

Improvise_a_Jazz_Solo_with_an_LSTM_Network_v4

June 18, 2025

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. At the end, you'll even be able to listen to your own music!

By the end of this assignment, you'll be able to:

- Apply an LSTM to a music generation task
- Generate your own jazz music with deep learning
- Use the flexible Functional API to create complex models

This is going to be a fun one. Let's get started!

1.1 Important Note on Submission to the AutoGrader

Before submitting your assignment to the AutoGrader, please make sure you are not doing the following:

1. You have not added any *extra print* statement(s) in the assignment.
2. You have not added any *extra* code cell(s) in the assignment.
3. You have not changed any of the function parameters.
4. You are not using any global variables inside your graded exercises. Unless specifically instructed to do so, please refrain from it and use the local variables instead.
5. You are not changing the assignment code where it is not required, like creating *extra* variables.

If you do any of the following, you will get something like, **Grader Error: Grader feedback not found** (or similarly unexpected) error upon submitting your assignment. Before asking for help/debugging the errors in your assignment, check for these first. If this is the case, and you don't remember the changes you have made, you can get a fresh copy of the assignment by following these [instructions](#).

1.2 Table of Contents

- Section ??
- Section ??
 - Section ??
 - Section ??
- Section ??

- Section ??
- Section ??
 - Section ??
 - * Section ??
 - * Section ??
 - Section ??
- Section ??

Packages

Run the following cell to load all the packages you'll need. This may take a few minutes!

```
[ ]: ### v2.1
```

```
[1]: import IPython
import sys
import matplotlib.pyplot as plt
import numpy as np
import tensorflow as tf

from music21 import *
from grammar import *
from qa import *
from preprocess import *
from music_utils import *
from data_utils import *
from outputs import *
from test_utils import *

from tensorflow.keras.layers import Dense, Activation, Dropout, Input, LSTM, Reshape, Lambda, RepeatVector
from tensorflow.keras.models import Model
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.utils import to_categorical
```

1 - Problem Statement

You would like to create a jazz music piece specially for a friend's birthday. However, you don't know how to play any instruments, or how to compose music. 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.

1.1 - Dataset

To get started, you'll 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:

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

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

The preprocessing of the musical data has been taken care of already, which for this notebook means it's been rendered in terms of musical “values.”

What are musical “values”? (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 you don’t need to worry about the details of music theory for this assignment.

Music as a sequence of values

- For the purposes of this assignment, all you need to know is that you’ll obtain a dataset of values, and will use an RNN model to generate sequences of values.
- Your music generation system will use 90 unique values.

Run the following code to load the raw music data and preprocess it into values. This might take a few minutes!

```
[3]: X, Y, n_values, indices_values, chords = load_music_utils('data/
    ↳original_metheny.mid')
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)
print('Number of chords', len(chords))
```

```
number of training examples: 60
Tx (length of sequence): 30
total # of unique values: 90
shape of X: (60, 30, 90)
Shape of Y: (30, 60, 90)
Number of chords 19
```

You have just loaded the following:

- X: This is an $(m, T_x, 90)$ dimensional array.
 - You 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 90 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 $(T_y, m, 90)$ dimensional array

- This is essentially the same as \mathbf{X} , but shifted one step to the left (to the past).
- Notice that the data in \mathbf{Y} is **reordered** to be dimension $(T_y, m, 90)$, where $T_y = T_x$. This format makes it more convenient to feed into the LSTM later.
- Similar to the dinosaur assignment, you’re using the previous values to predict the next value.

* So your sequence model will try to predict $y^{(t)}$ given $x^{(1)}, \dots, x^{(t)}$.

- **n_values**: The number of unique values in this dataset. This should be 90.
- **indices_values**: python dictionary mapping integers 0 through 89 to musical values.
- **chords**: Chords used in the input midi

1.2 - Model Overview

Here is the architecture of the model you’ll use. It’s similar to the Dinosaur model, except that you’ll implement it in Keras.

Figure 1: Basic LSTM model

- $X = (x^{(1)}, x^{(2)}, \dots, x^{(T_x)})$ is a window of size T_x scanned over the musical corpus.
- Each $x^{(t)}$ is an index corresponding to a value.
- $\hat{y}^{(t)}$ is the prediction for the next value.
- You’ll be training the model on random snippets of 30 values taken from a much longer piece of music.
 - Thus, you won’t bother to set the first input $x^{(1)} = \vec{0}$, since most of these snippets of audio start somewhere in the middle of a piece of music.
 - You’re setting each of the snippets to have the same length $T_x = 30$ to make vectorization easier.

1.3 Overview of Section 2 and 3

In Section 2, you’re going to train a model that predicts the next note in a style similar to the jazz music it’s trained on. The training is contained in the weights and biases of the model.

Then, in Section 3, you’re going to use those weights and biases in a new model that predicts a series of notes, and using the previous note to predict the next note.

- The weights and biases are transferred to the new model using the global shared layers (`LSTM_cell`, `tensor`, `resaper`) described below

2 - Building the Model

Now, you’ll build and train a model that will learn musical patterns. * The model takes input \mathbf{X} of shape $(m, T_x, 90)$ and labels \mathbf{Y} of shape $(T_y, m, 90)$. * You’ll use an LSTM with hidden states that have $n_a = 64$ dimensions.

```
[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 $x^{(1)}, x^{(2)}, \dots, x^{(T_x)}$ is given in advance, then Keras has simple built-in functions to build the model.
- However, for **sequence generation, at test time you won't know all the values of $x^{(t)}$ in advance.**
- Instead, you'll generate them one at a time using $x^{(t)} = y^{(t-1)}$.
 - 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 (you'll use global variables for this).
 2. Call these objects when propagating the input.

3 types of layers

- The layer objects you need for global variables have been defined.
 - Just 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.

```
[5]: n_values = 90 # number of music values
      reshaper = Reshape((1, n_values)) # Used in Step 2.B of □
      ↪ djmodel(), below
      LSTM_cell = LSTM(n_a, return_state = True) # Used in Step 2.C
      densor = Dense(n_values, activation='softmax') # Used in Step 2.D
```

- `reshaper`, `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 1 - djmodel

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_values)*

Step 1: Outputs

- Create an empty list “outputs” to save the outputs of the LSTM Cell at every time step.

Step 2: Loop through time steps

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

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,:]
```

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

- Use the `reshape()` layer. This 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=[previous_hidden_state, previous_cell_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 `dense`.

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. There are two ways to instantiate the Model object. One is by subclassing, which you won't use here. Instead, you'll use the highly flexible Functional API, which you may remember from an earlier assignment in this course! With the Functional API, you'll start from your Input, then specify the model's forward pass with chained layer calls, and finally create the model from inputs and outputs.
- Specify the inputs and output like so:

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

* Then, choose the appropriate variables for the input tensor, hidden state, cell state, and output.

- See the documentation for [Model](#)

```
[15]: # UNQ_C1 (UNIQUE CELL IDENTIFIER, DO NOT EDIT)
# GRADED FUNCTION: djmodel

def djmodel(Tx, LSTM_cell, densor, reshaper):
    """
    Implement the djmodel composed of Tx LSTM cells where each cell is
    responsible
    for learning the following note based on the previous note and context.
    Each cell has the following schema:
        [X_{t}, a_{t-1}, c0_{t-1}] -> RESHAPE() -> LSTM() -> DENSE()
    Arguments:
        Tx -- length of the sequences in the corpus
        LSTM_cell -- LSTM layer instance
        densor -- Dense layer instance
        reshaper -- Reshape layer instance

    Returns:
        model -- a keras instance model with inputs [X, a0, c0]
    """
    # Get the shape of input values
    n_values = densor.units

    # Get the number of the hidden state vector
    n_a = LSTM_cell.units

    # 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 (1
→line)
outputs = []

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

    # Step 2.A: select the "t"th time step vector from X.
    x = X[:,t,:]
    # Step 2.B: Use reshaper to reshape x to be (1, n_values) (1 line)
    x = reshaper(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: append 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.
- You will use Tx=30.
- This cell may take a few seconds to run.

```

[16]: ### YOU CANNOT EDIT THIS CELL

model = djmodel(Tx=30, LSTM_cell=LSTM_cell, densor=densor, reshaper=reshaper)

```

```

[17]: ### YOU CANNOT EDIT THIS CELL

# UNIT TEST
output = summary(model)
comparator(output, djmodel_out)

```

All tests passed!

```

[18]: # Check your model
model.summary()

```


Model: "functional_3"

Layer (type)	Output Shape	Param #	Connected to
input_3 (InputLayer)	[(None, 30, 90)]	0	
tf_op_layer_strided_slice (Tens	[(None, 90)]	0	input_3[0][0]
reshape (Reshape)	(None, 1, 90)	0	
tf_op_layer_strided_slice[0][0]			
tf_op_layer_strided_slice_1[0][0]			
tf_op_layer_strided_slice_2[0][0]			
tf_op_layer_strided_slice_3[0][0]			
tf_op_layer_strided_slice_4[0][0]			
tf_op_layer_strided_slice_5[0][0]			
tf_op_layer_strided_slice_6[0][0]			
tf_op_layer_strided_slice_7[0][0]			
tf_op_layer_strided_slice_8[0][0]			
tf_op_layer_strided_slice_9[0][0]			
tf_op_layer_strided_slice_10[0][0]			
tf_op_layer_strided_slice_11[0][0]			
tf_op_layer_strided_slice_12[0][0]			
tf_op_layer_strided_slice_13[0][0]			
tf_op_layer_strided_slice_14[0][0]			
tf_op_layer_strided_slice_15[0][0]			
tf_op_layer_strided_slice_16[0][0]			
tf_op_layer_strided_slice_17[0][0]			
tf_op_layer_strided_slice_18[0][0]			
tf_op_layer_strided_slice_19[0][0]			
tf_op_layer_strided_slice_20[0][0]			
tf_op_layer_strided_slice_21[0][0]			
tf_op_layer_strided_slice_22[0][0]			
tf_op_layer_strided_slice_23[0][0]			
tf_op_layer_strided_slice_24[0][0]			
tf_op_layer_strided_slice_25[0][0]			
tf_op_layer_strided_slice_26[0][0]			
tf_op_layer_strided_slice_27[0][0]			
tf_op_layer_strided_slice_28[0][0]			
tf_op_layer_strided_slice_29[0][0]			
a0 (InputLayer)	[(None, 64)]	0	

c0 (InputLayer)	[(None, 64)]	0	

tf_op_layer_strided_slice_1 (Te	[(None, 90)]	0	input_3[0] [0]

lstm (LSTM)	[(None, 64), (None, 39680		reshape[30] [0] a0[0] [0] c0[0] [0] reshape[31] [0] lstm[30] [1] lstm[30] [2] reshape[32] [0] lstm[31] [1] lstm[31] [2] reshape[33] [0] lstm[32] [1] lstm[32] [2] reshape[34] [0] lstm[33] [1] lstm[33] [2] reshape[35] [0] lstm[34] [1] lstm[34] [2] reshape[36] [0] lstm[35] [1] lstm[35] [2] reshape[37] [0] lstm[36] [1] lstm[36] [2] reshape[38] [0] lstm[37] [1] lstm[37] [2] reshape[39] [0] lstm[38] [1] lstm[38] [2] reshape[40] [0] lstm[39] [1] lstm[39] [2] reshape[41] [0] lstm[40] [1] lstm[40] [2] reshape[42] [0] lstm[41] [1] lstm[41] [2] reshape[43] [0] lstm[42] [1] lstm[42] [2]

```
reshape[44] [0]
lstm[43] [1]
lstm[43] [2]
reshape[45] [0]
lstm[44] [1]
lstm[44] [2]
reshape[46] [0]
lstm[45] [1]
lstm[45] [2]
reshape[47] [0]
lstm[46] [1]
lstm[46] [2]
reshape[48] [0]
lstm[47] [1]
lstm[47] [2]
reshape[49] [0]
lstm[48] [1]
lstm[48] [2]
reshape[50] [0]
lstm[49] [1]
lstm[49] [2]
reshape[51] [0]
lstm[50] [1]
lstm[50] [2]
reshape[52] [0]
lstm[51] [1]
lstm[51] [2]
reshape[53] [0]
lstm[52] [1]
lstm[52] [2]
reshape[54] [0]
lstm[53] [1]
lstm[53] [2]
reshape[55] [0]
lstm[54] [1]
lstm[54] [2]
reshape[56] [0]
lstm[55] [1]
lstm[55] [2]
reshape[57] [0]
lstm[56] [1]
lstm[56] [2]
reshape[58] [0]
lstm[57] [1]
lstm[57] [2]
reshape[59] [0]
lstm[58] [1]
lstm[58] [2]
```

tf_op_layer_strided_slice_2 (Te [(None, 90)])	0	input_3[0][0]

tf_op_layer_strided_slice_3 (Te [(None, 90)])	0	input_3[0][0]

tf_op_layer_strided_slice_4 (Te [(None, 90)])	0	input_3[0][0]

tf_op_layer_strided_slice_5 (Te [(None, 90)])	0	input_3[0][0]

tf_op_layer_strided_slice_6 (Te [(None, 90)])	0	input_3[0][0]

tf_op_layer_strided_slice_7 (Te [(None, 90)])	0	input_3[0][0]

tf_op_layer_strided_slice_8 (Te [(None, 90)])	0	input_3[0][0]

tf_op_layer_strided_slice_9 (Te [(None, 90)])	0	input_3[0][0]

tf_op_layer_strided_slice_10 (T [(None, 90)])	0	input_3[0][0]

tf_op_layer_strided_slice_11 (T [(None, 90)])	0	input_3[0][0]

tf_op_layer_strided_slice_12 (T [(None, 90)])	0	input_3[0][0]

tf_op_layer_strided_slice_13 (T [(None, 90)])	0	input_3[0][0]

tf_op_layer_strided_slice_14 (T [(None, 90)])	0	input_3[0][0]

tf_op_layer_strided_slice_15 (T [(None, 90)])	0	input_3[0][0]

tf_op_layer_strided_slice_16 (T [(None, 90)])	0	input_3[0][0]

tf_op_layer_strided_slice_17 (T [(None, 90)])	0	input_3[0][0]

tf_op_layer_strided_slice_18 (T [(None, 90)])	0	input_3[0][0]
tf_op_layer_strided_slice_19 (T [(None, 90)])	0	input_3[0][0]
tf_op_layer_strided_slice_20 (T [(None, 90)])	0	input_3[0][0]
tf_op_layer_strided_slice_21 (T [(None, 90)])	0	input_3[0][0]
tf_op_layer_strided_slice_22 (T [(None, 90)])	0	input_3[0][0]
tf_op_layer_strided_slice_23 (T [(None, 90)])	0	input_3[0][0]
tf_op_layer_strided_slice_24 (T [(None, 90)])	0	input_3[0][0]
tf_op_layer_strided_slice_25 (T [(None, 90)])	0	input_3[0][0]
tf_op_layer_strided_slice_26 (T [(None, 90)])	0	input_3[0][0]
tf_op_layer_strided_slice_27 (T [(None, 90)])	0	input_3[0][0]
tf_op_layer_strided_slice_28 (T [(None, 90)])	0	input_3[0][0]
tf_op_layer_strided_slice_29 (T [(None, 90)])	0	input_3[0][0]
dense (Dense)	(None, 90)	5850
		lstm[30][1]
		lstm[31][1]
		lstm[32][1]
		lstm[33][1]
		lstm[34][1]
		lstm[35][1]
		lstm[36][1]
		lstm[37][1]
		lstm[38][1]
		lstm[39][1]

```
lstm[40] [1]
lstm[41] [1]
lstm[42] [1]
lstm[43] [1]
lstm[44] [1]
lstm[45] [1]
lstm[46] [1]
lstm[47] [1]
lstm[48] [1]
lstm[49] [1]
lstm[50] [1]
lstm[51] [1]
lstm[52] [1]
lstm[53] [1]
lstm[54] [1]
lstm[55] [1]
lstm[56] [1]
lstm[57] [1]
lstm[58] [1]
lstm[59] [1]
```

```
=====
Total params: 45,530
Trainable params: 45,530
Non-trainable params: 0
-----
```

Expected Output

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

```
Total params: 45,530
Trainable params: 45,530
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)

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

model.compile(optimizer=opt, loss='categorical_crossentropy',
               metrics=['accuracy'])
```

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

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

Train the model You're now ready to fit the model!

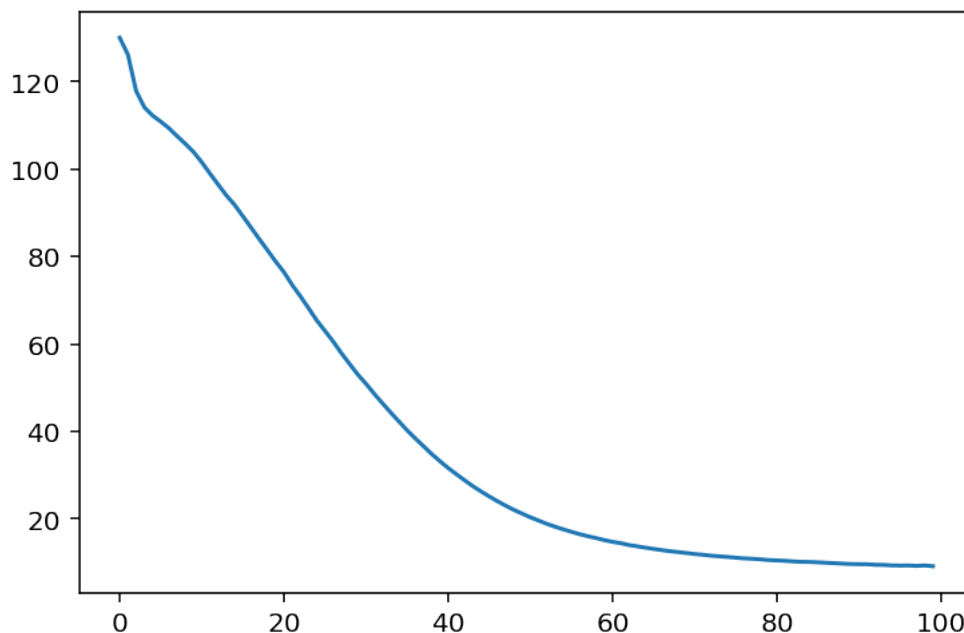
- You'll turn `Y` into a list, since the cost function expects `Y` to be provided in this format.
 - `list(Y)` is a list with 30 items, where each of the list items is of shape (60,90).
 - Train for 100 epochs (This will take a few minutes).

```
[21]: history = model.fit([X, a0, c0], list(Y), epochs=100, verbose = 0)
```

```
[22]: print(f"loss at epoch 1: {history.history['loss'][0]}")
      print(f"loss at epoch 100: {history.history['loss'][99]}")
      plt.plot(history.history['loss'])
```

```
loss at epoch 1: 130.07989501953125
loss at epoch 100: 9.14416217803955
```

```
[22]: [<matplotlib.lines.Line2D at 0x7efbbcbca7d0>]
```



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:

```
loss at epoch 1: 129.88641357421875
...
```

Scroll to the bottom to check Epoch 100

```
loss at epoch 100: 9.21483039855957
```

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

3 - Generating Music

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

3.1 - Predicting & Sampling

Figure 2: Generating new values in an LSTM

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, `dense`.

Initialization

- You'll initialize the following to be zeros:
 - `x0`
 - hidden state `a0`
 - cell state `c0`

Exercise 2 - `music_inference_model`

Implement `music_inference_model()` 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 1: Create an empty list "outputs" to save the outputs of the LSTM Cell at every time step.

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=[previous_hidden_state, previous_cell_state])
```

- Choose the appropriate variables for `input_x`, `hidden_state`, and `cell_state`

2.B: Compute the output by applying `dense` to compute a softmax on 'a' to get the output for the current step.

2.C: Append the output to the list `outputs`.

2.D: Convert the last output into a new input for the next time step. You will do this in 2 substeps:
- Get the index of the maximum value of the predicted output using `tf.math.argmax` along the last axis. - Convert the index into its `n_values`-one-hot encoding using `tf.one_hot`.

2.E: Use `RepeatVector(1)(x)` to convert the output of the one-hot encoding with shape=(None, 90) into a tensor with shape=(None, 1, 90)

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

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

- 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.

```
[25]: # UNQ_C2 (UNIQUE CELL IDENTIFIER, DO NOT EDIT)
# GRADED FUNCTION: music_inference_model

def music_inference_model(LSTM_cell, densor, Ty=100):
    """
    Uses the trained "LSTM_cell" and "densor" from model() to generate a
    →sequence of values.

    Arguments:
    LSTM_cell -- the trained "LSTM_cell" from model(), Keras layer object
    densor -- the trained "densor" from model(), Keras layer object
    Ty -- integer, number of time steps to generate

    Returns:
    inference_model -- Keras model instance
    """

    # Get the shape of input values
    n_values = densor.units
    # Get the number of the hidden state vector
    n_a = LSTM_cell.units

    # 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 predicted
    ↪ values (1 line)
    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. Use "x", not "x0" (1 line)
        _, a, c = LSTM_cell(x, initial_state=[a, c])

        # Step 2.B: Apply Dense layer to the hidden state output of the
    ↪ LSTM_cell (1 line)
        out = densor(a)
        # Step 2.C: Append the prediction "out" to "outputs". out.shape =
    ↪ (None, 90) (1 line)
        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 = tf.math.argmax(out, axis=-1)
        x = tf.one_hot(x, n_values)
        # Step 2.E:
        # Use RepeatVector(1) to convert x into a tensor with shape=(None, 1,
    ↪ 90)
        x = RepeatVector(1)(x)

    # Step 3: Create model instance with the correct "inputs" and "outputs" (1
    ↪ line)
    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.

```

[26]: ### YOU CANNOT EDIT THIS CELL
inference_model = music_inference_model(LSTM_cell, densor, Ty = 50)

```

```

[27]: ### YOU CANNOT EDIT THIS CELL

# UNIT TESTS
# Convert NumPy arrays to TensorFlow tensors
x_initializer = tf.convert_to_tensor(np.zeros((1, 1, n_values)), dtype=tf.
    ↪ float32)
a_initializer = tf.convert_to_tensor(np.zeros((1, n_a)), dtype=tf.float32)

```

```

c_initializer = tf.convert_to_tensor(np.zeros((1, n_a)), dtype=tf.float32)

# Use @tf.function to wrap the prediction and allow shape relaxation
@tf.function(experimental_relax_shapes=True)
def predict_model(x, a, c):
    return inference_model([x, a, c])

# Get model outputs
outputs = inference_model.outputs
print('Outputs = ', len(outputs))
print('Single output shape = ', outputs[0].shape)

# Assertions for outputs
expected_num_outputs = 50
expected_output_shape = (None, 90)

assert len(outputs) == expected_num_outputs, f"Expected {expected_num_outputs}␣
→outputs, but got {len(outputs)}"
assert outputs[0].shape.as_list() == list(expected_output_shape), f"Expected␣
→output shape {expected_output_shape}, but got {outputs[0].shape}"

# Call the wrapped prediction function
pred = predict_model(x_initializer, a_initializer, c_initializer)
print('len(pred) = ', len(pred))
print('pred[0].shape = ', pred[0].shape)

# Assertions for predictions
expected_pred_length = 50
expected_pred_shape = (1, 90)

assert len(pred) == expected_pred_length, f"Expected {expected_pred_length}␣
→predictions, but got {len(pred)}"
assert pred[0].shape == expected_pred_shape, f"Expected prediction shape␣
→{expected_pred_shape}, but got {pred[0].shape}"

inference_summary = summary(inference_model)
comparator(inference_summary, music_inference_model_out)

```

```

Outputs = 50
Single output shape = (None, 90)
len(pred) = 50
pred[0].shape = (1, 90)
All tests passed!

```

Expected Output:

```
Outputs = 50
```

```

Single otuput shape = (None, 90)
len(pred) = 50
pred[0].shape = (1, 90)
All tests passed!

```

```

[28]: # Check the inference model
inference_model.summary()

```

```

Model: "functional_5"

```

Layer (type)	Output Shape	Param #	Connected to
input_5 (InputLayer)	[(None, 1, 90)]	0	
a0 (InputLayer)	[(None, 64)]	0	
c0 (InputLayer)	[(None, 64)]	0	
lstm (LSTM)	[(None, 64), (None, 39680		input_5[0] [0] a0[0] [0] c0[0] [0]
repeat_vector[0] [0]			lstm[61] [1] lstm[61] [2]
repeat_vector_1[0] [0]			lstm[62] [1] lstm[62] [2]
repeat_vector_2[0] [0]			lstm[63] [1] lstm[63] [2]
repeat_vector_3[0] [0]			lstm[64] [1] lstm[64] [2]
repeat_vector_4[0] [0]			lstm[65] [1] lstm[65] [2]
repeat_vector_5[0] [0]			lstm[66] [1] lstm[66] [2]
repeat_vector_6[0] [0]			lstm[67] [1] lstm[67] [2]
repeat_vector_7[0] [0]			

	lstm[68] [1]
	lstm[68] [2]
repeat_vector_8[0] [0]	
	lstm[69] [1]
	lstm[69] [2]
repeat_vector_9[0] [0]	
	lstm[70] [1]
	lstm[70] [2]
repeat_vector_10[0] [0]	
	lstm[71] [1]
	lstm[71] [2]
repeat_vector_11[0] [0]	
	lstm[72] [1]
	lstm[72] [2]
repeat_vector_12[0] [0]	
	lstm[73] [1]
	lstm[73] [2]
repeat_vector_13[0] [0]	
	lstm[74] [1]
	lstm[74] [2]
repeat_vector_14[0] [0]	
	lstm[75] [1]
	lstm[75] [2]
repeat_vector_15[0] [0]	
	lstm[76] [1]
	lstm[76] [2]
repeat_vector_16[0] [0]	
	lstm[77] [1]
	lstm[77] [2]
repeat_vector_17[0] [0]	
	lstm[78] [1]
	lstm[78] [2]
repeat_vector_18[0] [0]	
	lstm[79] [1]
	lstm[79] [2]
repeat_vector_19[0] [0]	
	lstm[80] [1]
	lstm[80] [2]
repeat_vector_20[0] [0]	
	lstm[81] [1]
	lstm[81] [2]
repeat_vector_21[0] [0]	
	lstm[82] [1]
	lstm[82] [2]
repeat_vector_22[0] [0]	
	lstm[83] [1]
	lstm[83] [2]
repeat_vector_23[0] [0]	

	lstm[84] [1]
	lstm[84] [2]
repeat_vector_24[0] [0]	
	lstm[85] [1]
	lstm[85] [2]
repeat_vector_25[0] [0]	
	lstm[86] [1]
	lstm[86] [2]
repeat_vector_26[0] [0]	
	lstm[87] [1]
	lstm[87] [2]
repeat_vector_27[0] [0]	
	lstm[88] [1]
	lstm[88] [2]
repeat_vector_28[0] [0]	
	lstm[89] [1]
	lstm[89] [2]
repeat_vector_29[0] [0]	
	lstm[90] [1]
	lstm[90] [2]
repeat_vector_30[0] [0]	
	lstm[91] [1]
	lstm[91] [2]
repeat_vector_31[0] [0]	
	lstm[92] [1]
	lstm[92] [2]
repeat_vector_32[0] [0]	
	lstm[93] [1]
	lstm[93] [2]
repeat_vector_33[0] [0]	
	lstm[94] [1]
	lstm[94] [2]
repeat_vector_34[0] [0]	
	lstm[95] [1]
	lstm[95] [2]
repeat_vector_35[0] [0]	
	lstm[96] [1]
	lstm[96] [2]
repeat_vector_36[0] [0]	
	lstm[97] [1]
	lstm[97] [2]
repeat_vector_37[0] [0]	
	lstm[98] [1]
	lstm[98] [2]
repeat_vector_38[0] [0]	
	lstm[99] [1]
	lstm[99] [2]
repeat_vector_39[0] [0]	

			lstm[100] [1]
			lstm[100] [2]
repeat_vector_40[0] [0]			
			lstm[101] [1]
			lstm[101] [2]
repeat_vector_41[0] [0]			
			lstm[102] [1]
			lstm[102] [2]
repeat_vector_42[0] [0]			
			lstm[103] [1]
			lstm[103] [2]
repeat_vector_43[0] [0]			
			lstm[104] [1]
			lstm[104] [2]
repeat_vector_44[0] [0]			
			lstm[105] [1]
			lstm[105] [2]
repeat_vector_45[0] [0]			
			lstm[106] [1]
			lstm[106] [2]
repeat_vector_46[0] [0]			
			lstm[107] [1]
			lstm[107] [2]
repeat_vector_47[0] [0]			
			lstm[108] [1]
			lstm[108] [2]
repeat_vector_48[0] [0]			
			lstm[109] [1]
			lstm[109] [2]

dense (Dense)	(None, 90)	5850	lstm[61] [1]
			lstm[62] [1]
			lstm[63] [1]
			lstm[64] [1]
			lstm[65] [1]
			lstm[66] [1]
			lstm[67] [1]
			lstm[68] [1]
			lstm[69] [1]
			lstm[70] [1]
			lstm[71] [1]
			lstm[72] [1]
			lstm[73] [1]
			lstm[74] [1]
			lstm[75] [1]
			lstm[76] [1]
			lstm[77] [1]

```

lstm[78] [1]
lstm[79] [1]
lstm[80] [1]
lstm[81] [1]
lstm[82] [1]
lstm[83] [1]
lstm[84] [1]
lstm[85] [1]
lstm[86] [1]
lstm[87] [1]
lstm[88] [1]
lstm[89] [1]
lstm[90] [1]
lstm[91] [1]
lstm[92] [1]
lstm[93] [1]
lstm[94] [1]
lstm[95] [1]
lstm[96] [1]
lstm[97] [1]
lstm[98] [1]
lstm[99] [1]
lstm[100] [1]
lstm[101] [1]
lstm[102] [1]
lstm[103] [1]
lstm[104] [1]
lstm[105] [1]
lstm[106] [1]
lstm[107] [1]
lstm[108] [1]
lstm[109] [1]
lstm[110] [1]

```

tf_op_layer_ArgMax_1 (TensorFlow [(None,)])	0	dense[61] [0]
---	---	---------------

tf_op_layer_OneHot (TensorFlow [(None, 90)])	0	
tf_op_layer_ArgMax_1 [0] [0]		

repeat_vector (RepeatVector) (None, 1, 90)	0	
tf_op_layer_OneHot [0] [0]		

tf_op_layer_ArgMax_2 (TensorFlow [(None,)])	0	dense[62] [0]
---	---	---------------

```

-----
tf_op_layer_OneHot_1 (TensorFlo [(None, 90)]      0
tf_op_layer_ArgMax_2[0][0]
-----

repeat_vector_1 (RepeatVector) (None, 1, 90)      0
tf_op_layer_OneHot_1[0][0]
-----

tf_op_layer_ArgMax_3 (TensorFlo [(None,)])        0      dense[63][0]
-----

tf_op_layer_OneHot_2 (TensorFlo [(None, 90)]      0
tf_op_layer_ArgMax_3[0][0]
-----

repeat_vector_2 (RepeatVector) (None, 1, 90)      0
tf_op_layer_OneHot_2[0][0]
-----

tf_op_layer_ArgMax_4 (TensorFlo [(None,)])        0      dense[64][0]
-----

tf_op_layer_OneHot_3 (TensorFlo [(None, 90)]      0
tf_op_layer_ArgMax_4[0][0]
-----

repeat_vector_3 (RepeatVector) (None, 1, 90)      0
tf_op_layer_OneHot_3[0][0]
-----

tf_op_layer_ArgMax_5 (TensorFlo [(None,)])        0      dense[65][0]
-----

tf_op_layer_OneHot_4 (TensorFlo [(None, 90)]      0
tf_op_layer_ArgMax_5[0][0]
-----

repeat_vector_4 (RepeatVector) (None, 1, 90)      0
tf_op_layer_OneHot_4[0][0]
-----

tf_op_layer_ArgMax_6 (TensorFlo [(None,)])        0      dense[66][0]
-----

tf_op_layer_OneHot_5 (TensorFlo [(None, 90)]      0
tf_op_layer_ArgMax_6[0][0]
-----

```

```

-----
repeat_vector_5 (RepeatVector) (None, 1, 90)      0
tf_op_layer_OneHot_5[0][0]
-----

tf_op_layer_ArgMax_7 (TensorFlo [(None,)])      0      dense[67][0]
-----

tf_op_layer_OneHot_6 (TensorFlo [(None, 90)])    0
tf_op_layer_ArgMax_7[0][0]
-----

repeat_vector_6 (RepeatVector) (None, 1, 90)      0
tf_op_layer_OneHot_6[0][0]
-----

tf_op_layer_ArgMax_8 (TensorFlo [(None,)])      0      dense[68][0]
-----

tf_op_layer_OneHot_7 (TensorFlo [(None, 90)])    0
tf_op_layer_ArgMax_8[0][0]
-----

repeat_vector_7 (RepeatVector) (None, 1, 90)      0
tf_op_layer_OneHot_7[0][0]
-----

tf_op_layer_ArgMax_9 (TensorFlo [(None,)])      0      dense[69][0]
-----

tf_op_layer_OneHot_8 (TensorFlo [(None, 90)])    0
tf_op_layer_ArgMax_9[0][0]
-----

repeat_vector_8 (RepeatVector) (None, 1, 90)      0
tf_op_layer_OneHot_8[0][0]
-----

tf_op_layer_ArgMax_10 (TensorFl [(None,)])      0      dense[70][0]
-----

tf_op_layer_OneHot_9 (TensorFlo [(None, 90)])    0
tf_op_layer_ArgMax_10[0][0]
-----

repeat_vector_9 (RepeatVector) (None, 1, 90)      0
tf_op_layer_OneHot_9[0][0]
-----

```

tf_op_layer_ArgMax_11 (TensorFl [(None,)]) 0 dense[71][0]

tf_op_layer_OneHot_10 (TensorFl [(None, 90)]) 0
tf_op_layer_ArgMax_11[0][0]

repeat_vector_10 (RepeatVector) (None, 1, 90) 0
tf_op_layer_OneHot_10[0][0]

tf_op_layer_ArgMax_12 (TensorFl [(None,)]) 0 dense[72][0]

tf_op_layer_OneHot_11 (TensorFl [(None, 90)]) 0
tf_op_layer_ArgMax_12[0][0]

repeat_vector_11 (RepeatVector) (None, 1, 90) 0
tf_op_layer_OneHot_11[0][0]

tf_op_layer_ArgMax_13 (TensorFl [(None,)]) 0 dense[73][0]

tf_op_layer_OneHot_12 (TensorFl [(None, 90)]) 0
tf_op_layer_ArgMax_13[0][0]

repeat_vector_12 (RepeatVector) (None, 1, 90) 0
tf_op_layer_OneHot_12[0][0]

tf_op_layer_ArgMax_14 (TensorFl [(None,)]) 0 dense[74][0]

tf_op_layer_OneHot_13 (TensorFl [(None, 90)]) 0
tf_op_layer_ArgMax_14[0][0]

repeat_vector_13 (RepeatVector) (None, 1, 90) 0
tf_op_layer_OneHot_13[0][0]

tf_op_layer_ArgMax_15 (TensorFl [(None,)]) 0 dense[75][0]

tf_op_layer_OneHot_14 (TensorFl [(None, 90)])	0	
tf_op_layer_ArgMax_15[0][0]		

repeat_vector_14 (RepeatVector) (None, 1, 90)	0	
tf_op_layer_OneHot_14[0][0]		

tf_op_layer_ArgMax_16 (TensorFl [(None,)])	0	dense[76][0]

tf_op_layer_OneHot_15 (TensorFl [(None, 90)])	0	
tf_op_layer_ArgMax_16[0][0]		

repeat_vector_15 (RepeatVector) (None, 1, 90)	0	
tf_op_layer_OneHot_15[0][0]		

tf_op_layer_ArgMax_17 (TensorFl [(None,)])	0	dense[77][0]

tf_op_layer_OneHot_16 (TensorFl [(None, 90)])	0	
tf_op_layer_ArgMax_17[0][0]		

repeat_vector_16 (RepeatVector) (None, 1, 90)	0	
tf_op_layer_OneHot_16[0][0]		

tf_op_layer_ArgMax_18 (TensorFl [(None,)])	0	dense[78][0]

tf_op_layer_OneHot_17 (TensorFl [(None, 90)])	0	
tf_op_layer_ArgMax_18[0][0]		

repeat_vector_17 (RepeatVector) (None, 1, 90)	0	
tf_op_layer_OneHot_17[0][0]		

tf_op_layer_ArgMax_19 (TensorFl [(None,)])	0	dense[79][0]

tf_op_layer_OneHot_18 (TensorFl [(None, 90)])	0	
tf_op_layer_ArgMax_19[0][0]		

repeat_vector_18 (RepeatVector) (None, 1, 90)	0	
tf_op_layer_OneHot_18[0][0]		

tf_op_layer_ArgMax_20 (TensorFl [(None,)])	0	dense[80][0]

tf_op_layer_OneHot_19 (TensorFl [(None, 90)])	0	
tf_op_layer_ArgMax_20[0][0]		

repeat_vector_19 (RepeatVector) (None, 1, 90)	0	
tf_op_layer_OneHot_19[0][0]		

tf_op_layer_ArgMax_21 (TensorFl [(None,)])	0	dense[81][0]

tf_op_layer_OneHot_20 (TensorFl [(None, 90)])	0	
tf_op_layer_ArgMax_21[0][0]		

repeat_vector_20 (RepeatVector) (None, 1, 90)	0	
tf_op_layer_OneHot_20[0][0]		

tf_op_layer_ArgMax_22 (TensorFl [(None,)])	0	dense[82][0]

tf_op_layer_OneHot_21 (TensorFl [(None, 90)])	0	
tf_op_layer_ArgMax_22[0][0]		

repeat_vector_21 (RepeatVector) (None, 1, 90)	0	
tf_op_layer_OneHot_21[0][0]		

tf_op_layer_ArgMax_23 (TensorFl [(None,)])	0	dense[83][0]

tf_op_layer_OneHot_22 (TensorFl [(None, 90)])	0	
tf_op_layer_ArgMax_23[0][0]		

repeat_vector_22 (RepeatVector) (None, 1, 90)	0	
tf_op_layer_OneHot_22[0][0]		

tf_op_layer_ArgMax_24 (TensorFl [(None,)])	0	dense[84] [0]

tf_op_layer_OneHot_23 (TensorFl [(None, 90)])	0	
tf_op_layer_ArgMax_24[0] [0]		

repeat_vector_23 (RepeatVector) (None, 1, 90)	0	
tf_op_layer_OneHot_23[0] [0]		

tf_op_layer_ArgMax_25 (TensorFl [(None,)])	0	dense[85] [0]

tf_op_layer_OneHot_24 (TensorFl [(None, 90)])	0	
tf_op_layer_ArgMax_25[0] [0]		

repeat_vector_24 (RepeatVector) (None, 1, 90)	0	
tf_op_layer_OneHot_24[0] [0]		

tf_op_layer_ArgMax_26 (TensorFl [(None,)])	0	dense[86] [0]

tf_op_layer_OneHot_25 (TensorFl [(None, 90)])	0	
tf_op_layer_ArgMax_26[0] [0]		

repeat_vector_25 (RepeatVector) (None, 1, 90)	0	
tf_op_layer_OneHot_25[0] [0]		

tf_op_layer_ArgMax_27 (TensorFl [(None,)])	0	dense[87] [0]

tf_op_layer_OneHot_26 (TensorFl [(None, 90)])	0	
tf_op_layer_ArgMax_27[0] [0]		

repeat_vector_26 (RepeatVector) (None, 1, 90)	0	
tf_op_layer_OneHot_26[0] [0]		

tf_op_layer_ArgMax_28 (TensorFl [(None,)])	0	dense[88] [0]

tf_op_layer_OneHot_27 (TensorFl [(None, 90)])	0	

```

tf_op_layer_ArgMax_28[0][0]
-----
repeat_vector_27 (RepeatVector) (None, 1, 90)      0
tf_op_layer_OneHot_27[0][0]
-----
tf_op_layer_ArgMax_29 (TensorFl [(None,)])          0      dense[89][0]
-----
tf_op_layer_OneHot_28 (TensorFl [(None, 90)])        0
tf_op_layer_ArgMax_29[0][0]
-----
repeat_vector_28 (RepeatVector) (None, 1, 90)      0
tf_op_layer_OneHot_28[0][0]
-----
tf_op_layer_ArgMax_30 (TensorFl [(None,)])          0      dense[90][0]
-----
tf_op_layer_OneHot_29 (TensorFl [(None, 90)])        0
tf_op_layer_ArgMax_30[0][0]
-----
repeat_vector_29 (RepeatVector) (None, 1, 90)      0
tf_op_layer_OneHot_29[0][0]
-----
tf_op_layer_ArgMax_31 (TensorFl [(None,)])          0      dense[91][0]
-----
tf_op_layer_OneHot_30 (TensorFl [(None, 90)])        0
tf_op_layer_ArgMax_31[0][0]
-----
repeat_vector_30 (RepeatVector) (None, 1, 90)      0
tf_op_layer_OneHot_30[0][0]
-----
tf_op_layer_ArgMax_32 (TensorFl [(None,)])          0      dense[92][0]
-----
tf_op_layer_OneHot_31 (TensorFl [(None, 90)])        0
tf_op_layer_ArgMax_32[0][0]
-----
repeat_vector_31 (RepeatVector) (None, 1, 90)      0

```

```

tf_op_layer_OneHot_31[0][0]
-----
tf_op_layer_ArgMax_33 (TensorFl [(None,)])          0          dense[93][0]
-----
tf_op_layer_OneHot_32 (TensorFl [(None, 90)])        0
tf_op_layer_ArgMax_33[0][0]
-----
repeat_vector_32 (RepeatVector) (None, 1, 90)        0
tf_op_layer_OneHot_32[0][0]
-----
tf_op_layer_ArgMax_34 (TensorFl [(None,)])          0          dense[94][0]
-----
tf_op_layer_OneHot_33 (TensorFl [(None, 90)])        0
tf_op_layer_ArgMax_34[0][0]
-----
repeat_vector_33 (RepeatVector) (None, 1, 90)        0
tf_op_layer_OneHot_33[0][0]
-----
tf_op_layer_ArgMax_35 (TensorFl [(None,)])          0          dense[95][0]
-----
tf_op_layer_OneHot_34 (TensorFl [(None, 90)])        0
tf_op_layer_ArgMax_35[0][0]
-----
repeat_vector_34 (RepeatVector) (None, 1, 90)        0
tf_op_layer_OneHot_34[0][0]
-----
tf_op_layer_ArgMax_36 (TensorFl [(None,)])          0          dense[96][0]
-----
tf_op_layer_OneHot_35 (TensorFl [(None, 90)])        0
tf_op_layer_ArgMax_36[0][0]
-----
repeat_vector_35 (RepeatVector) (None, 1, 90)        0
tf_op_layer_OneHot_35[0][0]
-----
tf_op_layer_ArgMax_37 (TensorFl [(None,)])          0          dense[97][0]

```



```

-----
tf_op_layer_OneHot_36 (TensorFl [(None, 90)])      0
tf_op_layer_ArgMax_37[0][0]
-----

repeat_vector_36 (RepeatVector) (None, 1, 90)      0
tf_op_layer_OneHot_36[0][0]
-----

tf_op_layer_ArgMax_38 (TensorFl [(None,)])          0      dense[98][0]
-----

tf_op_layer_OneHot_37 (TensorFl [(None, 90)])      0
tf_op_layer_ArgMax_38[0][0]
-----

repeat_vector_37 (RepeatVector) (None, 1, 90)      0
tf_op_layer_OneHot_37[0][0]
-----

tf_op_layer_ArgMax_39 (TensorFl [(None,)])          0      dense[99][0]
-----

tf_op_layer_OneHot_38 (TensorFl [(None, 90)])      0
tf_op_layer_ArgMax_39[0][0]
-----

repeat_vector_38 (RepeatVector) (None, 1, 90)      0
tf_op_layer_OneHot_38[0][0]
-----

tf_op_layer_ArgMax_40 (TensorFl [(None,)])          0      dense[100][0]
-----

tf_op_layer_OneHot_39 (TensorFl [(None, 90)])      0
tf_op_layer_ArgMax_40[0][0]
-----

repeat_vector_39 (RepeatVector) (None, 1, 90)      0
tf_op_layer_OneHot_39[0][0]
-----

tf_op_layer_ArgMax_41 (TensorFl [(None,)])          0      dense[101][0]
-----

tf_op_layer_OneHot_40 (TensorFl [(None, 90)])      0
tf_op_layer_ArgMax_41[0][0]

```

```

-----
repeat_vector_40 (RepeatVector) (None, 1, 90)      0
tf_op_layer_OneHot_40[0][0]
-----
tf_op_layer_ArgMax_42 (TensorFl [(None,)])          0      dense[102][0]
-----
tf_op_layer_OneHot_41 (TensorFl [(None, 90)])        0
tf_op_layer_ArgMax_42[0][0]
-----
repeat_vector_41 (RepeatVector) (None, 1, 90)      0
tf_op_layer_OneHot_41[0][0]
-----
tf_op_layer_ArgMax_43 (TensorFl [(None,)])          0      dense[103][0]
-----
tf_op_layer_OneHot_42 (TensorFl [(None, 90)])        0
tf_op_layer_ArgMax_43[0][0]
-----
repeat_vector_42 (RepeatVector) (None, 1, 90)      0
tf_op_layer_OneHot_42[0][0]
-----
tf_op_layer_ArgMax_44 (TensorFl [(None,)])          0      dense[104][0]
-----
tf_op_layer_OneHot_43 (TensorFl [(None, 90)])        0
tf_op_layer_ArgMax_44[0][0]
-----
repeat_vector_43 (RepeatVector) (None, 1, 90)      0
tf_op_layer_OneHot_43[0][0]
-----
tf_op_layer_ArgMax_45 (TensorFl [(None,)])          0      dense[105][0]
-----
tf_op_layer_OneHot_44 (TensorFl [(None, 90)])        0
tf_op_layer_ArgMax_45[0][0]
-----
repeat_vector_44 (RepeatVector) (None, 1, 90)      0
tf_op_layer_OneHot_44[0][0]

```

```

-----
tf_op_layer_ArgMax_46 (TensorFl [(None,)])      0      dense[106][0]
-----
tf_op_layer_OneHot_45 (TensorFl [(None, 90)])    0
tf_op_layer_ArgMax_46[0][0]
-----
repeat_vector_45 (RepeatVector) (None, 1, 90)    0
tf_op_layer_OneHot_45[0][0]
-----
tf_op_layer_ArgMax_47 (TensorFl [(None,)])      0      dense[107][0]
-----
tf_op_layer_OneHot_46 (TensorFl [(None, 90)])    0
tf_op_layer_ArgMax_47[0][0]
-----
repeat_vector_46 (RepeatVector) (None, 1, 90)    0
tf_op_layer_OneHot_46[0][0]
-----
tf_op_layer_ArgMax_48 (TensorFl [(None,)])      0      dense[108][0]
-----
tf_op_layer_OneHot_47 (TensorFl [(None, 90)])    0
tf_op_layer_ArgMax_48[0][0]
-----
repeat_vector_47 (RepeatVector) (None, 1, 90)    0
tf_op_layer_OneHot_47[0][0]
-----
tf_op_layer_ArgMax_49 (TensorFl [(None,)])      0      dense[109][0]
-----
tf_op_layer_OneHot_48 (TensorFl [(None, 90)])    0
tf_op_layer_ArgMax_49[0][0]
-----
repeat_vector_48 (RepeatVector) (None, 1, 90)    0
tf_op_layer_OneHot_48[0][0]
=====
Total params: 45,530
Trainable params: 45,530

```

Non-trainable params: 0

Expected Output

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

Total params: 45,530
Trainable params: 45,530
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 .

```
[29]: x_initializer = np.zeros((1, 1, n_values))  
      a_initializer = np.zeros((1, n_a))  
      c_initializer = np.zeros((1, n_a))
```

Exercise 3 - predict_and_sample

Implement `predict_and_sample()`.

This function takes many arguments, including the inputs `x_initializer`, `a_initializer`, and `c_initializer`.

In order to predict the output corresponding to this input, you'll 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``

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).

```
[30]: # UNQ_C3 (UNIQUE CELL IDENTIFIER, DO NOT EDIT)
# GRADED FUNCTION: predict_and_sample

def predict_and_sample(inference_model, x_initializer = x_initializer,
    ↪a_initializer = a_initializer,
    ↪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, 90), one-hot vector
    ↪initializing the values generation
    a_initializer -- numpy array of shape (1, n_a), initializing the hidden
    ↪state of the LSTM_cell
    c_initializer -- numpy array of shape (1, n_a), initializing the cell state
    ↪of the LSTM_cel

    Returns:
    results -- numpy-array of shape (Ty, 90), matrix of one-hot vectors
    ↪representing the values generated
    indices -- numpy-array of shape (Ty, 1), matrix of indices representing the
    ↪values generated
    """

    n_values = x_initializer.shape[2]

    ### START CODE HERE ###
    # Step 1: Use your inference model to predict an output sequence given
    ↪x_initializer, a_initializer and c_initializer.
    pred = inference_model.predict([x_initializer, a_initializer,
    ↪c_initializer])
    # Step 2: Convert "pred" into an np.array() of indices with the maximum
    ↪probabilities
    indices = np.argmax(pred, axis = -1)
    # Step 3: Convert indices to one-hot vectors, the shape of the results
    ↪should be (Ty, n_values)
    results = to_categorical(indices, num_classes=n_values)
    ### END CODE HERE ###

    return results, indices
```

[31]: `### YOU CANNOT EDIT THIS CELL`

```
results, indices = predict_and_sample(inference_model, x_initializer,
    ↪ a_initializer, c_initializer)

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]) = 62
np.argmax(results[17]) = 89
list(indices[12:18]) = [array([62]), array([88]), array([18]), array([69]),
array([78]), array([89])]
```

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. Perhaps with the following values:
`np.argmax(results[12]) =`
26
`np.argmax(results[17]) =`
7
`list(indices[12:18]) =`
[array([26]), array([18]), array([53]), array([27]), array([40]), array([7])]

3.2 - Generate Music

Finally! You're 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's difficult to generate music that sounds good without it. The post-processing does things like clean up the generated audio by making sure the same sound is not repeated too many times, or 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 model. But this post-processing does make a huge difference, so you should use it in your 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.

[32]: `out_stream = generate_music(inference_model, indices_values, chords)`

Predicting new values for different set of chords.

Generated 32 sounds using the predicted values for the set of chords ("1") and after pruning

Generated 32 sounds using the predicted values for the set of chords ("2") and after pruning

Generated 32 sounds using the predicted values for the set of chords ("3") and after pruning

Generated 31 sounds using the predicted values for the set of chords ("4") and after pruning

Generated 32 sounds using the predicted values for the set of chords ("5") and after pruning

Your generated music is saved in output/my_music.midi

Using a basic midi to wav parser you can have a rough idea about the audio clip generated by this model. The parser is very limited.

```
[33]: mid2wav('output/my_music.midi')
      IPython.display.Audio('./output/rendered.wav')
```

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

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 generated using this algorithm:

```
[34]: IPython.display.Audio('./data/30s_trained_model.wav')
```

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

1.4 Congratulations!

You’ve completed this assignment, and generated your own jazz solo! The Coltranes would be proud.

By now, you’ve:

- Applied an LSTM to a music generation task
- Generated your own jazz music with deep learning
- Used the flexible Functional API to create a more complex model

This was a lengthy task. You should be proud of your hard work, and hopefully you have some good music to show for it. Cheers and see you next time!

What you should remember:

- A sequence model can be used to generate musical values, which are then post-processed into midi music.

- You can use a fairly similar model for tasks ranging from generating dinosaur names to generating original music, with the only 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 $1, \dots, T_x$.

4 - 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](#)

Finally, a shoutout to François Germain for valuable feedback.

[]: