Improvise_a_Jazz_Solo_with_an_LSTM_Network_v3a

February 16, 2020

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.

1.1 Updates

If you were working on the notebook before this update...

- The current notebook is version "3a".
- You can find your original work saved in the notebook with the previous version name ("v3")
- To view the file directory, go to the menu "File->Open", and this will open a new tab that shows the file directory.

List of updates

- `djmodel`
  - Explains `Input` layer and its parameter `shape`.
  - Explains `Lambda` layer and replaces the given solution with hints and sample code (to improve the learning experience).
  - Adds hints for using the Keras `Model`.
- `music_inference_model`
  - Explains each line of code in the `one_hot` function.
  - Explains how to apply `one_hot` with a Lambda layer instead of giving the code solution (to improve the learning experience).
  - Adds instructions on defining the `Model`.
- `predict_and_sample`
  - Provides detailed instructions for each step.
  - Clarifies which variable/function to use for inference.
- Spelling, grammar and wording corrections.

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

```
In [1]: 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.
```

1.2 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 network.

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

1.2.1 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 [2]: IPython.display.Audio('./data/30s_seq.mp3')
```

```
Out[2]: <IPython.lib.display.Audio object>
```

We have taken care of the preprocessing of the musical data to render it in terms of musical "values."

Details about music (optional) You can informally think of each "value" as a note, which comprises a pitch and 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 (playing 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.

Music as a sequence of values

- 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 [3]: X, Y, n_values, indices_values = load_music_utils()
        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 X:', X.shape)
        print('Shape of Y:', Y.shape)

number of training examples: 60
Tx (length of sequence): 30
total # of unique values: 78
shape of X: (60, 30, 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.
    - For example, X[i,t,:] is a one-hot vector representing the value of the i-th example at time t.
- `Y`: a $(T_y, m, 78)$ dimensional array
  - This is essentially the same as `X`, but shifted one step to the left (to the past).
  - Notice that 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 into the LSTM later.
  - Similar to the dinosaur assignment, we're using the previous values to predict the next value.
    - So our sequence model will try to predict $y^{\langle t \rangle}$ given $x^{\langle 1 \rangle}, \ldots, x^{\langle t \rangle}$.
- `n_values`: The number of unique values in this dataset. This should be 78.
- `indices_values`: python dictionary mapping integers 0 through 77 to musical values.

1.2.2 1.2 - Overview of our model

Here is the architecture of the model we will use. This is similar to the Dinosaurus model, except that you will implement it in Keras.

- $X = (x^{\langle 1 \rangle}, x^{\langle 2 \rangle}, \cdots, x^{\langle T_x \rangle})$ is a window of size T_x scanned over the musical corpus.
- Each $x^{\langle t \rangle}$ is an index corresponding to a value.
- $\hat{y}^t$ is the prediction for the next value.
- 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 $x^{\langle 1 \rangle} = \vec{0}$, since most of these snippets of audio start somewhere in the middle of a piece of music.
  - We are setting each of the snippets to have the same length $T_x = 30$ to make vectorization easier.

1.3 Overview of parts 2 and 3

- We're going to train a model that predicts the next note in a style that is similar to the jazz music that it's trained on. The training is contained in the weights and biases of the model.
- In Part 3, we're then going to use those weights and biases in a new model which predicts a series of notes, using the previous note to predict the next note.
- The weights and biases are transferred to the new model using 'global shared layers' described below"

1.4 2 - Building the model

- In this part you will build and train a model that will learn musical patterns.
- The model takes input X of shape $(m, T_x, 78)$ and labels Y of shape $(T_y, m, 78)$.
- We will use an LSTM with hidden states that have $n_a = 64$ dimensions.

```
In [4]: # number of dimensions for the hidden state of each LSTM cell.
        n_a = 64
```

Sequence generation uses a for-loop

- If you're building an RNN where, at test time, the entire input sequence $x^{\langle 1 \rangle}, x^{\langle 2 \rangle}, \ldots, x^{\langle T_x \rangle}$ is given in advance, 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 $x^{\langle t \rangle} = y^{\langle t-1 \rangle}$.
  - The input at time "t" is the prediction at the previous time step "t-1".
- So you'll need to implement your own for-loop to iterate over the time steps.

Shareable weights

- The function `djmodel()` will call the LSTM layer T_x times using a for-loop.
- It is important that all T_x copies have the same weights.
  - The T_x steps should have shared weights that aren't re-initialized.
- Referencing a globally defined shared layer will utilize the same layer-object instance at each time step.
- 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.

3 types of layers

- We have defined the layers objects you need as global variables.

- Please run the next cell to create them.
- Please read the Keras documentation and understand these layers:
  - Reshape(): Reshapes an output to a certain shape.
  - LSTM(): Long Short-Term Memory layer
  - Dense(): A regular fully-connected neural network layer.

```
In [5]: n_values = 78 # number of music values
        reshapor = Reshape((1, n_values))                        # Used in Step 2.B
        LSTM_cell = LSTM(n_a, return_state = True)         # Used in Step 2.C
        densor = Dense(n_values, activation='softmax')     # Used in Step 2.D
```

- `reshapor`, `LSTM_cell` and `densor` are globally defined layer objects, that you'll use to implement `djmodel()`.
- In order to propagate a Keras tensor object X through one of these layers, use `layer_object()`.
  - For one input, use `layer_object(X)`
  - For more than one input, put the inputs in a list: `layer_object([X1,X2])`

Exercise: Implement `djmodel()`.

Inputs (given)

- The `Input()` layer is used for defining the input `X` as well as the initial hidden state 'a0' and cell state `c0`.
- The `shape` parameter takes a tuple that does not include the batch dimension (`m`).
  - For example,

```
X = Input(shape=(Tx, n_values)) # X has 3 dimensions and not 2: (m, Tx, n_v
```

1. Create an empty list "outputs" to save the outputs of the LSTM Cell at every time step.

Step 2: Loop through time steps (TODO)

- Loop for $t \in 1, \ldots, T_x$:

2A. Select the 't' time-step vector from X.

- X has the shape (m, Tx, n_values).
- The shape of the 't' selection should be (n_values,).
- Recall that if you were implementing in numpy instead of Keras, you would extract a slice from a 3D numpy array like this:

```
var1 = array1[:,1,:]
```

Lambda layer

- Since we're using Keras, we need to define this step inside a custom layer.
- In Keras, this is a Lambda layer Lambda
- As an example, a Lambda layer that takes the previous layer and adds '1' looks like this

```
lambda_layer1 = Lambda(lambda z: z + 1)(previous_layer)
```

- The previous layer in this case is `X`.
- `z` is a local variable of the lambda function.
  - The `previous_layer` gets passed into the parameter `z` in the lowercase `lambda` function.
  - You can choose the name of the variable to be something else if you want.
- The operation after the colon ':' should be the operation to extract a slice from the previous layer.
- **Hint**: You'll be using the variable `t` within the definition of the lambda layer even though it isn't passed in as an argument to Lambda.

2B. Reshape x to be (1,n_values).

- Use the `reshapor()` layer. It is a function that takes the previous layer as its input argument.

2C. Run x through one step of LSTM_cell.

- Initialize the `LSTM_cell` with the previous step's hidden state a and cell state c.
- Use the following formatting:

```
next_hidden_state, _, next_cell_state = LSTM_cell(inputs=input_x, initial_state
```

  - Choose appropriate variables for inputs, hidden state and cell state.

2D. Dense layer

- Propagate the LSTM's hidden state through a dense+softmax layer using `densor`.

2E. Append output

- Append the output to the list of "outputs".

Step 3: After the loop, create the model

- Use the Keras `Model` object to create a model.
- specify the inputs and outputs:

```
model = Model(inputs=[input_x, initial_hidden_state, initial_cell_state], outpu
```

  - Choose the appropriate variables for the input tensor, hidden state, cell state, and output.

- See the documentation for Model

```
In [6]: # 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 instance model with n_a activations
            """

            # Define the input layer and specify the shape
            X = Input(shape=(Tx, n_values))

            # Define the initial hidden state a0 and initial cell state c0
```

```
    # using `Input`
    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
```

Create the model object

- 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 [7]: model = djmodel(Tx = 30 , n_a = 64, n_values = 78)

In [8]: # Check your model
        model.summary()
```

Layer (type)	Output Shape	Param #	Connected to
input_1 (InputLayer)	(None, 30, 78)	0	
lambda_1 (Lambda)	(None, 78)	0	input_1[0][0]

reshape_1 (Reshape)	(None, 1, 78)	0	lambda_1[0][0] lambda_2[0][0] lambda_3[0][0] lambda_4[0][0] lambda_5[0][0] lambda_6[0][0] lambda_7[0][0] lambda_8[0][0] lambda_9[0][0] lambda_10[0][0] lambda_11[0][0] lambda_12[0][0] lambda_13[0][0] lambda_14[0][0] lambda_15[0][0] lambda_16[0][0] lambda_17[0][0] lambda_18[0][0] lambda_19[0][0] lambda_20[0][0] lambda_21[0][0] lambda_22[0][0] lambda_23[0][0] lambda_24[0][0] lambda_25[0][0] lambda_26[0][0] lambda_27[0][0] lambda_28[0][0] lambda_29[0][0] lambda_30[0][0]
a0 (InputLayer)	(None, 64)	0	
c0 (InputLayer)	(None, 64)	0	
lambda_2 (Lambda)	(None, 78)	0	input_1[0][0]
lstm_1 (LSTM)	[(None, 64), (None, 6	36608	reshape_1[0][0] a0[0][0] c0[0][0] reshape_1[1][0] lstm_1[0][0] lstm_1[0][2] reshape_1[2][0] lstm_1[1][0] lstm_1[1][2] reshape_1[3][0]

```
lstm_1[2][0]
lstm_1[2][2]
reshape_1[4][0]
lstm_1[3][0]
lstm_1[3][2]
reshape_1[5][0]
lstm_1[4][0]
lstm_1[4][2]
reshape_1[6][0]
lstm_1[5][0]
lstm_1[5][2]
reshape_1[7][0]
lstm_1[6][0]
lstm_1[6][2]
reshape_1[8][0]
lstm_1[7][0]
lstm_1[7][2]
reshape_1[9][0]
lstm_1[8][0]
lstm_1[8][2]
reshape_1[10][0]
lstm_1[9][0]
lstm_1[9][2]
reshape_1[11][0]
lstm_1[10][0]
lstm_1[10][2]
reshape_1[12][0]
lstm_1[11][0]
lstm_1[11][2]
reshape_1[13][0]
lstm_1[12][0]
lstm_1[12][2]
reshape_1[14][0]
lstm_1[13][0]
lstm_1[13][2]
reshape_1[15][0]
lstm_1[14][0]
lstm_1[14][2]
reshape_1[16][0]
lstm_1[15][0]
lstm_1[15][2]
reshape_1[17][0]
lstm_1[16][0]
lstm_1[16][2]
reshape_1[18][0]
lstm_1[17][0]
lstm_1[17][2]
reshape_1[19][0]
```

			lstm_1[18][0] lstm_1[18][2] reshape_1[20][0]
			lstm_1[19][0] lstm_1[19][2] reshape_1[21][0]
			lstm_1[20][0] lstm_1[20][2] reshape_1[22][0]
			lstm_1[21][0] lstm_1[21][2] reshape_1[23][0]
			lstm_1[22][0] lstm_1[22][2] reshape_1[24][0]
			lstm_1[23][0] lstm_1[23][2] reshape_1[25][0]
			lstm_1[24][0] lstm_1[24][2] reshape_1[26][0]
			lstm_1[25][0] lstm_1[25][2] reshape_1[27][0]
			lstm_1[26][0] lstm_1[26][2] reshape_1[28][0]
			lstm_1[27][0] lstm_1[27][2] reshape_1[29][0]
			lstm_1[28][0] lstm_1[28][2]
lambda_3 (Lambda)	(None, 78)	0	input_1[0][0]
lambda_4 (Lambda)	(None, 78)	0	input_1[0][0]
lambda_5 (Lambda)	(None, 78)	0	input_1[0][0]
lambda_6 (Lambda)	(None, 78)	0	input_1[0][0]
lambda_7 (Lambda)	(None, 78)	0	input_1[0][0]
lambda_8 (Lambda)	(None, 78)	0	input_1[0][0]
lambda_9 (Lambda)	(None, 78)	0	input_1[0][0]
lambda_10 (Lambda)	(None, 78)	0	input_1[0][0]

lambda_11 (Lambda)	(None, 78)	0	input_1[0][0]
lambda_12 (Lambda)	(None, 78)	0	input_1[0][0]
lambda_13 (Lambda)	(None, 78)	0	input_1[0][0]
lambda_14 (Lambda)	(None, 78)	0	input_1[0][0]
lambda_15 (Lambda)	(None, 78)	0	input_1[0][0]
lambda_16 (Lambda)	(None, 78)	0	input_1[0][0]
lambda_17 (Lambda)	(None, 78)	0	input_1[0][0]
lambda_18 (Lambda)	(None, 78)	0	input_1[0][0]
lambda_19 (Lambda)	(None, 78)	0	input_1[0][0]
lambda_20 (Lambda)	(None, 78)	0	input_1[0][0]
lambda_21 (Lambda)	(None, 78)	0	input_1[0][0]
lambda_22 (Lambda)	(None, 78)	0	input_1[0][0]
lambda_23 (Lambda)	(None, 78)	0	input_1[0][0]
lambda_24 (Lambda)	(None, 78)	0	input_1[0][0]
lambda_25 (Lambda)	(None, 78)	0	input_1[0][0]
lambda_26 (Lambda)	(None, 78)	0	input_1[0][0]
lambda_27 (Lambda)	(None, 78)	0	input_1[0][0]
lambda_28 (Lambda)	(None, 78)	0	input_1[0][0]
lambda_29 (Lambda)	(None, 78)	0	input_1[0][0]
lambda_30 (Lambda)	(None, 78)	0	input_1[0][0]
dense_1 (Dense)	(None, 78)	5070	lstm_1[0][0] lstm_1[1][0] lstm_1[2][0] lstm_1[3][0] lstm_1[4][0] lstm_1[5][0] lstm_1[6][0]

```
                                                                    lstm_1[7][0]
                                                                    lstm_1[8][0]
                                                                    lstm_1[9][0]
                                                                    lstm_1[10][0]
                                                                    lstm_1[11][0]
                                                                    lstm_1[12][0]
                                                                    lstm_1[13][0]
                                                                    lstm_1[14][0]
                                                                    lstm_1[15][0]
                                                                    lstm_1[16][0]
                                                                    lstm_1[17][0]
                                                                    lstm_1[18][0]
                                                                    lstm_1[19][0]
                                                                    lstm_1[20][0]
                                                                    lstm_1[21][0]
                                                                    lstm_1[22][0]
                                                                    lstm_1[23][0]
                                                                    lstm_1[24][0]
                                                                    lstm_1[25][0]
                                                                    lstm_1[26][0]
                                                                    lstm_1[27][0]
                                                                    lstm_1[28][0]
                                                                    lstm_1[29][0]
==================================================================================================
Total params: 41,678
Trainable params: 41,678
Non-trainable params: 0
__________________________________________________________________________________________________
```

Expected Output

Scroll to the bottom of the output, and you'll see the following:

```
Total params: 41,678
Trainable params: 41,678
Non-trainable params: 0
```

Compile the model for training

- You now need to compile your model to be trained.
- We will use:
  - optimizer: Adam optimizer
  - Loss function: categorical cross-entropy (for multi-class classification)

```
In [9]: opt = Adam(lr=0.01, beta_1=0.9, beta_2=0.999, decay=0.01)

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

Initialize hidden state and cell state Finally, let's initialize `a0` and `c0` for the LSTM's initial state to be zero.

```
In [10]: m = 60
         a0 = np.zeros((m, n_a))
         c0 = np.zeros((m, n_a))
```

Train the model

- Lets now fit the model!
- We will turn `Y` into a list, since the cost function expects `Y` to be provided in this format
  - `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 [11]: model.fit([X, a0, c0], list(Y), epochs=100)
```

```
Epoch 1/100
60/60 [==============================] - 5s - loss: 125.8584 - dense_1_loss_1: 4.35
Epoch 2/100
60/60 [==============================] - 0s - loss: 122.9525 - dense_1_loss_1: 4.33
Epoch 3/100
60/60 [==============================] - 0s - loss: 116.6183 - dense_1_loss_1: 4.31
Epoch 4/100
60/60 [==============================] - 0s - loss: 113.6502 - dense_1_loss_1: 4.29
Epoch 5/100
60/60 [==============================] - 0s - loss: 110.7826 - dense_1_loss_1: 4.27
Epoch 6/100
60/60 [==============================] - 0s - loss: 108.1485 - dense_1_loss_1: 4.26
Epoch 7/100
60/60 [==============================] - 0s - loss: 105.5043 - dense_1_loss_1: 4.24
Epoch 8/100
60/60 [==============================] - 0s - loss: 102.2788 - dense_1_loss_1: 4.23
Epoch 9/100
60/60 [==============================] - 0s - loss: 98.5535 - dense_1_loss_1: 4.224
Epoch 10/100
60/60 [==============================] - 0s - loss: 94.9006 - dense_1_loss_1: 4.213
Epoch 11/100
60/60 [==============================] - 0s - loss: 90.8013 - dense_1_loss_1: 4.204
Epoch 12/100
60/60 [==============================] - 0s - loss: 86.5049 - dense_1_loss_1: 4.194
Epoch 13/100
60/60 [==============================] - 0s - loss: 82.3308 - dense_1_loss_1: 4.185
Epoch 14/100
60/60 [==============================] - 0s - loss: 78.2256 - dense_1_loss_1: 4.177
Epoch 15/100
60/60 [==============================] - 0s - loss: 74.4312 - dense_1_loss_1: 4.170
Epoch 16/100
60/60 [==============================] - 0s - loss: 70.9315 - dense_1_loss_1: 4.162
```

```
Epoch 17/100
60/60 [==============================] - 0s - loss: 67.3165 - dense_1_loss_1: 4.155
Epoch 18/100
60/60 [==============================] - 0s - loss: 64.1914 - dense_1_loss_1: 4.147
Epoch 19/100
60/60 [==============================] - 0s - loss: 61.0370 - dense_1_loss_1: 4.138
Epoch 20/100
60/60 [==============================] - 0s - loss: 57.8097 - dense_1_loss_1: 4.129
Epoch 21/100
60/60 [==============================] - 0s - loss: 55.0789 - dense_1_loss_1: 4.120
Epoch 22/100
60/60 [==============================] - 0s - loss: 52.2065 - dense_1_loss_1: 4.112
Epoch 23/100
60/60 [==============================] - 0s - loss: 49.4801 - dense_1_loss_1: 4.103
Epoch 24/100
60/60 [==============================] - 0s - loss: 46.9355 - dense_1_loss_1: 4.094
Epoch 25/100
60/60 [==============================] - 0s - loss: 44.5206 - dense_1_loss_1: 4.085
Epoch 26/100
60/60 [==============================] - 0s - loss: 42.1026 - dense_1_loss_1: 4.076
Epoch 27/100
60/60 [==============================] - 0s - loss: 39.8206 - dense_1_loss_1: 4.066
Epoch 28/100
60/60 [==============================] - 0s - loss: 37.7957 - dense_1_loss_1: 4.059
Epoch 29/100
60/60 [==============================] - 0s - loss: 35.7132 - dense_1_loss_1: 4.050
Epoch 30/100
60/60 [==============================] - 0s - loss: 33.7394 - dense_1_loss_1: 4.043
Epoch 31/100
60/60 [==============================] - 0s - loss: 31.7958 - dense_1_loss_1: 4.035
Epoch 32/100
60/60 [==============================] - 0s - loss: 29.9671 - dense_1_loss_1: 4.027
Epoch 33/100
60/60 [==============================] - 0s - loss: 28.3287 - dense_1_loss_1: 4.020
Epoch 34/100
60/60 [==============================] - 0s - loss: 26.6942 - dense_1_loss_1: 4.012
Epoch 35/100
60/60 [==============================] - 0s - loss: 25.2419 - dense_1_loss_1: 4.006
Epoch 36/100
60/60 [==============================] - 0s - loss: 23.8434 - dense_1_loss_1: 4.000
Epoch 37/100
60/60 [==============================] - 0s - loss: 22.5347 - dense_1_loss_1: 3.993
Epoch 38/100
60/60 [==============================] - 0s - loss: 21.2686 - dense_1_loss_1: 3.986
Epoch 39/100
60/60 [==============================] - 0s - loss: 20.1210 - dense_1_loss_1: 3.981
Epoch 40/100
60/60 [==============================] - 0s - loss: 19.0262 - dense_1_loss_1: 3.974
```

```
Epoch 41/100
60/60 [==============================] - 0s - loss: 18.0316 - dense_1_loss_1: 3.969
Epoch 42/100
60/60 [==============================] - 0s - loss: 17.1298 - dense_1_loss_1: 3.963
Epoch 43/100
60/60 [==============================] - 0s - loss: 16.3071 - dense_1_loss_1: 3.958
Epoch 44/100
60/60 [==============================] - 0s - loss: 15.5303 - dense_1_loss_1: 3.952
Epoch 45/100
60/60 [==============================] - 0s - loss: 14.8421 - dense_1_loss_1: 3.948
Epoch 46/100
60/60 [==============================] - 0s - loss: 14.1692 - dense_1_loss_1: 3.942
Epoch 47/100
60/60 [==============================] - 0s - loss: 13.5986 - dense_1_loss_1: 3.937
Epoch 48/100
60/60 [==============================] - 0s - loss: 13.0382 - dense_1_loss_1: 3.932
Epoch 49/100
60/60 [==============================] - 0s - loss: 12.5348 - dense_1_loss_1: 3.927
Epoch 50/100
60/60 [==============================] - 0s - loss: 12.0785 - dense_1_loss_1: 3.923
Epoch 51/100
60/60 [==============================] - 0s - loss: 11.6649 - dense_1_loss_1: 3.918
Epoch 52/100
60/60 [==============================] - 0s - loss: 11.2744 - dense_1_loss_1: 3.914
Epoch 53/100
60/60 [==============================] - 0s - loss: 10.9344 - dense_1_loss_1: 3.910
Epoch 54/100
60/60 [==============================] - 0s - loss: 10.6155 - dense_1_loss_1: 3.906
Epoch 55/100
60/60 [==============================] - 0s - loss: 10.3152 - dense_1_loss_1: 3.901
Epoch 56/100
60/60 [==============================] - 0s - loss: 10.0458 - dense_1_loss_1: 3.897
Epoch 57/100
60/60 [==============================] - 0s - loss: 9.7952 - dense_1_loss_1: 3.8936
Epoch 58/100
60/60 [==============================] - 0s - loss: 9.5636 - dense_1_loss_1: 3.8894
Epoch 59/100
60/60 [==============================] - 0s - loss: 9.3439 - dense_1_loss_1: 3.8856
Epoch 60/100
60/60 [==============================] - 0s - loss: 9.1455 - dense_1_loss_1: 3.8816
Epoch 61/100
60/60 [==============================] - 0s - loss: 8.9587 - dense_1_loss_1: 3.8774
Epoch 62/100
60/60 [==============================] - 0s - loss: 8.7912 - dense_1_loss_1: 3.8735
Epoch 63/100
60/60 [==============================] - 0s - loss: 8.6295 - dense_1_loss_1: 3.8699
Epoch 64/100
60/60 [==============================] - 0s - loss: 8.4776 - dense_1_loss_1: 3.8658
```

```
Epoch 65/100
60/60 [==============================] - 0s - loss: 8.3366 - dense_1_loss_1: 3.8622
Epoch 66/100
60/60 [==============================] - 0s - loss: 8.2027 - dense_1_loss_1: 3.8584
Epoch 67/100
60/60 [==============================] - 0s - loss: 8.0790 - dense_1_loss_1: 3.8545
Epoch 68/100
60/60 [==============================] - 0s - loss: 7.9629 - dense_1_loss_1: 3.8507
Epoch 69/100
60/60 [==============================] - 0s - loss: 7.8523 - dense_1_loss_1: 3.8469
Epoch 70/100
60/60 [==============================] - 0s - loss: 7.7477 - dense_1_loss_1: 3.8433
Epoch 71/100
60/60 [==============================] - 0s - loss: 7.6478 - dense_1_loss_1: 3.8396
Epoch 72/100
60/60 [==============================] - 0s - loss: 7.5529 - dense_1_loss_1: 3.8361
Epoch 73/100
60/60 [==============================] - 0s - loss: 7.4652 - dense_1_loss_1: 3.8322
Epoch 74/100
60/60 [==============================] - 0s - loss: 7.3780 - dense_1_loss_1: 3.8284
Epoch 75/100
60/60 [==============================] - 0s - loss: 7.2991 - dense_1_loss_1: 3.8251
Epoch 76/100
60/60 [==============================] - 0s - loss: 7.2205 - dense_1_loss_1: 3.8215
Epoch 77/100
60/60 [==============================] - 0s - loss: 7.1490 - dense_1_loss_1: 3.8179
Epoch 78/100
60/60 [==============================] - 0s - loss: 7.0784 - dense_1_loss_1: 3.8142
Epoch 79/100
60/60 [==============================] - 0s - loss: 7.0085 - dense_1_loss_1: 3.8110
Epoch 80/100
60/60 [==============================] - 0s - loss: 6.9440 - dense_1_loss_1: 3.8074
Epoch 81/100
60/60 [==============================] - 0s - loss: 6.8817 - dense_1_loss_1: 3.8039
Epoch 82/100
60/60 [==============================] - 0s - loss: 6.8228 - dense_1_loss_1: 3.8005
Epoch 83/100
60/60 [==============================] - 0s - loss: 6.7628 - dense_1_loss_1: 3.7972
Epoch 84/100
60/60 [==============================] - 0s - loss: 6.7082 - dense_1_loss_1: 3.7938
Epoch 85/100
60/60 [==============================] - 0s - loss: 6.6549 - dense_1_loss_1: 3.7903
Epoch 86/100
60/60 [==============================] - 0s - loss: 6.6036 - dense_1_loss_1: 3.7873
Epoch 87/100
60/60 [==============================] - 0s - loss: 6.5545 - dense_1_loss_1: 3.7836
Epoch 88/100
60/60 [==============================] - 0s - loss: 6.5056 - dense_1_loss_1: 3.7802
```

```
Epoch 89/100
60/60 [==============================] - 0s - loss: 6.4587 - dense_1_loss_1: 3.7769
Epoch 90/100
60/60 [==============================] - 0s - loss: 6.4149 - dense_1_loss_1: 3.7736
Epoch 91/100
60/60 [==============================] - 0s - loss: 6.3722 - dense_1_loss_1: 3.7706
Epoch 92/100
60/60 [==============================] - 0s - loss: 6.3287 - dense_1_loss_1: 3.7671
Epoch 93/100
60/60 [==============================] - 0s - loss: 6.2889 - dense_1_loss_1: 3.7637
Epoch 94/100
60/60 [==============================] - 0s - loss: 6.2499 - dense_1_loss_1: 3.7610
Epoch 95/100
60/60 [==============================] - 0s - loss: 6.2111 - dense_1_loss_1: 3.7576
Epoch 96/100
60/60 [==============================] - 0s - loss: 6.1743 - dense_1_loss_1: 3.7544
Epoch 97/100
60/60 [==============================] - 0s - loss: 6.1373 - dense_1_loss_1: 3.7514
Epoch 98/100
60/60 [==============================] - 0s - loss: 6.1041 - dense_1_loss_1: 3.7482
Epoch 99/100
60/60 [==============================] - 0s - loss: 6.0702 - dense_1_loss_1: 3.7450
Epoch 100/100
60/60 [==============================] - 0s - loss: 6.0375 - dense_1_loss_1: 3.7421
```

```
Out[11]: <keras.callbacks.History at 0x7f2d3fc25ac8>
```

Expected Output The model loss will start high, (100 or so), and after 100 epochs, it should be in the single digits. These won't be the exact number that you'll see, due to random initialization of weights.
For example:

```
Epoch 1/100
60/60 [==============================] - 3s - loss: 125.7673
...
```

Scroll to the bottom to check Epoch 100

```
...
Epoch 100/100
60/60 [==============================] - 0s - loss: 6.1861
```

Now that you have trained a model, let's go to the final section to implement an inference algorithm, and generate some music!

1.5 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. * Get a new output activation as well as cell state. * The new activation 'a' can then be used to generate the output using the fully connected layer, densor.

Initialization

- We will initialize the following to be zeros:
  - x0
  - hidden state a0
  - cell state c0

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 takes in the input layer, as well as the previous step's 'c' and 'a' to generate the current step's 'c' and 'a'.

  ```
  next_hidden_state, _, next_cell_state = LSTM_cell(input_x, initial_state=[previ
  ```

  - Choose the appropriate variables for the input_x, hidden_state, and cell_state
- Step 2.B: Compute the output by applying densor to compute a softmax on 'a' to get the output for the current step.
- Step 2.C: Append the output to the list outputs.
- Step 2.D: Sample x to be the one-hot version of 'out'.
- This allows you to pass it to the next LSTM's step.

- We have provided the definition of one_hot(x) in the 'music_utils.py' file and imported it. Here is the definition of one_hot

  ```
  def one_hot(x):
  x = K.argmax(x)
  x = tf.one_hot(indices=x, depth=78)
  x = RepeatVector(1)(x)
  return x
  ```

  Here is what the one_hot function is doing:

- argmax: within the vector x, find the position with the maximum value and return the index of that position.
  - For example: argmax of [-1,0,1] finds that 1 is the maximum value, and returns the index position, which is 2. Read the documentation for keras.argmax.

- one_hot: takes a list of indices and the depth of the one-hot vector (number of categories, which is 78 in this assignment). It converts each index into the one-hot vector representation. For instance, if the indices is [2], and the depth is 5, then the one-hot vector returned is [0,0,1,0,0]. Check out the documentation for tf.one_hot for more examples and explanations.
- RepeatVector(n): This takes a vector and duplicates it `n` times. Notice that we had it repeat 1 time. This may seem like it's not doing anything. If you look at the documentation for RepeatVector, you'll notice that if x is a vector with dimension (m,5) and it gets passed into `RepeatVector(1)`, then the output is (m,1,5). In other words, it adds an additional dimension (of length 1) to the resulting vector.
- Apply the custom one_hot encoding using the Lambda layer. You saw earlier that the Lambda layer can be used like this:

```
result = Lambda(lambda x: x + 1)(input_var)
```

If you pre-define a function, you can do the same thing:

```
def add_one(x)
    return x + 1

# use the add_one function inside of the Lambda function
result = Lambda(add_one)(input_var)
```

Step 3: Inference Model: This is how to use the Keras `Model`.

```
model = Model(inputs=[input_x, initial_hidden_state, initial_cell_state], outputs=t
```

- Choose the appropriate variables for the input tensor, hidden state, cell state, and output.
- **Hint**: the inputs to the model are the **initial** inputs and states.

```
In [12]: # 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, number 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",
        # Set "x" to be the one-hot representation of the selected value
        # See instructions above.
        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 [13]: inference_model = music_inference_model(LSTM_cell, densor, n_values = 78,

In [14]: # Check the inference model
         inference_model.summary()
```

```
_________________________________________________________________________________
Layer (type)                 Output Shape              Param #     Connected to
=================================================================================
```

input_2 (InputLayer)	(None, 1, 78)	0	
a0 (InputLayer)	(None, 64)	0	
c0 (InputLayer)	(None, 64)	0	
lstm_1 (LSTM)	[(None, 64), (None, 6	36608	input_2[0][0] a0[0][0] c0[0][0] lambda_31[0][0] lstm_1[30][0] lstm_1[30][2] lambda_32[0][0] lstm_1[31][0] lstm_1[31][2] lambda_33[0][0] lstm_1[32][0] lstm_1[32][2] lambda_34[0][0] lstm_1[33][0] lstm_1[33][2] lambda_35[0][0] lstm_1[34][0] lstm_1[34][2] lambda_36[0][0] lstm_1[35][0] lstm_1[35][2] lambda_37[0][0] lstm_1[36][0] lstm_1[36][2] lambda_38[0][0] lstm_1[37][0] lstm_1[37][2] lambda_39[0][0] lstm_1[38][0] lstm_1[38][2] lambda_40[0][0] lstm_1[39][0] lstm_1[39][2] lambda_41[0][0] lstm_1[40][0] lstm_1[40][2] lambda_42[0][0] lstm_1[41][0] lstm_1[41][2] lambda_43[0][0] lstm_1[42][0] lstm_1[42][2]

lambda_44[0][0]
lstm_1[43][0]
lstm_1[43][2]
lambda_45[0][0]
lstm_1[44][0]
lstm_1[44][2]
lambda_46[0][0]
lstm_1[45][0]
lstm_1[45][2]
lambda_47[0][0]
lstm_1[46][0]
lstm_1[46][2]
lambda_48[0][0]
lstm_1[47][0]
lstm_1[47][2]
lambda_49[0][0]
lstm_1[48][0]
lstm_1[48][2]
lambda_50[0][0]
lstm_1[49][0]
lstm_1[49][2]
lambda_51[0][0]
lstm_1[50][0]
lstm_1[50][2]
lambda_52[0][0]
lstm_1[51][0]
lstm_1[51][2]
lambda_53[0][0]
lstm_1[52][0]
lstm_1[52][2]
lambda_54[0][0]
lstm_1[53][0]
lstm_1[53][2]
lambda_55[0][0]
lstm_1[54][0]
lstm_1[54][2]
lambda_56[0][0]
lstm_1[55][0]
lstm_1[55][2]
lambda_57[0][0]
lstm_1[56][0]
lstm_1[56][2]
lambda_58[0][0]
lstm_1[57][0]
lstm_1[57][2]
lambda_59[0][0]
lstm_1[58][0]
lstm_1[58][2]

lambda_60[0][0]
lstm_1[59][0]
lstm_1[59][2]
lambda_61[0][0]
lstm_1[60][0]
lstm_1[60][2]
lambda_62[0][0]
lstm_1[61][0]
lstm_1[61][2]
lambda_63[0][0]
lstm_1[62][0]
lstm_1[62][2]
lambda_64[0][0]
lstm_1[63][0]
lstm_1[63][2]
lambda_65[0][0]
lstm_1[64][0]
lstm_1[64][2]
lambda_66[0][0]
lstm_1[65][0]
lstm_1[65][2]
lambda_67[0][0]
lstm_1[66][0]
lstm_1[66][2]
lambda_68[0][0]
lstm_1[67][0]
lstm_1[67][2]
lambda_69[0][0]
lstm_1[68][0]
lstm_1[68][2]
lambda_70[0][0]
lstm_1[69][0]
lstm_1[69][2]
lambda_71[0][0]
lstm_1[70][0]
lstm_1[70][2]
lambda_72[0][0]
lstm_1[71][0]
lstm_1[71][2]
lambda_73[0][0]
lstm_1[72][0]
lstm_1[72][2]
lambda_74[0][0]
lstm_1[73][0]
lstm_1[73][2]
lambda_75[0][0]
lstm_1[74][0]
lstm_1[74][2]

			lambda_76[0][0]
			lstm_1[75][0]
			lstm_1[75][2]
			lambda_77[0][0]
			lstm_1[76][0]
			lstm_1[76][2]
			lambda_78[0][0]
			lstm_1[77][0]
			lstm_1[77][2]
			lambda_79[0][0]
			lstm_1[78][0]
			lstm_1[78][2]
dense_1 (Dense)	(None, 78)	5070	lstm_1[30][0]
			lstm_1[31][0]
			lstm_1[32][0]
			lstm_1[33][0]
			lstm_1[34][0]
			lstm_1[35][0]
			lstm_1[36][0]
			lstm_1[37][0]
			lstm_1[38][0]
			lstm_1[39][0]
			lstm_1[40][0]
			lstm_1[41][0]
			lstm_1[42][0]
			lstm_1[43][0]
			lstm_1[44][0]
			lstm_1[45][0]
			lstm_1[46][0]
			lstm_1[47][0]
			lstm_1[48][0]
			lstm_1[49][0]
			lstm_1[50][0]
			lstm_1[51][0]
			lstm_1[52][0]
			lstm_1[53][0]
			lstm_1[54][0]
			lstm_1[55][0]
			lstm_1[56][0]
			lstm_1[57][0]
			lstm_1[58][0]
			lstm_1[59][0]
			lstm_1[60][0]
			lstm_1[61][0]
			lstm_1[62][0]
			lstm_1[63][0]
			lstm_1[64][0]

			lstm_1[65][0] lstm_1[66][0] lstm_1[67][0] lstm_1[68][0] lstm_1[69][0] lstm_1[70][0] lstm_1[71][0] lstm_1[72][0] lstm_1[73][0] lstm_1[74][0] lstm_1[75][0] lstm_1[76][0] lstm_1[77][0] lstm_1[78][0] lstm_1[79][0]
lambda_31 (Lambda)	(None, 1, 78)	0	dense_1[30][0]
lambda_32 (Lambda)	(None, 1, 78)	0	dense_1[31][0]
lambda_33 (Lambda)	(None, 1, 78)	0	dense_1[32][0]
lambda_34 (Lambda)	(None, 1, 78)	0	dense_1[33][0]
lambda_35 (Lambda)	(None, 1, 78)	0	dense_1[34][0]
lambda_36 (Lambda)	(None, 1, 78)	0	dense_1[35][0]
lambda_37 (Lambda)	(None, 1, 78)	0	dense_1[36][0]
lambda_38 (Lambda)	(None, 1, 78)	0	dense_1[37][0]
lambda_39 (Lambda)	(None, 1, 78)	0	dense_1[38][0]
lambda_40 (Lambda)	(None, 1, 78)	0	dense_1[39][0]
lambda_41 (Lambda)	(None, 1, 78)	0	dense_1[40][0]
lambda_42 (Lambda)	(None, 1, 78)	0	dense_1[41][0]
lambda_43 (Lambda)	(None, 1, 78)	0	dense_1[42][0]
lambda_44 (Lambda)	(None, 1, 78)	0	dense_1[43][0]
lambda_45 (Lambda)	(None, 1, 78)	0	dense_1[44][0]
lambda_46 (Lambda)	(None, 1, 78)	0	dense_1[45][0]

lambda_47 (Lambda)	(None, 1, 78)	0	dense_1[46][0]
lambda_48 (Lambda)	(None, 1, 78)	0	dense_1[47][0]
lambda_49 (Lambda)	(None, 1, 78)	0	dense_1[48][0]
lambda_50 (Lambda)	(None, 1, 78)	0	dense_1[49][0]
lambda_51 (Lambda)	(None, 1, 78)	0	dense_1[50][0]
lambda_52 (Lambda)	(None, 1, 78)	0	dense_1[51][0]
lambda_53 (Lambda)	(None, 1, 78)	0	dense_1[52][0]
lambda_54 (Lambda)	(None, 1, 78)	0	dense_1[53][0]
lambda_55 (Lambda)	(None, 1, 78)	0	dense_1[54][0]
lambda_56 (Lambda)	(None, 1, 78)	0	dense_1[55][0]
lambda_57 (Lambda)	(None, 1, 78)	0	dense_1[56][0]
lambda_58 (Lambda)	(None, 1, 78)	0	dense_1[57][0]
lambda_59 (Lambda)	(None, 1, 78)	0	dense_1[58][0]
lambda_60 (Lambda)	(None, 1, 78)	0	dense_1[59][0]
lambda_61 (Lambda)	(None, 1, 78)	0	dense_1[60][0]
lambda_62 (Lambda)	(None, 1, 78)	0	dense_1[61][0]
lambda_63 (Lambda)	(None, 1, 78)	0	dense_1[62][0]
lambda_64 (Lambda)	(None, 1, 78)	0	dense_1[63][0]
lambda_65 (Lambda)	(None, 1, 78)	0	dense_1[64][0]
lambda_66 (Lambda)	(None, 1, 78)	0	dense_1[65][0]
lambda_67 (Lambda)	(None, 1, 78)	0	dense_1[66][0]
lambda_68 (Lambda)	(None, 1, 78)	0	dense_1[67][0]
lambda_69 (Lambda)	(None, 1, 78)	0	dense_1[68][0]
lambda_70 (Lambda)	(None, 1, 78)	0	dense_1[69][0]

```
lambda_71 (Lambda)               (None, 1, 78)         0           dense_1[70][0]
__________________________________________________________________________________
lambda_72 (Lambda)               (None, 1, 78)         0           dense_1[71][0]
__________________________________________________________________________________
lambda_73 (Lambda)               (None, 1, 78)         0           dense_1[72][0]
__________________________________________________________________________________
lambda_74 (Lambda)               (None, 1, 78)         0           dense_1[73][0]
__________________________________________________________________________________
lambda_75 (Lambda)               (None, 1, 78)         0           dense_1[74][0]
__________________________________________________________________________________
lambda_76 (Lambda)               (None, 1, 78)         0           dense_1[75][0]
__________________________________________________________________________________
lambda_77 (Lambda)               (None, 1, 78)         0           dense_1[76][0]
__________________________________________________________________________________
lambda_78 (Lambda)               (None, 1, 78)         0           dense_1[77][0]
__________________________________________________________________________________
lambda_79 (Lambda)               (None, 1, 78)         0           dense_1[78][0]
==================================================================================
Total params: 41,678
Trainable params: 41,678
Non-trainable params: 0
__________________________________________________________________________________
```

** Expected Output** If you scroll to the bottom of the output, you'll see:

```
Total params: 41,678
Trainable params: 41,678
Non-trainable params: 0
```

Initialize inference model The following code creates the zero-valued vectors you will use to initialize `x` and the LSTM state variables `a` and `c`.

```
In [15]: 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:

Step 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).

  ```
  inference_model.predict([input_x_init, hidden_state_init, cell_state_init])
  ```

- Choose the appropriate input arguments to `predict` from the input arguments of this `predict_and_sample` function.

Step 2

- Convert `pred` into a numpy array of T_y indices.
  - Each index is computed by taking the `argmax` of an element of the `pred` list.
  - Use numpy.argmax.
  - Set the `axis` parameter.
    - Remember that the shape of the prediction is (m, T_y, n_{values})

Step 3

- Convert the indices into their one-hot vector representations.
  - Use to_categorical.
  - Set the `num_classes` parameter. Note that for grading purposes: you'll need to either:
    - Use a dimension from the given parameters of `predict_and_sample()` (for example, one of the dimensions of x_initializer has the value for the number of distinct classes).
    - Or just hard code the number of distinct classes (will pass the grader as well).
    - Note that using a global variable such as n_values will not work for grading purposes.

In [18]:
```
# 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=2)
```

```
            # Step 3: Convert indices to one-hot vectors, the shape of the results
            results = to_categorical(indices)
            ### END CODE HERE ###

            return results, indices
```

```
In [19]: results, indices = predict_and_sample(inference_model, x_initializer, a_ir
         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]) = 18
np.argmax(results[17]) = 3
list(indices[12:18]) = [array([18]), array([32]), array([46]), array([71]), array([
```

Expected (Approximate) Output:

- Your results **may likely 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[17]) =
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 of 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 let's use it in our implementation as well.

Let's 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 [20]: out_stream = generate_music(inference_model)

Predicting new values for different set of chords.
Generated 50 sounds using the predicted values for the set of chords ("1") and afte
Generated 49 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 50 sounds using the predicted values for the set of chords ("4") and afte
Generated 50 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 a reference, here is a 30 second audio clip we generated using this algorithm.

```
In [21]: IPython.display.Audio('./data/30s_trained_model.mp3')

Out[21]: <IPython.lib.display.Audio object>
```

1.5.1 Congratulations!

You have come to the end of the notebook.

1.6 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.

```
In [ ]:
```