Improvise a Jazz Solo with an LSTM Network - v3

May 21, 2019

1 Improvise a Jazz Solo with an LSTM Network

Welcome to your final programming assignment of this week! In this notebook, you will implement a model that uses an LSTM to generate music. You will even be able to listen to your own music at the end of the assignment.

You will learn to: - Apply an LSTM to music generation. - Generate your own jazz music with deep learning.

Please run the following cell to load all the packages required in this assignment. This may take a few minutes.

In [ ]:
```
from __future__ import print_function
import IPython
import sys
from music21 import *
import numpy as np
from grammar import *
from qa import *
from preprocess import *
from music_utils import *
from data_utils import *
from keras.models import load_model, Model
from keras.layers import Dense, Activation, Dropout, Input, LSTM, Reshape,
from keras.initializers import glorot_uniform
from keras.utils import to_categorical
from keras.optimizers import Adam
from keras import backend as K
```

```
Using TensorFlow backend.
```

1.1 1 - Problem statement

You would like to create a jazz music piece specially for a friend's birthday. However, you don't know any instruments or music composition. Fortunately, you know deep learning and will solve this problem using an LSTM netwok.

You will train a network to generate novel jazz solos in a style representative of a body of performed work.

1.1.1 1.1 - Dataset

You will train your algorithm on a corpus of Jazz music. Run the cell below to listen to a snippet of the audio from the training set:

```
In [3]: IPython.display.Audio('./data/30s_seq.mp3')
```

```
Out[3]: <IPython.lib.display.Audio object>
```

We have taken care of the preprocessing of the musical data to render it in terms of musical "values." You can informally think of each "value" as a note, which comprises a pitch and a duration. For example, if you press down a specific piano key for 0.5 seconds, then you have just played a note. In music theory, a "value" is actually more complicated than this–specifically, it also captures the information needed to play multiple notes at the same time. For example, when playing a music piece, you might press down two piano keys at the same time (playng multiple notes at the same time generates what's called a "chord"). But we don't need to worry about the details of music theory for this assignment. For the purpose of this assignment, all you need to know is that we will obtain a dataset of values, and will learn an RNN model to generate sequences of values.

Our music generation system will use 78 unique values. Run the following code to load the raw music data and preprocess it into values. This might take a few minutes.

```
In [4]: X, Y, n_values, indices_values = load_music_utils()
        print('shape of X:', X.shape)
        print('number of training examples:', X.shape[0])
        print('Tx (length of sequence):', X.shape[1])
        print('total # of unique values:', n_values)
        print('Shape of Y:', Y.shape)
```

```
shape of X: (60, 30, 78)
number of training examples: 60
Tx (length of sequence): 30
total # of unique values: 78
Shape of Y: (30, 60, 78)
```

You have just loaded the following:

- `X`: This is an (m, T_x, 78) dimensional array. We have m training examples, each of which is a snippet of $T_x = 30$ musical values. At each time step, the input is one of 78 different possible values, represented as a one-hot vector. Thus for example, X[i,t,:] is a one-hot vector representating the value of the i-th example at time t.
- `Y`: This is essentially the same as `X`, but shifted one step to the left (to the past). Similar to the dinosaurus assignment, we're interested in the network using the previous values to predict the next value, so our sequence model will try to predict $y^{\langle t \rangle}$ given $x^{\langle 1 \rangle}, \ldots, x^{\langle t \rangle}$. However, the data in `Y` is reordered to be dimension $(T_y, m, 78)$, where $T_y = T_x$. This format makes it more convenient to feed to the LSTM later.
- `n_values`: The number of unique values in this dataset. This should be 78.
- `indices_values`: python dictionary mapping from 0-77 to musical values.

1.1.2 1.2 - Overview of our model

Here is the architecture of the model we will use. This is similar to the Dinosaurus model you had used in the previous notebook, except that in you will be implementing it in Keras. The architecture is as follows:

We will be training the model on random snippets of 30 values taken from a much longer piece of music. Thus, we won't bother to set the first input $x^{\langle 1 \rangle} = \vec{0}$, which we had done previously to denote the start of a dinosaur name, since now most of these snippets of audio start somewhere in the middle of a piece of music. We are setting each of the snippts to have the same length $T_x = 30$ to make vectorization easier.

1.2 2 - Building the model

In this part you will build and train a model that will learn musical patterns. To do so, you will need to build a model that takes in X of shape $(m, T_x, 78)$ and Y of shape $(T_y, m, 78)$. We will use an LSTM with 64 dimensional hidden states. Lets set `n_a = 64`.

```
In [5]: n_a = 64
```

Here's how you can create a Keras model with multiple inputs and outputs. If you're building an RNN where even at test time entire input sequence $x^{\langle 1 \rangle}, x^{\langle 2 \rangle}, \ldots, x^{\langle T_x \rangle}$ were *given in advance*, for example if the inputs were words and the output was a label, then Keras has simple built-in functions to build the model. However, for sequence generation, at test time we don't know all the values of $x^{\langle t \rangle}$ in advance; instead we generate them one at a time using $x^{\langle t \rangle} = y^{\langle t-1 \rangle}$. So the code will be a bit more complicated, and you'll need to implement your own for-loop to iterate over the different time steps.

The function `djmodel()` will call the LSTM layer T_x times using a for-loop, and it is important that all T_x copies have the same weights. I.e., it should not re-initiaiize the weights every time—the T_x steps should have shared weights. The key steps for implementing layers with shareable weights in Keras are: 1. Define the layer objects (we will use global variables for this). 2. Call these objects when propagating the input.

We have defined the layers objects you need as global variables. Please run the next cell to create them. Please check the Keras documentation to make sure you understand what these layers are: Reshape(), LSTM(), Dense().

```
In [6]: reshapor = Reshape((1, 78))                        # Used in Step 2.B of dj
        LSTM_cell = LSTM(n_a, return_state = True)         # Used in Step 2.C
        densor = Dense(n_values, activation='softmax')     # Used in Step 2.D
```

Each of `reshapor`, `LSTM_cell` and `densor` are now layer objects, and you can use them to implement `djmodel()`. In order to propagate a Keras tensor object X through one of these layers, use `layer_object(X)` (or `layer_object([X,Y])` if it requires multiple inputs.). For example, `reshapor(X)` will propagate X through the `Reshape((1,78))` layer defined above.

Exercise: Implement `djmodel()`. You will need to carry out 2 steps:

1. Create an empty list "outputs" to save the outputs of the LSTM Cell at every time step.
2. Loop for $t \in 1, \ldots, T_x$:

   A. Select the "t"th time-step vector from X. The shape of this selection should be (78,). To do so, create a custom Lambda layer in Keras by using this line of code:

```
x = Lambda(lambda x: X[:,t,:])(X)
```

Look over the Keras documentation to figure out what this does. It is creating a "temporary" or "unnamed" function (that's what Lambda functions are) that extracts out the appropriate one-hot vector, and making this function a Keras `Layer` object to apply to `X`.

B. Reshape x to be (1,78). You may find the `reshapor()` layer (defined below) helpful.

C. Run x through one step of LSTM_cell. Remember to initialize the LSTM_cell with the previous step's hidden state a and cell state c. Use the following formatting:

```
a, _, c = LSTM_cell(input_x, initial_state=[previous hidden state, previous cel
```

D. Propagate the LSTM's output activation value through a dense+softmax layer using `densor`.

E. Append the predicted value to the list of "outputs"

In [9]:
```
# GRADED FUNCTION: djmodel

def djmodel(Tx, n_a, n_values):
    """
    Implement the model

    Arguments:
    Tx -- length of the sequence in a corpus
    n_a -- the number of activations used in our model
    n_values -- number of unique values in the music data

    Returns:
    model -- a keras model with the
    """

    # Define the input of your model with a shape
    X = Input(shape=(Tx, n_values))

    # Define s0, initial hidden state for the decoder LSTM
    a0 = Input(shape=(n_a,), name='a0')
    c0 = Input(shape=(n_a,), name='c0')
    a = a0
    c = c0

    ### START CODE HERE ###
    # Step 1: Create empty list to append the outputs while you iterate (≈
    outputs = []

    # Step 2: Loop
    for t in range(Tx):

        # Step 2.A: select the "t"th time step vector from X.
```

```
        x = Lambda(lambda x: X[:,t,:])(X)
        # Step 2.B: Use reshapor to reshape x to be (1, n_values) (≈1 line
        x = reshapor(x)
        # Step 2.C: Perform one step of the LSTM_cell
        a, _, c = LSTM_cell(x, initial_state=[a, c])
        # Step 2.D: Apply densor to the hidden state output of LSTM_Cell
        out = densor(a)
        # Step 2.E: add the output to "outputs"
        outputs.append(out)

    # Step 3: Create model instance
    model = Model(inputs=[X, a0, c0], outputs=outputs)

    ### END CODE HERE ###

    return model
```

Run the following cell to define your model. We will use `Tx=30`, `n_a=64` (the dimension of the LSTM activations), and `n_values=78`. This cell may take a few seconds to run.

```
In [10]: model = djmodel(Tx = 30 , n_a = 64, n_values = 78)
```

You now need to compile your model to be trained. We will Adam and a categorical cross-entropy loss.

```
In [11]: opt = Adam(lr=0.01, beta_1=0.9, beta_2=0.999, decay=0.01)

         model.compile(optimizer=opt, loss='categorical_crossentropy', metrics=['ac
```

Finally, lets initialize `a0` and `c0` for the LSTM's initial state to be zero.

```
In [12]: m = 60
         a0 = np.zeros((m, n_a))
         c0 = np.zeros((m, n_a))
```

Lets now fit the model! We will turn `Y` to a list before doing so, since the cost function expects `Y` to be provided in this format (one list item per time-step). So `list(Y)` is a list with 30 items, where each of the list items is of shape (60,78). Lets train for 100 epochs. This will take a few minutes.

```
In [13]: model.fit([X, a0, c0], list(Y), epochs=100)
```

```
Epoch 1/100
60/60 [==============================] - 3s - loss: 125.8634 - dense_1_loss_1: 4.35
Epoch 2/100
60/60 [==============================] - 0s - loss: 122.8028 - dense_1_loss_1: 4.33
Epoch 3/100
60/60 [==============================] - 0s - loss: 116.5398 - dense_1_loss_1: 4.30
Epoch 4/100
```

```
60/60 [==============================] - 0s - loss: 113.3836 - dense_1_loss_1: 4.28
Epoch 5/100
60/60 [==============================] - 0s - loss: 110.3218 - dense_1_loss_1: 4.27
Epoch 6/100
60/60 [==============================] - 0s - loss: 109.0249 - dense_1_loss_1: 4.25
Epoch 7/100
60/60 [==============================] - 0s - loss: 105.3585 - dense_1_loss_1: 4.24
Epoch 8/100
60/60 [==============================] - 0s - loss: 102.4433 - dense_1_loss_1: 4.23
Epoch 9/100
60/60 [==============================] - 0s - loss: 98.6752 - dense_1_loss_1: 4.218
Epoch 10/100
60/60 [==============================] - 0s - loss: 95.1143 - dense_1_loss_1: 4.205
Epoch 11/100
60/60 [==============================] - 0s - loss: 91.0622 - dense_1_loss_1: 4.194
Epoch 12/100
60/60 [==============================] - 0s - loss: 86.6006 - dense_1_loss_1: 4.183
Epoch 13/100
60/60 [==============================] - 0s - loss: 82.6505 - dense_1_loss_1: 4.172
Epoch 14/100
60/60 [==============================] - 0s - loss: 78.9350 - dense_1_loss_1: 4.165
Epoch 15/100
60/60 [==============================] - 0s - loss: 74.8533 - dense_1_loss_1: 4.157
Epoch 16/100
60/60 [==============================] - 0s - loss: 71.4829 - dense_1_loss_1: 4.149
Epoch 17/100
60/60 [==============================] - 0s - loss: 67.9508 - dense_1_loss_1: 4.141
Epoch 18/100
60/60 [==============================] - 0s - loss: 64.3750 - dense_1_loss_1: 4.132
Epoch 19/100
60/60 [==============================] - 0s - loss: 61.7160 - dense_1_loss_1: 4.123
Epoch 20/100
60/60 [==============================] - 0s - loss: 58.3391 - dense_1_loss_1: 4.114
Epoch 21/100
60/60 [==============================] - 0s - loss: 55.4512 - dense_1_loss_1: 4.106
Epoch 22/100
60/60 [==============================] - 0s - loss: 52.2643 - dense_1_loss_1: 4.097
Epoch 23/100
60/60 [==============================] - 0s - loss: 49.3293 - dense_1_loss_1: 4.089
Epoch 24/100
60/60 [==============================] - 0s - loss: 46.5327 - dense_1_loss_1: 4.082
Epoch 25/100
60/60 [==============================] - 0s - loss: 43.7493 - dense_1_loss_1: 4.073
Epoch 26/100
60/60 [==============================] - 0s - loss: 41.2463 - dense_1_loss_1: 4.065
Epoch 27/100
60/60 [==============================] - 0s - loss: 38.7064 - dense_1_loss_1: 4.057
Epoch 28/100
```

```
60/60 [==============================] - 0s - loss: 36.4020 - dense_1_loss_1: 4.050
Epoch 29/100
60/60 [==============================] - 0s - loss: 34.1644 - dense_1_loss_1: 4.043
Epoch 30/100
60/60 [==============================] - 0s - loss: 31.9531 - dense_1_loss_1: 4.036
Epoch 31/100
60/60 [==============================] - 0s - loss: 29.9833 - dense_1_loss_1: 4.029
Epoch 32/100
60/60 [==============================] - 0s - loss: 28.0779 - dense_1_loss_1: 4.022
Epoch 33/100
60/60 [==============================] - 0s - loss: 26.2915 - dense_1_loss_1: 4.016
Epoch 34/100
60/60 [==============================] - 0s - loss: 24.5710 - dense_1_loss_1: 4.008
Epoch 35/100
60/60 [==============================] - 0s - loss: 23.0551 - dense_1_loss_1: 4.002
Epoch 36/100
60/60 [==============================] - 0s - loss: 21.6218 - dense_1_loss_1: 3.996
Epoch 37/100
60/60 [==============================] - 0s - loss: 20.3248 - dense_1_loss_1: 3.990
Epoch 38/100
60/60 [==============================] - 0s - loss: 19.1017 - dense_1_loss_1: 3.984
Epoch 39/100
60/60 [==============================] - 0s - loss: 18.0207 - dense_1_loss_1: 3.979
Epoch 40/100
60/60 [==============================] - 0s - loss: 17.0298 - dense_1_loss_1: 3.973
Epoch 41/100
60/60 [==============================] - 0s - loss: 16.1186 - dense_1_loss_1: 3.968
Epoch 42/100
60/60 [==============================] - 0s - loss: 15.2889 - dense_1_loss_1: 3.962
Epoch 43/100
60/60 [==============================] - 0s - loss: 14.5539 - dense_1_loss_1: 3.958
Epoch 44/100
60/60 [==============================] - 0s - loss: 13.8774 - dense_1_loss_1: 3.953
Epoch 45/100
60/60 [==============================] - 0s - loss: 13.2549 - dense_1_loss_1: 3.949
Epoch 46/100
60/60 [==============================] - 0s - loss: 12.7003 - dense_1_loss_1: 3.943
Epoch 47/100
60/60 [==============================] - 0s - loss: 12.2025 - dense_1_loss_1: 3.939
Epoch 48/100
60/60 [==============================] - 0s - loss: 11.7328 - dense_1_loss_1: 3.935
Epoch 49/100
60/60 [==============================] - 0s - loss: 11.3260 - dense_1_loss_1: 3.931
Epoch 50/100
60/60 [==============================] - 0s - loss: 10.9416 - dense_1_loss_1: 3.926
Epoch 51/100
60/60 [==============================] - 0s - loss: 10.5942 - dense_1_loss_1: 3.922
Epoch 52/100
```

```
60/60 [==============================] - 0s - loss: 10.2788 - dense_1_loss_1: 3.918
Epoch 53/100
60/60 [==============================] - 0s - loss: 9.9941 - dense_1_loss_1: 3.9141
Epoch 54/100
60/60 [==============================] - 0s - loss: 9.7281 - dense_1_loss_1: 3.9102
Epoch 55/100
60/60 [==============================] - 0s - loss: 9.4879 - dense_1_loss_1: 3.9061
Epoch 56/100
60/60 [==============================] - 0s - loss: 9.2565 - dense_1_loss_1: 3.9026
Epoch 57/100
60/60 [==============================] - 0s - loss: 9.0535 - dense_1_loss_1: 3.8988
Epoch 58/100
60/60 [==============================] - 0s - loss: 8.8615 - dense_1_loss_1: 3.8949
Epoch 59/100
60/60 [==============================] - 0s - loss: 8.6813 - dense_1_loss_1: 3.8916
Epoch 60/100
60/60 [==============================] - 0s - loss: 8.5163 - dense_1_loss_1: 3.8878
Epoch 61/100
60/60 [==============================] - 0s - loss: 8.3623 - dense_1_loss_1: 3.8840
Epoch 62/100
60/60 [==============================] - 0s - loss: 8.2229 - dense_1_loss_1: 3.8804
Epoch 63/100
60/60 [==============================] - 0s - loss: 8.0867 - dense_1_loss_1: 3.8771
Epoch 64/100
60/60 [==============================] - 0s - loss: 7.9590 - dense_1_loss_1: 3.8733
Epoch 65/100
60/60 [==============================] - 0s - loss: 7.8436 - dense_1_loss_1: 3.8701
Epoch 66/100
60/60 [==============================] - 0s - loss: 7.7313 - dense_1_loss_1: 3.8666
Epoch 67/100
60/60 [==============================] - 0s - loss: 7.6292 - dense_1_loss_1: 3.8630
Epoch 68/100
60/60 [==============================] - 0s - loss: 7.5309 - dense_1_loss_1: 3.8597
Epoch 69/100
60/60 [==============================] - 0s - loss: 7.4367 - dense_1_loss_1: 3.8563
Epoch 70/100
60/60 [==============================] - 0s - loss: 7.3490 - dense_1_loss_1: 3.8529
Epoch 71/100
60/60 [==============================] - 0s - loss: 7.2654 - dense_1_loss_1: 3.8495
Epoch 72/100
60/60 [==============================] - 0s - loss: 7.1852 - dense_1_loss_1: 3.8465
Epoch 73/100
60/60 [==============================] - 0s - loss: 7.1116 - dense_1_loss_1: 3.8429
Epoch 74/100
60/60 [==============================] - 0s - loss: 7.0377 - dense_1_loss_1: 3.8396
Epoch 75/100
60/60 [==============================] - 0s - loss: 6.9702 - dense_1_loss_1: 3.8366
Epoch 76/100
```

```
60/60 [==============================] - 0s - loss: 6.9039 - dense_1_loss_1: 3.8333
Epoch 77/100
60/60 [==============================] - 0s - loss: 6.8433 - dense_1_loss_1: 3.8302
Epoch 78/100
60/60 [==============================] - 0s - loss: 6.7832 - dense_1_loss_1: 3.8268
Epoch 79/100
60/60 [==============================] - 0s - loss: 6.7233 - dense_1_loss_1: 3.8240
Epoch 80/100
60/60 [==============================] - 0s - loss: 6.6695 - dense_1_loss_1: 3.8207
Epoch 81/100
60/60 [==============================] - 0s - loss: 6.6152 - dense_1_loss_1: 3.8177
Epoch 82/100
60/60 [==============================] - 0s - loss: 6.5658 - dense_1_loss_1: 3.8145
Epoch 83/100
60/60 [==============================] - 0s - loss: 6.5143 - dense_1_loss_1: 3.8115
Epoch 84/100
60/60 [==============================] - 0s - loss: 6.4672 - dense_1_loss_1: 3.8085
Epoch 85/100
60/60 [==============================] - 0s - loss: 6.4213 - dense_1_loss_1: 3.8054
Epoch 86/100
60/60 [==============================] - 0s - loss: 6.3776 - dense_1_loss_1: 3.8026
Epoch 87/100
60/60 [==============================] - 0s - loss: 6.3358 - dense_1_loss_1: 3.7994
Epoch 88/100
60/60 [==============================] - 0s - loss: 6.2938 - dense_1_loss_1: 3.7964
Epoch 89/100
60/60 [==============================] - 0s - loss: 6.2532 - dense_1_loss_1: 3.7934
Epoch 90/100
60/60 [==============================] - 0s - loss: 6.2153 - dense_1_loss_1: 3.7905
Epoch 91/100
60/60 [==============================] - 0s - loss: 6.1783 - dense_1_loss_1: 3.7878
Epoch 92/100
60/60 [==============================] - 0s - loss: 6.1409 - dense_1_loss_1: 3.7848
Epoch 93/100
60/60 [==============================] - 0s - loss: 6.1065 - dense_1_loss_1: 3.7818
Epoch 94/100
60/60 [==============================] - 0s - loss: 6.0723 - dense_1_loss_1: 3.7793
Epoch 95/100
60/60 [==============================] - 0s - loss: 6.0393 - dense_1_loss_1: 3.7763
Epoch 96/100
60/60 [==============================] - 0s - loss: 6.0071 - dense_1_loss_1: 3.7735
Epoch 97/100
60/60 [==============================] - 0s - loss: 5.9751 - dense_1_loss_1: 3.7708
Epoch 98/100
60/60 [==============================] - 0s - loss: 5.9461 - dense_1_loss_1: 3.7679
Epoch 99/100
60/60 [==============================] - 0s - loss: 5.9166 - dense_1_loss_1: 3.7650
Epoch 100/100
```

```
60/60 [==============================] - 0s - loss: 5.8878 - dense_1_loss_1: 3.7625
```

```
Out[13]: <keras.callbacks.History at 0x7f8e6b495c88>
```

You should see the model loss going down. Now that you have trained a model, lets go on the the final section to implement an inference algorithm, and generate some music!

1.3 3 - Generating music

You now have a trained model which has learned the patterns of the jazz soloist. Lets now use this model to synthesize new music.

3.1 - Predicting & Sampling At each step of sampling, you will take as input the activation `a` and cell state `c` from the previous state of the LSTM, forward propagate by one step, and get a new output activation as well as cell state. The new activation `a` can then be used to generate the output, using `densor` as before.

To start off the model, we will initialize `x0` as well as the LSTM activation and and cell value `a0` and `c0` to be zeros.

Exercise: Implement the function below to sample a sequence of musical values. Here are some of the key steps you'll need to implement inside the for-loop that generates the T_y output characters:

Step 2.A: Use `LSTM_Cell`, which inputs the previous step's `c` and `a` to generate the current step's `c` and `a`.

Step 2.B: Use `densor` (defined previously) to compute a softmax on `a` to get the output for the current step.

Step 2.C: Save the output you have just generated by appending it to `outputs`.

Step 2.D: Sample x to the be "out"'s one-hot version (the prediction) so that you can pass it to the next LSTM's step. We have already provided this line of code, which uses a Lambda function.

```
x = Lambda(one_hot)(out)
```

[Minor technical note: Rather than sampling a value at random according to the probabilities in `out`, this line of code actually chooses the single most likely note at each step using an argmax.]

```
In [14]: # GRADED FUNCTION: music_inference_model

         def music_inference_model(LSTM_cell, densor, n_values = 78, n_a = 64, Ty =
             """
             Uses the trained "LSTM_cell" and "densor" from model() to generate a s

             Arguments:
             LSTM_cell -- the trained "LSTM_cell" from model(), Keras layer object
             densor -- the trained "densor" from model(), Keras layer object
             n_values -- integer, umber of unique values
             n_a -- number of units in the LSTM_cell
             Ty -- integer, number of time steps to generate
```

```
    Returns:
    inference_model -- Keras model instance
    """

    # Define the input of your model with a shape
    x0 = Input(shape=(1, n_values))

    # Define s0, initial hidden state for the decoder LSTM
    a0 = Input(shape=(n_a,), name='a0')
    c0 = Input(shape=(n_a,), name='c0')
    a = a0
    c = c0
    x = x0

    ### START CODE HERE ###
    # Step 1: Create an empty list of "outputs" to later store your predic
    outputs = []

    # Step 2: Loop over Ty and generate a value at every time step
    for t in range(Ty):

        # Step 2.A: Perform one step of LSTM_cell (≈1 line)
        a, _, c = LSTM_cell(x, initial_state=[a, c])

        # Step 2.B: Apply Dense layer to the hidden state output of the LS
        out = densor(a)

        # Step 2.C: Append the prediction "out" to "outputs". out.shape =
        outputs.append(out)

        # Step 2.D: Select the next value according to "out", and set "x"
        #           selected value, which will be passed as the input to l
        #           the line of code you need to do this.
        x = Lambda(one_hot)(out)

    # Step 3: Create model instance with the correct "inputs" and "outputs
    inference_model = Model(inputs=[x0, a0, c0], outputs=outputs)

    ### END CODE HERE ###

    return inference_model
```

Run the cell below to define your inference model. This model is hard coded to generate 50 values.

```
In [27]: inference_model = music_inference_model(LSTM_cell, densor, n_values = 78,
```

Finally, this creates the zero-valued vectors you will use to initialize `x` and the LSTM state variables `a` and `c`.

```
In [28]: x_initializer = np.zeros((1, 1, 78))
         a_initializer = np.zeros((1, n_a))
         c_initializer = np.zeros((1, n_a))
```

Exercise: Implement `predict_and_sample()`. This function takes many arguments including the inputs [x_initializer, a_initializer, c_initializer]. In order to predict the output corresponding to this input, you will need to carry-out 3 steps: 1. Use your inference model to predict an output given your set of inputs. The output `pred` should be a list of length T_y where each element is a numpy-array of shape (1, n_values). 2. Convert `pred` into a numpy array of T_y indices. Each index corresponds is computed by taking the `argmax` of an element of the `pred` list. Hint. 3. Convert the indices into their one-hot vector representations. Hint.

```
In [29]: # GRADED FUNCTION: predict_and_sample

         def predict_and_sample(inference_model, x_initializer = x_initializer, a_i
                                c_initializer = c_initializer):
             """
             Predicts the next value of values using the inference model.

             Arguments:
             inference_model -- Keras model instance for inference time
             x_initializer -- numpy array of shape (1, 1, 78), one-hot vector initi
             a_initializer -- numpy array of shape (1, n_a), initializing the hidde
             c_initializer -- numpy array of shape (1, n_a), initializing the cell

             Returns:
             results -- numpy-array of shape (Ty, 78), matrix of one-hot vectors re
             indices -- numpy-array of shape (Ty, 1), matrix of indices representin
             """

             ### START CODE HERE ###
             # Step 1: Use your inference model to predict an output sequence given
             pred = inference_model.predict([x_initializer, a_initializer, c_initia
             # Step 2: Convert "pred" into an np.array() of indices with the maximu
             indices = np.argmax(pred, axis = -1)
             # Step 3: Convert indices to one-hot vectors, the shape of the results
             results = to_categorical(indices)
             ### END CODE HERE ###

             return results, indices
```

```
In [30]: results, indices = predict_and_sample(inference_model, x_initializer, a_in
         print("np.argmax(results[12]) =", np.argmax(results[12]))
         print("np.argmax(results[17]) =", np.argmax(results[17]))
         print("list(indices[12:18]) =", list(indices[12:18]))
```

```
np.argmax(results[12]) = 49
np.argmax(results[17]) = 12
```

```
list(indices[12:18]) = [array([49]), array([20]), array([63]), array([15]), array([
```

Expected Output: Your results may differ because Keras' results are not completely predictable. However, if you have trained your LSTM_cell with model.fit() for exactly 100 epochs as described above, you should very likely observe a sequence of indices that are not all identical. Moreover, you should observe that: np.argmax(results[12]) is the first element of list(indices[12:18]) and np.argmax(results[17]) is the last element of list(indices[12:18]).

np.argmax(results[12]) =
1
np.argmax(results[12]) =
42
list(indices[12:18]) =
[array([1]), array([42]), array([54]), array([17]), array([1]), array([42])]

3.3 - Generate music Finally, you are ready to generate music. Your RNN generates a sequence of values. The following code generates music by first calling your `predict_and_sample()` function. These values are then post-processed into musical chords (meaning that multiple values or notes can be played at the same time).

Most computational music algorithms use some post-processing because it is difficult to generate music that sounds good without such post-processing. The post-processing does things such as clean up the generated audio by making sure the same sound is not repeated too many times, that two successive notes are not too far from each other in pitch, and so on. One could argue that a lot of these post-processing steps are hacks; also, a lot the music generation literature has also focused on hand-crafting post-processors, and a lot of the output quality depends on the quality of the post-processing and not just the quality of the RNN. But this post-processing does make a huge difference, so lets use it in our implementation as well.

Lets make some music!

Run the following cell to generate music and record it into your `out_stream`. This can take a couple of minutes.

```
In [31]: out_stream = generate_music(inference_model)

Predicting new values for different set of chords.
Generated 51 sounds using the predicted values for the set of chords ("1") and afte
Generated 51 sounds using the predicted values for the set of chords ("2") and afte
Generated 51 sounds using the predicted values for the set of chords ("3") and afte
Generated 51 sounds using the predicted values for the set of chords ("4") and afte
Generated 51 sounds using the predicted values for the set of chords ("5") and afte
Your generated music is saved in output/my_music.midi
```

To listen to your music, click File->Open... Then go to "output/" and download "my_music.midi". Either play it on your computer with an application that can read midi files if you have one, or use one of the free online "MIDI to mp3" conversion tools to convert this to mp3.

As reference, here also is a 30sec audio clip we generated using this algorithm.

```
In [33]: IPython.display.Audio('./data/30s_trained_model.mp3')

Out[33]: <IPython.lib.display.Audio object>
```

1.3.1 Congratulations!

You have come to the end of the notebook.

Here's what you should remember: - A sequence model can be used to generate musical values, which are then post-processed into midi music. - Fairly similar models can be used to generate dinosaur names or to generate music, with the major difference being the input fed to the model. - In Keras, sequence generation involves defining layers with shared weights, which are then repeated for the different time steps $1, \ldots, T_x$.

Congratulations on completing this assignment and generating a jazz solo!

References

The ideas presented in this notebook came primarily from three computational music papers cited below. The implementation here also took significant inspiration and used many components from Ji-Sung Kim's github repository.

- Ji-Sung Kim, 2016, deepjazz
- Jon Gillick, Kevin Tang and Robert Keller, 2009. Learning Jazz Grammars
- Robert Keller and David Morrison, 2007, A Grammatical Approach to Automatic Improvisation
- François Pachet, 1999, Surprising Harmonies

We're also grateful to François Germain for valuable feedback.