

---

# CSE 253 Homework Assignment 4

## Generating Music with Recurrent Networks

---

**Sainan Liu**  
A13291871  
sall131@eng.ucsd.edu  
**Shiwei Song**  
A53206591  
shs163@eng.ucsd.edu

**Hao-en Sung**  
A53204772  
wrangle1005@gmail.com  
**Haifeng Huang**  
A53208823  
hah086@eng.ucsd.edu

### Abstract

In this assignment, we are asked to train a recurrent neural network to learn the structure of an ABC notation music file through prediction. The method we use is to gradually increase the batch size (sequence size) in training and add the temperature parameter in the activation function in the Softmax layer for music generation. The sequences are separated as 80% for training and 20% for validation. We are able to achieve 52% accuracy with the training data, and over 51% accuracy with the validation data. We found that more audible tunes can be generated with a lower temperature, especially when  $T \leq 1$ , and the best training sequence length is between 50 to 70 characters. Then we changed the number of neurons in the hidden layer. We found that more neurons would lead to a better result. We also experimented on adding a dropout after the hidden layer. Larger dropout rate would lead to larger loss. However, larger dropout seemed to lead to more complex tunes. Adagrad achieves slightly better result than RMSprop at the end of training but RMSprop converges faster than Adagrad at the beginning 50 epochs. Different neurons at hidden layer are performing different tasks, some of them can recognize the body of the music and others can recognize the start header of the music.

## 1 Introduction

In this experiment, we have used a one-layer recurrent neural network to learn the structure of an ABC notation music file through prediction. We have tested the following hyper-parameters:

- We have tested the neural network with 3 different temperatures:  $T=1$ ,  $T=2$  and  $T=0.5$ , where  $T$  is used in the final Softmax layer for music generation.
- We have tried  $p=.1$ ,  $.2$  and  $.3$  in the dropout layer to see if it affects the training speed, and if it improves the results.
- The performance with Adagrad and RMSprop have been both tested and compared.

Training and validation loss over number of epochs are used as performance measurement. Finally, feature Heatmaps are generated for further analysis.

## 2 Method

An one-layer RNN is used to learn the structure of an ABC notation music file through prediction.

## 2.1 Data Generation

We have first randomly sliced the training data of size 110 with possibly up to 50% overlapping, and yield 3794 sequences for training and 949 sequences for testing. To train the network with sequences of increasing length  $k$ , we simply took the first  $k$  characters of each sequence. This number may vary slightly between experiment, but within each experiment section, they are consistent.

## 2.2 Training Parameters

- We have decided to use 64 sequences of length  $k$  in a batch to train the network, where  $k = [20, 50, 70, 90, 110]$  incrementally for every 50 epoch (with early stopping = 6 non-decreasing losses, this number might be lower). For example, for the first 50 epoch, we would train the network with batches of 64 sequences with 20 character in length. Each character of a sequence will be trained continuously one after another, and the state would be reset after every 20-character sequence. For every batch, 65 sequences are trained simultaneously to speed up this process.
- Only one hidden layer is used. Initially 100 neurons are used in our hidden layer. Then, different number of neurons has been tested and compared.
- Initially Adagrad with learning rate of 0.01 is used as the optimizer. Later on, RMSprop with learning rate of 0.001 has been used instead to compare with RMSprop's performance.
- Softmax is used as output with cross entropy loss during training.

## 2.3 Music Generation

- Softmax with a temperature parameter is used for music generation.
- Music is typically primed with 50 characters.
- A n-sided coin flip method is used to predict the next character after priming.

## 2.4 Hyper Parameters

The following hyper parameters are used for the result generation:

1. Temperature = [1, 2, 0.5]
2. Number of neurons in the hidden layer = [50, 75, 100, 150]
3. Dropout  $p = [.1, .2, .3]$
4. Optimizers = [Adagrad(lr=0.01), RMSprop(lr=0.001)]

## 3 Results

### (a) 2 sample music pieces per temperature

midi list:  
q4.a.1.1.mid  
q4.a.1.2.mid  
q4.a.2.1.mid  
q4.a.dot5.1.mid  
q4.a.dot5.2.mid

#### 1. T=1

**First generated music with T=1:**

X=1

T:Pinsa

vMtrpan mfseri  
 D:Pore  
 M:iranscri  
 R:FoeHu Aatin  
 R:-eud  
 C:TT  
 4  
 K:D  
 A:Tiecrit et/o  
 he  
 R:Baatir  
 eof arl|aomrin et/o/ /|  
 3/ |2 D2G E2B|B2BBGd |de |ge||ge|2G2B 2/A/ GA A2 |  
 E2 A B2de2 ||Ad edA||A2A B2 e2 e2 d f2d |AB|AE|2FFF |ABc 2/ gg b  
 g ede | fdB dee|eaaa d2e dBGAd ede |eaB dBd|2 B3B/ BAB|cd| dB/ |  
 2eedd ed||| dcA | A2 e2e| d2 A|3ec/d/ / | B2 2 |2d e  
 <:Varde ere lele  
 C:Transl tar  
 Tn  
 C2 AAG d2e | d2e|ed |e|fae a3ced gdB|| f2 dadea  
 d2e dBG Bd FFBBB||dB A/ B  
 gg3 B G2AF A2 |2 |B2G A2 AA G

Pinsa

TT



Figure 1: First music from the T=1 code above in standard music notation.

#### Second generated music with T=1:

X:110  
 d:Cineaeaneetaanss thllouen lacdoBhun/e da |irt

T:2/4  
 L:1/  
 /4/a  
 K:D  
 FA GFD|B  
 <B3 G2D||  
 GagtM dB|B | B2 B2 |2dd/e ddd||G BGG G2E||BAE|| B2AA/||2 D2B |2c|B2B  
 |: dGa g G2G2  
 A2A B2ea E2BA|e  
 aAdd |Bd bgf ede deB|A2D |2G | A2c2// |E AEA EAG/B AA|||G  
 G2e :Ahe oe vation  
 Z:Dlrhem eontereaiaipe  
 C:iranscrette  
 <start>  
 X:1/t f2 BBD|DG|FAAA  
 |2e|2g | |2 BAd A BB3 B3 Bc| d2 eaa 2/e/ /| f2 fBe d2d |2 edaaga gB | d2 ||

2/4



Figure 2: Second music from the T=1 code above in standard music notation.

## 2. T=2

### First generated music with T=2:

X:1

T:Coun iich the Moc  
 oDueheaTerl  
 R::Sa e sreant1

K:Fr G2na Thenhavv nuhin  
 Z2 eedd 2 agf eae |2 aafaer Tnr a-

```

Z:ada
rrhen
i
Z:DFeees ani u cSn s_
00
: o
ieaaoren.i
Z2g eag3d e2|B2||2a e3d d|2 2efb ea/ cA 2 E2 ||
-4/, |

```

[Empty lines have been deleted for better note generation online.]

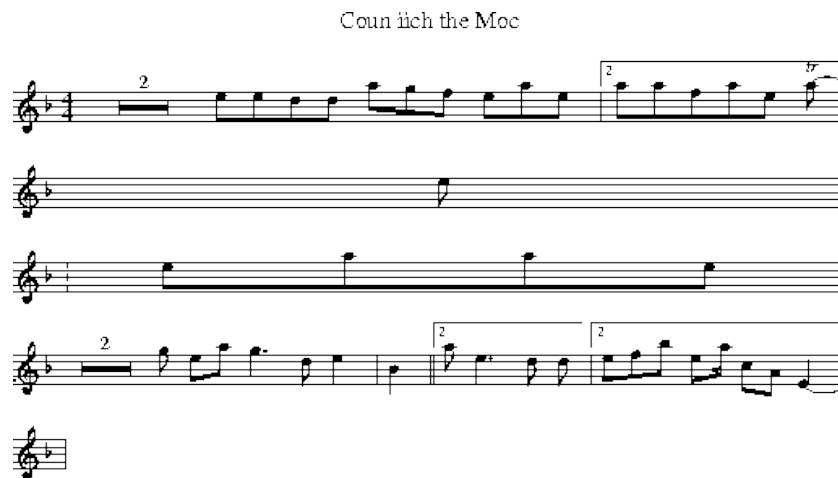


Figure 3: First music from the T=2 code above in standard music notation.

**Partial tune from the second generated music with T=2:  
There is no playable midi file available for this tune.**

```

X:1
T:Coanyhettiet
Z:Tiran
g
t' Tollee@ o mei ita

K:
2||1 a
Z: or
Zsta
M2a
Rd enBF:hleraoi
i e ir
GoCnl r

4
2|2g
edd ede||
Go"

```

The musical score for 'The Rose Tree' is presented in four staves. The first staff is in 4/4 time and begins with a treble clef, a key signature of one sharp (F#), and a common time signature (C). It contains a single eighth note on the G line of the staff, preceded by a repeat sign and a first ending bracket labeled '1'. The second staff is in 4/4 time and begins with a treble clef and a key signature of one sharp (F#). It contains a single eighth note on the G line of the staff, preceded by a repeat sign and a first ending bracket labeled '1'. The third staff is in 4/4 time and begins with a treble clef and a key signature of one sharp (F#). It contains a single eighth note on the G line of the staff, followed by a half note on the A line, a quarter note on the B line, and a quarter note on the C line. The fourth staff is in 4/4 time and begins with a treble clef and a key signature of one sharp (F#). It contains a single eighth note on the G line of the staff, followed by a half note on the A line, a quarter note on the B line, and a quarter note on the C line.

### 3. T=0.5

```
X:11
T:Far
the Colle

T:Mnrtar|e al the Crance
C:Trance
C:Tradce
C:Faghl|toute observation maile(err toute observation
mailto:galouvielle@free.fr
M:C|
K:G
TA cAGA|F
G2A G B2BB2 BA | d2 d2ed e2G GDG|Bef |dB|dBccd|A2 B2 |Ad|ed
e2e|d2G |Gde A2 |BBD E2G FG3 G G2d d d2 e d2d edB|| A2c |2e |
f2 ed d2d|e2ded cAe |e | f2 ee | edded| f2e/dd2 ed | e2B |2B|B2BGA
| dd | d2d| f2 | B2 BA G2F G G2G | B2 ||
```

Trance  
Trance  
Fapiltoute observation mail(enn toute observation mailto:galouvielle@free.fr



### Second generated music with T=0.5:

X:1

T: Bir enl

'is eeealt terenee

A: Provence

C: Mariation <mailto:galouvielle@free.fr>

M: C/4

L: 1/8

T: Bar tichel BELLON - 2005-07-06

T: 6/8

K: D

BAD E2G | A2G E2F G2A|BBc d e2 d2 | A2 AA|BG3 EAA |2e | e2d eee d2  
dAG|AGd e e2d | d2 d2 A2 |B||A|| d2d |AA||FGE |AB |d2 d2G|AGE E2G|F2  
| d2G A2 | A2 A2 | c2 BA B2B|| c2 2 |2A| A2 2 |2A| B2GBA B2 |AG | A2 A2F  
|A2DD2 G2|| G2 G2 | G2 e e2d2 | d2 d2 | B2 d2 | B2 BA |G | d2 2 | G



Figure 6: Second music from the T=0.5 code above in standard music notation.

**(b) Training loss and validation loss vs number of epochs on data**

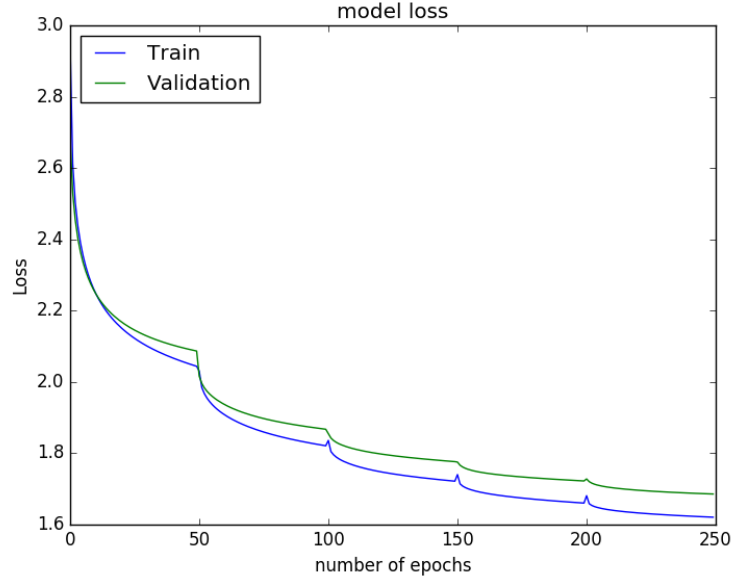


Figure 7: Training and validation loss vs number of epochs with sequence length increment for at most every 50 epoch. Sequence length = [20, 50, 70, 90, 110]; Batch size = 64; #Hidden = 100; optimizer=Adagrad.

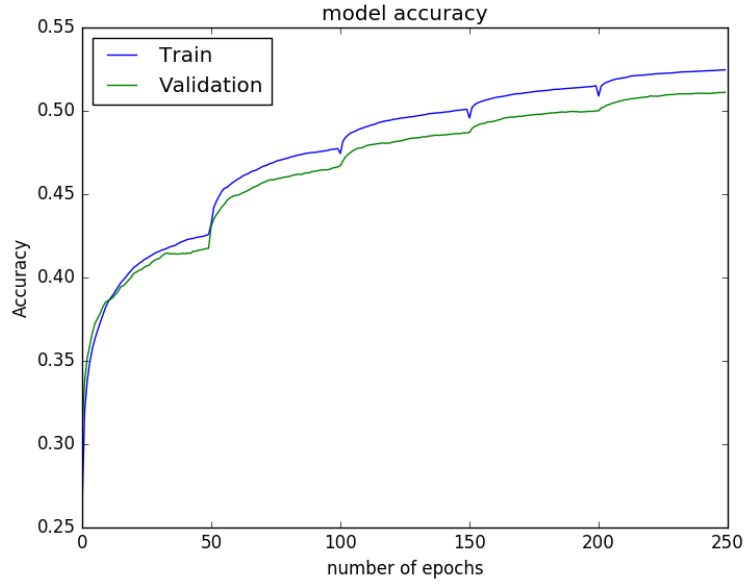


Figure 8: Training and validation accuracies vs number of epochs with sequence length increment for at most every 50 epoch. Sequence length = [20, 50, 70, 90, 110]; Batch size = 64; #Hidden = 100; optimizer=Adagrad.



**(c) 50, 75 and 150 neurons in hidden layer**

We change the number of neurons in the hidden layer to 50, 75 and 150.

1. 50 hidden units

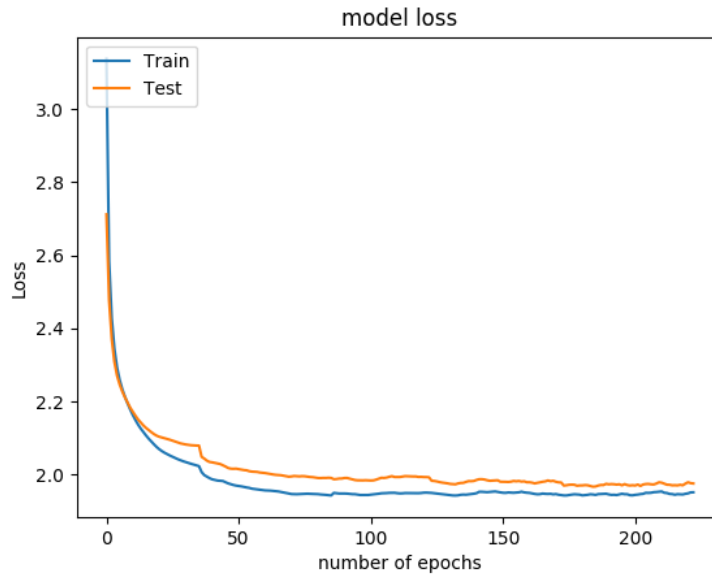


Figure 9: Training and validation loss vs number of epochs with 50 hidden units

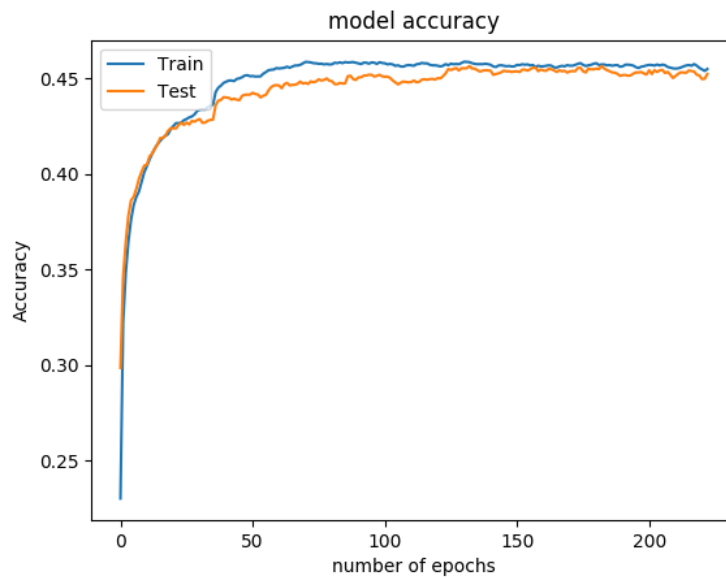


Figure 10: Training and validation accuracies vs number of epochs with 50 hidden units

2. 75 hidden units

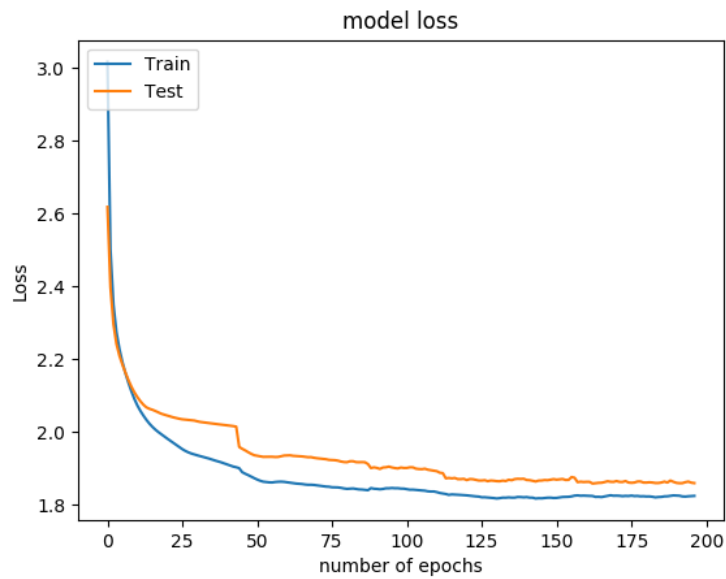


Figure 11: Training and validation loss vs number of epochs with 75 hidden units

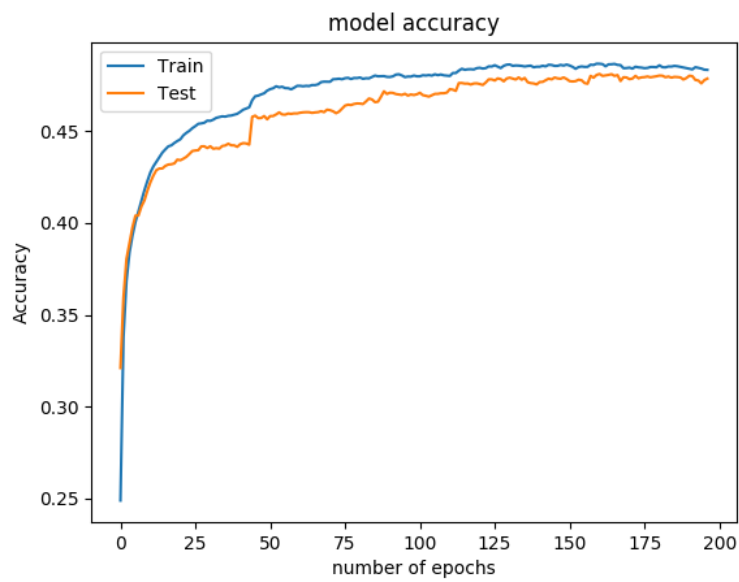


Figure 12: Training and validation accuracies vs number of epochs with 75 hidden units

### 3. 150 hidden units

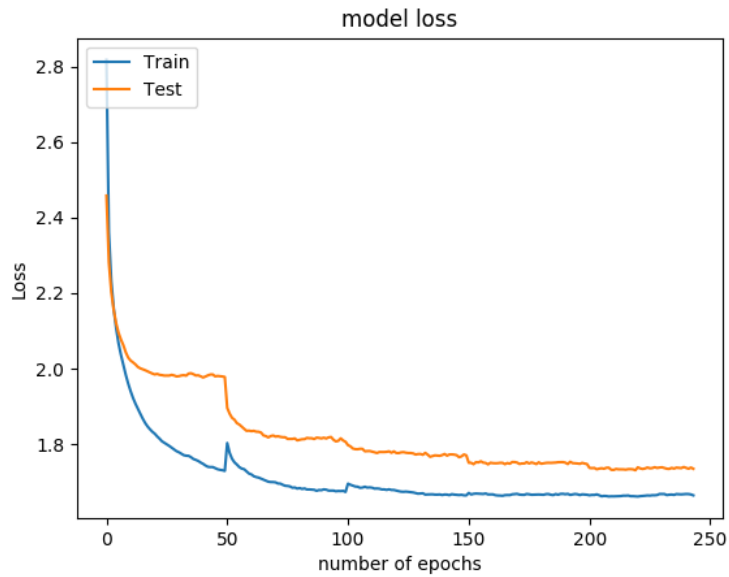


Figure 13: Training and validation loss vs number of epochs with 150 hidden units

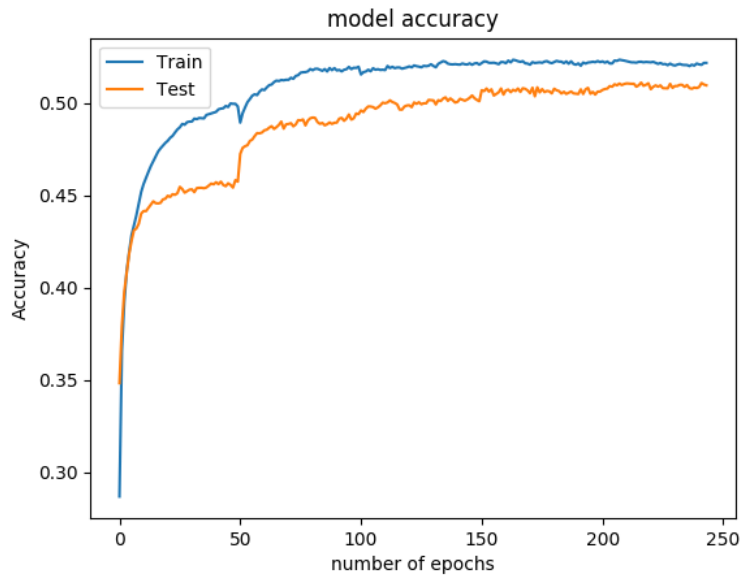


Figure 14: Training and validation accuracies vs number of epochs with 150 hidden units

As number of neurons in hidden layer increases, the loss decreases. The more neurons the network contains, the more information the network contains, the better result we get.

#### (d) dropout $p = .1, .2, .3$

In this part, we add a dropout layer before the hidden layer with dropout  $p = 0.1, 0.2, 0.3$ .

midi list:

Q4d\_dot1.mid

Q4d\_dot2.mid  
Q4d\_dot3.mid

1.  $p = 0.1$

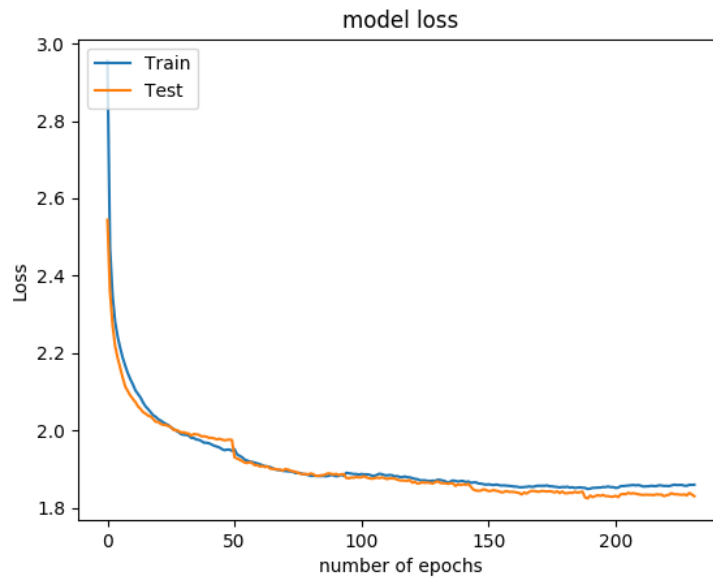


Figure 15: Training and validation loss vs number of epochs with  $p = 0.1$

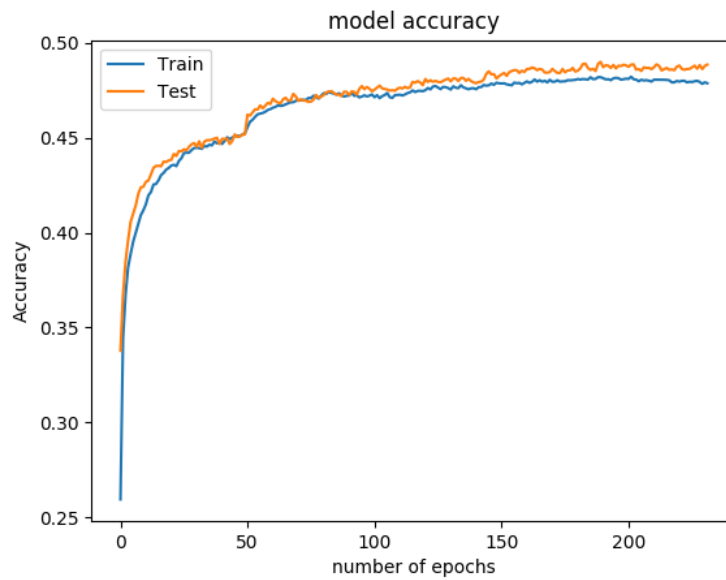


Figure 16: Training and validation accuracies vs number of epochs with  $p = 0.1$

**Music generated with  $p = 0.1$**

X:13

T:6/8  
 K:G  
 A2 | eeo|eeh2  
 M:6aa  
 R:Bar  
 .dandee  
 R:srtorre obsereiae a  
 Z2  
 d:T2  
 t torn  
 eartnle  
 R: 2 2 |3Bf | d2g d d3dcd c c2Bc ||  
 | |2 |2 1ag e2e | d2d| e3eg |2B e2B 2 ||B2 dB |23de  
 Z | B2 B B2 |2|2A| B2B A  
 | |2 | e2 |2B |  
 B2A EG3 A G2d2|3dcf| d2 d2 |3aff |  
 d2A/ / | e2 | c2c |2 |A| |G2 BA d2B||fe |d || e2 2  
 GE3 |2B A2E||d2 d2 d2 2 ||

6/8



Figure 17: Music generated with  $p = 0.1$

2.  $p = 0.2$

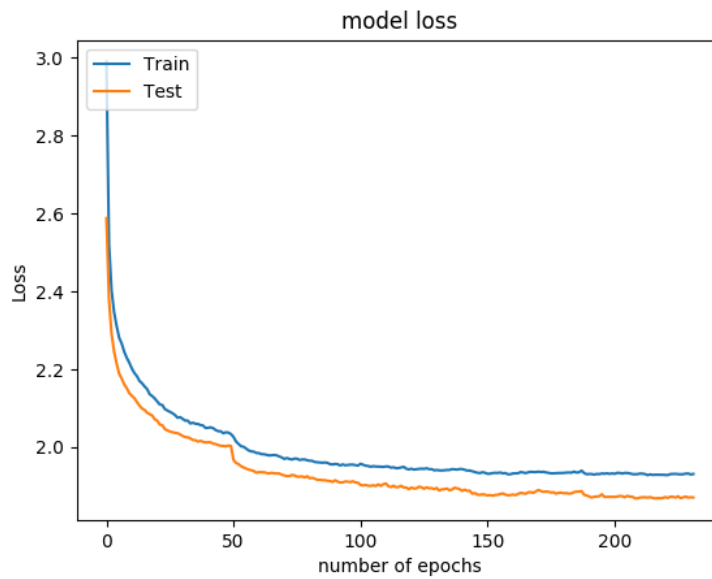


Figure 18: Training and validation loss vs number of epochs with  $p = 0.2$

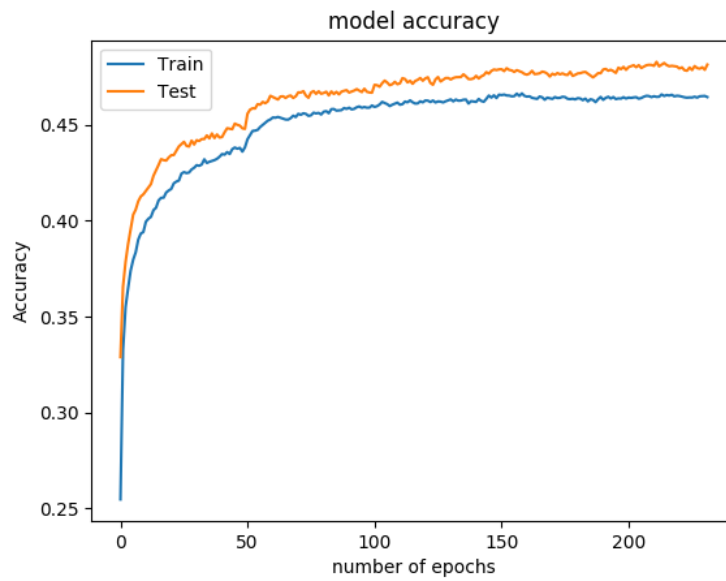


Figure 19: Training and validation accuracies vs number of epochs with  $p = 0.2$

### Music generated with $p = 0.2$

X:1

M:1/8

K:G leeh T

var

Z2 |ee |d2 r Blanc ee itnn

R:saroht  
 Z:6/8  
 K:D  
 a3 B2 ce01|oe nge  
 R:|eel er  
 onhelto e The adl.t  
 T:Burthaeaa h ig  
  
 Z:T  
 Z:id:hn-sea  
 R:id:h|o  
 Dd e/ | d2 ee AAAc |2FF/ |  
 <2 cd |/ :|  
 4  
 L:1/8  
 K:Dmo  
 Behaenc Bhrtithe t  
 Z:T  
 BhB A2c |2d| dde |  
 d2B e2 | B2 A  
 <2 | A2G G2 || Aer g e2 2 2|/ | B2G2B2A |Ad|A |||  
 ||2 | e2fg e2 c/  
 |:   
 2  
 AAA|h2/ | B2d | d3Bd| e  
 e  
 c3cd  
 2d edd 2A FDF 2 /2 /2 /2 /B A2G |AdBeeBdBA AAG 2 c2 ||

Burthaeaa h ig



Figure 20: Music generated with  $p = 0.2$

3.  $p = 0.3$

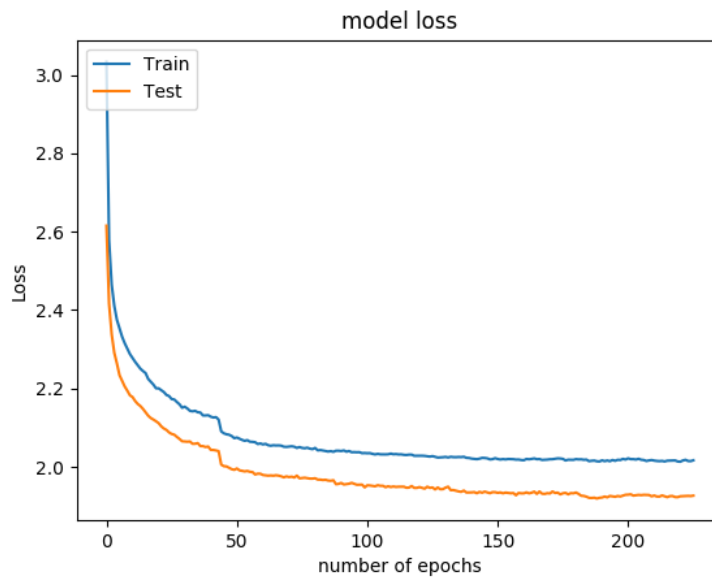


Figure 21: Training and validation loss vs number of epochs with  $p = 0.3$

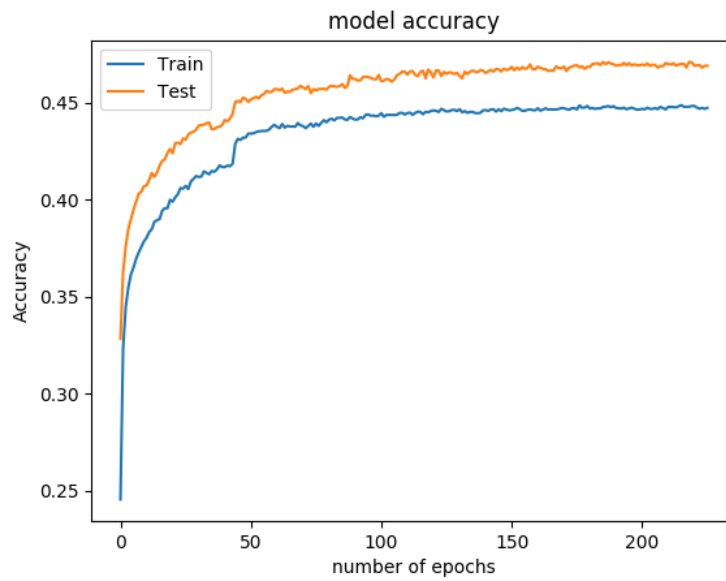


Figure 22: Training and validation accuracies vs number of epochs with  $p = 0.3$

### Music generated with $p = 0.3$

X:1

T:ClæM's:l s |ionene

ai

4

K:D

G



B2GG2 |Be |> 2 g2AB| B2B2 | B2d|deee | Tag  
 |a/ e2B/|/ A2AB/A/ / / 2 |AA AA |ed | e2dB2 |2 |d eg3 g2e | egfg |  
 egg dd |2d |d|e|dd2/|Ac | ded| d2BB2d e| e2gg |ed| dgfgg2 | e2e2 |  
 |2ed|| eg-an 1  
 Z: :2  
 AEit |e ga F2D | 2  
 K2 | B2B |F2| A2F |2F||2D A2| ag | B2 | B2B2 | B2B | B2d A  
 B2 d2 BB/2/ |//2eAB d22/ e2/d/ / e c2B2 2 | B3BBA | B 2  
 L:Frór:O' - a2a A3/ |2 e d2g3eBA 22 B |2B |AG| A2e/ |d 2ed  
 Z: B2 2|2B |  
 |: ged |2e || ea ed |Ac/ |2| B2BBd || e2 eed| e2 |2 A2 G2  
 |areMondMin Bi e rt>ohetorleee Bnlbeeente  
 R:sanne  
 Rat eene dh n tr32|EFG 2 | f2 dAc/ | B2A d2dd | d2d eed |  
 |dAG F2d e2 | d2B| B3BB| |2ear | B2B|2 |2  
 |2d | e2 e c|2c/ | d2d2 2 d2BA A2 B2 |AG|A B2 A2A A2B AGBBA A2B |  
 2|g2gg/ /|ad g3dcc |dd | 2 |B2d: 2e eAA |  
 |2/ ed | B2BA B2c |2B |

ClæM's:l s lionene



Figure 23: Music generated with  $p = 0.3$

Dropout decreases the training speed. The music generated with larger dropout seems to be more complex.

### (e) Adagrad vs RMSprop

In this problem, we are going to use different optimization techniques Adagrad and RMSProp, and compare the performance of them.

The training and validation loss and accuracy for RMSprop is shown in Figure.24 and Figure.25. Here we are using initial learning rate 0.001. The training and validation loss and accuracy for Adagrad is shown in Figure.26 and Figure.27. Here we are using initial learning rate 0.01.

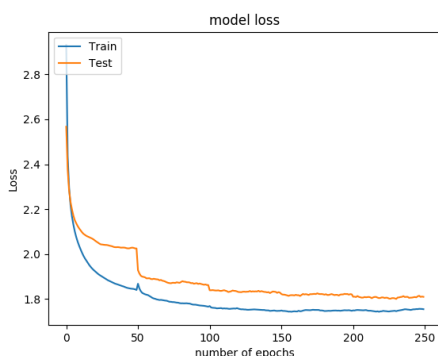


Figure 24: Training and Validation Loss for RMSprop

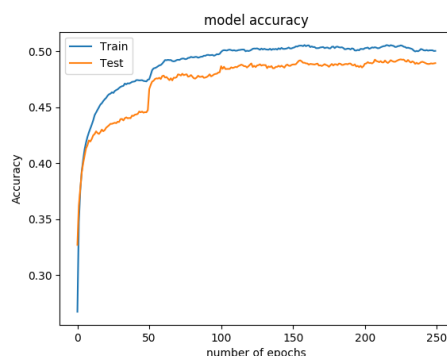


Figure 25: Training and Validation Accuracy for RMSprop

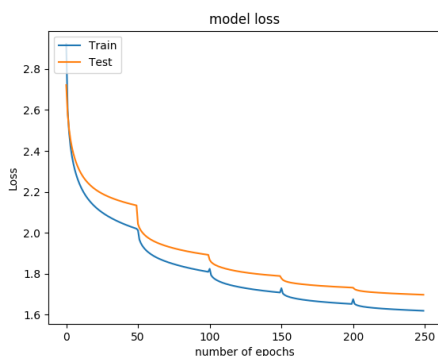


Figure 26: Training and Validation Loss for Adagrad

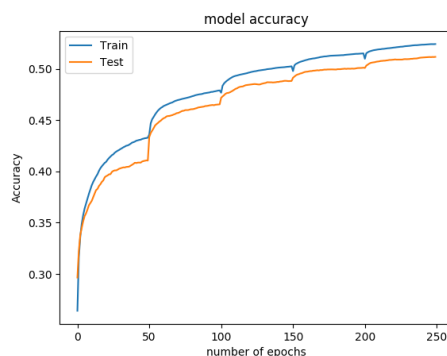


Figure 27: Training and Validation Accuracy for Adagrad

### (f) Feature Evaluation

In this problem, we are going to show activations of two hidden neurons and see if these neurons can recognize different parts of a piece of music. Figure.28 to Figure.29 show the activations of neurons.

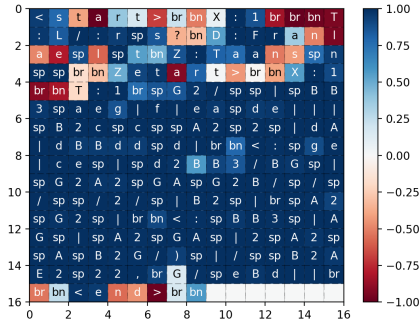


Figure 28: Activation For Neural 1

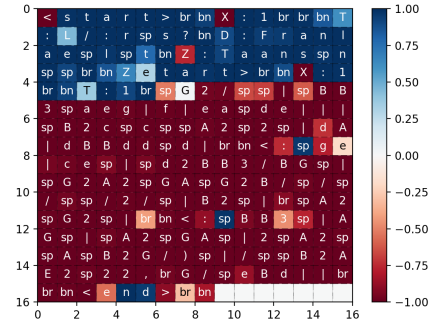


Figure 29: Activation For Neural 2

## 4 Discussion

### (a) 2 sample music pieces per temperature

For these images, we used Adagrad with learning rate of 0.01, and 100 hidden neurons. We found that with a lower temperature, more complete and playable tunes can be generated. As temperature goes up to 1 we start to have tunes that are not playable, or ongoing tunes without an end token. As temperature rise up to 2, there are barely any tunes that have an end token, let alone playable. We only found one complete tune that is playable. For the second tune at  $T=2$ , we have manually selected the partial code instead, although this tune happens to display in standard music format, there was no midi file that can be generated for it. After many repetitions, in the end, we were not able to find another tune that contains an end token and playable when  $T=2$ . This is consistent with our expectation, because as  $T \rightarrow \infty$ , all labels have nearly the same probability, which gives more randomness to the music generation. On another hand, the lower the temperature, the behavior tends to be more deterministic, we might tend to see more repeated notes when  $T$  is too small, although these tunes were not selected to display here.

### (b) Training loss and validation loss vs number of epochs on data

For these images, we used Adagrad with learning rate of 0.01, and 100 hidden neurons. We first notice that the validation loss is higher than training loss, which is expected. We also notice that the training loss decreases much slower later on. It seems to be plateaued at a very high loss, and low accuracy rate around 52%. This may indicate that we don't have a network structure that is complicated enough to capture the structure of the tunes. We found that earlier on in the training process, with shorter sequences, such as 20 characters, the training and validation performances tend to plateau at higher loss value. Once we increase the sequence length, the loss seem to decrease further then plateau at lower value. However, when we use 70 characters or above, the loss start to decrease much slower with the increase of sequence length. This shows that the best sequence length to capture the structure might be between 50 to 70 characters.

Additionally, we have also tried to take the maximum output and feed that back into the input and train it to produce the next input, but that ends up decreasing the accuracy immediately to 20% after the first epoch, and the accuracy stopped increasing or increasing super slowly after 29%. The generated music is not audible at all. Hence we abandoned that method pretty earlier on.

### (c) 50, 75 and 150 neurons in hidden layer

We changed the number of neurons in the hidden layer to 50, 75 and 150 in the part and kept other parameters as initial settings. We found that as the number of neurons in hidden layer increases, the loss decreases and the accuracy increases.

This is a reasonable outcome as we made the model more complicated and added more trainable parameters. So the more neurons the network contains, the more information the network could contain and the better result it would generate.

#### (d) dropout $p = .1, .2, .3$

In this part, we added a dropout after the hidden layer. We tried with different dropout rate  $p = .1, .2, .3$ . We found that as dropout rate increases, the loss increases and the accuracy decreases. Also adding dropout would decrease the training speed. However, there's a trend that the tunes generated with larger dropout seemed to be more complicated. We think one possible reason for this could be the randomness the dropout brings into the model. So larger dropout would lead to larger randomness and more complex tunes.

#### (e) Adagrad vs RMSprop

We can see from the figures that Adagrad achieves higher accuracy and lower loss than RMSprop. The update rule for Adagrad for  $i$ -th parameter  $\theta$  at time step  $t$  is

$$\theta_{t+1,i} = \theta_{t,i} - \frac{\eta}{\sqrt{G_{t,ii} + \epsilon}} g_{t,i} \quad (1)$$

where  $\eta$  is the initial learning rate,  $g_{t,i}$  is the gradient of objective function with respect to parameter  $\theta$ ,  $G_t$  is a diagonal matrix where each diagonal element  $G_{t,ii}$  is the sum of the squares of the gradients with respect to  $i$  up to time step  $t$ . while  $\epsilon$  is a smoothing term that avoids division by zero (usually on the order of  $1e8$ ). The flaw of Adagrad is that the learning rate will eventually becomes infinitely small and network will stop learning. We can see that Adagrad has significant drop in loss and increase in accuracy at 50, 100, 150, 200 epochs.

The update rule for RMSprop for  $i$ -th parameter  $\theta$  at time step  $t$  is

$$E[g^2]_t = 0.9E[g^2]_{t-1} + 0.1g_t^2 \quad (2)$$

$$\theta_{t+1,i} = \theta_{t,i} - \frac{\eta}{\sqrt{E[g^2]_t + \epsilon}} g_{t,i} \quad (3)$$

RMSprop divides the learning rate by an exponentially decaying average of squared gradients, so its learning rate is decreasing slower than Adagrad, thus we can see that RMSprop converges faster than Adagrad at the beginning 50 epochs.

The reason why Adagrad achieves better result is that its initial learning rate is 10 times bigger than RMSprop.

#### (f) Feature Evaluation

Because we only have 249 characters in this music, we append 7 null characters in the end. We produce the heatmap for each neuron at hidden layer and choose two with the best representation of the music. From Figure.28 we can see that the first neuron is able to recognize body of the music and from Figure.28 we can see that the second neuron is able to recognize the start header of the music.

## 5 Summary

In summary, training with incremental sequence length allow us to achieve 52% accuracy with the training data, and over 51% accuracy with the validation data with Adagrad of 0.01 learning rate with 100 hidden neurons. We found that higher the temperature causes more randomness in the music generation, whereas, lower the temperature generates more deterministic tunes. More audible tunes can be generated with a lower temperature, when  $T \leq 1$  in this case. More hidden neurons will lead to better result. Adding dropout will decrease training speed and add more randomness to tunes generated. Adagrad achieves slightly better result than RMSprop at the end of training but RMSprop converges faster than Adagrad at the beginning 50 epochs. We found that neurons at hidden layer can recognize different part of the music, for the two neurons we have shown, the first neuron can recognize the body of the music and the second one can recognize the start header of the music.

## **6 Contributions**

Sainan and Shiwei were in charge of the initial development of the code for question 3. Sainan was in charge of report section a and b; Shiwei was in charge of report section c and d; Hao-en was in charge of report section e. Haifeng is in charge of report section f.

## Codes

### Codes for Main Report

Listing 1: RNNTrain.py for question 3

```
1 from os.path import isfile, isdir
2 from os import makedirs
3 from keras.layers.recurrent import SimpleRNN
4 from keras.layers.core import Dense
5 from keras.models import Sequential
6 from keras.optimizers import RMSprop, Adagrad
7 from keras.callbacks import EarlyStopping, Callback
8 from utilities import loadTunes, partition, label2code, prepDataSeq, processSample, savePkl, loadPkl
9 import time
10 from tqdm import *
11 import numpy as np
12 import random
13 from keras.utils import np_utils
14 from keras import backend as K
15 LR = 0.01
16 QUESTION_ID = 'Q3'
17 N_HIDDEN_UNIT = 100
18 INCREMENT_TRAINING = True
19 BATCH_SIZE = 64 # How many samples will be processed simultaneously.
20 PRIME_LEN = 50 # Prime the music generation with PRIME_LEN characters.
21 MAX_EPOCH = 50 # How many epoch will be run.
22 EARLY_STOPPING = 6 # Stop training if validation set's loss stop decreasing.
23 DATA_STORAGE = '../data/'
24 # For results and models
25 MODEL_STORAGE = '../model/'
26 RESULT_STORAGE = '../result/'
27 MODEL_FILE = '%smymyRNN%s.h5'%(MODEL_STORAGE, QUESTION_ID)
28 TEMP_WEIGHT_FILE = '%stemp_test_weight%s.h5'%(MODEL_STORAGE, QUESTION_ID)
29 WEIGHT_FILE = '%sweight_me%d_lr%s.h5'%(RESULT_STORAGE, MAX_EPOCH, LR, QUESTION_ID)
30
31 RESULT_FILE = '%sresult_%g_%s.pkl'%(RESULT_STORAGE, LR, QUESTION_ID)
32 ACC_FILE = '%saccuracy_%g_%s.png'%(RESULT_STORAGE, LR, QUESTION_ID)
33 LOSS_FILE = '%sloss_%g_%s.png'%(RESULT_STORAGE, LR, QUESTION_ID)
34 # For generateMusic
35 TUNE_STORAGE = '%stunes/tune_%s/'%(RESULT_STORAGE, QUESTION_ID)
36
37 END_TOKEN = "<end>"
38 TUNE_MAX_LEN = 5000 # maximum length of the tune is around 4900, this ensures that we will stop
39
40 TEMPERATURE = 2
41 MAX_SEQ_LEN = 110
42
43 def getTemperature():
44     return TEMPERATURE
45
46 def getResult():
47     acc, loss, val_acc, val_loss = loadPkl(RESULT_FILE)
48     return acc, loss, val_acc, val_loss
49 def temperature_activation(a):
50     T = TEMPERATURE
51     return K.softmax(a/T)
52
53 def getTestModel(output_dim = 94, lr=LR, n_hidden_units = N_HIDDEN_UNIT, modelfile = MODEL_FILE):
54     # as the first layer in a Sequential model
55     model = Sequential()
56     input_length = 1 # number of timesteps.
57     input_dim = output_dim # number of features after one-hot encoding.
58     model.add(SimpleRNN(n_hidden_units,
59                         batch_input_shape = (BATCH_SIZE, input_length, input_dim),
```

```

60         return_sequences=False, # return last output in the output sequence fo
61         stateful=True, # last state for every sample at index i in a batch wi
62         unroll = True)) # network will be unrolled, speedup TF.
63     model.add(Dense(output_dim, activation=temperature_activation))
64     opt = RMSprop(lr=lr)
65     model.compile(optimizer=opt,
66                   loss='categorical_crossentropy',
67                   metrics=['accuracy'])
68     model.summary()
69     model.save(modelfile)
70     return model
71
72 def getModel(output_dim = 94, lr=LR, n_hidden_units = N_HIDDEN_UNIT, modelfile = MODEL_FILE):
73     # as the first layer in a Sequential model
74     model = Sequential()
75     input_length = 1 # number of timesteps.
76     input_dim = output_dim # number of features after one-hot encoding.
77     model.add(SimpleRNN(n_hidden_units,
78                         batch_input_shape = (BATCH_SIZE, input_length, input_dim),
79                         return_sequences=False, # return last output in the output sequence fo
80                         stateful=True, # last state for every sample at index i in a batch wi
81                         unroll = True)) # network will be unrolled, speedup TF.
82     model.add(Dense(output_dim, activation='softmax'))
83     opt = Adagrad(lr=lr)
84     model.compile(optimizer=opt,
85                   loss='categorical_crossentropy',
86                   metrics=['accuracy'])
87     model.summary()
88     model.save(modelfile)
89     return model
90
91 def getTestModelWithWeights():
92     # Get default model with the final weights loaded.
93
94     if not isfile(WEIGHT_FILE):
95         print "WARNING: Can not load %s, please run RNNTrain.py first."%(WEIGHT_FILE)
96         return None
97     else:
98         model = getTestModel()
99         model.load_weights(WEIGHT_FILE)
100         return model
101
102 def trainModel(model, seqLen, Xtrain, ytrain, Xvalid, yvalid, nClasses, encodedTunes, le, incre
103     print np.array(Xtrain).shape, np.array(ytrain).shape, np.array(Xvalid).shape, np.array(yv
104     WEIGHT_FILE_STORE = '../result/weight_me%d_lr%g_seq%d%s.h5'%(MAX_EPOCH, LR, seqLen, QUES
105     print "Training with learning rate %g with sequence %d"%(LR, seqLen)
106     mean_tr_accs = []
107     mean_tr_losses = []
108     mean_te_accs = []
109     mean_te_losses = []
110     previous_epoch = 0
111     # Load weights if incrementalW is true, and previous weights exist.
112     if incrementalTraining and isfile(WEIGHT_FILE):
113         model.load_weights(WEIGHT_FILE)
114         print "Loaded weights from %s"%(WEIGHT_FILE)
115         mean_tr_accs, mean_tr_losses, mean_te_accs, mean_te_losses = loadPkl(RESETL.FILE)
116         print "Loaded history.pkl from %s"%(RESULT_FILE)
117         previous_epoch = len(mean_tr_accs)
118     elif incrementalTraining and not isfile(WEIGHT_FILE):
119         print "Did not find previous weights. Cannot load previous weights."
120     elif isfile(WEIGHT_FILE):
121         print "Over-writing previous weights %s"%(WEIGHT_FILE)
122     else:
123         print "Creating %s for the first time."%(WEIGHT_FILE)
124

```

```

125 pre_test_loss = float("inf")
126 incre = 0
127 model_pre = model
128 optimal_model = None
129 for epoch in range(MAX_EPOCH):
130     print "train_part_Epoch:%d/%d"%(previous_epoch+epoch, previous_epoch+MAX_EPOCH)
131     tr_accs = []
132     tr_losses = []
133     n_batch = len(Xtrain)/BATCH_SIZE
134     print '%d_samples %d_batches with batch_size of %d'%(len(Xtrain),
135                                                             n_batch,
136                                                             BATCH_SIZE)
137     for i in tqdm(range(n_batch)):
138         x_seqs = np.array(Xtrain[i*BATCH_SIZE:(i+1)*BATCH_SIZE])
139         y_seqs = np.array(ytrain[i*BATCH_SIZE:(i+1)*BATCH_SIZE])
140         for j in range(seqLen):
141             # conver every feature to (numsample, lenth, dim) format.
142             # here x_hot is in (batch_size, 1, nClasses) format
143             # y_hot is in (batch_size, nClasses) format
144             x_hot, y_hot = processSample(x_seqs[:, [j]],
145                                         y_seqs[:, [j]],
146                                         nClasses)
147             tr_loss, tr_acc = model.train_on_batch(x_hot, y_hot)
148             tr_accs.append(tr_acc)
149             tr_losses.append(tr_loss)
150         model.reset_states()
151     mean_tr_acc = np.mean(tr_accs)
152     mean_tr_accs.append(mean_tr_acc)
153     mean_tr_loss = np.mean(tr_losses)
154     mean_tr_losses.append(mean_tr_loss)
155     print ('accuracy_training = %f'%format(mean_tr_acc))
156     print ('loss_training = %f'%format(mean_tr_loss))
157     print ('-----')
158
159     te_accs = []
160     te_losses = []
161     n_batch = len(Xvalid)/BATCH_SIZE
162     for i in tqdm(range(n_batch)):
163         x_seqs = np.array(Xvalid[i*BATCH_SIZE:(i+1)*BATCH_SIZE])
164         y_seqs = np.array(yvalid[i*BATCH_SIZE:(i+1)*BATCH_SIZE])
165         for j in range(seqLen):
166             x_hot, y_hot = processSample(x_seqs[:, [j]],
167                                         y_seqs[:, [j]],
168                                         nClasses)
169             te_loss, te_acc = model.test_on_batch(x_hot,
170                                                  y_hot)
171             te_accs.append(te_acc)
172             te_losses.append(te_loss)
173         model.reset_states()
174
175     mean_te_acc = np.mean(te_accs)
176     mean_te_accs.append(mean_te_acc)
177     mean_te_loss = np.mean(te_losses)
178     mean_te_losses.append(mean_te_loss)
179     print ('accuracy_testing = %g'%format(mean_te_acc))
180     print ('loss_testing = %g'%format(mean_te_loss))
181     print ('-----')
182
183     # Early Stopping
184     if pre_test_loss <= mean_te_loss:
185         incre += 1
186         if incre == 1:
187             optimal_model = model_pre
188     else:
189         incre = 0

```



```

190         model_pre = model
191         optimal_model = None
192
193     if incre >= EARLY_STOPPING:
194         print "Early_stopping_at_%d_-_%d_steps"%(epoch, EARLY_STOPPING)
195         break
196
197     pre_test_loss = mean_te_loss
198
199     # Check music generation every 20 epoch.
200     if epoch%20 ==0:
201         model.save_weights(TEMP_WEIGHT_FILE)
202         te_model = getTestModel()
203         te_model.load_weights(TEMP_WEIGHT_FILE)
204         final_tune = generateMusic(te_model, encodedTunes, le)
205         filename = '%s%d.txt'%(TUNE_STORAGE, epoch + previous_epoch)
206         outputfile = open(filename, 'w')
207         outputfile.write("%s" % ''.join(final_tune))
208
209     # Save weights.
210     if optimal_model != None:
211         model = optimal_model
212         data = mean_tr_accs[: -EARLY_STOPPING], mean_tr_losses[: -EARLY_STOPPING], mean_te_accs[: -EARLY_STOPPING], mean_te_losses[: -EARLY_STOPPING]
213     else:
214         data = mean_tr_accs, mean_tr_losses, mean_te_accs, mean_te_losses
215
216     model.save_weights(WEIGHT_FILE)
217     model.save_weights(WEIGHT_FILE_STORE)
218
219     savePkl(data, RESULT_FILE)
220     mean_tr_accs, mean_tr_losses, mean_te_accs, mean_te_losses = loadPkl(RESULT_FILE)
221     savefig([mean_tr_accs, mean_te_accs], 'model_accuracy',
222             'number_of_epochs',
223             'Accuracy',
224             ['Train', 'Validation'],
225             ACC_FILE)
226     savefig([mean_tr_losses, mean_te_losses], 'model_loss',
227             'number_of_epochs',
228             'Loss',
229             ['Train', 'Validation'],
230             LOSS_FILE)
231     print "Done Saving"
232
233 def generateMusic(model, encodedTunes, le, maxTunes = 6, maxTuneLen = TUNE_MAXLEN):
234     nClasses = len(list(le.classes_))
235     # prime the network with a sequence randomly selected from tunes.
236     random.shuffle(encodedTunes)
237     start_seqs = []
238     chosen_tune = random.randint(0, len(encodedTunes)-1)
239     # Make sure that the chosen tune is long enough to prime the song.
240     while PRIME_LEN > len(encodedTunes[chosen_tune]):
241         chosen_tune = random.randint(0, len(encodedTunes)-1)
242     # Copy the same selected tune for Batch_Size
243     for i in range(BATCH_SIZE):
244         start_seqs.append(encodedTunes[chosen_tune][:PRIME_LEN])
245     y_prime = le.inverse_transform(start_seqs[0])
246     print "Prime the sequence with [0]x%d: \n-----\n%s\n-----"%(BATCH_SIZE, ''.join(y_prime))
247     start_X = np.array(start_seqs)
248     prime_pred = []
249     for j in range(PRIME_LEN):
250         # convert every feature to (numsample, lenth, dim) format.
251         # here x_hot is in (batch_size, 1, nClasses) format
252         # y_hot is in (batch_size, nClasses) format
253         x_hot = np_utils.to_categorical(start_X[:, [j]], nClasses)
254         x_hot = np.expand_dims(x_hot, axis=1)

```

```

255     y_pred_prob = model.predict_on_batch(x_hot)
256     y_pred_code = np.argmax(y_pred_prob[0])
257     # take the prediction for the char j in first prime sequence.
258     prime_pred.append(le.inverse_transform(y_pred_code))
259
260     print "prime_pred_info", len(prime_pred)
261     # convert last prob prime output to label to display and check.
262     print "Prime_result:\n————\n%s\n————"%' '.join(prime_pred))
263
264     lastToken = []
265     # Take the last predicted character, continue feeding it to model as input.
266     x_next_hot = prob2input(y_pred_prob)
267     more_preds = [prime_pred[-1]]
268     tunes_count = 0
269     while ((''.join(lastToken) != END_TOKEN or tunes_count < maxTunes) and len(more_preds) < maxTunes):
270         y_pred_prob = model.predict_on_batch(x_next_hot)
271         x_next_hot = prob2input(y_pred_prob)
272         y_pred_code = np.argmax(y_pred_prob[0])
273         pred_char = le.inverse_transform(y_pred_code)
274         lastToken.append(pred_char)
275         if len(lastToken) == 6:
276             lastToken = lastToken[1:]
277             more_preds.append(pred_char)
278         if ''.join(lastToken) == END_TOKEN:
279             tunes_count += 1
280     result = [y_prime.tolist()[0]] # Add first character
281     result.extend(prime_pred) # Add rest to result.
282     result.extend(more_preds)
283     return result
284
285 def runRNN(seqLen):
286     # Generate result folders for result and models.
287     if not isdir(MODEL_STORAGE):
288         makedirs(MODEL_STORAGE)
289
290     if not isdir(RESULT_STORAGE):
291         makedirs(RESULT_STORAGE)
292
293     # Generate folders for tunes storage.
294     if not isdir(TUNE_STORAGE):
295         makedirs(TUNE_STORAGE)
296
297     PRE_PROCESS_STORE = '%spre_preprocessed_seq%d-%s'%(DATA_STORAGE, MAX_SEQ_LEN, QUESTION_ID)
298     preprocesspk1 = '%s.pkl'%(PRE_PROCESS_STORE)
299     preprocessstxt = '%s.txt'%(PRE_PROCESS_STORE)
300
301     if not isfile(preprocesspk1):
302         print "Read_data_from_file_%s."%(preprocesspk1)
303         tunes = loadTunes()
304         print "Encode_chars_to_ints."
305         encoded_tunes, label_encoder = label2code(tunes)
306         print "Chop_txt_to_sequences_of_%d_length"%(MAX_SEQ_LEN)
307         X, y = prepDataSeq(encoded_tunes, MAX_SEQ_LEN)
308         print "Found_total_%d_sequences"%(len(X))
309         x_train, y_train, x_valid, y_valid = partition(X, y)
310         print "Done_partition."
311         data = x_train, y_train, x_valid, y_valid, encoded_tunes, label_encoder
312         savePk1(data, preprocesspk1)
313     else:
314         x_train, y_train, x_valid, y_valid, encoded_tunes, label_encoder = loadPk1(preprocesspk1)
315
316     f = open(preprocessstxt, 'w')
317     for seq in x_train:
318         f.write(''.join(label_encoder.inverse_transform(seq)))
319         f.write('\n————\n')

```

```

320     print "Done saving ."
321
322     nClasses =len(list(label_encoder.classes_))
323     print "There are %d tunes. %d classes."%(len(encoded_tunes), nClasses)
324     print "%d sequences for training, %d sequences for testing"%(len(x_train), len(x_valid))
325     #print label_encoder.classes_
326
327     model = getModel(nClasses)
328     trainModel(model, seqLen, x_train, y_train, x_valid, y_valid, nClasses, encoded_tunes, lab
329
330 if __name__ == "__main__":
331     for seq_len in [20, 50, 70, 90, 110]:
332         runRNN(seq_len)

```

Listing 2: utilities.py for data extraction and generation

```

1
2 from os.path import isfile
3 import numpy as np
4 import random
5 from sklearn import preprocessing
6 from keras.utils import np_utils
7 import cPickle as pickle
8 import matplotlib.pyplot as plt
9
10 def savePkl(dataset, pklfile):
11     # Save small pkl files.
12     f = file(pklfile, 'wb')
13     pickle.dump(dataset, f, protocol=pickle.HIGHEST_PROTOCOL)
14     f.close
15
16 def loadPkl(pklfile):
17     # Load small pkl files.
18     f = open(pklfile, 'rb')
19     dataset = pickle.load(f)
20     f.close
21     return dataset
22 # This file provide tools that can be used to read the data.
23 def loadTunes(filepath='../data/input.txt'):
24     f = open(filepath, 'r')
25     tunes = []
26     starting_seqs = []
27     tune = []
28     for line in f:
29         tune.extend(list(line))
30         if line == '<end>\r\n':
31             tunes.append(tune)
32             tune = []
33     print "Found %d tunes"%(len(tunes))
34     statspath = '../data/stats.txt' # stores the length for every tune.
35     statsfile = open(statspath, 'w')
36     for i in range(len(tunes)):
37         statsfile.write("%d:\t%d\n"%(i, len(tunes[i])))
38     print "tunes ranges from %d to %d characters"%(min(map(len, tunes)), max(map(len, tunes)))
39     return tunes
40
41 def label2code(tunes):
42     le = preprocessing.LabelEncoder()
43     all_chars = []
44     map(all_chars.extend, tunes)
45     le.fit(all_chars)
46     print 'found %d classes.'%(len(list(le.classes_)))
47     new_tunes = map(le.transform, tunes)
48     return new_tunes, le
49

```

```

50 def prepDataSeq(data, sequenceLen, sequential = False, noTuneSeparation= True, overlapping = True)
51     # slice random sequences from all tunes. Not aligned with the beginning of the file.
52     X = []
53     y = []
54     if sequential:
55         print "Sequentially connected sequences will be generated."
56     else:
57         print "Randomly selected sequences will be generated."
58     if noTuneSeparation:
59         print "The file will be chopped as a whole sequence."
60     else:
61         print "Each tune will be chopped as an independent sequence."
62     if overlapping:
63         print "No overlapping sequences will be generated."
64     else:
65         print "Overlapping sequences will be generated."
66     if noTuneSeparation:
67         # Sequences are randomly drawn from the entire input.txt file.
68         all_chars = []
69         map(all_chars.extend, data)
70         N = len(all_chars)
71         valid_start_max = N - 1 - 2*sequenceLen
72         start = 0
73         while start < valid_start_max:
74             if not sequential:
75                 start = random.randint(start, start+sequenceLen)
76             end = start + sequenceLen
77             x_seq = all_chars[start:end]
78             X.append(x_seq)
79             y_seq = all_chars[start+1:end+1]
80             y.append(y_seq)
81             if overlapping and not sequential:
82                 start = end - sequenceLen/2
83             else:
84                 start = end
85     else:
86         # Sequences are only drawn from within every tune sequence.
87         for tune in data:
88             N = len(tune)
89             valid_start_max = N - 1 - 2*sequenceLen
90             # randomly select one sequence start per sequenceLen sequentially from X.
91             start = 0
92             while start <= valid_start_max:
93                 if not sequential:
94                     start = random.randint(start, start+sequenceLen)
95                 end = start + sequenceLen
96                 x_seq = tune[start:end]
97                 X.append(x_seq)
98                 y_seq = tune[start+1:end+1]
99                 y.append(y_seq)
100                 if overlapping and not sequential:
101                     start = end - sequenceLen/2
102                 else:
103                     start = end
104             # both X and y are shape of (sequences x sequenceLen)
105
106             # Add last sequence if one does not exist, add it for half of the time.
107             if start != N-1 and random.random() > 0.5 and N-1 >= sequenceLen:
108                 X.append(tune[-sequenceLen-1:-1])
109                 y.append(tune[-sequenceLen:])
110                 if len(X[-1]) != sequenceLen or len(y[-1]) != sequenceLen:
111                     print len(X[-1]), len(y[-1])
112     return X, y
113
114 def partition(X, y):

```

```

115     data = zip(X, y)
116     random.shuffle(data)
117     # 80% for training, 20% for validation
118     n_sequences = len(data)
119     n_train = int(n_sequences*0.8)
120     n_valid = n_sequences - n_train
121     data_train = zip(*data[:n_train])
122     data_test = zip(*data[n_train:])
123     return list(data_train[0]), list(data_train[1]), list(data_test[0]), list(data_test[1])
124
125 def processSample(x, y, nClasses):
126     xhot = np_utils.to_categorical(x, nClasses)
127     xhot = np.expand_dims(xhot, axis=1)
128     yhot = np_utils.to_categorical(y, nClasses)
129     return xhot, yhot
130
131 def prob2input(y_pred_prob):
132     nSamples = y_pred_prob.shape[0]
133     nClasses = y_pred_prob.shape[1]
134     y_pred_ints = []
135     for i in range(nSamples):
136         y_pred_int = np.random.choice(range(nClasses), p=y_pred_prob[i, :])
137         y_pred_ints.append(y_pred_int)
138     y_pred_hot = np_utils.to_categorical(y_pred_ints, nClasses)
139     y_pred_hot = np.expand_dims(y_pred_hot, axis=1)
140     return y_pred_hot
141
142 def savefig(results, title='', xlabel='', ylabel='', legends = [], savepath = '',
Xs = [], display = False, overwrite = True):
143     if not isfile(savepath) or overwrite:
144         print "Save %s ..."%(savepath)
145
146         if Xs == []:
147             Xs = range(len(results[0]))
148         print "#_iterations:", len(Xs)
149         for Ys in results:
150             plt.plot(Xs, Ys)
151
152             plt.title(title)
153             plt.ylabel(ylabel)
154             plt.xlabel(xlabel)
155
156             if legends !=[]:
157                 plt.legend(legends, loc='upper_left')
158             plt.savefig(savepath)
159             print "Done_saving_acc_figure."
160             if display:
161                 plt.show()
162             plt.clf()
163
164 if __name__ == "__main__":
165     tunes = readTxt()
166     samples = np.random.randint(len(tunes), size=3)
167     sample_len = 50
168     print "example_start_sequences_of_length %d:\n%s\n%s\n%s"%(sample_len,
169                                                                    ''.join(tunes[samples[0]][:sample_len]),
170                                                                    ''.join(tunes[samples[1]][:sample_len]),
171                                                                    ''.join(tunes[samples[2]][:sample_len]))
172     new_tunes, label_encoder = label2code(tunes)
173     sequence_len = 30
174     X, y = prepDataSeq(new_tunes, sequence_len)
175     print "Found_total_%d_sequences"%(len(X))
176     x_train, y_train, x_test, y_test = partition(X, y)
177     print "%d_train_sequences, %d_test_sequences"%(len(x_train), len(x_test))
178     print "train_data_dimension", np.array(x_train).shape

```

```
179     print "label_data_dimension", np.array(x_test).shape
```

Listing 3: ReportQ4ab.py for report generation for q4.a and b

```
1 from os.path import isfile, isdir
2 from os import makedirs
3
4 from RNNTrain import getTestModelWithWeights, generateMusic, getTemperature, getResult
5 from utilities import loadTunes, label2code, savefig
6
7 QUESTION_ID = 'Q4a'
8 MODEL_STORAGE = '../model/'
9 RESULT_STORAGE = '../result/'
10 TUNE_STORAGE = '%stunes/tune_%s/'%(RESULT_STORAGE, QUESTION_ID)
11 MAX_TUNE = 30
12 TUNE_MAX_LEN = 5000
13
14 def reportQ4a():
15     if not isdir(MODEL_STORAGE):
16         makedirs(MODEL_STORAGE)
17
18     if not isdir(RESULT_STORAGE):
19         makedirs(RESULT_STORAGE)
20
21     # Generate folders for tunes storage.
22     if not isdir(TUNE_STORAGE):
23         makedirs(TUNE_STORAGE)
24     model = getTestModelWithWeights()
25     if model == None:
26         return
27     print "Read_tunes_from_input.txt_file."
28     tunes = loadTunes()
29     print "Encode_chars_to_ints."
30     encoded_tunes, label_encoder = label2code(tunes)
31     # for T in [1, 2, 0.5]: we need to change the TEMPERATURE global variable to theses values
32     for num in range(MAX_TUNE):
33         final_tune = generateMusic(model, encoded_tunes, label_encoder, 1, TUNE_MAX_LEN)
34         filename = '%s%g.%d.txt'%(TUNE_STORAGE, getTemperature(), num)
35         outputfile = open(filename, 'w')
36         # print type(final_tune), len(final_tune), final_tune[:50]
37         outputfile.write("%s" % ''.join(final_tune))
38         model.reset_states()
39
40 def reportQ4b():
41     acc, loss, val_acc, val_loss = getResult()
42     savefig([acc, val_acc], 'model_accuracy',
43            'number_of_epochs',
44            'Accuracy',
45            ['Train', 'Validation'],
46            '%sq4_b/q4_b_acc.png'%RESULT_STORAGE)
47     savefig([loss, val_loss], 'model_loss',
48            'number_of_epochs',
49            'Loss',
50            ['Train', 'Validation'],
51            '%sq4_b/q4_b_loss.png'%RESULT_STORAGE)
52
53 if __name__ == '__main__':
54     reportQ4a()
55     reportQ4b()
```

Listing 4: ReportQ4e.py for report generation for q4.e

```
1 from os.path import isfile, isdir
2 from os import makedirs
3 from keras.layers.recurrent import SimpleRNN
```

```

4 from keras.layers.core import Dense
5 from keras.models import Sequential
6 from keras.optimizers import RMSprop, Adagrad
7 from keras.callbacks import EarlyStopping, Callback
8 from utilities import loadTunes, partition, label2code, prepDataSeq, processSample, savePkl, l
9 import time
10 from tqdm import *
11 import numpy as np
12 import random
13 from keras.utils import np_utils
14 from keras import backend as K
15 LR = 0.001
16 QUESTION_ID = 'Q4e-RMSprop'
17 N_HIDDEN_UNIT = 100
18 INCREMENT_TRAINING = True
19 BATCH_SIZE = 64 # How many samples will be processed simultaneously.
20 PRIME_LEN = 30 # Prime the music generation with PRIME_LEN characters.
21 MAX_EPOCH = 50 # How many epoch will be run.
22 EARLY_STOPPING = 6 # Stop training if validation set's loss stop decreasing.
23 DATA_STORAGE = '../data/'
24 # For results and models
25 MODEL_STORAGE = '../model/'
26 RESULT_STORAGE = '../result/'
27 MODEL_FILE = '%smvRNN%s.h5'%(MODEL_STORAGE, QUESTION_ID)
28 TEMP_MODEL_FILE = '%stemp_test_model%s.h5'%(MODEL_STORAGE, QUESTION_ID)
29 WEIGHT_FILE = '%sweight_me%d_lr%s.h5'%(RESULT_STORAGE, MAX_EPOCH, LR, QUESTION_ID)
30
31 RESULT_FILE = '%sresult_%g_%s.pkl'%(RESULT_STORAGE, LR, QUESTION_ID)
32 ACC_FILE = '%saccuracy_%g_%s.png'%(RESULT_STORAGE, LR, QUESTION_ID)
33 LOSS_FILE = '%sloss_%g_%s.png'%(RESULT_STORAGE, LR, QUESTION_ID)
34 # For generateMusic
35 TUNE_STORAGE = '%stunes/tune_%s/'%(RESULT_STORAGE, QUESTION_ID)
36
37 END_TOKEN = "<end>"
38 TUNE_MAX_LEN = 5000 # maximum length of the tune is around 4900, this ensures that we will sto
39
40 TEMPERATURE = 1
41 MAX_SEQ_LEN = 110
42 def getTemperature():
43     return TEMPERATURE
44
45 def temperature_activation(a):
46     T = TEMPERATURE
47     return K.softmax(a/T)
48
49 def getTestModel(output_dim = 94, lr=LR, n_hidden_units = N_HIDDEN_UNIT, modelfile = MODEL_FILE
50     # as the first layer in a Sequential model
51     model = Sequential()
52     input_length = 1 # number of timesteps.
53     input_dim = output_dim # number of features after one-hot encoding.
54     model.add(SimpleRNN(n_hidden_units,
55         batch_input_shape = (BATCH_SIZE, input_length, input_dim),
56         return_sequences=False, # return last output in the output sequence fo
57         stateful=True, # last state for every sample at index i in a batch wi
58         unroll = True)) # network will be unrolled, speedup TF.
59     model.add(Dense(output_dim, activation=temperature_activation))
60     opt = RMSprop(lr=lr)
61     #opt = Adagrad(lr=lr)
62     model.compile(optimizer=opt,
63         loss='categorical_crossentropy',
64         metrics=['accuracy'])
65     model.summary()
66     model.save(modelfile)
67     return model
68

```

```

69 def getModel(output_dim = 94, lr=LR, n_hidden_units =N_HIDDEN_UNIT, modelfile = MODEL_FILE):
70     # as the first layer in a Sequential model
71     model = Sequential()
72     input_length = 1 # number of timesteps.
73     input_dim = output_dim # number of features after one-hot encoding.
74     model.add(SimpleRNN(n_hidden_units ,
75                         batch_input_shape = (BATCH_SIZE, input_length, input_dim),
76                         return_sequences=False, # return last output in the output sequence for
77                         stateful=True, # last state for every sample at index i in a batch with
78                         unroll = True)) # network will be unrolled, speedup TF.
79     model.add(Dense(output_dim, activation='softmax'))
80     opt = RMSprop(lr=lr)
81     #opt = Adagrad(lr=lr)
82     model.compile(optimizer=opt,
83                  loss='categorical_crossentropy',
84                  metrics=['accuracy'])
85     model.summary()
86     model.save(modelfile)
87     return model
88
89 def getTestModelWithWeights():
90     # Get default model with the final weights loaded.
91
92     if not isfile(WEIGHT_FILE):
93         print "WARNING: Can not load %s, please run RNNTrain.py first."%(WEIGHT_FILE)
94         return None
95     else:
96         model = getTestModel()
97         model.load_weights(WEIGHT_FILE)
98         return model
99
100 def trainModel(model, seqLen, Xtrain, ytrain, Xvalid, yvalid, nClasses, encodedTunes, le, incre
101 print np.array(Xtrain).shape, np.array(ytrain).shape, np.array(Xvalid).shape, np.array(yv
102 WEIGHT_FILE_STORE = './result/weight_me%d_lr%g_seq%d%s.h5'%(MAX_EPOCH, LR, seqLen, QUES
103 print "Training with learning rate %g with sequence %d"%(LR, seqLen)
104 mean_tr_accs =[]
105 mean_tr_losses =[]
106 mean_te_accs =[]
107 mean_te_losses =[]
108 previous_epoch = 0
109 # Load weights if incrementalW is true, and previous weights exist.
110 if incrementalTraining and isfile(WEIGHT_FILE):
111     model.load_weights(WEIGHT_FILE)
112     print "Loaded weights from %s"%(WEIGHT_FILE)
113     mean_tr_accs, mean_tr_losses, mean_te_accs, mean_te_losses = loadPkl(RESET_FILE)
114     print "Loaded history.pkl from %s"%(RESULT_FILE)
115     previous_epoch = len(mean_tr_accs)
116 elif incrementalTraining and not isfile(WEIGHT_FILE):
117     print "Did not find previous weights. Cannot load previous weights."
118 elif isfile(WEIGHT_FILE):
119     print "Over-writing previous weights %s"%(WEIGHT_FILE)
120 else:
121     print "Creating %s for the first time."%(WEIGHT_FILE)
122
123 pre_test_loss = float("inf")
124 incre = 0
125 model_pre = model
126 optimal_model = None
127 for epoch in range(MAX_EPOCH):
128     print "train part Epoch:%d/%d"%(previous_epoch+epoch, previous_epoch+MAX_EPOCH)
129     tr_accs = []
130     tr_losses = []
131     n_batch = len(Xtrain)/BATCH_SIZE
132     print '%d samples %d batches with batch size of %d'%(len(Xtrain),
133                                     n_batch,

```



```

134                                                                 BATCH.SIZE)
135
136     for i in tqdm(range(n_batch)):
137         x_seqs = np.array(Xtrain[i*BATCH.SIZE:(i+1)*BATCH.SIZE])
138         y_seqs = np.array(ytrain[i*BATCH.SIZE:(i+1)*BATCH.SIZE])
139         for j in range(seqLen):
140             # conver every feature to (numsample, lenth, dim) format.
141             # here x_hot is in (batch_size, l, nClasses) format
142             # y_hot is in (batch_size, nClasses) format
143             x_hot, y_hot = processSample(x_seqs[:, [j]],
144                                         y_seqs[:, [j]],
145                                         nClasses)
146             tr_loss, tr_acc = model.train_on_batch(x_hot, y_hot)
147             tr_accs.append(tr_acc)
148             tr_losses.append(tr_loss)
149         model.reset_states()
150         mean_tr_acc = np.mean(tr_accs)
151         mean_tr_accs.append(mean_tr_acc)
152         mean_tr_loss = np.mean(tr_losses)
153         mean_tr_losses.append(mean_tr_loss)
154         print('accuracy_training = {}'.format(mean_tr_acc))
155         print('loss_training = {}'.format(mean_tr_loss))
156         print('-----')
157
158     te_accs = []
159     te_losses = []
160     n_batch = len(Xvalid)/BATCH.SIZE
161     for i in tqdm(range(n_batch)):
162         x_seqs = np.array(Xvalid[i*BATCH.SIZE:(i+1)*BATCH.SIZE])
163         y_seqs = np.array(yvalid[i*BATCH.SIZE:(i+1)*BATCH.SIZE])
164         for j in range(seqLen):
165             x_hot, y_hot = processSample(x_seqs[:, [j]],
166                                         y_seqs[:, [j]],
167                                         nClasses)
168             te_loss, te_acc = model.test_on_batch(x_hot,
169                                                  y_hot)
170             te_accs.append(te_acc)
171             te_losses.append(te_loss)
172         model.reset_states()
173
174     mean_te_acc = np.mean(te_accs)
175     mean_te_accs.append(mean_te_acc)
176     mean_te_loss = np.mean(te_losses)
177     mean_te_losses.append(mean_te_loss)
178     print('accuracy_testing = %g'%(mean_te_acc))
179     print('loss_testing = %g'%(mean_te_loss))
180     print('-----')
181
182     # Early Stopping
183     if pre_test_loss <= mean_te_loss:
184         incre += 1
185         if incre == 1:
186             optimal_model = model_pre
187     else:
188         incre = 0
189         model_pre = model
190         optimal_model = None
191
192     if incre >= EARLY_STOPPING:
193         print("Early stopping at %d-%d steps"%(epoch, EARLY_STOPPING))
194         break
195
196     pre_test_loss = mean_te_loss
197
198     # Check music generation every 20 epoch.
199     if epoch%20 ==0:

```

```

199         model.save_weights(TEMP_MODEL_FILE)
200         te_model = getTestModel()
201         te_model.load_weights(TEMP_MODEL_FILE)
202         final_tune = generateMusic(te_model, encodedTunes, le)
203         filename = '%s%d.txt'%(TUNE_STORAGE, epoch + previous_epoch)
204         outputfile = open(filename, 'w')
205         outputfile.write("%s" % ''.join(final_tune))
206
207     # Save weights.
208     if optimal_model != None:
209         model = optimal_model
210         data = mean_tr_accs[: -EARLY_STOPPING], mean_tr_losses[: -EARLY_STOPPING], mean_te_accs[: -EARLY_STOPPING]
211     else:
212         data = mean_tr_accs, mean_tr_losses, mean_te_accs, mean_te_losses
213
214     model.save_weights(WEIGHT_FILE)
215     model.save_weights(WEIGHT_FILE_STORE)
216
217     savePkl(data, RESULT_FILE)
218     mean_tr_accs, mean_tr_losses, mean_te_accs, mean_te_losses = loadPkl(RESULT_FILE)
219     savefig([mean_tr_accs, mean_te_accs], 'model_accuracy',
220             'number_of_epochs',
221             'Accuracy',
222             ['Train', 'Test'],
223             ACC_FILE)
224     savefig([mean_tr_losses, mean_te_losses], 'model_loss',
225             'number_of_epochs',
226             'Loss',
227             ['Train', 'Test'],
228             LOSS_FILE)
229     print "Done Saving"
230
231 def generateMusic(model, encodedTunes, le, maxTunes = 6, maxTuneLen = TUNE_MAXLEN):
232     nClasses = len(list(le.classes_))
233     # prime the network with a sequence randomly selected from tunes.
234     random.shuffle(encodedTunes)
235     start_seqs = []
236     chosen_tune = random.randint(0, len(encodedTunes)-1)
237     # Copy the same selected tune for Batch Size
238     for i in range(BATCH_SIZE):
239         start_seqs.append(encodedTunes[chosen_tune][:PRIME_LEN])
240     y_prime = le.inverse_transform(start_seqs[0])
241     print "Prime the sequence with [0]x%d:\n-----\n%s\n-----"%(BATCH_SIZE, ''.join(y_prime))
242     start_X = np.array(start_seqs)
243     prime_pred = []
244     for j in range(PRIME_LEN):
245         # convert every feature to (numsample, lenth, dim) format.
246         # here x_hot is in (batch_size, 1, nClasses) format
247         # y_hot is in (batch_size, nClasses) format
248         x_hot = np_utils.to_categorical(start_X[:, [j]], nClasses)
249         x_hot = np.expand_dims(x_hot, axis=1)
250         y_pred_prob = model.predict_on_batch(x_hot)
251         y_pred_code = np.argmax(y_pred_prob[0])
252         # take the prediction for the char j in first prime sequence.
253         prime_pred.append(le.inverse_transform(y_pred_code))
254
255     print "prime_pred_info", len(prime_pred)
256     # convert last prob prime output to label to display and check.
257     print "Prime result:\n-----\n%s\n-----"%(''.join(prime_pred))
258
259     lastToken = []
260     # Take the last predicted character, continue feeding it to model as input.
261     x_next_hot = prob2input(y_pred_prob)
262     more_preds = [prime_pred[-1]]
263     tunes_count = 0

```

```

264     while((''.join(lastToken) != END_TOKEN or tunes_count < maxTunes) and len(more_preds) < m
265         y_pred_prob = model.predict_on_batch(x_next_hot)
266         x_next_hot = prob2input(y_pred_prob)
267         y_pred_code = np.argmax(y_pred_prob[0])
268         pred_char = le.inverse_transform(y_pred_code)
269         lastToken.append(pred_char)
270         if len(lastToken) == 6:
271             lastToken = lastToken[1:]
272         more_preds.append(pred_char)
273         if ''.join(lastToken) == END_TOKEN:
274             tunes_count +=1
275     result = [y_prime.tolist()[0]] # Add first character
276     result.extend(prime_pred) # Add rest to result.
277     result.extend(more_preds)
278     return result
279
280 def runRNN(seqLen):
281     # Generate result folders for result and models.
282     if not isdir(MODEL.STORAGE):
283         makedirs(MODEL.STORAGE)
284
285     if not isdir(RESULT.STORAGE):
286         makedirs(RESULT.STORAGE)
287
288     # Generate folders for tunes storage.
289     if not isdir(TUNE.STORAGE):
290         makedirs(TUNE.STORAGE)
291
292     PRE_PROCESS_STORE = '%spre_processed_seq%d-%s'%(DATA.STORAGE, MAX_SEQ_LEN, QUESTION_ID)
293     preprocesspk1 = '%s.pk1'%(PRE_PROCESS_STORE)
294     preprocessstxt = '%s.txt'%(PRE_PROCESS_STORE)
295
296     if not isfile(preprocesspk1):
297         print "Read data from file %s."%(preprocesspk1)
298         tunes = loadTunes()
299         print "Encode chars to ints."
300         encoded_tunes, label_encoder = label2code(tunes)
301         print "Chop txt to sequences of %d length"%(MAX_SEQ_LEN)
302         X, y = prepDataSeq(encoded_tunes, MAX_SEQ_LEN)
303         print "Found total %d sequences"%(len(X))
304         x_train, y_train, x_valid, y_valid = partition(X, y)
305         print "Done partition."
306         data = x_train, y_train, x_valid, y_valid, encoded_tunes, label_encoder
307         savePkl(data, preprocesspk1)
308     else:
309         x_train, y_train, x_valid, y_valid, encoded_tunes, label_encoder = loadPkl(preprocesspk1)
310
311     f = open(preprocessstxt, 'w')
312     for seq in x_train:
313         f.write(''.join(label_encoder.inverse_transform(seq)))
314         f.write('\n-----\n')
315     print "Done saving."
316
317     nClasses = len(list(label_encoder.classes_))
318     print "There are %d tunes. %d classes."%(len(encoded_tunes), nClasses)
319     print "%d sequences for training, %d sequences for testing"%(len(x_train), len(x_valid))
320     #print label_encoder.classes_
321
322     model = getModel(nClasses)
323     trainModel(model, seqLen, x_train, y_train, x_valid, y_valid, nClasses, encoded_tunes, label_encoder)
324
325 if __name__ == "__main__":
326     for seq_len in [20, 50, 70, 90, 110]:
327         runRNN(seq_len)

```

Listing 5: ReportQ4f.py for report generation for q4.f

```

1  from os.path import isfile, isdir
2  from os import makedirs
3  from keras.layers.recurrent import SimpleRNN
4  from keras.layers.core import Dense
5  from keras.models import Sequential
6  from keras.optimizers import RMSprop, Adagrad
7  from keras.callbacks import EarlyStopping, Callback
8  from utilities import loadTunes, partition, label2code, prepDataSeq, processSample, savePkl, l
9  import time
10 from tqdm import *
11 import numpy as np
12 import matplotlib.pyplot as plt
13 from matplotlib import cm
14 import random
15 from keras.utils import np_utils
16 from keras import backend as K
17 from keras.models import Model
18 from itertools import izip
19
20 LR = 0.01
21 QUESTION_ID = 'Q4e'
22 N_HIDDEN_UNIT = 100
23 INCREMENT_TRAINING = True
24 BATCH_SIZE = 64 # How many samples will be processed simultaneously.
25 PRIME_LEN = 30 # Prime the music generation with PRIME_LEN characters.
26 MAX_EPOCH = 50 # How many epoch will be run.
27 EARLY_STOPPING = 6 # Stop training if validation set's loss stop decreasing.
28 DATA_STORAGE = '../data/'
29 # For results and models
30 MODEL_STORAGE = '../model/'
31 RESULT_STORAGE = '../result/'
32 MODEL_FILE = '%smvRNN_%s.h5'%(MODEL_STORAGE, QUESTION_ID)
33 TEMP_MODEL_FILE = '%stemp_test_model_%s.h5'%(MODEL_STORAGE, QUESTION_ID)
34 WEIGHT_FILE = '%sweight_me%d_lr%s.h5'%(RESULT_STORAGE, MAX_EPOCH, LR, QUESTION_ID)
35
36 RESULT_FILE = '%sresult_%g_%s.pkl'%(RESULT_STORAGE, LR, QUESTION_ID)
37 ACC_FILE = '%saccuracy_%g_%s.png'%(RESULT_STORAGE, LR, QUESTION_ID)
38 LOSS_FILE = '%sloss_%g_%s.png'%(RESULT_STORAGE, LR, QUESTION_ID)
39 # For generateMusic
40 TUNE_STORAGE = '%stunes/tune_%s/'%(RESULT_STORAGE, QUESTION_ID)
41
42 END_TOKEN = "<end>"
43 TUNE_MAX_LEN = 5000 # maximum length of the tune is around 4900, this ensures that we will sto
44
45 TEMPERATURE = 1
46 MAX_SEQ_LEN = 110
47 def getTemperature():
48     return TEMPERATURE
49
50 def temperature_activation(a):
51     T = TEMPERATURE
52     return K.softmax(a/T)
53
54 def getTestModel(output_dim = 94, lr=LR, n_hidden_units =N_HIDDEN_UNIT, modelfile = MODEL_FILE
55     # as the first layer in a Sequential model
56     model = Sequential()
57     input_length = 1 # number of timesteps.
58     input_dim = output_dim # number of features after one-hot encoding.
59     model.add(SimpleRNN(n_hidden_units,
60                         batch_input_shape = (BATCH_SIZE, input_length, input_dim),
61                         return_sequences=False, # return last output in the output sequence fo
62                         stateful=True, # last state for every sample at index i in a batch wi
63                         unroll = True)) # network will be unrolled, speedup TF.

```

```

64     model.add(Dense(output_dim, activation=temperature_activation))
65     #opt = RMSprop(lr=lr)
66     opt = Adagrad(lr=lr)
67     model.compile(optimizer=opt,
68                   loss='categorical_crossentropy',
69                   metrics=['accuracy'])
70     model.summary()
71     model.save(modelfile)
72     return model
73
74 def getModel(output_dim = 94, lr=LR, n_hidden_units =N_HIDDEN_UNIT, modelfile = MODEL_FILE):
75     # as the first layer in a Sequential model
76     model = Sequential()
77     input_length = 1 # number of timesteps.
78     input_dim = output_dim # number of features after one-hot encoding.
79     model.add(SimpleRNN(n_hidden_units,
80                         batch_input_shape = (BATCH_SIZE, input_length, input_dim),
81                         return_sequences=False, # return last output in the output sequence for
82                         stateful=True, # last state for every sample at index i in a batch with
83                         unroll = True)) # network will be unrolled, speedup TF.
84     model.add(Dense(output_dim, activation='softmax'))
85     #opt = RMSprop(lr=lr)
86     opt = Adagrad(lr=lr)
87     model.compile(optimizer=opt,
88                   loss='categorical_crossentropy',
89                   metrics=['accuracy'])
90     model.summary()
91     model.save(modelfile)
92     return model
93
94 def getTestModelWithWeights():
95     # Get default model with the final weights loaded.
96
97     if not isfile(WEIGHT_FILE):
98         print "WARNING: Can not load %s, please run RNNTrain.py first."%(WEIGHT_FILE)
99         return None
100    else:
101        model = getTestModel()
102        model.load_weights(WEIGHT_FILE)
103        return model
104
105 def trainModel(model, seqLen, Xtrain, ytrain, Xvalid, yvalid, nClasses, encodedTunes, le, incrementalTraining):
106     print np.array(Xtrain).shape, np.array(ytrain).shape, np.array(Xvalid).shape, np.array(yvalid).shape
107     WEIGHT_FILE_STORE = '../result/weight_me%d_lr%g_seq%d_%s.h5'%(MAX_EPOCH, LR, seqLen, QUES)
108     print "Training with learning rate %g with sequence %d"%(LR, seqLen)
109     mean_tr_accs = []
110     mean_tr_losses = []
111     mean_te_accs = []
112     mean_te_losses = []
113     previous_epoch = 0
114     # Load weights if incrementalTraining is true, and previous weights exist.
115     if incrementalTraining and isfile(WEIGHT_FILE):
116         model.load_weights(WEIGHT_FILE)
117         print "Loaded weights from %s"%(WEIGHT_FILE)
118         mean_tr_accs, mean_tr_losses, mean_te_accs, mean_te_losses = loadPkl(RESET_FILE)
119         print "Loaded history pkl from %s"%(RESET_FILE)
120         previous_epoch = len(mean_tr_accs)
121     elif incrementalTraining and not isfile(WEIGHT_FILE):
122         print "Did not find previous weights. Cannot load previous weights."
123     elif isfile(WEIGHT_FILE):
124         print "Over-writing previous weights %s"%(WEIGHT_FILE)
125     else:
126         print "Creating %s for the first time."%(WEIGHT_FILE)
127
128     pre_test_loss = float("inf")

```

```

129     incre = 0
130     model_pre = model
131     optimal_model = None
132     for epoch in range(MAX_EPOCH):
133         print "train_part_Epoch:%d/%d"%(previous_epoch+epoch, previous_epoch+MAX_EPOCH)
134         tr_accs = []
135         tr_losses = []
136         n_batch = len(Xtrain)/BATCH_SIZE
137         print '%d samples %d batches with batch size of %d'%(len(Xtrain),
138                                                                n_batch,
139                                                                BATCH_SIZE)
140         for i in tqdm(range(n_batch)):
141             x_seqs = np.array(Xtrain[i*BATCH_SIZE:(i+1)*BATCH_SIZE])
142             y_seqs = np.array(ytrain[i*BATCH_SIZE:(i+1)*BATCH_SIZE])
143             for j in range(seqLen):
144                 # conver every feature to (numsample, lenth, dim) format.
145                 # here x_hot is in (batch_size, 1, nClasses) format
146                 # y_hot is in (batch_size, nClasses) format
147                 x_hot, y_hot = processSample(x_seqs[:, [j]],
148                                             y_seqs[:, [j]],
149                                             nClasses)
150                 tr_loss, tr_acc = model.train_on_batch(x_hot, y_hot)
151                 tr_accs.append(tr_acc)
152                 tr_losses.append(tr_loss)
153             model.reset_states()
154             mean_tr_acc = np.mean(tr_accs)
155             mean_tr_accs.append(mean_tr_acc)
156             mean_tr_loss = np.mean(tr_losses)
157             mean_tr_losses.append(mean_tr_loss)
158             print('accuracy_training = {}'.format(mean_tr_acc))
159             print('loss_training = {}'.format(mean_tr_loss))
160             print('-----')
161
162         te_accs = []
163         te_losses = []
164         n_batch = len(Xvalid)/BATCH_SIZE
165         for i in tqdm(range(n_batch)):
166             x_seqs = np.array(Xvalid[i*BATCH_SIZE:(i+1)*BATCH_SIZE])
167             y_seqs = np.array(yvalid[i*BATCH_SIZE:(i+1)*BATCH_SIZE])
168             for j in range(seqLen):
169                 x_hot, y_hot = processSample(x_seqs[:, [j]],
170                                             y_seqs[:, [j]],
171                                             nClasses)
172                 te_loss, te_acc = model.test_on_batch(x_hot,
173                                                       y_hot)
174                 te_accs.append(te_acc)
175                 te_losses.append(te_loss)
176             model.reset_states()
177
178             mean_te_acc = np.mean(te_accs)
179             mean_te_accs.append(mean_te_acc)
180             mean_te_loss = np.mean(te_losses)
181             mean_te_losses.append(mean_te_loss)
182             print('accuracy_testing = %g'%(mean_te_acc))
183             print('loss_testing = %g'%(mean_te_loss))
184             print('-----')
185
186         # Early Stopping
187         if pre_test_loss <= mean_te_loss:
188             incre += 1
189             if incre == 1:
190                 optimal_model = model_pre
191         else:
192             incre = 0
193             model_pre = model

```

```

194         optimal_model = None
195
196     if incre >= EARLY_STOPPING:
197         print "Early stopping at %d-%d steps"%(epoch, EARLY_STOPPING)
198         break
199
200     pre_test_loss = mean_te_loss
201
202     # Check music generation every 20 epoch.
203     if epoch%20 ==0:
204         model.save_weights(TEMP_MODEL_FILE)
205         te_model = getTestModel()
206         te_model.load_weights(TEMP_MODEL_FILE)
207         final_tune = generateMusic(te_model, encodedTunes, le)
208         filename = '%s%d.txt'%(TUNE_STORAGE, epoch + previous_epoch)
209         outputfile = open(filename, 'w')
210         outputfile.write("%s" % ''.join(final_tune))
211
212     # Save weights.
213     if optimal_model != None:
214         model = optimal_model
215         data = mean_tr_accs[:-EARLY_STOPPING], mean_tr_losses[:-EARLY_STOPPING], mean_te_accs[
216     else:
217         data = mean_tr_accs, mean_tr_losses, mean_te_accs, mean_te_losses
218
219     model.save_weights(WEIGHT_FILE)
220     model.save_weights(WEIGHT_FILE_STORE)
221
222     savePkl(data, RESULT_FILE)
223     mean_tr_accs, mean_tr_losses, mean_te_accs, mean_te_losses = loadPkl(RESULT_FILE)
224     savefig([mean_tr_accs, mean_te_accs], 'model_accuracy',
225             'number_of_epochs',
226             'Accuracy',
227             ['Train', 'Test'],
228             ACC_FILE)
229     savefig([mean_tr_losses, mean_te_losses], 'model_loss',
230             'number_of_epochs',
231             'Loss',
232             ['Train', 'Test'],
233             LOSS_FILE)
234     print "Done Saving"
235
236 def generateMusic(model, encodedTunes, le, maxTunes = 6, maxTuneLen = TUNE_MAX_LEN):
237     nClasses = len(list(le.classes_))
238     # prime the network with a sequence randomly selected from tunes.
239     random.shuffle(encodedTunes)
240     start_seqs = []
241     chosen_tune = random.randint(0, len(encodedTunes)-1)
242     # Copy the same selected tune for Batch_Size
243     for i in range(BATCH_SIZE):
244         start_seqs.append(encodedTunes[chosen_tune][:PRIME_LEN])
245     y_prime = le.inverse_transform(start_seqs[0])
246     print "Prime the sequence with [0]x%d: \n-----\n%s\n-----"%(BATCH_SIZE, ''.join(y_prime))
247     start_X = np.array(start_seqs)
248     prime_pred = []
249     for j in range(PRIME_LEN):
250         # convert every feature to (numsample, lenth, dim) format.
251         # here x_hot is in (batch_size, 1, nClasses) format
252         # y_hot is in (batch_size, nClasses) format
253         x_hot = np_utils.to_categorical(start_X[:, [j]], nClasses)
254         x_hot = np.expand_dims(x_hot, axis=1)
255         y_pred_prob = model.predict_on_batch(x_hot)
256         y_pred_code = np.argmax(y_pred_prob[0])
257         # take the prediction for the char j in first prime sequence.
258         prime_pred.append(le.inverse_transform(y_pred_code))

```

```

259
260 print "prime_pred_info", len(prime_pred)
261 # convert last prob prime output to label to display and check.
262 print "Prime_result:\n————\n%s\n————"%( ''.join(prime_pred))
263
264 lastToken = []
265 # Take the last predicted character, continue feeding it to model as input.
266 x_next_hot = prob2input(y_pred_prob)
267 more_preds = [prime_pred[-1]]
268 tunes_count = 0
269 while(( ''.join(lastToken) != END_TOKEN or tunes_count < maxTunes) and len(more_preds) < maxTunes):
270     y_pred_prob = model.predict_on_batch(x_next_hot)
271     x_next_hot = prob2input(y_pred_prob)
272     y_pred_code = np.argmax(y_pred_prob[0])
273     pred_char = le.inverse_transform(y_pred_code)
274     lastToken.append(pred_char)
275     if len(lastToken) == 6:
276         lastToken = lastToken[1:]
277     more_preds.append(pred_char)
278     if ''.join(lastToken) == END_TOKEN:
279         tunes_count +=1
280 result = [y_prime.tolist()[0]] # Add first character
281 result.extend(prime_pred) # Add rest to result.
282 result.extend(more_preds)
283 return result
284
285 def runRNN(seqLen):
286     # Generate result folders for result and models.
287     if not isdir(MODEL_STORAGE):
288         makedirs(MODEL_STORAGE)
289
290     if not isdir(RESULT_STORAGE):
291         makedirs(RESULT_STORAGE)
292
293     # Generate folders for tunes storage.
294     if not isdir(TUNE_STORAGE):
295         makedirs(TUNE_STORAGE)
296
297     PRE_PROCESS_STORE = '%spre_processed_seq%d_%s'%(DATA_STORAGE, MAX_SEQ_LEN, QUESTION_ID)
298     preprocesspkl = '%s.pkl'%(PRE_PROCESS_STORE)
299     preprocesstxt = '%s.txt'%(PRE_PROCESS_STORE)
300
301     if not isfile(preprocesspkl):
302         print "Read_data_from_file_%s"%(preprocesspkl)
303         tunes = loadTunes()
304         print "Encode_chars_to_ints."
305         encoded_tunes, label_encoder = label2code(tunes)
306         print "Chop_txt_to_sequences_of_%d_length" %(MAX_SEQ_LEN)
307         X, y = prepDataSeq(encoded_tunes, MAX_SEQ_LEN)
308         print "Found_total_%d_sequences"%(len(X))
309         x_train, y_train, x_valid, y_valid = partition(X, y)
310         print "Done_partition."
311         data = x_train, y_train, x_valid, y_valid, encoded_tunes, label_encoder
312         savePkl(data, preprocesspkl)
313     else:
314         x_train, y_train, x_valid, y_valid, encoded_tunes, label_encoder = loadPkl(preprocesspkl)
315
316     f = open(preprocesstxt, 'w')
317     for seq in x_train:
318         f.write(' '.join(label_encoder.inverse_transform(seq)))
319         f.write('\n————\n')
320     print "Done_saving."
321
322     nClasses = len(list(label_encoder.classes_))
323     print "There_are_%d_tunes_%d_classes"%(len(encoded_tunes), nClasses)

```



```

324     print "%d_sequences_for_training, %d_sequences_for_testing"%(len(x_train), len(x_valid))
325     #print label_encoder.classes_
326
327     model = getModel(nClasses)
328     trainModel(model, seqLen, x_train, y_train, x_valid, y_valid, nClasses, encoded_tunes, label_encoder)
329
330 def show_values(pc, ax, dchar, fmt="%s", **kw):
331     invisible_char = dict()
332     invisible_char['\t'] = 'bt'
333     invisible_char['\n'] = 'bn'
334     invisible_char['\r'] = 'br'
335     invisible_char['_'] = 'sp'
336     invisible_char[chr(127)] = ''
337
338     pc.update_scalarmappable()
339     for p, color in izip(pc.get_paths(), pc.get_facecolors(), dchar):
340         x, y = p.vertices[:2, :].mean(0)
341         if np.all(color[:3] > 0.5):
342             color = (0.0, 0.0, 0.0)
343         else:
344             color = (1.0, 1.0, 1.0)
345
346         if char in invisible_char:
347             char = invisible_char[char]
348         ax.text(x, y, fmt % char, ha="center", va="center", color=color, **kw)
349
350 if __name__ == "__main__":
351     model = getModel()
352     model.load_weights(WEIGHT_FILE)
353
354     nmodel = Model(input=model.input, output=model.layers[-2].output)
355     nmodel.summary()
356
357     PRE_PROCESS_STORE = '%spre_processed_seq%d%s'%(DATA_STORAGE, MAX_SEQ_LEN, QUESTION_ID)
358     preprocesspk1 = '%s.pkl'%(PRE_PROCESS_STORE)
359     print "Read_data_from_file_%s"%(preprocesspk1)
360     tunes = loadTunes()
361     print "Encode_chars_to_ints."
362     encoded_tunes, label_encoder = label2code(tunes)
363
364     nClasses = len(list(label_encoder.classes_))
365
366     print "Read_data_from_file_generated_music_file."
367     tunes = loadTunes('../result/tunes/tune_Q4e/240.txt')
368     print "Encode_chars_to_ints."
369     encoded_tunes = map(label_encoder.transform, tunes)
370
371     X = encoded_tunes[5]
372     n = len(X)
373     print 'Length_of_selected_tune: %d'%(n)
374
375     for neuronID in range(100):
376         print 'Choosing_NeuronID: %d'%(neuronID)
377
378         echar = []
379         value = []
380         x_seqs = np.array([X for i in xrange(BATCH_SIZE)])
381         for i in range(n):
382             # conver every feature to (numsample, lenth, dim) format.
383             # here x_hot is in (batch_size, 1, nClasses) format
384             # y_hot is in (batch_size, nClasses) format
385             x_hot = np_utils.to_categorical(x_seqs[:, [i]], nClasses)
386             x_hot = np.expand_dims(x_hot, axis=1)
387
388             a_pred = nmodel.predict_on_batch(x_hot)

```

```

389
390         echar.append(x_seqs[0, i])
391         value.append(a_pred[0, neuronID])
392
393     dchar = label_encoder.inverse_transform(echar)
394     dchar = np.hstack((dchar, np.array([chr(127) for i in xrange(256-n)])))
395     value = np.hstack((value, np.array([0 for i in xrange(256-n)])))
396     value = np.reshape(value, (16,16))
397
398     fig, ax = plt.subplots()
399     heatmap = ax.pcolor(value, edgecolors='k', linestyle='dashed', linewidths=0.2, cmap=
400     plt.gca().invert_yaxis()
401
402     plt.colorbar(heatmap)
403     show_values(heatmap, ax, dchar)
404     plt.savefig('../result/heatmaps/heatmap_' + str(neuronID) + '.png', dpi=200)

```