Melody Generation using LSTM RNN

VI^{th} Semester Mini Project

By

ABHINAV MISHRA	-	IIM2014003
KSHITIJ TRIPATHI	-	ISM2014501
LAKSHAY GUPTA	-	IIT2014044
SWARNIMA DIKSHIT	-	IWM2014003
UTKARSH SAHU	-	IIT2014080

Indian Institute Of Information Technology
ALLAHABAD

A project report submitted to the Indian Institute Of Information Technology, Allahabad in accordance with the requirements of 6th semester mini project.

Under Guidance Of
DR. K.P.SINGH

June 26, 2017

Abstract

Recent work in deep machine learning has led to more powerful artificial neural network designs, including Recurrent Neural Networks (RNN) that can process input sequences of arbitrary length. We focus on a special kind of RNN known as a Long-Short-Term-Memory (LSTM) network. LSTM networks have enhanced memory capability, creating the possibility of using them for learning and generating music and language. This thesis focuses on Melody generation using LSTM networks. For music generation, an LSTM model in implemented using char-RNN model given by Andrez Karpathy in the Python programming language, using the Numpy library. I collected a data set of 12000 songs in abc notation, to serve as the LSTM training input. The network learns a probabilistic model of sequences of musical notes from the input data that allows it to generate new melodies. We have also tried to give a rating measure to the generated melodies by using KL Divergence as well as giving rating using probabilistic measure. For both implementations, I discuss the overall performance, design of the model, and adjustments made in order to improve performance.

DEDICATION AND ACKNOWLEDGEMENTS

We would like to extend our sincere thanks to DR. K.P.SINGH for guiding us throughout the journey of our project. We are deeply indebted to our mentor for suggesting us our project topic MELODY GENERATION USING LSTM-RNN which we found really interesting to work on. He was a constant source of assistance and ideas.

CANDIDATE'S DECLARATION

We declare that the work in this project was carried out in accordance with the requirements of the Institute's Regulations and Code of Practice and that it has not been submitted for any other academic award. Except where indicated by specific reference in the text, the work is the candidate's own work. Work done in collaboration with, or with the assistance of, others, is indicated as such. Any views expressed in the report are genuine.

ABHINAV MISHRA	-	IIM2014003
KSHITIJ TRIPATHI	-	ISM2014501
LAKSHAY GUPTA	-	IIT2014044
SWARNIMA DIKSHIT	-	IWM2014003
UTKARSH SAHU	-	IIT2014080

SIGNED: ..... DATE: .....

CERTIFICATE FROM SUPERVISOR

This is to certify that the statement made by the candidates is correct to the best of my knowledge and belief. The project titled "MELODY GENERATION USING LSTM-RNN" is a record of candidate's work carried out by them under my guidance and supervision. I do hereby recommend that it should be accepted in the fulfillment of the requirements of the VI^{th} semester mini project at IIIT Allahabad.

DR. K.P.SINGH

SIGNED: ..... DATE: .....

TABLE OF CONTENTS

List of Figures

Figure **Page**

CHAPTER 1

INTRODUCTION

Music is the ultimate language. Many amazing composers throughout history have composed pieces that were both creative and deliberate. Composers such as Bach and Mozart were well known for being very precise in crafting pieces with a great deal of underlying musical structure. Is it possible then for a computer to also learn to create such musical structure? Algorithmic composition has been tried throughout history. Earliest methods include using a dice and previous compositions and then selecting a note based on the dice throw. This algorithm was suggested by Mozart in the 18th century.

1.1 Motivation

In recent years neural networks have become widely popular and are often mentioned along with terms such as machine learning, deep learning, data mining, and big data. Deep learning methods perform better than traditional machine learning approaches on virtually every single metric. From Google's DeepDream that can learn an artist's style, to AlphaGo learning an immensely complicated game as Go, the programs are capable of learning to solve problems in a way our brains can do naturally. To clarify, deep learning, first recognized in the 80's, is one paradigm for performing machine learning. Unlike other machine learning algorithms that rely on hard-coded feature extraction and domain expertise, deep learning models are more powerful because they are capable of automatically discovering representations needed for detection or classification based on the rawdata they are fed. [7] For this thesis, we focus on a type of

machine learning technique known as artificial neural networks. When we stack multiple hidden layers in the neural networks, they are considered deep learning. Before diving into the architecture of LSTM networks, we will begin by studying the architecture of a regular neural network, then touch upon recurrent neural network and its issues, and how LSTMs resolve that issue.

1.2 Feedforward Neural Networks

Neural network is a machine learning technique inspired by the structure of the brain. The basic foundational unit is called a neuron. Every neuron accepts a set of inputs and each input is given a specific weight. The neuron then computes some function on the weighted input. Functions performed throughout the network by the neurons include both linear and nonlinear – these nonlinear functions are what allow neural networks to learn complex nonlinear patterns. Nonlinear functions include sigmoid, tanh, ReLU, and Elu; these functions have relatively simple derivatives, which is an important characteristic that will be discussed later in this section. Whatever value the functions computes from the weighted inputs are the outputs of the neuron that are then transmitted as inputs to succeeding neuron(s). The connected neurons then form a network, hence the name neural network. The basic structure of a neural network consists of three types of layers: input layer, hidden layer, and output layer. The diagram below is an example of a neural network's structure. Figure 1.1 shows an example ANN.

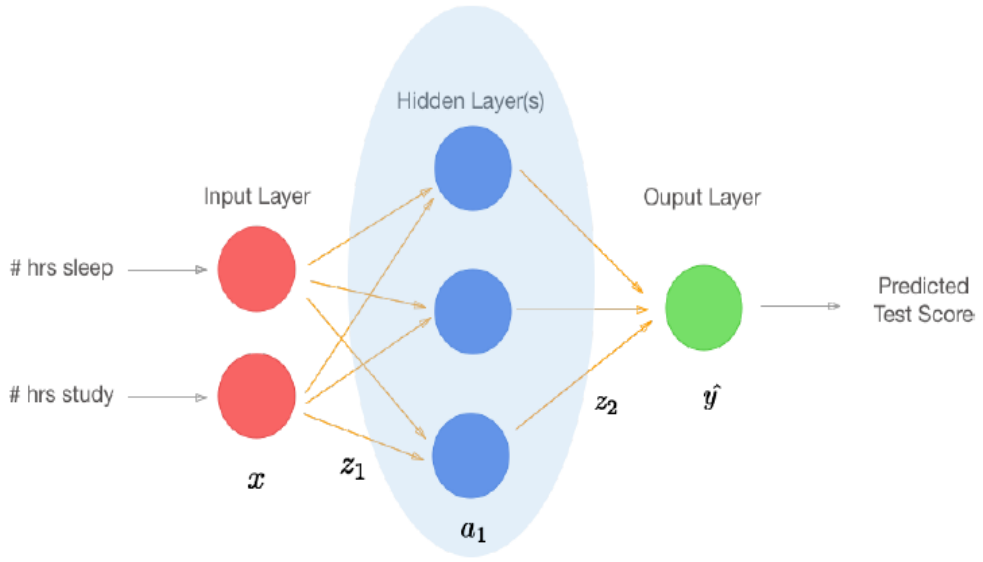


Figure 1.1: An example of a neural network

1.2.1 Forward Propagation

The first step in a neural network is the forward propagation. Given an input, the network makes a prediction on what the output would be. To propagate the input across the layers, we perform functions like that of below:

$$z_1 = xW_1 + b_1$$
$$a_1 = \tanh(z_1)$$
$$z_2 = a_1W_2 + b_2$$
$$\hat{y} = softmax(z_2)$$

The equations z1, z2 are linear functions with x as input and W, b are weights and biases. The a1 in the hidden linear performs a nonlinear activation function tanh. The tanh function takes in the inputs z1 and output values in the range of [-1. 1]. In general, the activation function condenses very large or very small values into a logistic space, and their relatively simple derivatives allow for gradient descent to be workable in backpropagation. In the output layer, we perform the softmax function that turns the values of z2 into a probability distribution where the highest value is the predicted output. The equations above are the steps that occur in the forward propagation. The next step is backpropagation, which is where the actual learning happens.

1.2.2 Backpropagation

Backpropagation is a way of computing gradients of expressions through recursive application of chain rule. [11] The backpropagation involves two steps: calculating the loss, and performing a gradient descent. We calculate the loss L by cross entropy loss to determine how off our predicted output is from the correct output ... We typically think of the input .. as given and fixed, and the weights and biases as the variables that we are able to modify. Because we randomly initialize the weights and biases, we expect our losses to be high at first. The goal of training is to adjust these parameters iteration by iteration so that eventually the loss is minimized as much as possible. We need to find the direction in which the weight-space improves the weight vector and minimizes our loss is the gradient descent. The gradient descent is an optimization function that adjusts weights according to the error. The gradient is another word for slope. The slope describes the relationship between the network fs error and a single weight, as in how much the error changes as the weight is adjusted. The relationship between network error and each of those weights is a derivative, !" !", which measures the degree to which a slight change in a weight causes a slight change in the error. [10] The weights are represented in matrix form in the network, and each weight matrix passes through activations and sums over several layers. Therefore, in order to find the derivative we need to use the chain rule to calculate the derivative of the error in relation to the weights. If we were to

apply the backpropagation formula for the equations listed from the forward propagation section, we have the following derivatives for the weights in respect to the loss:

$$z_1 = xW_1 + b_1$$
$$a_1 = \tanh(z_1)$$
$$z_2 = a_1 W_2 + b_2$$
$$\hat{y} = softmax(z_2)$$

$$\frac{\delta L}{\delta W_1} = \frac{\delta L}{\delta z_2} \cdot \frac{\delta z_2}{\delta a_1} \cdot \frac{\delta a_1}{\delta z_1} \cdot \frac{\delta z_1}{\delta W_1} = x^T \cdot (1 - tanh^2 z_1) \cdot \hat{y} - y \cdot W_2^T$$

$$\frac{\delta L}{\delta W_2} = \frac{\delta L}{\delta z_2} \cdot \frac{\delta z_2}{\delta W_2} = a_1^T \cdot \hat{y} - y$$

We continually adjust the model's weights in response to the error it produces iteration after iteration until the error can no longer be reduced.

1.3 Recurrent Neural Networks

1.3.1 Forward Propagation

While traditional feedforward neural networks perform well in classification tasks, they are limited to looking at individual instances rather than analyzing sequences of inputs. Sequences can be of arbitrary length, have complex time dependencies and patterns, and have high dimensionality. Some examples of sequential datasets include text, genomes, music, speech, text, handwriting, change of price in stock markets, and even images, which can be decomposed into a series of patches and treated as a sequence. [10] Recurrent neural networks are built upon neurons like feedforward neural networks but have additional connections between layers.

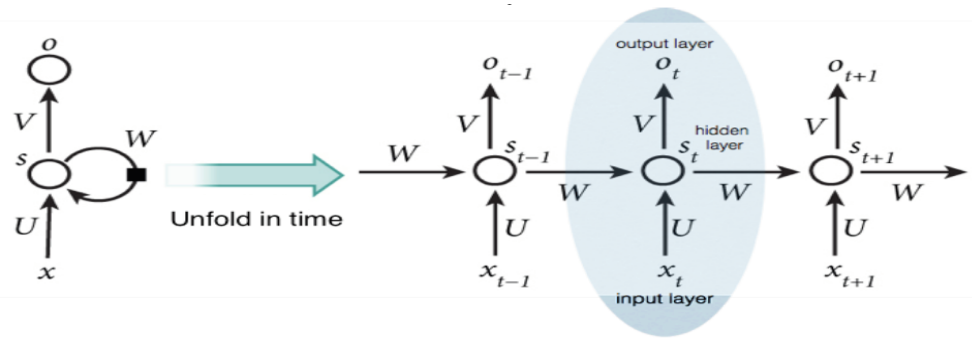


Figure 1.2: Recurrent Neural Network in Timesteps

The diagram above illustrates how the workings of the RNN, when unfolded in time, is very similar to feedforward neural networks. The area highlighted in blue is similar to what is happening in the diagram of the feedforward neural network. There is the input layer x_t hidden layer s_t, and output layer o_t. U, V, and W are the parameters or the weights that the model needs to learn. The difference between the feedforward neural network and the RNN is that there is an additional input, s_{t-1}, fed into the hidden layer s_t. If the network path highlighted in blue is the current time step t, then the previous that is the network at timestep t-1, and the network after happens at time step t+1, in which the current hidden layer s_t will be fed into s_{t+1} along with x_{t+1}. In the hidden layer, we apply an activation function to the sum of the previous hidden layer state and current input x_t. While the left hand side of diagram 2 seems to suggest that RNNs have a cyclic cycle, the connection between previous time step and current time step in the hidden state is still acyclic; this is important to recognize because the network needs to be acyclic in order for backpropagation to be possible. The diagram below illustrates what is happening in the hidden state of a RNN:

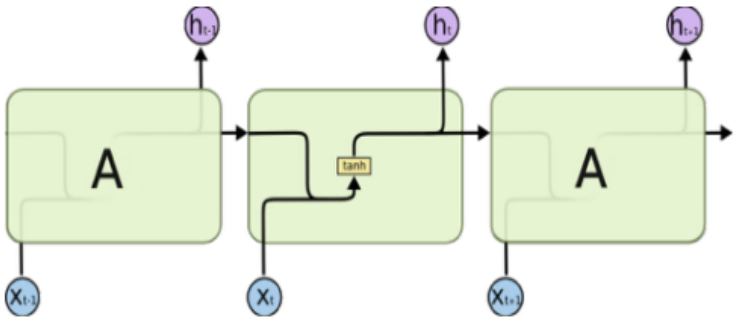


Figure 1.3: Hidden State of a RNN

Mathematically, we represent the step happening in the hidden state h_t as:

$$h_t = tanh(W * x_t + U * h_{t-1})$$

W and U are weight matrices; they are filters that determine how much importance to accord to both the present input and the previous hidden state. When we feed in the previous hidden state, it contains traces of all those that preceded h_{t-1}; this is how the RNN is able to have a persistent memory.

1.3.2 Back-propagation Through Time

In the back-propagation, we calculate the error the weight matrices generate, and then adjust their weights until the error cannot go any lower. As we see in *Figure 1.2*, the weight matrix W is carried through each time step. In order to compute the gradient for the current W, we need to perform the chain rule through a series of previous time steps. Because of this, we call the process back propagation through time (BPTT). If the sequences are quite long, the BPTT can take a long time; thus, in practice many people truncate the back-propagation to few steps instead of all the way to the beginning.

1.4 Long Short-Term Memory

1.4.1 Cell State and the Gates

LSTM was first introduced in 1997 by Sepp Hochrieiter and Jürgen Schmidhuber. LSTMs are capable of bridging time intervals in excess of 1000 time steps even in case of noisy, incompressible input sequences, without loss of short time lag capabilities. The architecture enforces constant error flow through internal states of special unit known as the memory cell.
There are three gates to the cell: the forget gate, input gate, and output gate. These gates are sigmoid functions that determine how much information to pass or block from the cell. Sigmoid functions takes in values and outputs them in the range of [0,1]. In terms of acting as a gate, a value of 0 means let nothing through, and a value of 1 means let everything through. These gates have their own weights that are adjusted via gradient descent. For the rest of the explanation for the forward propagation of the LSTM, we will refer to the diagram below.

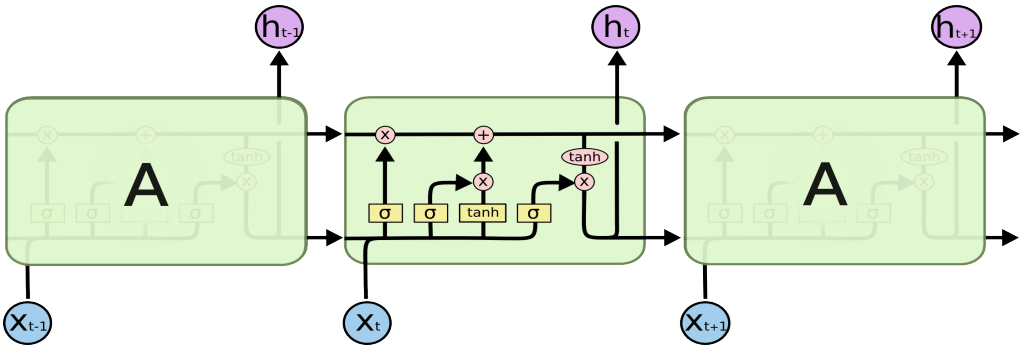
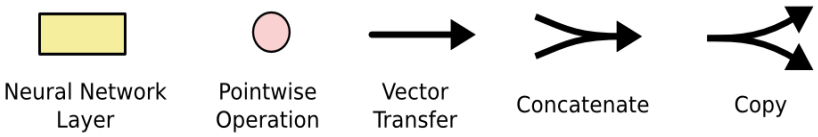


Figure 1.4: Hidden state of a LSTM RNN

Notation for explaining the *Figure 1.4* is as follows:



In further equations, h_{t-1} is the previous hidden state, x_t is the current input, W is the weight matrix, and b is the bias.

1.4.2 Forget Gate

The first step in our LSTM is to decide what information we're going to throw away from the cell state. This decision is made by a sigmoid layer called the "forget gate layer." It looks at h_{t1} and x_t, and outputs a number between 0 and 1 for each number in the cell state C_{t1}. A 1 represents "completely keep this" while a 0 represents "completely get rid

of this". For example, if we are moving from one music piece to the next in the input dataset, then we can forget all of the information related to the old music piece.

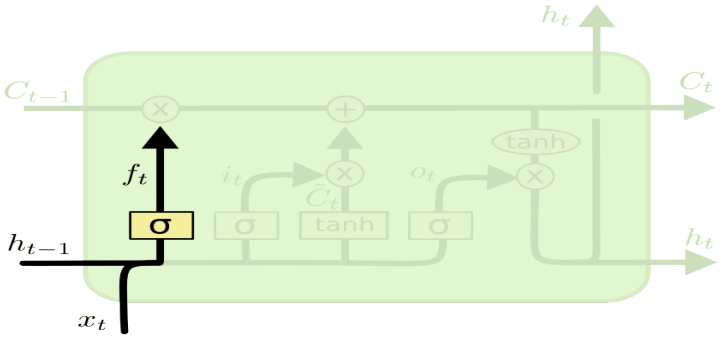


Figure 1.5: LSTM Forget Gate

$$f_t = \sigma(W_f.[h_{t-1}, x_t] + b_i)$$

1.4.3 Input Gate

The next step involves two parts. First, the input gate determines what new information to store in the memory cell. Next, a tanh layer creates a vector of new candidate values to be added to the state. From the example of the music learning model, we are inputting the first few sequences of notes of the new piece.

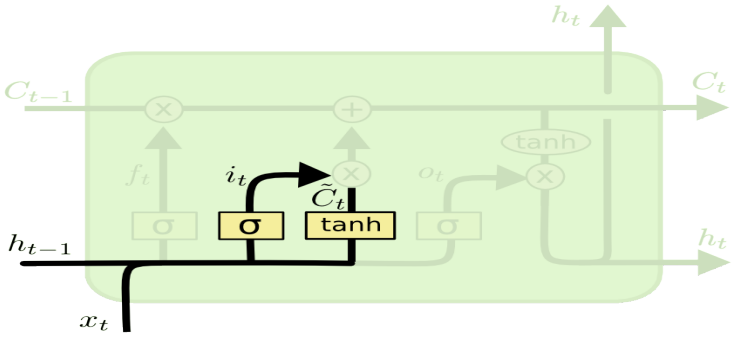


Figure 1.6: LSTM Input Gate

$$i_t = \sigma(W_i.[h_{t-1}, x_t] + b_i)$$
$$\widetilde{C}_t = tanh(W_c.[h_{t-1}, x_t] + b_c$$

1.4.4 Updating the Cell Memory

At this point we have determined what to forget and what to input, but we have not actually changed the memory cell state.

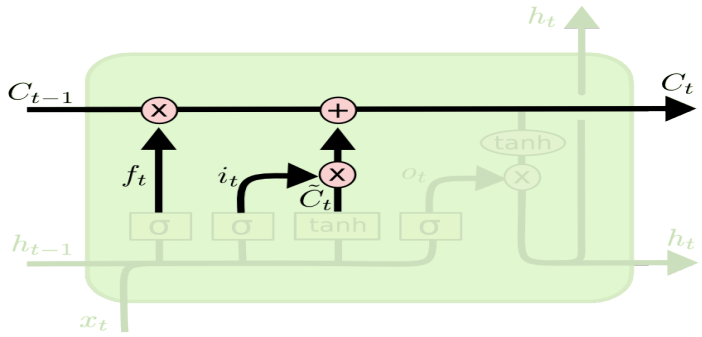


Figure 1.7: LSTM Input Gate

$$C_t = f_t * C_{t-1} + i_t * \widetilde{C}_t$$

To update the old cell state, C_{t-1} is multiplied with the vector f_t , and then added to $i_t * t$

1.4.5 Output Gate

To determine what to output from the memory cell, we again apply the sigmoid function to the previous hidden state and current input, then multiply that with tanh applied to the new memory cell (this will make the values between -1 and 1). In the music learning model example, we want to output information that will be helpful in predicting the next sequence of notes.

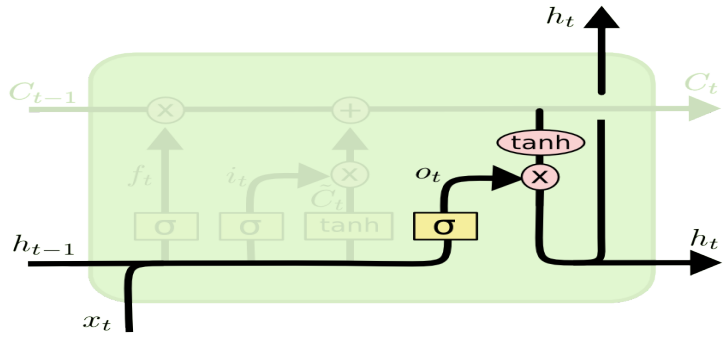


Figure 1.8: LSTM Output Gate

$$o_t = \sigma(W_o.[h_{t-1}, x_t] + b_o)$$

$$h_t = o_t * tanh(C_t)$$

CHAPTER 2

OBJECTIVE

We aim to use a Recurrent Neural Network that will be trained on our vast music database(ABC format) so that it can generate musical sequence that will hold a long term structure.

We also aim to create some metrics which can be used as a rating measure for given music sequence. This measure will help our RNN to classify the generated music in terms of whetheer the generated piece is close to already human likeable music.

CHAPTER 3

LITERARY SURVEY

Table 3.1: Literature Survey

S.No.	Research Paper	Institute	Algorithm	Dataset	Conclusion
1	Automatic Music Composition with Simple Probabilistic Generative Grammars	CIC-IPN Mexico	A generative grammar is generated by training the model on a vast music dataset. And the generative grammar can be later used to make music.	Not mentioned	A turing like test was performed to rate melodies in a dataset containing some human generated and some machine generated melodies. Some of the machine generated music surpassed human compositions.
2	Music generation using with Markov Models.	Stellenbosch University	A probabilistic Markov Model was generated by a learning model using a dataset. And the model was then used to create music.	Music compositions of two different artists was used.	A Turing Test/Survey was performed and people were asked to rate top three compositions. Human Compositions still surpassed machine music but 72% failed to differentiate between humans or computer or preferred machine compositions to human compositions.
3	A First Look at Music Compositionusing LSTM Recurrent Neural Networks	Google AMI	A RNN was trained on a vast database and two experiments were performed. In the first RNN was used to learn chord structure. In the second experiment a RNN was made to learn musical structure.	Self-made dataset of classical music compositions in both sheet music and MIDI formats	In the first experiment the system successfully learned chord structure. In the second experiment system was able to maintain long term structure in the generated pieces.
4	Jazz Melody Generation from Recurrent Network Learningof Several Human Melodies	Smith College	A RNN was trained on some jazz compositions and later it was checked whether the RNN could produce new pieces.	A self-acquired dataset of jazz compositions.	The RNN was able to generate music that was as pleasant as human generated music.

CHAPTER 4

METHODOLOGY

4.1 Acquiring the ABC Nottingham Dataset

- We used the ABC Nottingham Dataset. Nottingham dataset is a database of over 14000 folk songs.We merged all the files into a single document

4.2 Training the LSTM-RNN

- We trained our LSTM RNN on the songs stored in the document.It reads the ABC files character by character and tries to optimize its weights and biases to generate the time sequences it observes in the notation. The weights and biases were stored for pickling our trained RNN.

4.3 Generating the HMM probability matrix

- We read the songs one by one and created a bi-gram matrix which stores the probability of one note occurring after a particular note. We used Laplacian smoothing to make sure that our metric also entertained the cases where our RNN was being combinatorially creative. We pickled this probability matrix for later use.

4.4 Generating the average note distribution histogram

- We read the songs one by one and stored the probabilities of all the characters encountered in the training data set.This was used to store a histogram. We pickled

this histogram for later use.

4.5 Generation of new compositions

- We use the pickled model to create music.We can decide the length of the generative sequence. Once a composition is generated we use our HMM probability matrix to calculate the log-prob of the composition. Moreover we create a histogram of the note distribution of the composition and use KL Divergence to rate our sequence.

CHAPTER 5

ABC NOTATION

The ABC notation is a shorthand form of musical notation. In basic form it uses the letters A through G to represent the given notes, with other elements used to place added value on these - sharp, flat, the length of the note, key, ornamentation.

ABC Notation was initially developed for use with folk and traditional European tunes, but later with the help of many researchers it was extended to many other forms of music with extended characters and header information.

abcnotation.com is the home page of the project and this is where we found the Nottingham Folk music database containing folk music based in abc notation which served as the main dataset for our project.

```
<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 5.1: Sample ABC Notation

CHAPTER 6

METRICS

Following are the metrics that we propose for rating the music generated through our system:

- TURING TEST

  Turing test is the most widely used rating measure for machine generated music. Turing test takes rating about the music from humans where the humans are uninformed about the music , whether it is generated by a machine or it is a human generated music. We would like to perform a small Turing test to check the efficiency of our algorithm.

- METRIC USING BIGRAM PROBABILITY MODEL

  - We created a 2d matrix which contains the bigram probability of all the characters present in our training database. The bigram probability of an event C | A can be defined as the conditional probability of event C happening after the occurrence of event A.
  - To provide a normalized answer we have initialized the probability of all events to be at least one to apply Laplace smoothing on the model.
  - Now in order to assign rating to a generated composition we calculate the probability of the the whole composition by multiplying the bigram probabilities

of all notes. This is done because a highly probable sequence will have a high score.

- Now this score/probability may be too low because all these probabilities lie between 0 and 1. Thus we would perform the above operation in log space . Thus all the calculations would be performed using log of probabilities.

$$p_1 * p_2 * p_3 * p_4 = exp(log(p_1) + log(p_2) + log(p_3) + log(p_4))$$

- METRIC USING KULLBACK-LEIBLER DIVERGENCE

  - We calculated the frequency of every character in each composition and calculated the histogram of probabilities of the composition.
  - Now after we have calculated the histogram for every composition we use them to calculate the average histogram for a composition which kind of represents the count of the notes appearing in a composition.
  - For every generated piece of music we find out the histogram of probabilities and find the KL-Divergence between the two histograms. A low value for the divergence represents a better piece of music that is similar to the training pieces.

CHAPTER 7

RESULTS

We aim to use a Recurrent Neural Network that will be trained on our vast music database(ABC format) so that it can generate musical sequence that will hold a long term structure.
We also aim to create some metrics which can be used as a rating measure for given music sequence. This measure will help our RNN to classify the generated music in terms of whether the generated piece is close to already human likeable music.
Following are results of some of the music pieces that we generated using our software.

S. No.	Kullback Divergence Score	HMM Score
Song 1	0.0637991987489	1437.71141965
Song 2	-0.0374434422859	1277.18542313
Song 3	0.16201365575	1427.0851656

A lower value of Kullback divergence score and lower value of HMM score indicates that piece is more similar to the already made music in the Nottingham database for folk music. Hence, it can be labelled as a comparatively better music for human listening.

CHAPTER 8

CONCLUSION

We have successfully trained our LSTM-RNN model on Nottingham Dataset and the generated music compositions are pleasing to hear and are almost comparable to the quality of actual human generated music. The compositions possess Long Term Structure. We also found out that our trained model is combinatorially creative.

We did a turing test among our peers and mostly people weren't able to differentiate between the generated composition and human generated music.

Our second metric scores of the generated compositions are coming between 1200-1500 which is comparable to those of the human generated music. This result concludes that the machine generated music is following sequential rules which the algorithm approximates during training process.

KL divergence of the generated music histogram and the average histogram of the training set is low which shows that the average note distribution remains similar.

CHAPTER 9

REQUIREMENTS

The following specifications are required for the development and deployment of the system:

- SOFTWARE REQUIREMENTS
  - Python
  - Numpy
  - EasyABC
- HARDWARE REQUIREMENTS
  - Minimum 8 GB RAM
- DATASET
  - Nottingham Folk Music Dataset

CHAPTER 10

APPLICATIONS

Following are some of the ways in which our project can be helpful -

- The field of art and music is very dynamic and diverse. The randomness in machines along with the evaluation techniques used to rate the music can help in generation of exotic and never heard pieces which may sound great to humans as the system develops overtime.
- The software can be used by musicians making music of the same variety as the training data set of the music of checking how does their music rate according to the evaluation criteria of the software which can give them a measure of closeness of their music to generally liked music.

APPENDIX A

CODE SNIPPETS

PROCEDURE: lossFun()

```
def lossFun(inputs, targets, hprev):
  """
  inputs,targets are both list of integers.
  hprev is Hx1 array of initial hidden state
  returns the loss, gradients on model parameters, and last hidden state
  """
  xs, hs, ys, ps = {}, {}, {}, {}
  hs[-1] = np.copy(hprev)
  loss = 0
  # forward pass
  for t in range(len(inputs)):
    xs[t] = np.zeros((vocab_size,1)) # encode in 1-of-k representation
    xs[t][inputs[t]] = 1
    hs[t] = np.tanh(np.dot(Wxh, xs[t]) + np.dot(Whh, hs[t-1]) + bh) # hidden state
    ys[t] = np.dot(Why, hs[t]) + by # unnormalized log probabilities for next chars
    ps[t] = np.exp(ys[t]) / np.sum(np.exp(ys[t])) # probabilities for next chars
    loss += -np.log(ps[t][targets[t],0]) # softmax (cross-entropy loss)
  # backward pass: compute gradients going backwards
  dWxh, dWhh, dWhy = np.zeros_like(Wxh), np.zeros_like(Whh), np.zeros_like(Why)
  dbh, dby = np.zeros_like(bh), np.zeros_like(by)
  dhnext = np.zeros_like(hs[0])
  for t in reversed(range(len(inputs))):
    dy = np.copy(ps[t])
    dy[targets[t]] -= 1
    dWhy += np.dot(dy, hs[t].T)
    dby += dy
    dh = np.dot(Why.T, dy) + dhnext # backprop into h
    dhraw = (1 - hs[t] * hs[t]) * dh # backprop through tanh nonlinearity
    dbh += dhraw
    dWxh += np.dot(dhraw, xs[t].T)
    dWhh += np.dot(dhraw, hs[t-1].T)
    dhnext = np.dot(Whh.T, dhraw)
  for dparam in [dWxh, dWhh, dWhy, dbh, dby]:
    np.clip(dparam, -5, 5, out=dparam) # clip to mitigate exploding gradients
  return loss, dWxh, dWhh, dWhy, dbh, dby, hs[len(inputs)-1]
```

PROCEDURE: sample()

```
def sample(h, seed_ix, n):
  """
  sample a sequence of integers from the model
  h is memory state, seed_ix is seed letter for first time step
  """
  x = np.zeros((vocab_size, 1))
  x[seed_ix] = 1
  ixes = []
  for t in range(n):
    h = np.tanh(np.dot(Wxh, x) + np.dot(Whh, h) + bh)
    y = np.dot(Why, h) + by
    p = np.exp(y) / np.sum(np.exp(y))
    ix = np.random.choice(range(vocab_size), p=p.ravel())
    x = np.zeros((vocab_size, 1))
    x[ix] = 1
    ixes.append(ix)
  return ixes
```

PROCEDURE: KulbackDivergence()

```
# KULBACK divergence
n, p = 0 , 0
no_song = 0
nochars=0
hist=np.zeros((vocab_size))
while p < data_size:
  if data[p]=='X':
    if nochars!=0:
      histogram = histogram + hist/nochars
    hist = np.zeros((vocab_size))
    no_song = no_song + 1
    nochars = 0

  d0=char_to_ix[data[p]]
  hist[d0]=hist[d0]+1
  nochars=nochars+1
  p = p+1;

if nochars!=0:
  histogram = histogram + hist/nochars

histogram = histogram/no_song
print(histogram)
```

PROCEDURE: HMM()

```
# Hidden Markov Model
bigram = np.ones((vocab_size,vocab_size))
n, p = 0 , 1

ct = 0;
unigram = np.ones(vocab_size)
while p < data_size:
  unigram[char_to_ix[data[p]]] = unigram[char_to_ix[data[p]]] + 1
  ct = ct + 1;
  bigram[char_to_ix[data[p-1]]][char_to_ix[data[p]]] = bigram[char_to_ix[data[p-1]]][char_to_ix[data[p]]]+1
  p = p+1

unigram = unigram/ct
bigram = bigram/ct

prob_mat = np.zeros((vocab_size,vocab_size))


for i in range(vocab_size):
  for j in range(vocab_size):
    prob_mat[i][j]  = bigram[i][j] /unigram[i]



```

PROCEDURE: TrainingLoop()

```
while True:
  try:

  # prepare inputs (we're sweeping from left to right in steps seq_length long)
    if p+seq_length+1 >= len(data) or n == 0:
      hprev = np.zeros((hidden_size,1)) # reset RNN memory
      p = 0 # go from start of data
    inputs = [char_to_ix[ch] for ch in data[p:p+seq_length]]    #convert character array to integer using map
    targets = [char_to_ix[ch] for ch in data[p+1:p+seq_length+1]] #convert character array to target integer vector

    # sample from the model now and then
    if n % 100 == 0:
      sample_ix = sample(hprev, inputs[0], 200)
      txt = ''.join(ix_to_char[ix] for ix in sample_ix)
      """print ("----\n" ,txt ," \n----" )"""
      #print (txt)
    # forward seq_length characters through the net and fetch gradient
    loss, dWxh, dWhh, dWhy, dbh, dby, hprev = lossFun(inputs, targets, hprev)
    smooth_loss = smooth_loss * 0.999 + loss * 0.001
    if n % 100 == 0: print ("iter ",n," loss ",smooth_loss) # print progress

    # perform parameter update with Adagrad
    for param, dparam, mem in zip([Wxh, Whh, Why, bh, by],
                                  [dWxh, dWhh, dWhy, dbh, dby],
                                  [mWxh, mWhh, mWhy, mbh, mby]):
      mem += dparam * dparam
      param += -learning_rate * dparam / np.sqrt(mem + 1e-8) # adagrad update

    p += seq_length # move data pointer
    n += 1 # iteration counter
```

BIBLIOGRAPHY

[1] Horacio Alberto García Salas, Alexander Gelbukh, Hiram Calvo, and Fernando Galindo Soria *"Automatic Music Composition with Simple Probabilistic Generative Grammarsy"*:

[2] walter,schulze and brink van der merwe *"Music Generationwith Markov Models"* :

[3] Allen Huang Raymond Wu *"Deep learning for music"*: International Joint Conference on Neural Networks, 2001

[4] Chun-Chi J., Risto Miikkulainen *"Creating Melodies with Evolving Recurrent Neural Networks"*:

[5] Douglas Eck,Jurgen Schmidhuber *"A First Look at Music Composition using LSTM Recurrent Neural Networks"*: Technical Report No. IDSIA-07-02 IDSIA / USI-SUPSI.

[6] I-Ting Liu , Bhiksha Ramakrishnan *"BACH IN 2014: MUSIC COMPOSITION WITH RECURRENT NEURAL NETWORK"*:

[7] *"Char RNN model"*: Federal Information Processing Standards Publication 197, December 18, 2015.
`http://github.com/users/andrejKarpthy`