入力するシーケンスを作らないといけません。出力は音符リストの各入力シーケンスの後の音符シーケンスに続く、最初の音符かコードになります。

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)

このコードの例は、各シーケンスの長さが１００の音符または、コードにしています。それが意味することは前の長さが100の音符がシーケンスの次の音符を予測するネットワークの予測を助けることです。違うシーケンスの長さを使ってネットワークを訓練させ、作曲ネットワークの確認を非常にお勧めします。

ネットワークのデータの準備は最後に、入力を正規化し、出力を[1-hotエンコード](https://machinelearningmastery.com/why-one-hot-encode-data-in-machine-learning/)します。

モデル

最終的に、モデル構造を設計します。モデルには４つの違ったタイプのレイヤーがあります：

**LSTM layersは** シーケンスを受け取り、シーケンス(return\_sequences=True)または、行列を返すリカレントニューラルネットレイヤーです。

**Dropout layers**は過剰適合を防ぐために、トレーニングの間は更新ごとに入力単位の割合を0にする設定からなる正規化手法です。割合はレイヤーを使ったパラメーターで決定されます。

**Dense layers**または、 **fully connected layers** は各入力ノードが各出力ノードと完全に繋がったニューラルネットワークレイヤーです。

**The Activation layer** ニューラルネットワークの出力ノードを計算する活性化関数を決定します。

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')

使用する異なるレイヤーについての情報が得られたところで、ネットワークモデルにそれらを追加します。

LSTM、Dense、そしてActivation layerの最初のパラメーターはレイヤーに必要なノードがいくつあるかです。Dropout layerの最初のパラメーターはトレーニング中に入力ユニットをどれだけ排除するかの割合です。

最初のレイヤーは *input\_shape* と呼ばれ、一意の値を提供しないといけません。パラメーターの目的はトレーニングするデータの形をネットワークに教えるためです。

最後のレイヤーは様々な出力とノードの数が常に含まれてないといけません。これはネットワークの出力がクラスに直接代入されているのを保証するためです。

このチュートリアルでは3つのLSTM layer、3つのDropout layer、2つのDense layer、そしてactivation layerで構成されるシンプルなネットワークを使います。予測の質を向上したいなら、ネットワークの構造をいじって、確認することをおすすめします。

トレーニングの各反復の損失を計算したいなら、 [多クラス交差エントロピー](https://rdipietro.github.io/friendly-intro-to-cross-entropy-loss/) を使います。なぜなら、各出力が一つのクラスのみ属し、2つ以上のクラスを処理するためです。そして、ネットワークを最適化したいときは、RMSpropオプティマイザがRNNでは使いやすく、とてもいい選択です。

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)

ネットワークの構造が決定したら、トレーニングを開始します。Keras に入っている*model.fit()*関数を使って、ネットワークを訓練します。 最初のパラメーターは、さっき用意した入力シーケンスのリストです。二個目のパラメーターはそれぞれの出力のリストです。チュートリアルでは、訓練のために200エポック（イテレーション）を用意しました。各バッチは64個のサンプルを含むネットワークに送られます。

確実に重労働を失わないためにいつでもトレーニングを損失なくストップできるようにモデルチェックポイントを使います。モデルチェックポイントは全てのエポックごとにネットワークノードの重みをファイルに保存する方法を提供します。これで重みがなくなることに心配しなくても損失値の値が満足なら、ニューラルネットワークの稼働を止めることが出来ます。そうしないと、重みをファイルに保存する機会を失い、全てのエポックが通過するまで待つ必要があります。

音楽作成

トレーニングが終わったら、何時間もトレーニングにかけたネットワークを楽しむ時間です。

ニューラルネットワークを使って作曲するには、以前の状態に戻す必要があります。簡単にするため、トレーニングセクションで使ったコードを再利用してデータを準備し、以前と同じようにネットワークモデルを構築します。ただし、ネットワークをトレーニングする代わりに、トレーニングセクションの間で保存した重みをネットワークに使います。

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')

これでトレーニング済みのモデルを使用して、音符の作成ができます。

自由に使える音符シーケンスがあるので、リストからランダムなインデックスを開始点にします。これにより、何も変更しなくて再実行しても、毎回異なった結果を出力することが出来ます。ただし、開始点を制御する場合、ランダム関数をコマンドライン引数にするだけです。

ここでは、ネットワークの出力をデコードするために、代入関数を作る必要があります。この関数は、数値データをカテゴリーデータにします（数値を楽譜に）。

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)]

ネットワークを使って500音の曲を作ることにしました。なぜなら約二分の曲であり、ネットワークに大量の音楽を作るためのスペースがあるからです。作成する音符ごとに、シーケンスをネットワークに送る必要があります。最初の送るシーケンスは開始インデックスの音符です。図2のように入力する後続のシーケンスごとに、最初の音符を削除して最後に前の反復を挿入します。

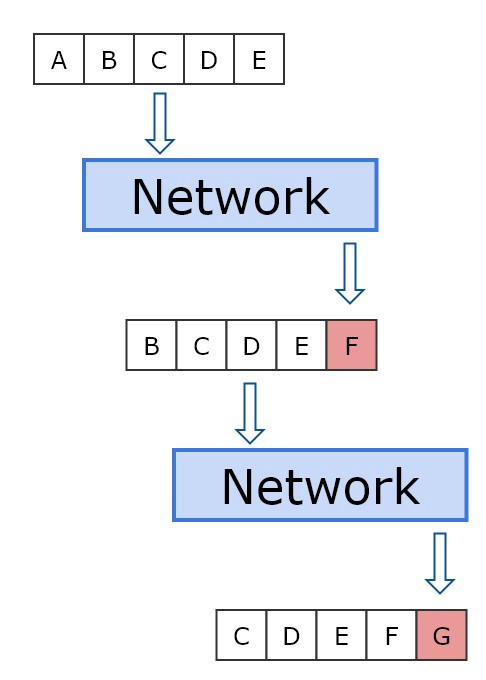


図2:最初の入力はABCDEです。それをネットワークに供給して得られる出力はFです。次の反復はシーケンスAを削除してシーケンスFを追加します。それを繰り返します。

ネットワークの出力から最も可能性が高い予測を決めるために、最も高い予測値のインデックスを抽出します。 出力した配列の数値のインデックスXが次の音符のXである確率に対応します。図3が説明を理解するのに役立つでしょう。

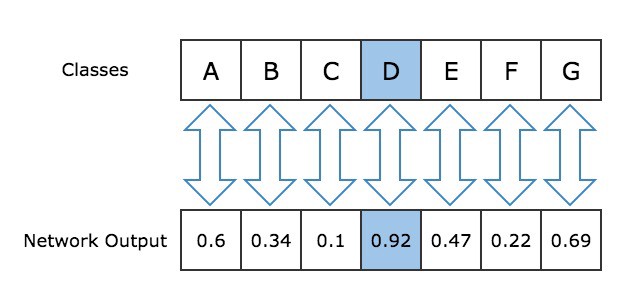


図3: ここでは、ネットワークからの出力予測とクラスの間の代入を確認します。ご覧の通り、予測値が最も高いのがDである可能性が高いため、最も可能性が高いクラスとしてDを選びます。

次に、すべてのネットワークからの出力を一つの配列に集めます。

音符とコードの全てがエンコードされた配列が出来たので、デコードと、音符とコードオブジェクトの配列を作りたいと思います。

また、デコードするのが音符かコードか識別する必要があります。

もしパターンが **Chord**なら、文字列を音符の配列に分割する必要があります。次に、各音符の文字列表現をループして、それぞれの音符オブジェクトを作成します。これらの音符を含む、Codeオブジェクトが生成できます。

もしパターンが **Note**なら、 音程の文字列表現を使い、Noteオブジェクトを作成します。

最後に間隔を0.5（前のセクションで決定したように）に増やし、 Note/Chordオブジェクトをリストに追加します。

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

ネットワークによって作られた音符とコードのリストができたので、リストのパラメーターでMusic21 Streamオブジェクトを作ることが出来ます。最後に、ネットワークで生成された音楽を含むMidiを作るために、Music21のツールキットである*write* 関数を使い、ストリームをファイルに変えます。

midi\_stream = stream.Stream(output\_notes)midi\_stream.write('midi', fp='test\_output.mid')

結果

結果に驚くときが来ました。 図4はLSTMを利用して作成された音楽の楽譜です。ひと目でいくつかの構造があることが分かります。特に、2ページの最後から3番めの行で顕著です。

音楽をかじっていて、楽譜を読める人なら楽譜に奇妙な音符が散らばっていることに気づくでしょう。これはニューラルネットワークが完璧な旋律を作ることが出来ない結果です。今の実装では、沢山の間違った音符があり、より良い結果を出すにはもっと大きなネットワークが必要です。



図4: LSTM によって作成された音楽の楽譜の例

この比較的浅いネットワークの結果から、1の音楽（注：原文ではSoundCloudが埋め込まれている）のように、とても印象的です。興味がある方のために、図4の楽譜は5の音楽です。

将来的にしたいこと

シンプルなLSTMと352ものクラスを利用することで、驚くべき結果と美しいメロディーを実現しました。しかし、改善点はまだあると思います。

まず、現時点では音符の長さの変化や違う音符の間隔は実装されていません。実装するためには、異なる間隔ごとのクラスと、音符感の休符を表す休符クラスを追加することが出来ます。

より多くのクラスを追加して満足の行く結果を得るためには、LSTMをかなり深くする必要があります。それにはより強力なパソコンが必要になります。私が使っているノートパソコンでは今のネットワークをトレーニングするのに約20時間かかりました。

2つ目に、始まりと終わりに断片を追加します。ネットワークは今、断片間の区別がないため、ある断片がどこで終わり、別の断片がどこから始まるか認識していません。これによりネットワークは断片を突然終了させることなく、最初から最後までピースを生成できます。

3つ目に、不明な音符を処理するメソッドを追加します。現在のネットワークのように不明な音符に遭遇すると、障害が発生します。その問題を解決する方法として、未知の音符に似ている音符または、コードを見つけることです。

最後に、データセットに沢山の楽器を追加します。現在、ネットワークは単一の楽器しか持たないmidiしか対応していません。オーケストラ全体を対応できるように拡張できるかどうかを確認するのは興味深いことです。

結論

このチュートリアルではLSTMで作曲するAIを作る方法を見せました。結果は完璧ではないかもしれませんが、それでもかなり印象的な音楽をニューラルネットワークは作ることが出来、将来的には複雑な楽曲を作るのに役立つと思います。