Queen's Smith SCHOOL OF BUSINESS

MUSIC GENERATION USING DEEP LEARNING

TEAM BAY
MMAI 894 – DEEP LEARNING

Contents

Introduction

There has been a significant advancement in the use of artificial intelligence for image processing and text processing tasks with great accuracy in recent years. Similarly, Artificial Intelligence can be used to model human-level creativity in the fields of arts and entertainment industries.

This report will outline a case study focused on generating new music by using deep learning techniques to learn existing music notes. The application of the newly generated music can be used in a variety of different ways. For example, it can assist musicians in generating ideas for new music or even be used to generate new standalone music in interactive voice response (IVR) systems. Additionally, it can work in tandem with computer vision techniques to suggest music depending on mood or emotion, detected by facial recognition. In this case, a mobile application could be developed where the application scans a user's face, identifies their emotion, and creates specific music to align with the user's current mood.

For our purposes, we sought to develop a deep learning model capable of generating new music that is deemed the same quality as human-generated music. In order to assess quality, the intent is that the new machine-generated music is indistinguishable to human-generated music as determined through a survey with various samples of human-created music files and machine-generated music files, where survey participants categorize the music files as either human- or machine-generated after listening. The expectation is that survey response data indicates that it is very difficult to tell human-created and machine-generated music apart.

The objectives of this specific use case can be highly applicable in many business contexts. In particular, given the highly competitive market dynamics of the music streaming industry, this type of capability or application can be highly valuable for companies in this space to differentiate and deliver added value to customers by providing highly customized music to their audience. Additionally, this can enable companies such as Spotify, Amazon, Apple, and others to generate their own content for user consumption without needing to license music from artists or mainstream music labels, reducing costs and providing a unique experience.

In order to develop this use case, we leveraged existing music data to train the deep learning model. The model learns existing patterns in the music data and generated new and meaningful music notes.

Data Collection

Music is a sequence of musical events and can be represented in three formats:

1. Sheet music

It is a sequence of musical notes separated by space that can have notes for single and multiple instruments. Below is a representation of sheet music:

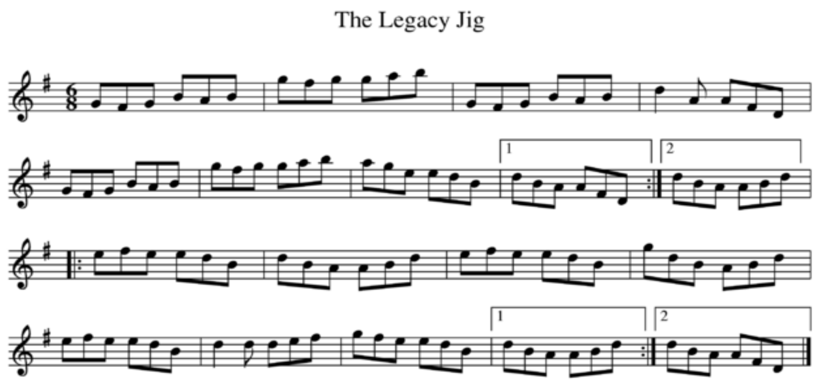


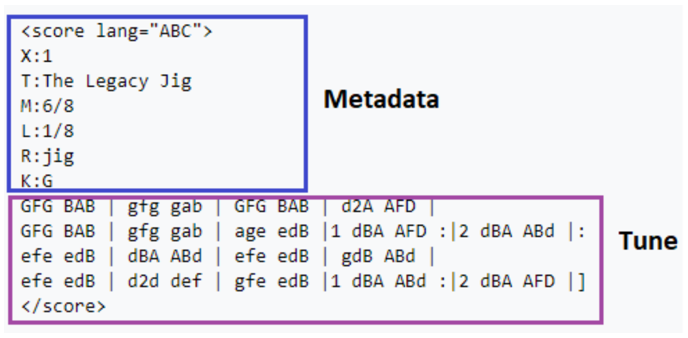
Figure 1: Representation of sheet music

2. <u>MIDI (Music Instruments and Digital Interface)</u>

MIDI is just a series of messages like "note on", "note off", “note/pitch", or "pitch bend”. A MIDI instrument interprets these messages to produce sound. A MIDI instrument can be a piece of hardware (electronic keyboard, synthesizer, etc.) or part of a software environment (Ableton, Garageband, digital performer, logic, etc.).

3. <u>ABC Notation</u>

The music notes are represented in alphabet characters, hence the name “ABC”. There are two parts in ABC-notation; the first part is the metadata of the music notes, which has details like the index, instances with more than one tune in a file (X:), the title (T:), the time signature (M:), the default note length (L:), the type of tune (R:) and the key (K:). Lines following the metadata represent the tune.



```
<score lang="ABC">
X:1
T:The Legacy Jig
M:6/8
L:1/8
R:jig
K:G
GFG BAB | gfg gab | GFG BAB | d2A AFD |
GFG BAB | gfg gab | age edB |1 dBA AFD :|2 dBA ABd |:
efe edB | dBA ABd | efe edB | gdB ABd |
efe edB | d2d def | gfe edB |1 dBA ABd :|2 dBA AFD |]
</score>
```

Figure 2: ABC notation with metadata and tune

For this case study, we used ABC notation of music notes for training our model. There are many open-source datasets available for this type of use-case using the ABC notation. Examples include Sourceforge, Trillian.mit.edu, among many others. ABC notation data in Trillian.mit.edu is presented in individual files which added complexity to our process. As such, we opted to download the data from Sourceforge (http://abc.sourceforge.net/NMD/) as it allowed for download of consolidated data and saved it in a text file (*input.txt*). The input file has 340 music notes.

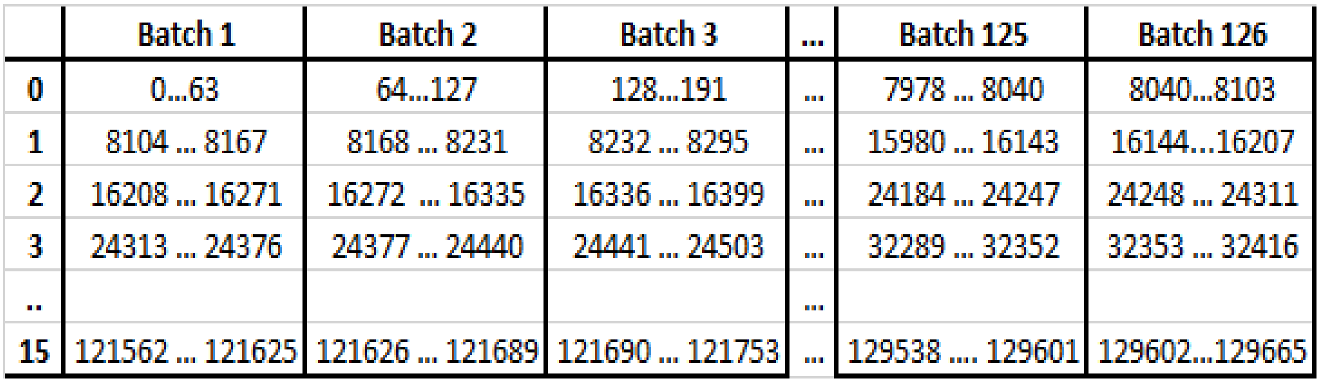
Data Preprocessing

Since the data input is a sequence of characters for the model to train, we used Recurrent Neural Networks (RNN) as they are known to work well with sequential data. The input data that we downloaded was clean, so data cleaning was not required. However, when giving the data as input to the model, we sent them in batches. The reason for dividing the data into batches was because the model learns as it processes each batch in order to achieve optimal weights.

We observed that a batch size of 16 and sequence length 64 characters worked well for our model.

We created a *read_batches()* function to divide the dataset into n-number of batches. In our *input.txt* file, there are 129,665 characters. Below are the details on how batches are divided:

- Total no. of characters = 129665
- Total number of unique characters = 86
- Batch size = 16
- # of characters in a batch = (129665/16) = 8104
- Sequence length in batches = 64
- Total number of batches in each epoch = 8104/64 = 126 batches

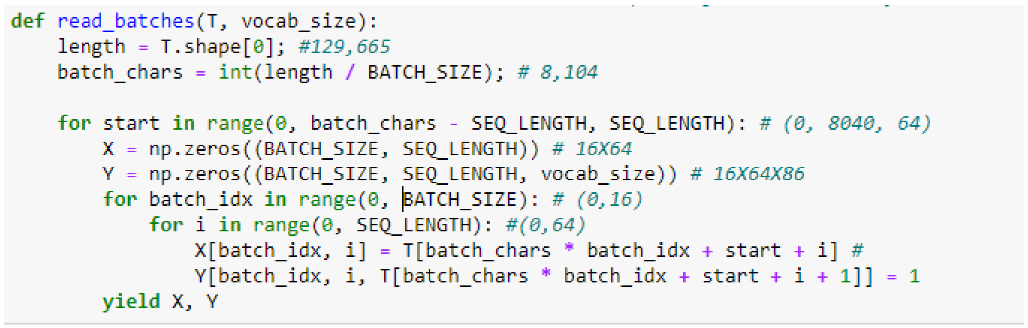


	Batch 1	Batch 2	Batch 3	...	Batch 125	Batch 126
0	0...63	64...127	128...191	...	7978 ... 8040	8040...8103
1	8104 ... 8167	8168 ... 8231	8232 ... 8295	...	15980 ... 16143	16144...16207
2	16208 ... 16271	16272 ... 16335	16336 ... 16399	...	24184 ... 24247	24248 ... 24311
3	24313 ... 24376	24377 ... 24440	24441 ... 24503	...	32289 ... 32352	32353 ... 32416
..				...		
15	121562 ... 121625	121626 ... 121689	121690 ... 121753	...	129538 129601	129602...129665

Figure 3: Batches using 16 batch size and 64 sequence length

We assigned an index to each character in the data and stored this information in a dictionary *char_to_idx*, so that we could train our neural network model.

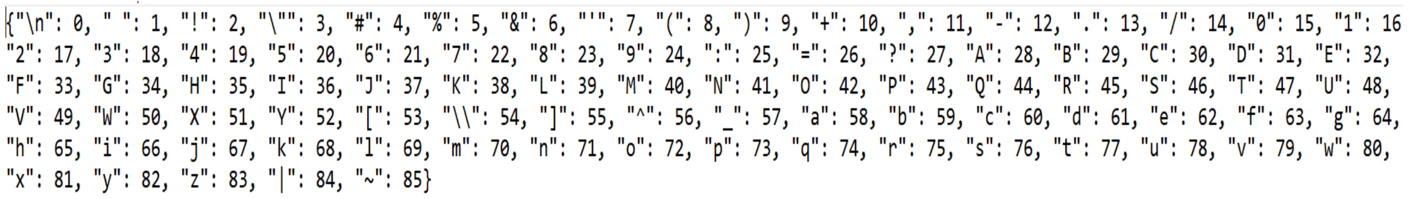
In the below function, we created batches with a sequence length of 64 characters where *X* consists of indices of characters for each of the sequence and *Y* is a tensor which is a one-hot encoded value corresponding to the index of the next character.



```
def read_batches(T, vocab_size):
    length = T.shape[0]; #129,665
    batch_chars = int(length / BATCH_SIZE); # 8,104

    for start in range(0, batch_chars - SEQ_LENGTH, SEQ_LENGTH): # (0, 8040, 64)
        X = np.zeros((BATCH_SIZE, SEQ_LENGTH)) # 16X64
        Y = np.zeros((BATCH_SIZE, SEQ_LENGTH, vocab_size)) # 16X64X86
        for batch_idx in range(0, BATCH_SIZE): # (0,16)
            for i in range(0, SEQ_LENGTH): #(0,64)
                X[batch_idx, i] = T[batch_chars * batch_idx + start + i] #
                Y[batch_idx, i, T[batch_chars * batch_idx + start + i + 1]] = 1
        yield X, Y
```

Figure 4: Read Batches function

Below is a snippet of char_to_indx dictionary:



```
{"\n": 0, " ": 1, "!": 2, "\"": 3, "#": 4, "%": 5, "&": 6, "'": 7, "(": 8, ")": 9, "+": 10, ",": 11, "-": 12, ".": 13, "/": 14, "0": 15, "1": 16
"2": 17, "3": 18, "4": 19, "5": 20, "6": 21, "7": 22, "8": 23, "9": 24, ":": 25, "=": 26, "?": 27, "A": 28, "B": 29, "C": 30, "D": 31, "E": 32,
"F": 33, "G": 34, "H": 35, "I": 36, "J": 37, "K": 38, "L": 39, "M": 40, "N": 41, "O": 42, "P": 43, "Q": 44, "R": 45, "S": 46, "T": 47, "U": 48,
"V": 49, "W": 50, "X": 51, "Y": 52, "[": 53, "\\": 54, "]": 55, "^": 56, "_": 57, "a": 58, "b": 59, "c": 60, "d": 61, "e": 62, "f": 63, "g": 64,
"h": 65, "i": 66, "j": 67, "k": 68, "l": 69, "m": 70, "n": 71, "o": 72, "p": 73, "q": 74, "r": 75, "s": 76, "t": 77, "u": 78, "v": 79, "w": 80,
"x": 81, "y": 82, "z": 83, "|": 84, "~": 85}
```

Figure 5: Snippet of char_to_indx

Model Building

Our model needed to learn from existing characters and generate new character sequences not present in the training data. So, for a given input character, the model has to predict the next character. This use case falls under the many-many type of RNN; below is an output of a many-many RNN. As observed, 'h' is the input to the model, and it predicts the probability if next character being an 'e'

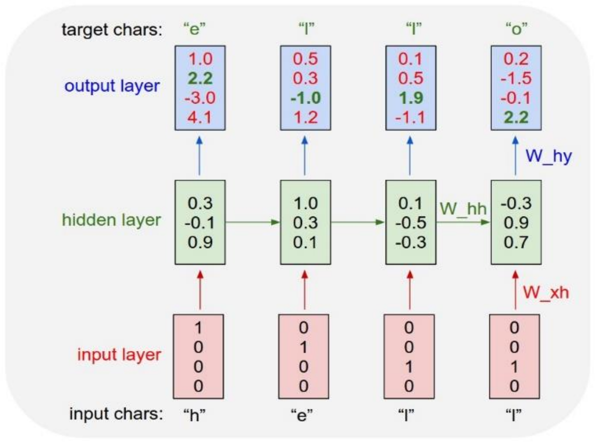
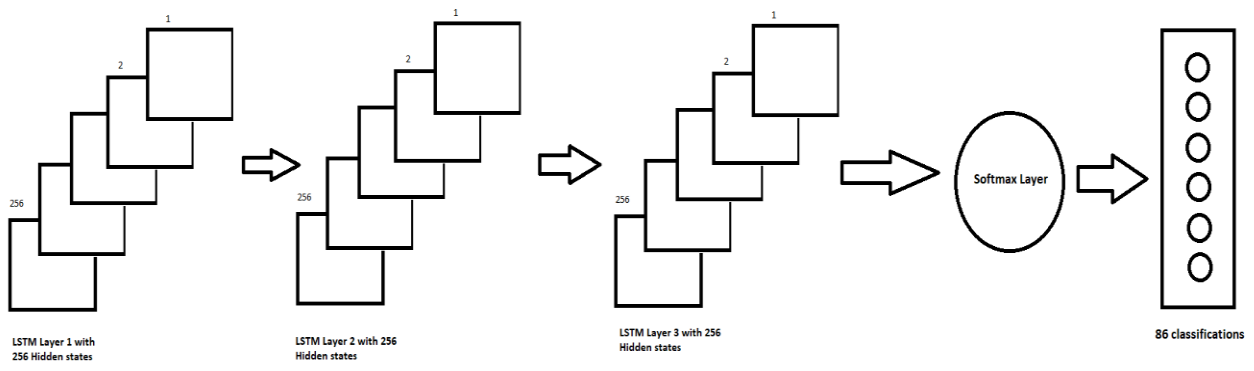


Figure 6: Many-many RNN example

We used a multi stack layered LSTM architecture presented below to train our model. Before deciding on this architecture, we tried multiple iterations with single layer LSTM, with 64 and 256 LSTM units and didn't gave us desired results.



1
2
256
LSTM Layer 1 with 256 Hidden states
1
2
256
LSTM Layer 2 with 256 Hidden states
1
2
256
LSTM Layer 3 with 256 Hidden states
Softmax Layer
86 classifications

Figure 7: LSTM architecture

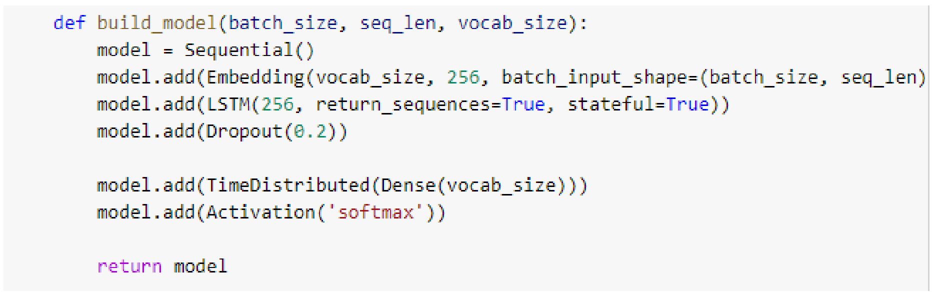
1. <u>Iteration 1 – Single LSTM layer</u>

We updated the *build_model()* function with a single LSTM layer and 256 units. After training the model with this architecture, even though the training loss was lower as the number of epochs increased, the model's new character sequence didn't give desired results. When we played the music notes of new character sequence generated, there was some noise. The output generated was as follows:

d/2c/2|"G"B2G G2B|"G"d2d d2d|"C"e2d ced|"C"efg "G"dBG|"Em"Ged "D7"cAG|"G"G3 - G2:|

A|"D"FED FAd|"Em"cde|"G"dcB "D"AFD|"G"GFG "A7"ABc|"D"dcB

The identified noise can be observed by listening to the music notes using the following web site: https://www.abcjs.net/abcjs-editor.html

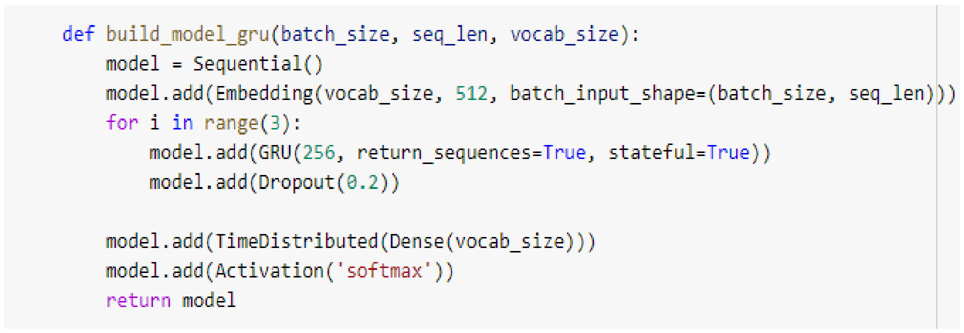


```
def build_model(batch_size, seq_len, vocab_size):
    model = Sequential()
    model.add(Embedding(vocab_size, 256, batch_input_shape=(batch_size, seq_len)
    model.add(LSTM(256, return_sequences=True, stateful=True))
    model.add(Dropout(0.2))

    model.add(TimeDistributed(Dense(vocab_size)))
    model.add(Activation('softmax'))

    return model
```

Figure 8: Code for iteration 1

2. Iteration 2 – Multi stack GRU layers

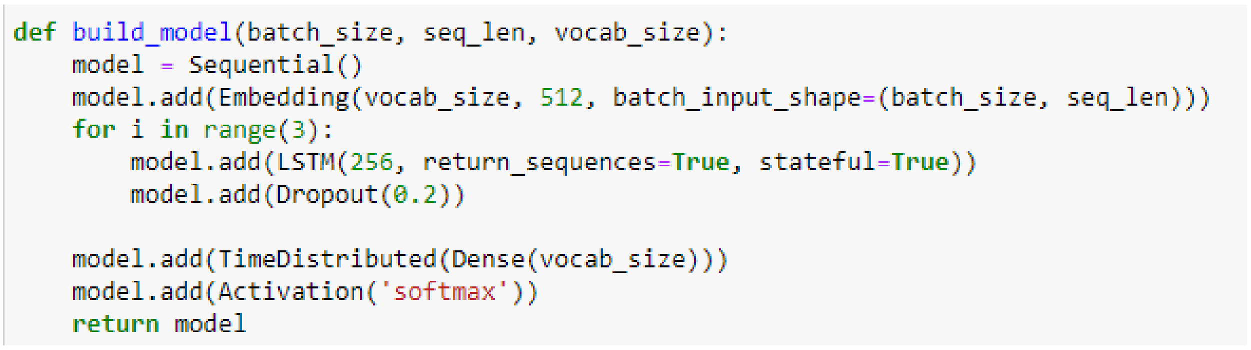
For iteration 2, we replaced the main model’s stacked LSTM layers with stacked GRU layers with same number of units (256). After training the model with this architecture, we ran the sample generator function. The music from new character sequence generated was almost similar to that of stack LSTM layers with similar shapes.



```
def build_model_gru(batch_size, seq_len, vocab_size):
    model = Sequential()
    model.add(Embedding(vocab_size, 512, batch_input_shape=(batch_size, seq_len)))
    for i in range(3):
        model.add(GRU(256, return_sequences=True, stateful=True))
        model.add(Dropout(0.2))

    model.add(TimeDistributed(Dense(vocab_size)))
    model.add(Activation('softmax'))
    return model
```

Figure 9: Code for iteration 2

The multi layered LSTM architecture we used to training our model has multiple LSTM units (256). Same input goes to all the LSTM units, where each unit will learn different aspects of the input characters. The output of each LSTM cells goes to the next timestamp. The below figure outlines the code used to build our model and we have summarized the key parameters within.



```
def build_model(batch_size, seq_len, vocab_size):
    model = Sequential()
    model.add(Embedding(vocab_size, 512, batch_input_shape=(batch_size, seq_len)))
    for i in range(3):
        model.add(LSTM(256, return_sequences=True, stateful=True))
        model.add(Dropout(0.2))

    model.add(TimeDistributed(Dense(vocab_size)))
    model.add(Activation('softmax'))
    return model
```

Figure 10: Code used to build model

- <u>Return_sequences=True</u>

By default, this parameter is false for LSTM. If we make it True, for every LSTM unit, we have to set this parameter to True if we want it to generate an output. Usually, for many-many architectures, this parameter is true.

- Stateful=True

The default value is False; if True, then we are stating that the output of 'Batch 1_First row' should go as an input to 'Batch 2_First row'. Essentially, the RNN will learn data from 0 to 8103 with continuity.

- TimeDistributed Dense layer

After every timestep, we use this parameter if we want to create a dense layer. It is similar to an MLP with 86 neurons (i.e. vocab_size).

- Softmax layer

At each timestep, the softmax layer will generate 86 classifications.

- Embedding layer

The embedding layer encodes the input sequence into a sequence of dense vectors of 512 dimensions.

- Dropout layer

This layer is used to avoid overfitting. It is present after each LSTM layer.

Model Training

Before the model is trained with input data, we have added some steps to simplify model training. Any neural network or machine learning algorithms work with numbers as input rather than characters. Hence, we formatted our input data which is a sequence of characters to integers and stored in *char_to_indx* dictionary.

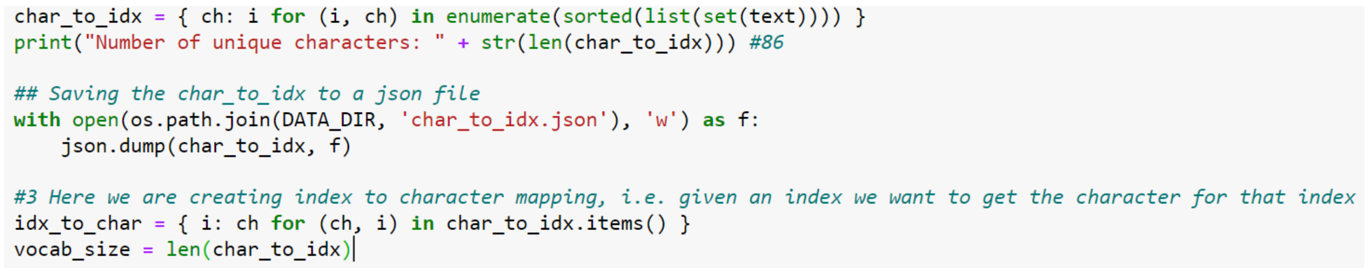
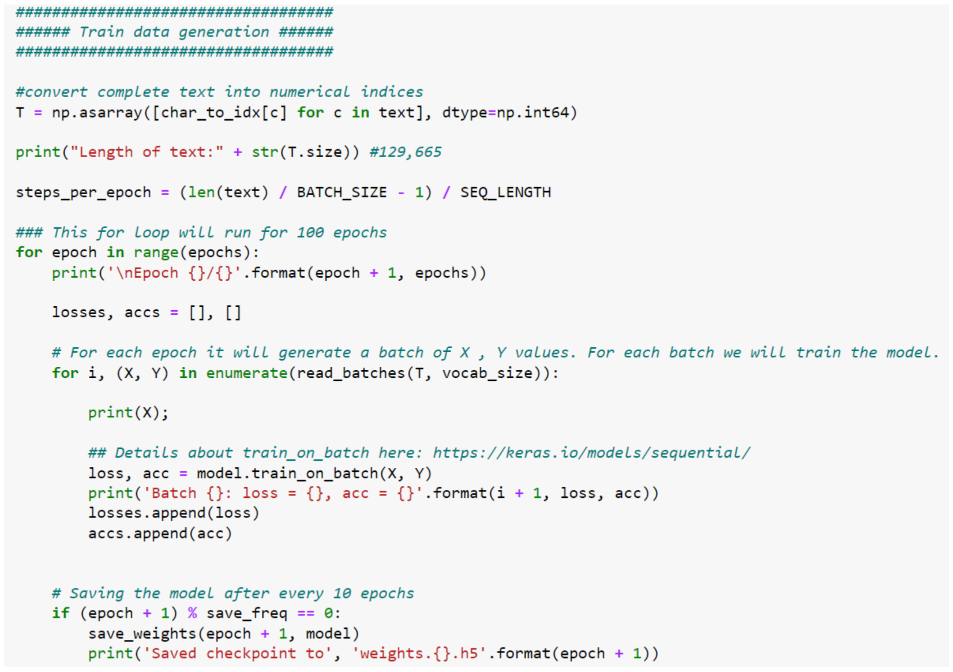

```
char_to_idx = { ch: i for (i, ch) in enumerate(sorted(list(set(text)))) }
print("Number of unique characters: " + str(len(char_to_idx))) #86

## Saving the char_to_idx to a json file
with open(os.path.join(DATA_DIR, 'char_to_idx.json'), 'w') as f:
    json.dump(char_to_idx, f)

#3 Here we are creating index to character mapping, i.e. given an index we want to get the character for that index
idx_to_char = { i: ch for (ch, i) in char_to_idx.items() }
vocab_size = len(char_to_idx)
```

Figure 11: Code used to encode characters to indices.

After the dictionary is created, the next step is to convert the input data to a sequence of numbers. As seen in the below screenshot, the *text* variable has the input data. A for loop is used to convert each character in the text data to an array of character indexes. The *steps_per_epoch* variable will have the number of batches the code has to execute for each epoch; in our case, there are 126 batches. Now text data is sent to the *read_batches()* function, where the batches are created and the model is trained for each batch.

Below is a sample out for each epoch-batch. *X* is a numpy array of character indexes of input data and the *train()* function is used to train our LSTM model. After every 10 epochs, the model is saved along with its respective weights in 'model' folder. This is done because when we generate new character sequences, we will be selecting the weights corresponding to the epoch with less training loss for better predictions.



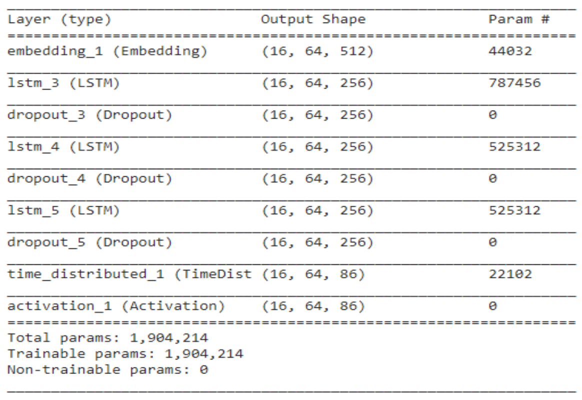
```
####################################
###### Train data generation ######
####################################

#convert complete text into numerical indices
T = np.asarray([char_to_idx[c] for c in text], dtype=np.int64)

print("Length of text:" + str(T.size)) #129,665

steps_per_epoch = (len(text) / BATCH_SIZE - 1) / SEQ_LENGTH

### This for loop will run for 100 epochs
for epoch in range(epochs):
    print('\nEpoch {}/{}'.format(epoch + 1, epochs))

    losses, accs = [], []

    # For each epoch it will generate a batch of X , Y values. For each batch we will train the model.
    for i, (X, Y) in enumerate(read_batches(T, vocab_size)):

        print(X);

        ## Details about train_on_batch here: https://keras.io/models/sequential/
        loss, acc = model.train_on_batch(X, Y)
        print('Batch {}: loss = {}, acc = {}'.format(i + 1, loss, acc))
        losses.append(loss)
        accs.append(acc)


    # Saving the model after every 10 epochs
    if (epoch + 1) % save_freq == 0:
        save_weights(epoch + 1, model)
        print('Saved checkpoint to', 'weights.{}.h5'.format(epoch + 1))
```

Figure 12: Code showing model is saved for every 10 epochs

Finally, the weights of the model will be updated through a backpropagation algorithm, and the model will learn to output a word.

Model Summary

Below is a summary of the model outlining the layer types, output shapes, and number of parameters as well as a plot which visualizes the structure of the network:



```
Layer (type)                 Output Shape              Param #
=================================================================
embedding_1 (Embedding)      (16, 64, 512)             44032
_________________________________________________________________
lstm_3 (LSTM)                (16, 64, 256)             787456
_________________________________________________________________
dropout_3 (Dropout)          (16, 64, 256)             0
_________________________________________________________________
lstm_4 (LSTM)                (16, 64, 256)             525312
_________________________________________________________________
dropout_4 (Dropout)          (16, 64, 256)             0
_________________________________________________________________
lstm_5 (LSTM)                (16, 64, 256)             525312
_________________________________________________________________
dropout_5 (Dropout)          (16, 64, 256)             0
_________________________________________________________________
time_distributed_1 (TimeDist (16, 64, 86)              22102
_________________________________________________________________
activation_1 (Activation)    (16, 64, 86)              0
=================================================================
Total params: 1,904,214
Trainable params: 1,904,214
Non-trainable params: 0
_________________________________________________________________
```

Figure 13: Model summary

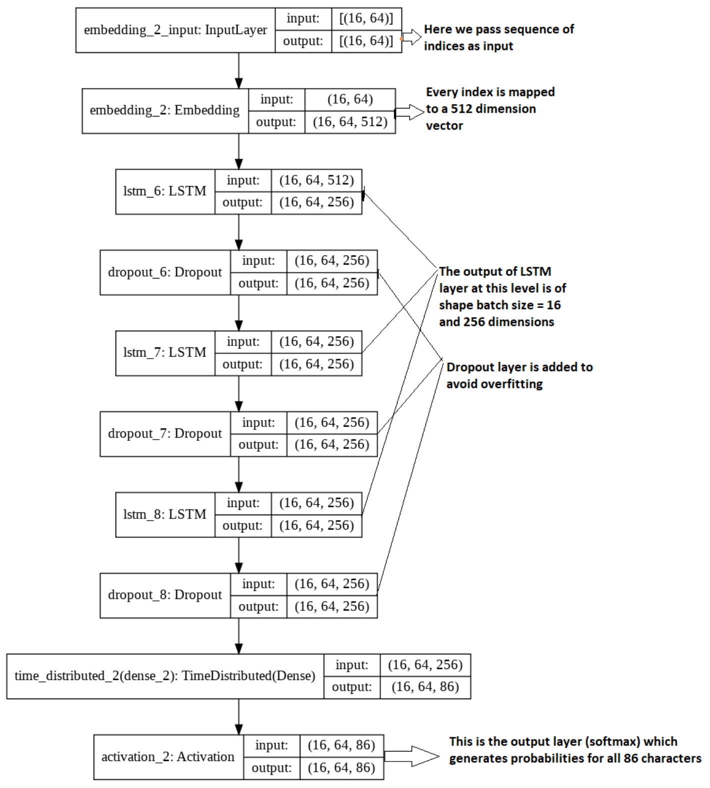

embedding_2_input: InputLayer | input: [(16, 64)] | output: [(16, 64)]
Here we pass sequence of indices as input
embedding_2: Embedding | input: (16, 64) | output: (16, 64, 512)
Every index is mapped to a 512 dimension vector
lstm_6: LSTM | input: (16, 64, 512) | output: (16, 64, 256)
dropout_6: Dropout | input: (16, 64, 256) | output: (16, 64, 256)
The output of LSTM layer at this level is of shape batch size = 16 and 256 dimensions
lstm_7: LSTM | input: (16, 64, 256) | output: (16, 64, 256)
Dropout layer is added to avoid overfitting
dropout_7: Dropout | input: (16, 64, 256) | output: (16, 64, 256)
lstm_8: LSTM | input: (16, 64, 256) | output: (16, 64, 256)
dropout_8: Dropout | input: (16, 64, 256) | output: (16, 64, 256)
time_distributed_2(dense_2): TimeDistributed(Dense) | input: (16, 64, 256) | output: (16, 64, 86)
activation_2: Activation | input: (16, 64, 86) | output: (16, 64, 86)
This is the output layer (softmax) which generates probabilities for all 86 characters

Figure 14: Plot of network structure

Performance of model training

As our model generates new character sequence, there won't be any validation or test datasets. Hence, we will have only training loss and accuracy to assess our model performance. Below are the plots for training loss and accuracy respectively. From the graphs, we can observe that as the number of epochs increases, the loss has reduced and accuracy improved, during the later stages of training, since the model moves towards convergence.

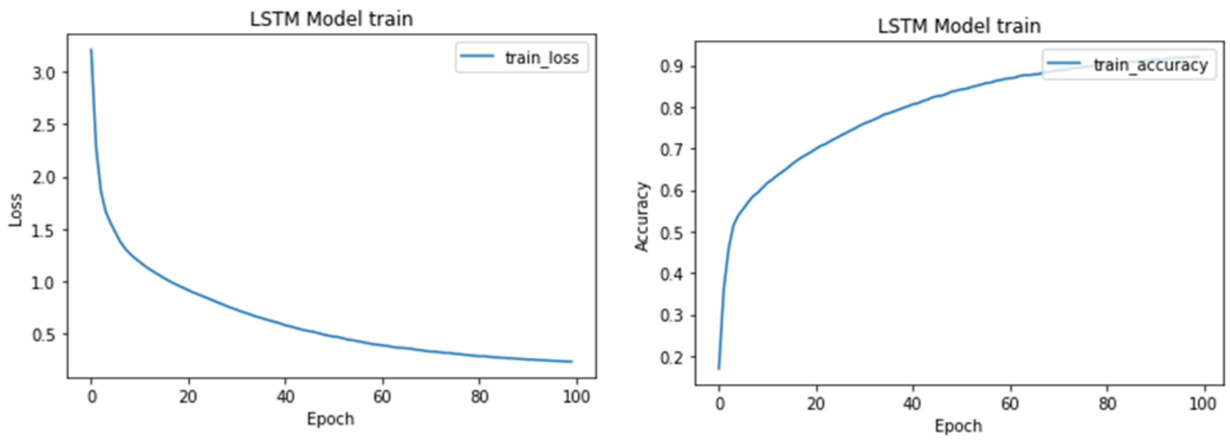
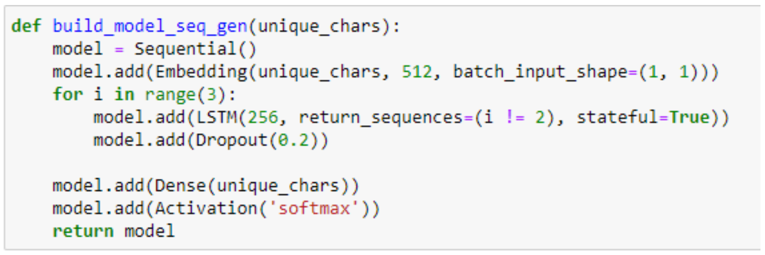


Figure 15: Training loss and accuracy plots

New Music Generation

Our model was trained for a set of characters with a batch size of 16. However, to generate new music, we will input a single character. Hence, we created a model similar to the trained model, except that *vocab_size* is a single character. The *build_model_seq_gen()* function is the architecture of a new model to generate characters' sequence for a given input character.

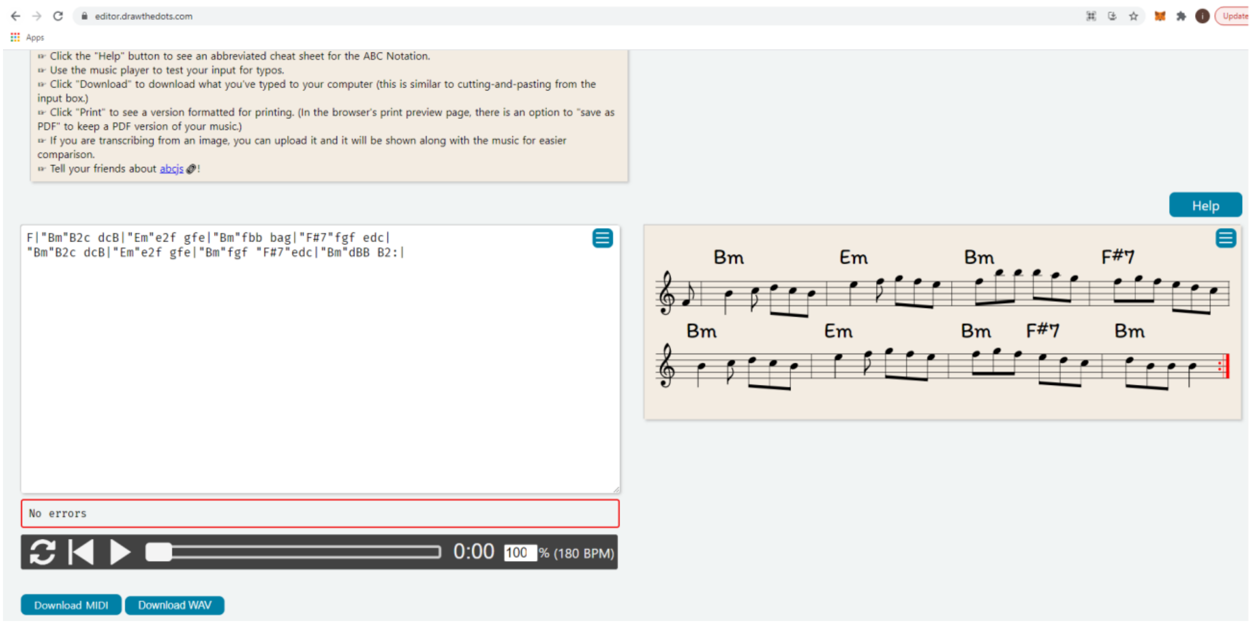


```
def build_model_seq_gen(unique_chars):
    model = Sequential()
    model.add(Embedding(unique_chars, 512, batch_input_shape=(1, 1)))
    for i in range(3):
        model.add(LSTM(256, return_sequences=(i != 2), stateful=True))
        model.add(Dropout(0.2))

    model.add(Dense(unique_chars))
    model.add(Activation('softmax'))
    return model
```

Figure 16: Build_model_seq_gen() function

The next task is to pick optimal weights generated by our trained models. These details were saved in model folder *weights.{epoch number}.h5*. These weights were used to run our new sequence generator model. The output will be an array of probability scores for each character. From this array, a new sequence will be printed randomly using probability scores. The *sample_seq_generator()* function is used to achieve this task.

Once the new music is generated in ABC notation, we can listen to the music by using the *Draw the Dots* web site to play the sequence (https://editor.drawthedots.com/)


editor.drawthedots.com

- Click the "Help" button to see an abbreviated cheat sheet for the ABC Notation.
- Use the music player to test your input for typos.
- Click "Download" to download what you've typed to your computer (this is similar to cutting-and-pasting from the input box.)
- Click "Print" to see a version formatted for printing. (In the browser's print preview page, there is an option to "save as PDF" to keep a PDF version of your music.)
- If you are transcribing from an image, you can upload it and it will be shown along with the music for easier comparison.
- Tell your friends about abcjs!

```
F|"Bm"B2c dcB|"Em"e2f gfe|"Bm"fbb bag|"F#7"fgf edc|
"Bm"B2c dcB|"Em"e2f gfe|"Bm"fgf "F#7"edc|"Bm"dBB B2:|
```

```
No errors
```

Figure 17: Draw the dots web site

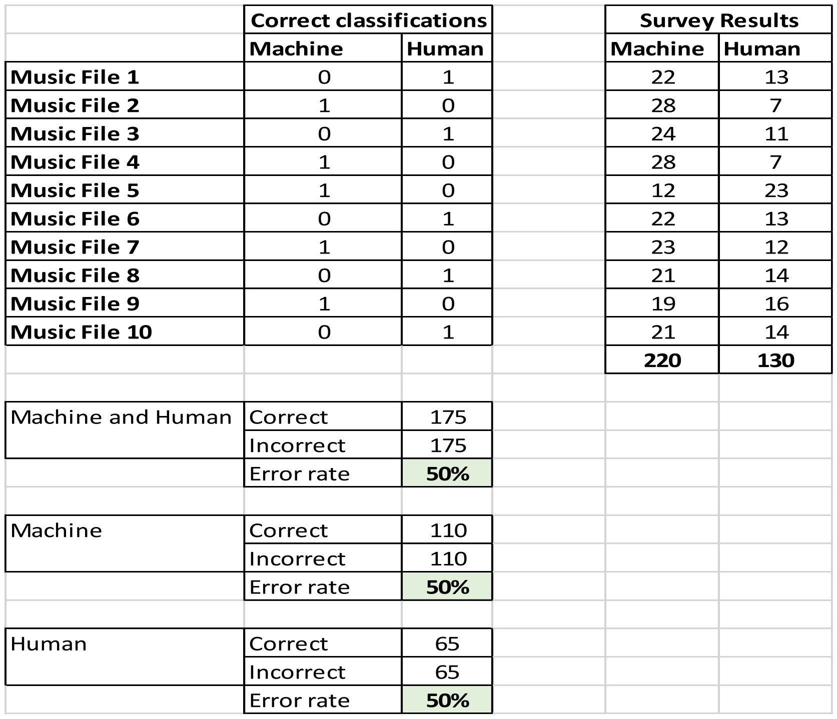
Model Evaluation

As our model is a new sequence generator model, we cannot evaluate by using test data. The new sequence generator is random, and in our case, the model generates new music notes that don't exist and we can't evaluate using test data. To assess our model's performance, we have created a survey to differentiate the music files between those that are machine-generated or that are played by real humans.

We asked participants to listen to 10 music files which has both machine generated and human generated music and classify them in their respective categories. As seen in the below figure, music files under category *Human=1* are taken from our input data and music files under category *Machine=1*, is the output generated by the model. In total, we had received 350 responses from MMAI 2021 cohort, of which survey respondents were able to classify 50% of music samples correctly with an error rate of 50%. Interestingly, our data shows that the participants were biased towards categorizing the music files as machine generated (Machine=220, Humans=130). This has shown that the music generated by the model is indistinguishable from human-created music.

Survey link: https://forms.office.com/r/di7dcMGtkn

SoundCloud: https://soundcloud.com/vamsi-b-338412141/sets/music-generation-using-deep-learning



	Correct classifications			**Survey Results**	
	Machine	**Human**		**Machine**	**Human**
Music File 1	0	1		22	13
Music File 2	1	0		28	7
Music File 3	0	1		24	11
Music File 4	1	0		28	7
Music File 5	1	0		12	23
Music File 6	0	1		22	13
Music File 7	1	0		23	12
Music File 8	0	1		21	14
Music File 9	1	0		19	16
Music File 10	0	1		21	14
				220	**130**

Machine and Human	Correct	175
	Incorrect	175
	Error rate	**50%**
Machine	Correct	110
	Incorrect	110
	Error rate	**50%**
Human	Correct	65
	Incorrect	65
	Error rate	**50%**

Figure 18: Survey results

For evaluation of this in a real world application such as the one we identified for the music streaming industry, we would look to metrics such as click through rate, conversion rate, rate that the file was repeatedly listened to, etc. to assess whether the machine-generated music was preferred over human-created content or was liked in general.

Productionizing Music Generation Model

This model is trained for music notes of a single instrument. However, in the real-world, musicians often use several instruments when composing music. As such, this model may fail to learn music notes of multiple instruments at once.

Also, to train the model, GPUs are required which may be a significant additional expense. For an input data of 340 music notes, training time was 10 mins with *12GB NVIDIA Tesla K80 GPU*, and without GPU it took 1.5 hours. To generate a new music, it takes approximately 10 seconds to generate a new music file of 40 seconds length. In a real world setting, there will be thousands and even millions of music tones. To process that vast repository of content, a system with very high configuration will be required.

The above concerns can be addressed by hosting the music files on the cloud, and if the music industry can collaborate with FANG (Facebook, Amazon, Netflix, Google) companies, then applications can be developed to reduce the development and training time. Further, as we are positioning this use case for the streaming music industry, which includes many top tech companies, there would be easy integration to the cloud to alleviate the concerns noted above.

Future work

The use case was trained using a single music instrument i.e. piano. In future the model can be extended for other music instruments like violin, table etc., and for music played with multiple instruments.   For this project, we focused on ABC notation dataset, there are other representations of music, for example MIDI format, can be converted to ABC notation or vice versa so that more data can be leveraged for training the model.

Implementing transfer learning techniques like Transformers or Generative Pre-trained Transformer (GPT) can also help in building a robust model. Using LSTM or GRU models can not see the whole ABC sequence, whereas a GPT model has a content window of 1024 which is much longer than almost every ABC sequence.

Additionally, this capability can work in tandem with computer vision techniques to suggest music depending on mood or emotion, detected by facial recognition. In this case, a mobile application could be developed where the app scans a user's face, identifies their emotion, and creates specific music to align with the user's current mood.

Conclusion

In conclusion, we developed a model to create new machine-generated music which we has been determined as being indistinguishable from human-created music based on the 350 survey responses we received. This was seen through the 50% error rate among survey participant responses indicating that a random guess is equally likely to determine whether the music file was created by a human or a model.

As such, our model performed well given this business objective and confirms feasibility for companies in the streaming music industry to offer this as a new service in order to differentiate themselves within the market and potentially attract greater market share. The ability to create machine-generated music can allow companies such as Spotify, Google, Amazon, Apple and other tech players, to generate their own content to reduce content licensing costs, reduce dependency on artists and music labels, attract and retain customers through customized music, and provide additional value through unique experiences.

References

Andrej Karpathy - Unreasonable Effectiveness of RNN

http://karpathy.github.io/2015/05/21/rnn-effectiveness/

Christopher, Olah. "Understanding LSTM Networks," August 27, 2015.

http://colah.github.io/posts/2015-08-Understanding-LSTMs/

Silipo, Rosaria. "Text Encoding: A Review," November 21,

2019. https://towardsdatascience.com/text-encoding-a-review-7c929514cccf.