Generation and Improvization of Jazz music using LSTM network

January 19, 2024

[137]:
```
### v1.1
```

[138]:
```
# Importing the necessary packages

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
```

[139]:
```
# This is a snipper of the audio from the training set

IPython.display.Audio('./data/30s_seq.wav')
```

[139]:
```
<IPython.lib.display.Audio object>
```

[140]:
```
# Here musical 'values' are defined as consisting of a pitch and duration. A␣
↪particular key when pressed for a particular
# duration is a musical value.

# The below code loads and preprocesses the raw music data into values
```

```
X, Y, n_values, indices_values, chords = load_music_utils('data/
↪original_metheny.mid')
```

[141]:
```
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
```

[142]:
```
# The shape of X is (m, T_x, 90) where m is the total number of training␣
↪examples, T_x is the number of musical values in 1
# training examples and 90 is the number of unique musical values (represented␣
↪as a one-hot vector).

# The shape of Y is (T_y, m, 90). This is the actual truth output where Y<t+1>␣
↪= X<t>

# Indices_values - A mapping or indexing of music data elements (e.g., a way to␣
↪convert between notes and their
# corresponding indices in a numerical representation).
```

[143]:
```
n_values = 90 # number of music values
n_a = 64 # number of dimensions for the hidden state of each LSTM cell.
reshaper = Reshape((1, n_values))  # Defining the reshaper layer
LSTM_cell = LSTM(n_a, return_state = True) # Defining the LSTM cell layer.␣
↪Please note that LSTM cell depends only on n_a
densor = Dense(n_values, activation='softmax') # The densor layer is to␣
↪calculate the y_hat from the output of the LSTM cell block
```

[144]:
```
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)) # We don't have to specify 'm' while␣
↪defining X. We define it for 1 training example
    # and it is applicable for the whole batch size. It is understood that X is␣
↪of shape (m, T_x, n_values)

    # Define the initial hidden state a0 and initial cell state c0
    # using `Input`
    a0 = Input(shape=(n_a,), name='a0') # It is understood that a0 is of shape␣
↪(m,n_a)
    c0 = Input(shape=(n_a,), name='c0') # It is understood that c0 is of shape␣
↪(m,n_a)
    a = a0 # Intial value of a
    c = c0 # Initial value of c

    outputs = []

    for t in range(Tx):

        # Select the "t"th time step vector from X.
        x = X[:,t,:]
        # Use reshaper to reshape x to be (1, n_values) because LSTM_cell␣
↪expects x to be in (m,1,n_values) format
        x = reshaper(x)
        # Perform one step of the LSTM_cell
        _, a, c = LSTM_cell(inputs=x, initial_state=[a, c])
        # Apply densor to the hidden state output of LSTM_Cell
        out = densor(a)
        # Append the output to "outputs"
        outputs.append(out)

    model = Model(inputs=[X, a0, c0], outputs=outputs) # In outputs, the outer␣
↪most level dimension is time step, then m and
```

```
    # then atlast n_values. That's why we had initially defined Y as (T_y, m,␣
↪n_values)

    return model
```

[145]:
```
# The function djmodel mainly calculates the 'outputs' which is then passed as␣
↪'outputs' to 'Model' and the whole function
# returns the defined model
```

[146]:
```
### YOU CANNOT EDIT THIS CELL

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

[147]:
```
### YOU CANNOT EDIT THIS CELL

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

```
All tests passed!
```

[148]:
```
# Check your model
#model.summary()
```

[149]:
```
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'])
```

[150]:
```
m = 60
a0 = np.zeros((m, n_a))
c0 = np.zeros((m, n_a))
```

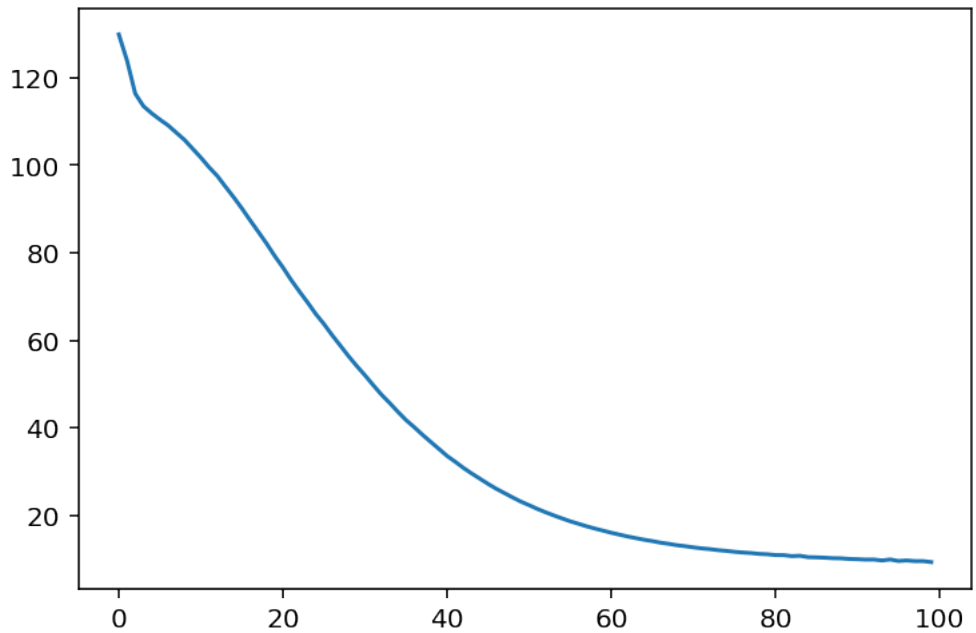
[151]:
```
history = model.fit([X, a0, c0], list(Y), epochs=100, verbose = 0) # list(np.
↪array([[1,2],[3,4],[5,6]])) = [array([1, 2]), array([3, 4]), array([5, 6])]
# Y is of shape (T_y, m, n_values). So list(Y) is a list of T_y elements, each␣
↪of shape (m, n_values)
# This is how we had defined 'outputs' while defining the 'model' using the␣
↪function 'djmodel'

# model.fit trains the weights of the 'LSTM_cell' and 'densor' layers
```

[152]:
```
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: 129.82150268554688
loss at epoch 100: 9.377666473388672
```

[152]: [<matplotlib.lines.Line2D at 0x7f7980612e50>]



[153]:
```
# Note to Self 1

# When you create an instance of 'Model' (which has been imported from␣
↪tensorflow.keras.models),
# you define the 'inputs' to the model and the 'outputs' generated by the model.
↪ You show the model the procedure to
# generate (predict) the 'outputs'.

# When you do model.fit(inputs = inputs, true_output, epochs, etc), here␣
↪'model' is an instance of already defined 'Model'.
# Now this line of code will start generating the 'outputs' as it was defined␣
↪in the 'Model' and will compare the
# predicted 'outputs' with the 'true_output' and update the weights of the␣
↪'LSTM_cell' and 'densor' layers.

# Note - Before you start model.fit(), it is necessary to model.compile() where␣
↪you define the optimization algorithm and
# the loss function to be used

# After the weights of 'LSTM_cell' and 'densor' layers have been trained, we␣
↪are defining another
```

```
# 'Model' (imported from tensorflow.keras.models) and this time we are␣
↪generating the 'outputs' in a different manner.
# In this model, only x<1> is an input and there is no x<2>, x<3>, so on. We␣
↪predict the y<1> from x<1> and then in y<1>,
# we find the index with maximum probability and convert y<1> into a one-hot␣
↪vector and feed that as x<2>.
# This is how we are generating y<1>, y<2>, y<3>,...y<t> in this 'Model'. This␣
↪would be our inference model.

# When you use model.predict(inputs), you only pass in the inputs and the model␣
↪returns the
# generated (predicted) 'outputs' in the manner it was mentioned while defining␣
↪the inference 'Model'.
```

[154]:
```
# Note to Self 2

# In the inference model, why are we feeding the y<1> as x<2>, y<2> as x<3> and␣
↪so on?

# We are predicting the entire sequence and we are being given just x<1>. If␣
↪the 'weights' are the right values (zero loss),
# then the predicted y<1> would be nothing but x<2> and that's why we are␣
↪feeding  y<1> (which is same as x<2>) while
# predicting y<2>.

# The weights have been trained in a such a manner that when input is x<t>, the␣
↪output is y<t> (which is same as x<t+1>
```

[155]:
```
# MUSIC INFERENCE
```

[156]:
```
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 model with a shape
    x0 = Input(shape=(1, n_values)) # Actually of shape (m, 1, n_values)


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

    # Create an empty list of "outputs" to later store your predicted values
    outputs = []

    # Loop over Ty and generate a value at every time step
    for t in range(Ty):
        # Perform one step of LSTM_cell.
        _, a, c = LSTM_cell(inputs=x, initial_state=[a, c])

        # Apply Dense layer to the hidden state output of the LSTM_cell
        out = densor(a)
        # Append the prediction "out" to "outputs". out.shape = (None, 90).
↪Actually
        outputs.append(out)

        x = tf.math.argmax(out, axis=1) # Gives the indice of maximum value
        x = tf.one_hot(x, depth=n_values) # Converts into one-hot
↪representation vector

        # RepeatVector(1) converts x into a tensor with shape=(None, 1, 90)
↪because LSTM_cell accepts x as a tensor of shape (None,1,90)
        x = RepeatVector(1)(x)

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

    return inference_model
```

[157]:
```
### YOU CANNOT EDIT THIS CELL
inference_model = music_inference_model(LSTM_cell, densor, Ty = 50)
```

[158]:
```
### YOU CANNOT EDIT THIS CELL

# UNIT TEST
```

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

All tests passed!

[159]:
```
# Check the inference model
#inference_model.summary()
```

[160]:
```
# Initializing the inputs to be fed into the Inference model

x_initializer = np.zeros((1, 1, n_values))
a_initializer = np.zeros((1, n_a))
c_initializer = np.zeros((1, n_a))

# We are passing just 1 training example
```

[161]:
```
# PREDICTING
```

[162]:
```
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]

    # Using the inference model, we predict an output sequence given␣
↪x_initializer, a_initializer and c_initializer.
    pred = inference_model.predict([x_initializer, a_initializer,␣
↪c_initializer])
```

```
    # The shape of pred is (T_y,1,90)

    # We convert "pred" into an np.array() of indices with the maximum␣
↪probabilities
    indices = np.argmax(pred, axis=2)
    # The shape of 'indices' is (T_y, 1)

    # We convert indices to one-hot vectors
    results = to_categorical(indices, num_classes=n_values)
    # The shape of 'results' is (T_y, n_values = 90)

    return results, indices
```

[163]:
```
# Note to self 3

# The shape of 'pred' is (T_y,1,90). This is because that's how we defined the␣
↪procedure to generate(predict) the
# 'outputs' while defining the inference 'Model'.

# Actually 'pred' is a list of numpy arrays. Each element of 'pred' is of shape␣
↪(1,90) and there are T_y = 50 such elements.
```

[164]:
```
# Example usage of np.argmax() and to_categorical()

indices = np.argmax([[1,2],[3,4],np.array((6,5))], axis = 1)
print(indices.shape)
to_categorical(indices, num_classes=5)

# Note to self 4
# In the above case, we are passing a 2-d array to np.argmax() and so the␣
↪resultant shape is (3,)
# Whereas inside the function 'predict_and_sample', we are passing a 3-d array␣
↪to np.argmax() and hence the resultant shape
# is (50,1)
```

```
(3,)
```

[164]:
```
array([[0., 1., 0., 0., 0.],
       [0., 1., 0., 0., 0.],
       [1., 0., 0., 0., 0.]], dtype=float32)
```

[165]:
```
# Example usage of np.argmax() on a 2-d array

print(np.array([[1,2],[3,4],[5,6]]).shape)
print()
print(np.argmax(np.array([[1,2],[3,4],[5,6]]), axis = 0))
print()
```

```
print(np.argmax(np.array([[1,2],[3,4],[5,6]]), axis = 1))
```

```
(3, 2)

[2 2]

[1 1 1]
```

[166]:
```
# Note to Self 5
# Example of usage of np.argmax() on a 3-dimensional array

array_example = np.array([
    [
        [1, 2, 3, 4],
        [5, 6, 7, 8],
        [9, 10, 11, 12]
    ],
    [
        [13, 14, 15, 16],
        [17, 18, 19, 20],
        [21, 22, 23, 24]
    ]
])

print(array_example.shape)
print()

maximum_0 = np.argmax(array_example, axis = 0)
maximum_1 = np.argmax(array_example, axis = 0)
maximum_2 = np.argmax(array_example, axis = 0)

# In a (2x3x4) array, the first dimension = 2 (no.of layers), second dimension␣
 ↪= 3 (no.of rows) and
# third dimension (no. of columns) = 4

print(maximum_0)
print()
# When axis = 0, it is along the layers direction. First we go from the lower␣
 ↪dimension (columns) to rows (higher dimension).
# But the filling of elements in 'maximum_0' is always from columns to rows to␣
 ↪higher dimension.

print(maximum_1)
print()
# When axis = 1, it is along the rows direction. First we go from the lower␣
 ↪dimension (columns) to layers (higher dimension).
```

```
# But the filling of elements in 'maximum_1' is always from columns to rows to␣
↪higher dimension.

print(maximum_2)
# When axis = 2, it is along the columns direction. First we go from the lower␣
↪dimension (rows) to layers (higher dimension).
# But the filling of elements in 'maximum_2' is always from columns to rows to␣
↪higher dimension.
```

```
(2, 3, 4)

[[1 1 1 1]
 [1 1 1 1]
 [1 1 1 1]]

[[1 1 1 1]
 [1 1 1 1]
 [1 1 1 1]]

[[1 1 1 1]
 [1 1 1 1]
 [1 1 1 1]]
```

[167]:
```
### 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]) = 1
np.argmax(results[17]) = 36
list(indices[12:18]) = [array([1]), array([35]), array([2]), array([14]),
array([57]), array([36])]
```

[168]:
```
print(indices.shape)
print(results.shape)
```

```
(50, 1)
(50, 90)
```

[169]:
```
# GENERATING MUSIC
```

[170]:
```
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 32 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
```

[171]:
```python
# Note to self 6

# Inside the function 'generate_music', there is the function␣
↪'predict_and_sample' which outputs the 'results' and 'indices'.
# Either 'indices' or 'results' can be converted to musical notes using the␣
↪'indices_values'

# This is how generate_music gives the ouput 'out_stream'

# And the generated music is saved in 'output/my_music.midi'
```

[172]:
```python
# 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.

mid2wav('output/my_music.midi')
IPython.display.Audio('./output/rendered.wav')

# The MIDI file ('output/my_music.midi') is the direct output of the music␣
↪generation LSTM model, and
# the WAV file ('rendered.wav') is the rendered audio version of that MIDI␣
↪file, allowing us to listen to the music our
# model has created. The conversion process does not alter the fundamental␣
↪musical content created by the model;
# it simply translates it into a form that can be audibly played back.
```

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

[173]:
```python
# Here is a 30 second audio clip generated using this algorithm

IPython.display.Audio('./data/30s_trained_model.wav')
```

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

[174]:
```python
# Note
# The 2 generated clips are different due to the following reasons :-

# Different Model States: The two clips might be outputs from the model at␣
↪different stages of training.

# Different Input Seeds: If the model generates music based on some initial␣
↪seed or input, variations in this input can lead to
# different outputs.

# Differences in Post-Processing: The process used to convert the MIDI or␣
↪generative output to WAV format might differ in
# terms of instruments used, synthesizer settings, effects applied
```

[175]:
```python
# A sequence model can be used to generate musical values, which are then␣
↪post-processed into midi music.
```