CSE 253 Homework Assignment 4
Generating Music with Recurrent Networks

Sainan Liu
A13291871
sal131@eng.ucsd.edu

Hao-en Sung
A53204772
wrangle1005@gmail.com

Shiwei Song
A53206591
shs163@eng.ucsd.edu

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
```

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

Second generated music with T=1:

```
X:110
d:Cineaeanetaanss 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 eontereaeaipe
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 ||
```

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 Thenhavr 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.]

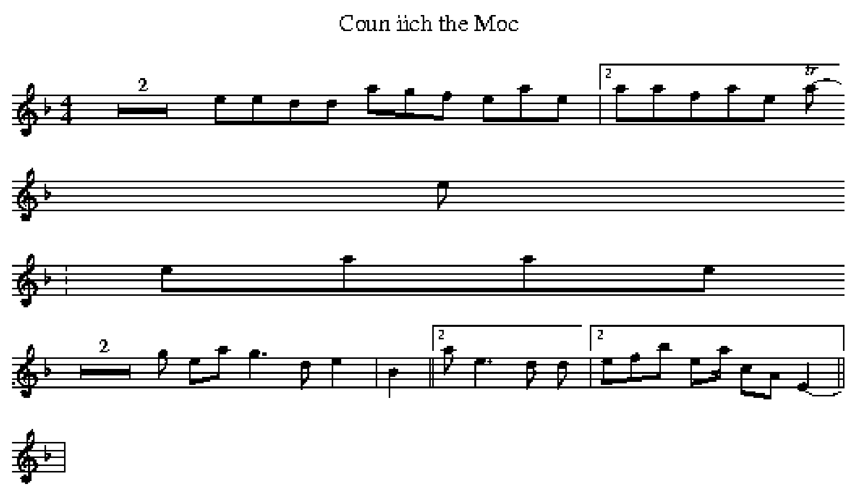

Figure 3: First music from the T=2 code above in standard music notation.

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"
```

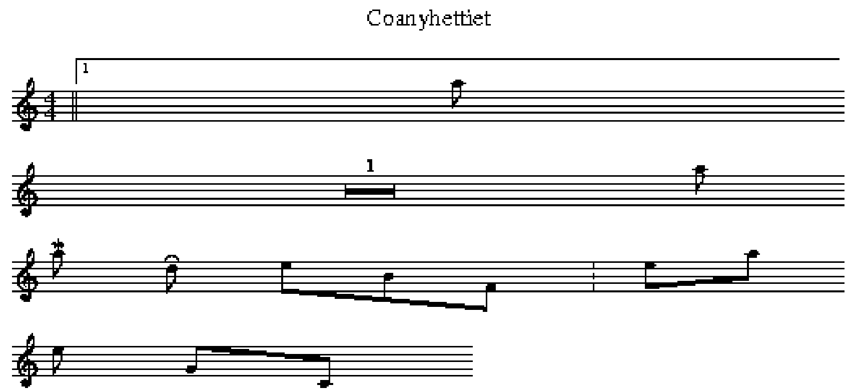

Figure 4: Second music from the T=2 code above in standard music notation.

3. T=0.5

First generated music with T=0.5:

```
X:11
T:Far
the Colle

T:Mnrtar|e  al the Crance
C:Trance
C:Tradce
C:Faghltoute 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 ||
```

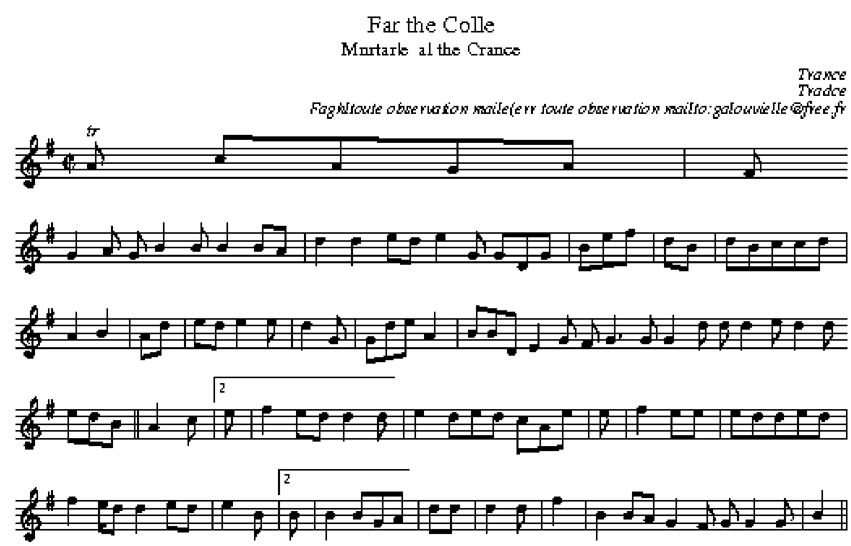

Figure 5: First music from the T=0.5 code above in standard music notation.

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

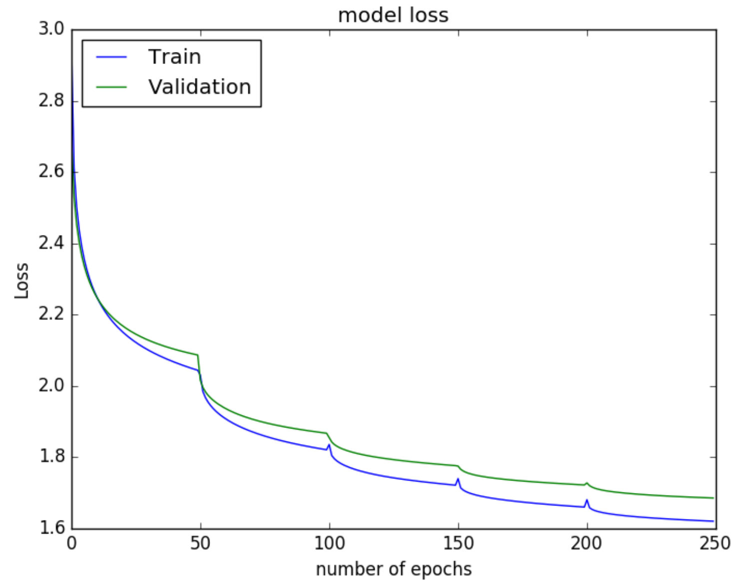

Figure 7: Training and validation loss vs number of epochs with sequence length increment for at most every 50 epoch. Sequence length = [20, 50, 70, 90, 110]; Batch size = 64; #Hidden = 100; optimizer=Adagrad.

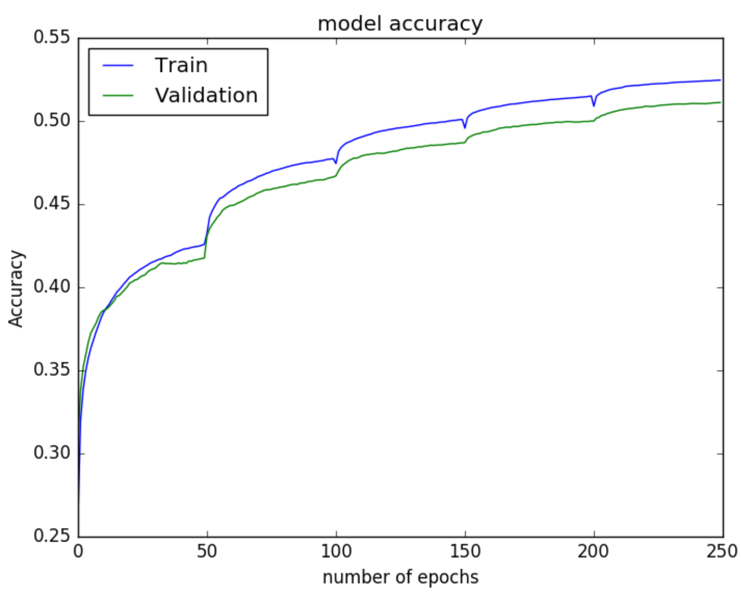

Figure 8: Training and validation accuracies vs number of epochs with sequence length increment for at most every 50 epoch. Sequence length = [20, 50, 70, 90, 110]; Batch size = 64; #Hidden = 100; optimizer=Adagrad.

(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

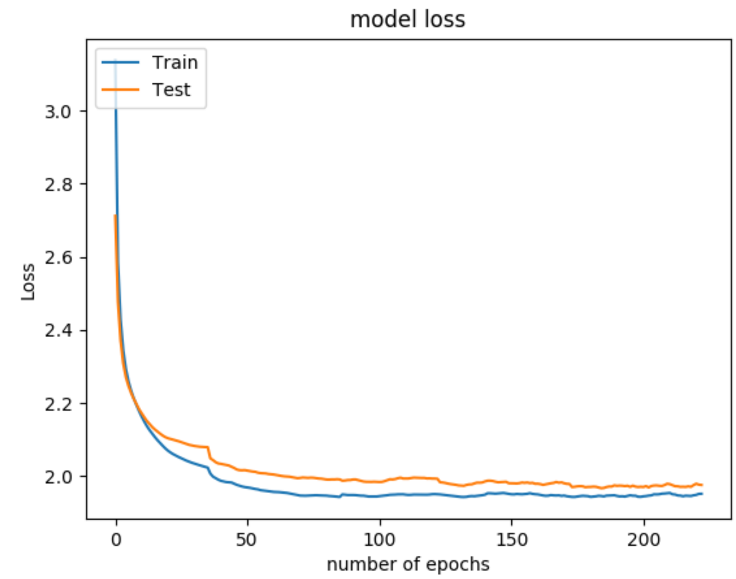

Figure 9: Training and validation loss vs number of epochs with 50 hidden units

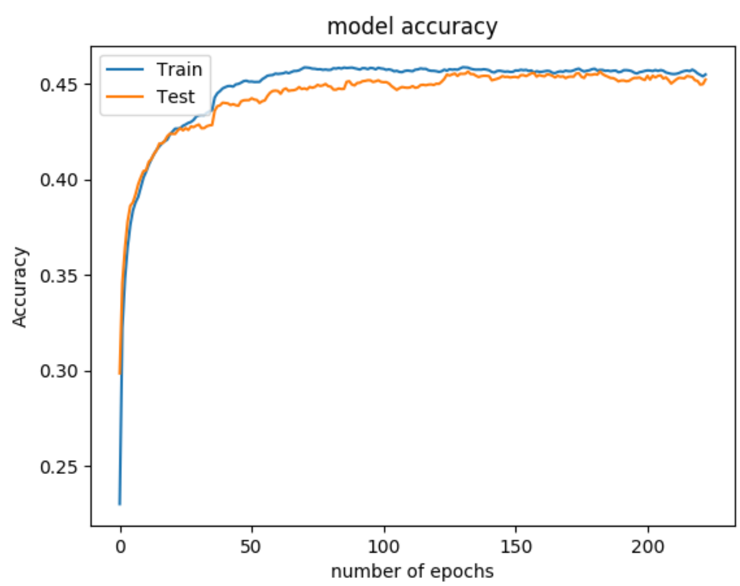

Figure 10: Training and validation accuracies vs number of epochs with 50 hidden units

2. 75 hidden units

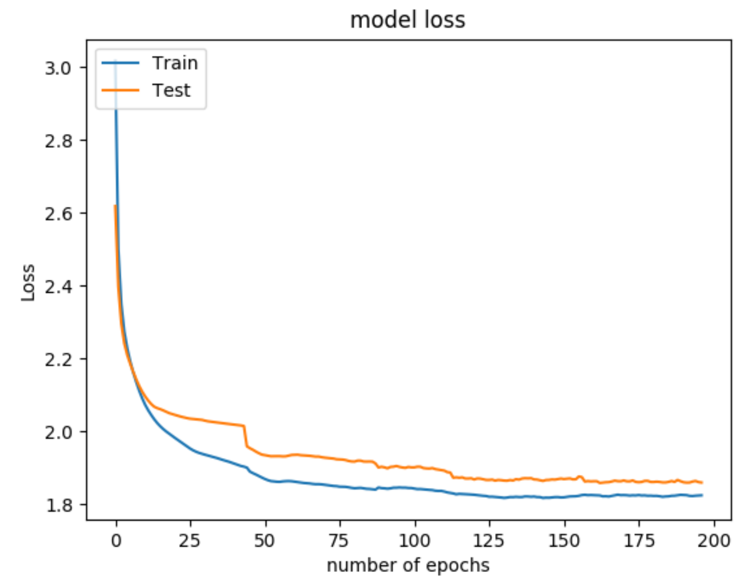

Figure 11: Training and validation loss vs number of epochs with 75 hidden units

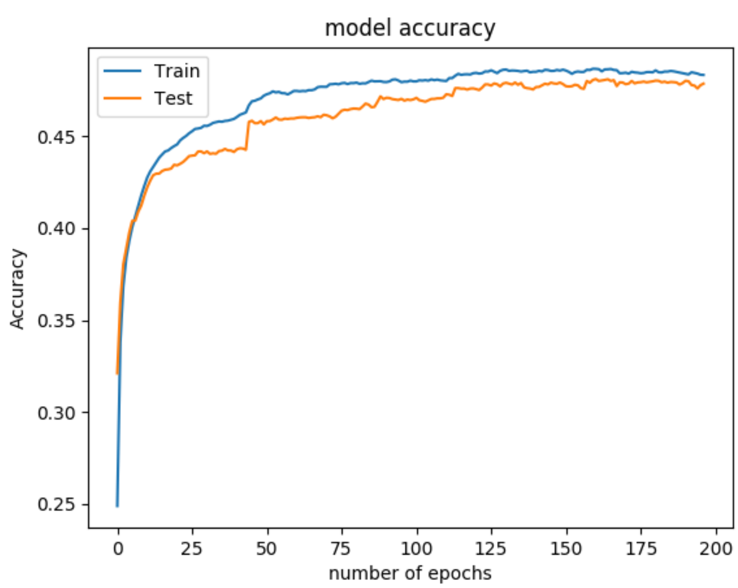

Figure 12: Training and validation accuracies vs number of epochs with 75 hidden units

3. 150 hidden units

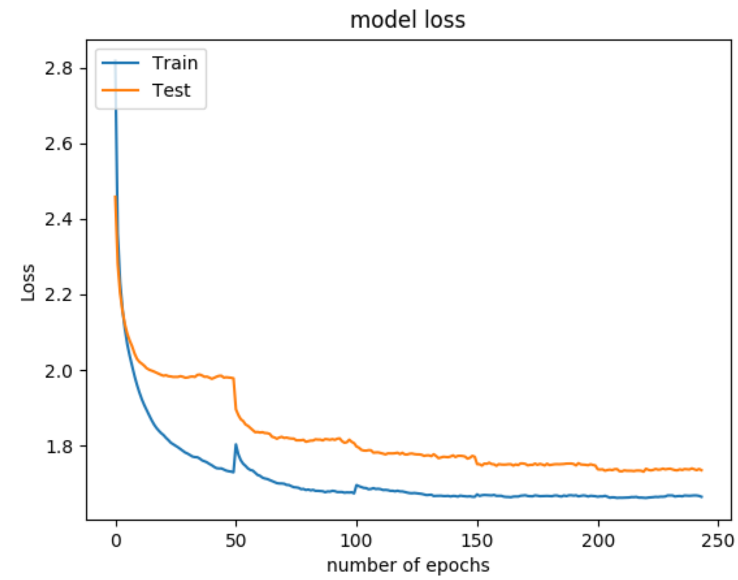

Figure 13: Training and validation loss vs number of epochs with 150 hidden units

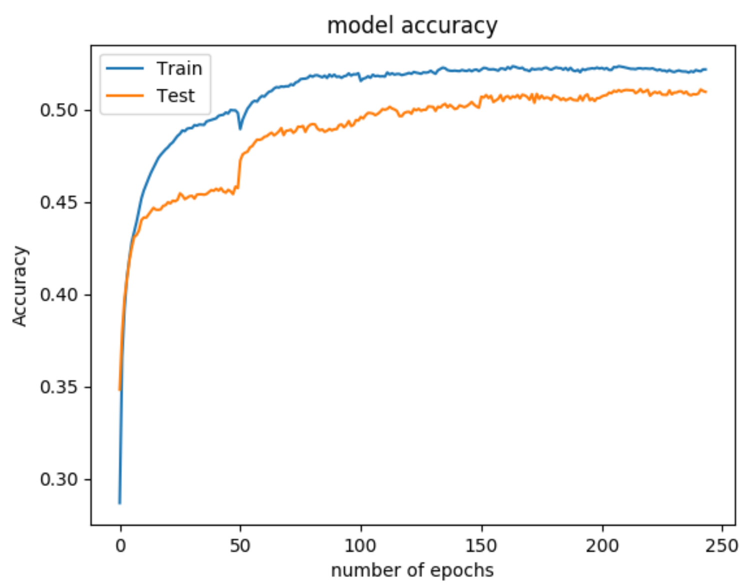

Figure 14: Training and validation accuracies vs number of epochs with 150 hidden units

As number of neurons in hidden layer increases, the loss decreases. The more neurons the network contains, the more information the network conatins, 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. $p = 0.1$

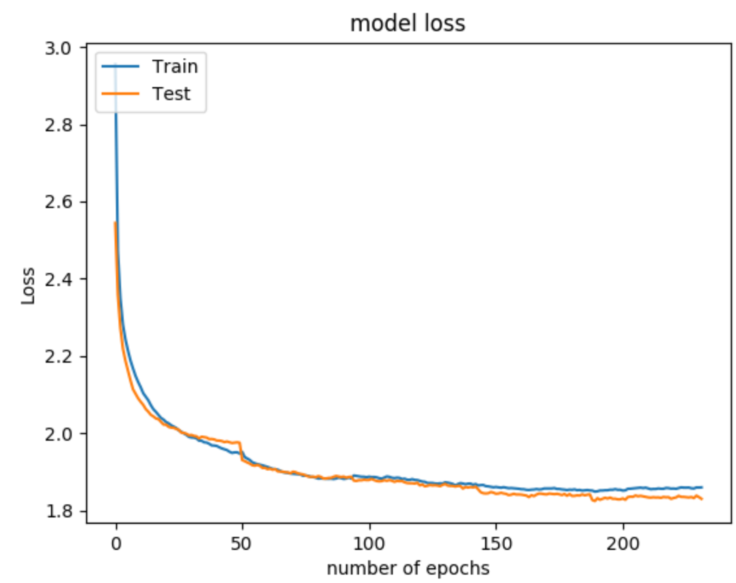

Figure 15: Training and validation loss vs number of epochs with $p = 0.1$

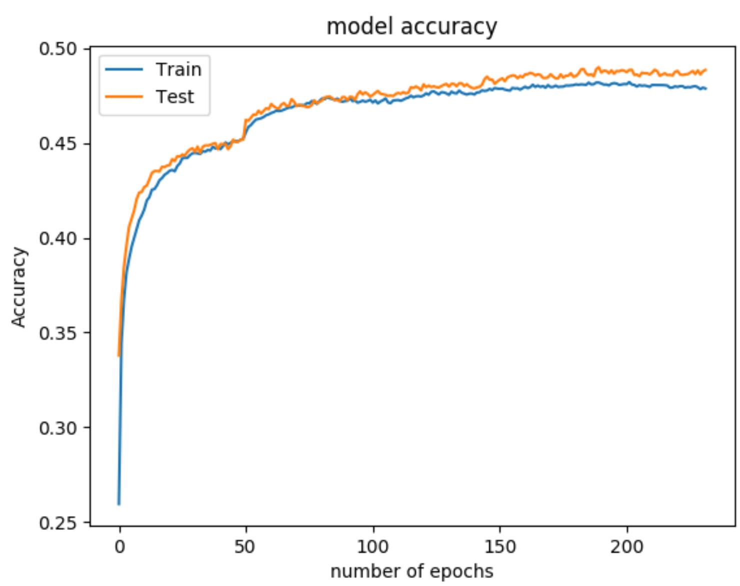

Figure 16: Training and validation accuracies vs number of epochs with $p = 0.1$

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 ||
```

Figure 17: Music generated with $p = 0.1$

2. $p = 0.2$

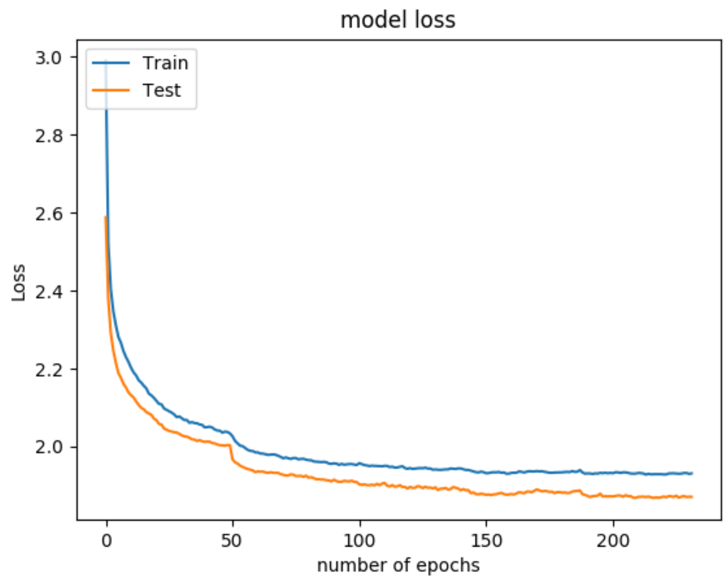

Figure 18: Training and validation loss vs number of epochs with $p = 0.2$

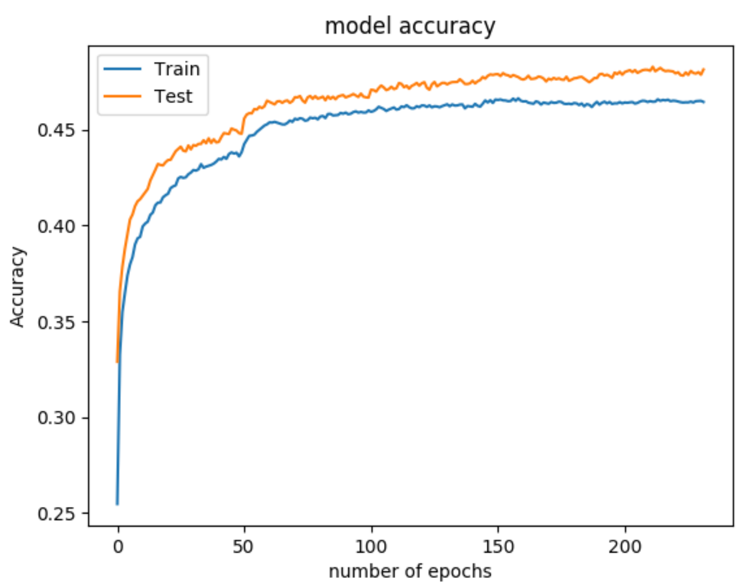

Figure 19: Training and validation accuracies vs number of epochs with $p = 0.2$

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 ceO1|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 ||
```

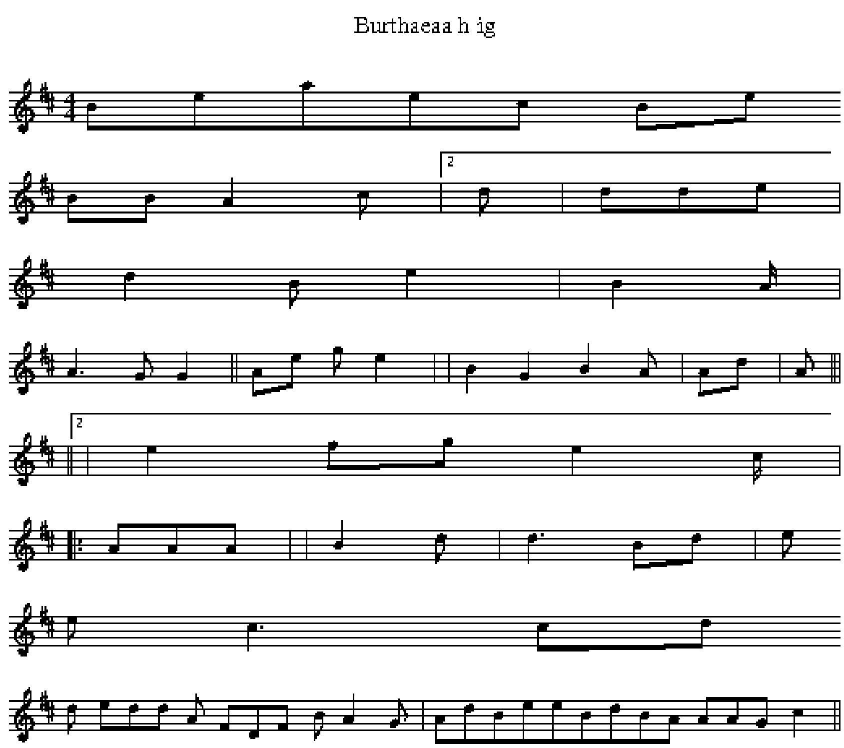

Figure 20: Music generated with $p = 0.2$

3. $p = 0.3$

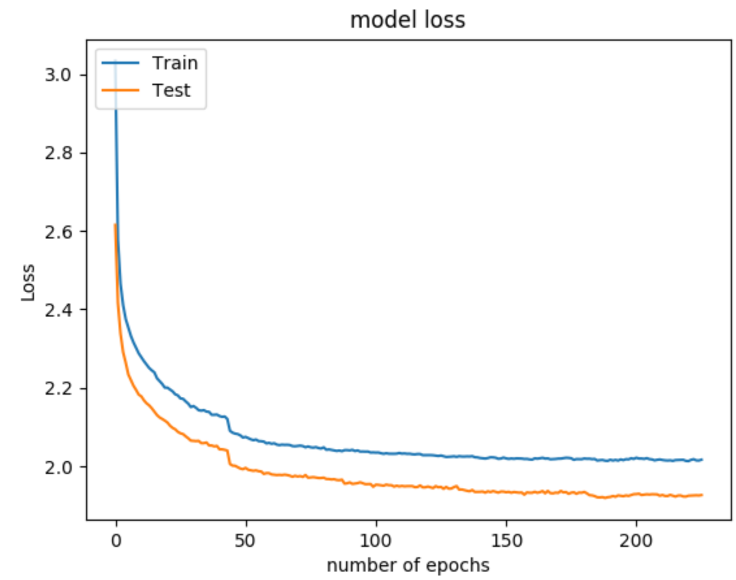

Figure 21: Training and validation loss vs number of epochs with $p = 0.3$

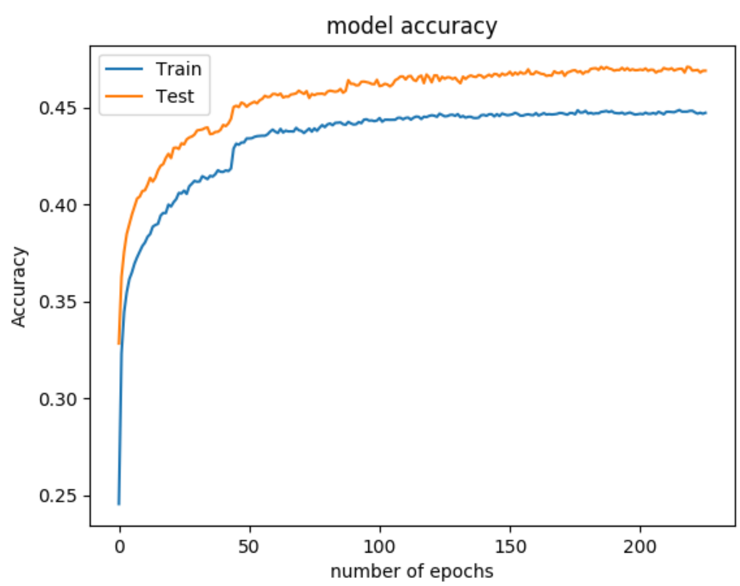

Figure 22: Training and validation accuracies vs number of epochs with $p = 0.3$

Music generated with $p = 0.3$

```
X:1

T:ClaeM'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  l
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:Fror: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 |
```

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.

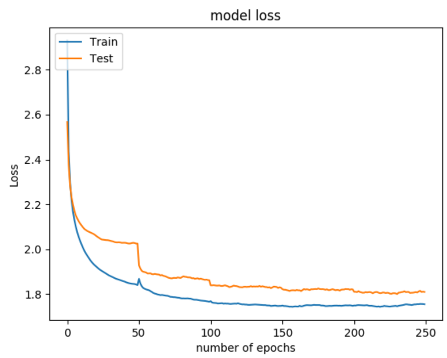

Figure 24: Training and Validation Loss for RMSprop

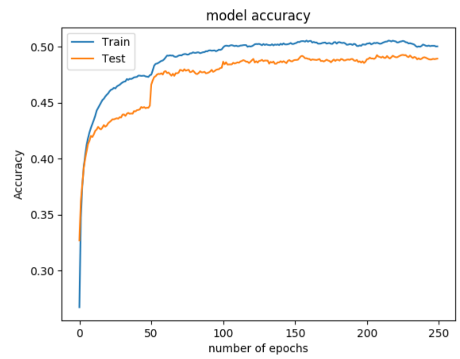

Figure 25: Training and Validation Accuracy for RMSprop

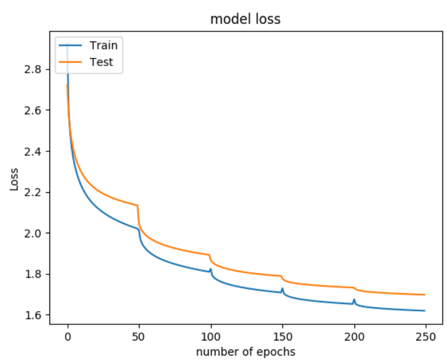

Figure 26: Training and Validation Loss for Adagrad

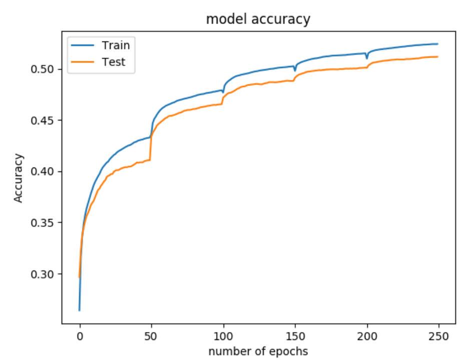

Figure 27: Training and Validation Accuracy for Adagrad

(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.

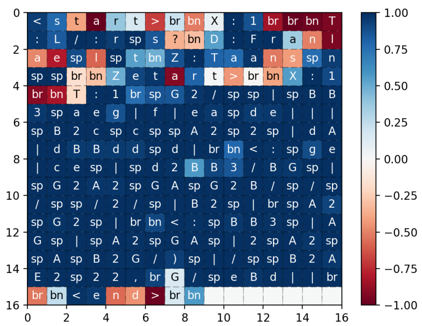

Figure 28: Activation For Neural 1

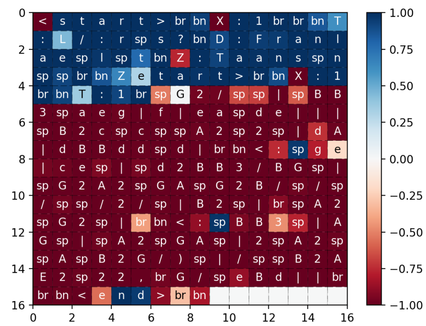

Figure 29: Activation For Neural 2

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 $T \to \infty$, 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 θ at time step t is

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

where η is the initial learning rate, $g_{t,i}$ is the gradient of objective function with respect to parameter θ, 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 ϵ 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 θ at time step t is

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

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

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

```python
from os.path import isfile, isdir
from os import makedirs
from keras.layers.recurrent import SimpleRNN
from keras.layers.core import Dense
from keras.models import Sequential
from keras.optimizers import RMSprop, Adagrad
from keras.callbacks import EarlyStopping, Callback
from utilities import loadTunes, partition, label2code, prepDataSeq, processSample, savePkl, l
import time
from tqdm import *
import numpy as np
import random
from keras.utils import np_utils
from keras import backend as K
LR = 0.01
QUESTION_ID = 'Q3'
N_HIDDEN_UNIT = 100
INCREMENT_TRAINING = True
BATCH_SIZE = 64 # How many samples will be processed simultaneously.
PRIME_LEN = 50 # Prime the music generation with PRIME_LEN characters.
MAX_EPOCH = 50 # How many epoch will be run.
EARLY_STOPPING = 6 # Stop training if validation set's loss stop decreasing.
DATA_STORAGE = '../data/'
# For results and models
MODEL_STORAGE = '../model/'
RESULT_STORAGE = '../result/'
MODEL_FILE = '%smyRNN_%s.h5'%(MODEL_STORAGE, QUESTION_ID)
TEMP_WEIGHT_FILE = '%stemp_test_weight_%s.h5'%(MODEL_STORAGE, QUESTION_ID)
WEIGHT_FILE = '%sweight_me%d_lr%g_%s.h5'%(RESULT_STORAGE, MAX_EPOCH, LR, QUESTION_ID)

RESULT_FILE = '%sresult_%g_%s.pkl'%(RESULT_STORAGE, LR, QUESTION_ID)
ACC_FILE = '%saccuracy_%g_%s.png'%(RESULT_STORAGE, LR, QUESTION_ID)
LOSS_FILE = '%sloss_%g_%s.png'%(RESULT_STORAGE, LR, QUESTION_ID)
# For generateMusic
TUNE_STORAGE = '%stunes/tune_%s/'%(RESULT_STORAGE, QUESTION_ID)

END_TOKEN ="<end>"
TUNE_MAX_LEN = 5000 # maximum length of the tune is around 4900, this ensures that we will sto

TEMPERATURE = 2
MAX_SEQ_LEN = 110

def getTemperature():
    return TEMPERATURE

def getResult():
    acc, loss, val_acc, val_loss = loadPkl(RESULT_FILE)
    return acc, loss, val_acc, val_loss
def temperature_activation(a):
    T = TEMPERATURE
    return K.softmax(a/T)

def getTestModel(output_dim = 94, lr=LR, n_hidden_units =N_HIDDEN_UNIT, modelfile = MODEL_FILE
    # as the first layer in a Sequential model
    model = Sequential()
    input_length = 1 # number of timesteps.
    input_dim = output_dim # number of features after one-hot encoding.
    model.add(SimpleRNN(n_hidden_units,
                        batch_input_shape = (BATCH_SIZE, input_length, input_dim),
```

```
                          return_sequences=False, # return last output in the output sequence fo
                          stateful=True, # last state for every sample at index i in a batch wi
                          unroll = True)) # network will be unrolled, speedup TF.
    model.add(Dense(output_dim, activation=temperature_activation))
    opt = RMSprop(lr=lr)
    model.compile(optimizer=opt,
                  loss='categorical_crossentropy',
                  metrics=['accuracy'])
    model.summary()
    model.save(modelfile)
    return model

def getModel(output_dim = 94, lr=LR, n_hidden_units =N_HIDDEN_UNIT, modelfile = MODEL_FILE):
    # as the first layer in a Sequential model
    model = Sequential()
    input_length = 1 # number of timesteps.
    input_dim = output_dim # number of features after one-hot encoding.
    model.add(SimpleRNN(n_hidden_units,
                          batch_input_shape = (BATCH_SIZE, input_length, input_dim),
                          return_sequences=False, # return last output in the output sequence fo
                          stateful=True, # last state for every sample at index i in a batch wi
                          unroll = True)) # network will be unrolled, speedup TF.
    model.add(Dense(output_dim, activation='softmax'))
    opt = Adagrad(lr=lr)
    model.compile(optimizer=opt,
                  loss='categorical_crossentropy',
                  metrics=['accuracy'])
    model.summary()
    model.save(modelfile)
    return model

def getTestModelWithWeights():
    # Get default model with the final weights loaded.

    if not isfile(WEIGHT_FILE):
        print "WARNING: Can not load %s, please run RNNTrain.py first."%(WEIGHT_FILE)
        return None
    else:
        model = getTestModel()
        model.load_weights(WEIGHT_FILE)
        return model

def trainModel(model,seqLen, Xtrain, ytrain, Xvalid, yvalid, nClasses, encodedTunes, le, incre
    print np.array(Xtrain).shape, np.array(ytrain).shape, np.array(Xvalid).shape, np.array(yva
    WEIGHT_FILE_STORE = '../result/weight_me%d_lr%g_seq%d_%s.h5'%(MAX_EPOCH, LR, seqLen, QUES
    print "Training with learning rate %g with sequence %d"%(LR, seqLen)
    mean_tr_accs =[]
    mean_tr_losses =[]
    mean_te_accs =[]
    mean_te_losses =[]
    previous_epoch = 0
    # Load weights if incrementW is true, and previous weights exist.
    if incrementalTraining and isfile(WEIGHT_FILE):
        model.load_weights(WEIGHT_FILE)
        print "Loaded weights from %s"%(WEIGHT_FILE)
        mean_tr_accs, mean_tr_losses, mean_te_accs, mean_te_losses = loadPkl(RESULT_FILE)
        print "Loaded history pkl from %s"%(RESULT_FILE)
        previous_epoch = len(mean_tr_accs)
    elif incrementalTraining and not isfile(WEIGHT_FILE):
        print "Did not find previous weights. Cannot load previous weights."
    elif isfile(WEIGHT_FILE):
        print "Over-writting previous weights %s"%(WEIGHT_FILE)
    else:
        print "Creating %s for the first time."%(WEIGHT_FILE)

```

```
    pre_test_loss = float("inf")
    incre = 0
    model_pre = model
    optimal_model = None
    for epoch in range(MAX_EPOCH):
        print "train part Epoch:%d/%d"%(previous_epoch+epoch, previous_epoch+MAX_EPOCH)
        tr_accs = []
        tr_losses = []
        n_batch = len(Xtrain)/BATCH_SIZE
        print '%d samples %d batches with batch size of %d'%(len(Xtrain),
                                                               n_batch,
                                                               BATCH_SIZE)
        for i in tqdm(range(n_batch)):
            x_seqs = np.array(Xtrain[i*BATCH_SIZE:(i+1)*BATCH_SIZE])
            y_seqs = np.array(ytrain[i*BATCH_SIZE:(i+1)*BATCH_SIZE])
            for j in range(seqLen):
                # conver every feature to (numsample, lenth, dim) format.
                # here x_hot is in (batch_size, 1, nClasses) format
                # y_hot is in (batch_size, nClasses) format
                x_hot, y_hot = processSample(x_seqs[:, [j]],
                                             y_seqs[:, [j]],
                                             nClasses)
                tr_loss, tr_acc = model.train_on_batch(x_hot, y_hot)
                tr_accs.append(tr_acc)
                tr_losses.append(tr_loss)
            model.reset_states()
        mean_tr_acc = np.mean(tr_accs)
        mean_tr_accs.append(mean_tr_acc)
        mean_tr_loss = np.mean(tr_losses)
        mean_tr_losses.append(mean_tr_loss)
        print ('accuracy trianing = {}'.format(mean_tr_acc))
        print ('loss training = {}'.format(mean_tr_loss))
        print ('___________________________________')

        te_accs = []
        te_losses = []
        n_batch = len(Xvalid)/BATCH_SIZE
        for i in tqdm(range(n_batch)):
            x_seqs = np.array(Xvalid[i*BATCH_SIZE:(i+1)*BATCH_SIZE])
            y_seqs = np.array(yvalid[i*BATCH_SIZE:(i+1)*BATCH_SIZE])
            for j in range(seqLen):
                x_hot, y_hot = processSample(x_seqs[:, [j]],
                                             y_seqs[:, [j]],
                                             nClasses)
                te_loss, te_acc = model.test_on_batch(x_hot,
                                                      y_hot)
                te_accs.append(te_acc)
                te_losses.append(te_loss)
            model.reset_states()

        mean_te_acc = np.mean(te_accs)
        mean_te_accs.append(mean_te_acc)
        mean_te_loss = np.mean(te_losses)
        mean_te_losses.append(mean_te_loss)
        print('accuracy testing =%g'%(mean_te_acc))
        print('loss testing =%g'%(mean_te_loss))
        print('___________________________________')

        # Early Stopping
        if pre_test_loss <= mean_te_loss:
            incre += 1
            if incre == 1:
                optimal_model = model_pre
        else:
            incre = 0
```

```
            model_pre = model
            optimal_model = None

        if incre >= EARLY_STOPPING:
            print "Early stpping at %d - %d steps"%(epoch, EARLY_STOPPING)
            break

        pre_test_loss = mean_te_loss

        # Check music generation every 20 epoch.
        if epoch%20 ==0:
            model.save_weights(TEMP_WEIGHT_FILE)
            te_model = getTestModel()
            te_model.load_weights(TEMP_WEIGHT_FILE)
            final_tune = generateMusic(te_model, encodedTunes, le)
            filename = '%s%d.txt'%(TUNE_STORAGE, epoch + previous_epoch)
            outputfile = open(filename, 'w')
            outputfile.write("%s" % ''.join(final_tune))

    # Save weights.
    if optimal_model != None:
        model = optimal_model
        data = mean_tr_accs[:-EARLY_STOPPING], mean_tr_losses[:-EARLY_STOPPING], mean_te_accs[
    else:
        data = mean_tr_accs, mean_tr_losses, mean_te_accs, mean_te_losses

    model.save_weights(WEIGHT_FILE)
    model.save_weights(WEIGHT_FILE_STORE)

    savePkl(data, RESULT_FILE)
    mean_tr_accs, mean_tr_losses, mean_te_accs, mean_te_losses = loadPkl(RESULT_FILE)
    savefig([mean_tr_accs, mean_te_accs], 'model accuracy',
            'number of epochs',
            'Accuracy',
            ['Train', 'Validation'],
            ACC_FILE)
    savefig([mean_tr_losses, mean_te_losses], 'model loss',
            'number of epochs',
            'Loss',
            ['Train', 'Validation'],
            LOSS_FILE)
    print "Done Saving"

def generateMusic(model, encodedTunes, le, maxTunes = 6, maxTuneLen = TUNE_MAX_LEN):
    nClasses =len(list(le.classes_))
    # prime the network with a sequence randomly selected from tunes.
    random.shuffle(encodedTunes)
    start_seqs = []
    chosen_tune = random.randint(0, len(encodedTunes)-1)
    # Make sure that the chosen tune is long enough to prime the song.
    while PRIME_LEN > len(encodedTunes[chosen_tune]):
        chosen_tune = random.randint(0, len(encodedTunes)-1)
    # Copy the same selected tune for Batch_Size
    for i in range(BATCH_SIZE):
        start_seqs.append(encodedTunes[chosen_tune][:PRIME_LEN])
    y_prime = le.inverse_transform(start_seqs[0])
    print "Prime the sequence with [0]x%d: \n--------\n%s\n--------"%(BATCH_SIZE, ''.join(y_prime)
    start_X = np.array(start_seqs)
    prime_pred = []
    for j in range(PRIME_LEN):
        # convert every feature to (numsample, lenth, dim) format.
        # here x_hot is in (batch_size, 1, nClasses) format
        # y_hot is in (batch_size, nClasses) format
        x_hot = np_utils.to_categorical(start_X[:, [j]], nClasses)
        x_hot = np.expand_dims(x_hot, axis=1)
```

```
        y_pred_prob = model.predict_on_batch(x_hot)
        y_pred_code = np.argmax(y_pred_prob[0])
        # take the prediction for the char j in first prime sequence.
        prime_pred.append(le.inverse_transform(y_pred_code))

    print "prime_pred info", len(prime_pred)
    # convert last prob prime output to label to display and check.
    print "Prime result:\n--------\n%s\n---------"%(''.join(prime_pred))

    lastToken = []
    # Take the last predicted character, continue feeding it to model as input.
    x_next_hot = prob2input(y_pred_prob)
    more_preds = [prime_pred[-1]]
    tunes_count = 0
    while((''.join(lastToken) != END_TOKEN or tunes_count < maxTunes) and len(more_preds) < ma
        y_pred_prob = model.predict_on_batch(x_next_hot)
        x_next_hot = prob2input(y_pred_prob)
        y_pred_code = np.argmax(y_pred_prob[0])
        pred_char = le.inverse_transform(y_pred_code)
        lastToken.append(pred_char)
        if len(lastToken) == 6:
            lastToken = lastToken[1:]
        more_preds.append(pred_char)
        if ''.join(lastToken) == END_TOKEN:
            tunes_count +=1
    result = [y_prime.tolist()[0]] # Add first character
    result.extend(prime_pred) # Add rest to result.
    result.extend(more_preds)
    return result

def runRNN(seqLen):
    # Generate result folders for result and models.
    if not isdir(MODEL_STORAGE):
        makedirs(MODEL_STORAGE)

    if not isdir(RESULT_STORAGE):
        makedirs(RESULT_STORAGE)

    # Generate folders for tunes storage.
    if not isdir(TUNE_STORAGE):
        makedirs(TUNE_STORAGE)

    PRE_PROCESS_STORE = '%spre_proecessed_seq%d_%s'%(DATA_STORAGE, MAX_SEQ_LEN, QUESTION_ID)
    preprocesspkl = '%s.pkl'%(PRE_PROCESS_STORE)
    preprocesstxt = '%s.txt'%(PRE_PROCESS_STORE)

    if not isfile(preprocesspkl):
        print "Read data from file %s."%(preprocesspkl)
        tunes = loadTunes()
        print "Encode chars to ints."
        encoded_tunes, label_encoder = label2code(tunes)
        print "Chop txt to sequences of %d length" %(MAX_SEQ_LEN)
        X, y = prepDataSeq(encoded_tunes, MAX_SEQ_LEN)
        print "Found total %d sequences"%(len(X))
        x_train, y_train, x_valid, y_valid = partition(X, y)
        print "Done partition."
        data = x_train, y_train, x_valid, y_valid, encoded_tunes, label_encoder
        savePkl(data, preprocesspkl)
    else:
        x_train, y_train, x_valid, y_valid, encoded_tunes, label_encoder = loadPkl(preprocessp

    f = open(preprocesstxt, 'w')
    for seq in x_train:
        f.write(''.join(label_encoder.inverse_transform(seq)))
        f.write('\n--------\n')
```

```
    print "Done saving."

    nClasses =len(list(label_encoder.classes_))
    print "There are %d tunes. %d classes."%(len(encoded_tunes), nClasses)
    print "%d sequences for training, %d sequences for testing"%(len(x_train), len(x_valid))
    #print label_encoder.classes_

    model = getModel(nClasses)
    trainModel(model, seqLen, x_train, y_train, x_valid, y_valid, nClasses, encoded_tunes, lab

if __name__ == "__main__":
    for seq_len in [20, 50, 70, 90, 110]:
        runRNN(seq_len)
```

Listing 2: utilities.py for data extraction and generation

```

from os.path import isfile
import numpy as np
import random
from sklearn import preprocessing
from keras.utils import np_utils
import cPickle as pickle
import matplotlib.pyplot as plt

def savePkl(dataset, pklfile):
    # Save small pkl files.
    f = file(pklfile, 'wb')
    pickle.dump(dataset, f, protocol=pickle.HIGHEST_PROTOCOL)
    f.close

def loadPkl(pklfile):
    # Load small pkl files.
    f = open(pklfile, 'rb')
    dataset = pickle.load(f)
    f.close
    return dataset
# This file provide tools that can be used to read the data.
def loadTunes(filepath='../data/input.txt'):
    f = open(filepath, 'r')
    tunes = []
    starting_seqs = []
    tune = []
    for line in f:
        tune.extend(list(line))
        if line == '<end>\r\n':
            tunes.append(tune)
            tune = []
    print "Found %d tunes"%(len(tunes))
    statspath = '../data/stats.txt' # stores the length for every tune.
    statsfile = open(statspath, 'w')
    for i in range(len(tunes)):
        statsfile.write("%d:\t%d\n"%(i, len(tunes[i])))
    print "tunes ranges from %d to %d characters"%(min(map(len, tunes)), max(map(len, tunes)))
    return tunes

def label2code(tunes):
    le = preprocessing.LabelEncoder()
    all_chars = []
    map(all_chars.extend, tunes)
    le.fit(all_chars)
    print 'found %d classes.'%(len(list(le.classes_)))
    new_tunes = map(le.transform, tunes)
    return new_tunes, le

```

```
def prepDataSeq(data, sequenceLen, sequential = False, noTuneSeparation= True, overlapping = T
    # slice random sequences from all tunes. Not aligned with the beginning of the file.
    X = []
    y = []
    if sequential:
        print "Sequentially connected sequences will be generated."
    else:
        print "Randomly selected sequences will be generated."
    if noTuneSeparation:
        print "The file will be chopped as a whole sequence."
    else:
        print "Each tune will be chopped as an independent sequence."
    if overlapping:
        print "No overlapping sequences will be generated."
    else:
        print "Overlapping sequences will be generated."
    if noTuneSeparation:
        # Sequences are randomly drawn from the entire input.txt file.
        all_chars = []
        map(all_chars.extend, data)
        N = len(all_chars)
        valid_start_max = N - 1 - 2*sequenceLen
        start  = 0
        while start < valid_start_max:
            if not sequential:
                start = random.randint(start, start+sequenceLen)
            end = start + sequenceLen
            x_seq = all_chars[start:end]
            X.append(x_seq)
            y_seq = all_chars[start+1:end+1]
            y.append(y_seq)
            if overlapping and not sequential:
                start = end - sequenceLen/2
            else:
                start = end
    else:
        # Sequences are only drawn from within every tune sequence.
        for tune in data:
            N = len(tune)
            valid_start_max = N - 1 - 2*sequenceLen
            # randomly select one sequence start per sequenceLen sequentailly from X.
            start = 0
            while start <= valid_start_max:
                if not sequential:
                    start = random.randint(start, start+sequenceLen)
                end = start + sequenceLen
                x_seq = tune[start:end]
                X.append(x_seq)
                y_seq = tune[start+1:end+1]
                y.append(y_seq)
                if overlapping and not sequential:
                    start = end - sequenceLen/2
                else:
                    start = end
            # both X and y are shape of (sequences x sequenceLen)

            # Add last sequence if one does not exist, add it for half of the time.
            if start != N-1 and random.random()>0.5 and N-1 >= sequenceLen:
                X.append(tune[-sequenceLen -1:-1])
                y.append(tune[-sequenceLen:])
                if len(X[-1]) != sequenceLen or  len(y[-1]) != sequenceLen:
                    print len(X[-1]), len(y[-1])
    return X, y

def partition(X, y):
```

```
    data = zip(X, y)
    random.shuffle(data)
    # 80% for training, 20% for validation
    n_sequences = len(data)
    n_train = int(n_sequences*0.8)
    n_valid = n_sequences - n_train
    data_train = zip(*data[:n_train])
    data_test = zip(*data[n_train:])
    return list(data_train[0]), list(data_train[1]), list(data_test[0]), list(data_test[1])

def processSample(x, y, nClasses):
    xhot = np_utils.to_categorical(x, nClasses)
    xhot = np.expand_dims(xhot, axis=1)
    yhot = np_utils.to_categorical(y, nClasses)
    return xhot, yhot

def prob2input(y_pred_prob):
    nSamples = y_pred_prob.shape[0]
    nClasses = y_pred_prob.shape[1]
    y_pred_ints = []
    for i in range(nSamples):
        y_pred_int = np.random.choice(range(nClasses), p=y_pred_prob[i, :])
        y_pred_ints.append(y_pred_int)
    y_pred_hot = np_utils.to_categorical(y_pred_ints, nClasses)
    y_pred_hot = np.expand_dims(y_pred_hot, axis=1)
    return y_pred_hot

def savefig(results, title='', xlabel='', ylabel='', legends = [], savepath = '',
Xs = [], display = False, overwrite = True):
    if not isfile(savepath) or overwrite:
        print "Save %s..."%(savepath)

        if Xs == []:
            Xs = range(len(results[0]))
        print "# iterations:", len(Xs)
        for Ys in results:
            plt.plot(Xs, Ys)

            plt.title(title)
            plt.ylabel(ylabel)
            plt.xlabel(xlabel)

        if legends !=[]:
            plt.legend(legends, loc='upper left')
        plt.savefig(savepath)
        print "Done saving acc figure."
        if display:
            plt.show()
        plt.clf()

if __name__ == "__main__":
    tunes = readTxt()
    samples = np.random.randint(len(tunes), size=3)
    sample_len = 50
    print "example start sequences of length %d:\n%s\n%s\n%s"%(sample_len,
                                                                      ''.join(tunes[samples[0]][:sam
                                                                      ''.join(tunes[samples[1]][:sam
                                                                      ''.join(tunes[samples[2]][:sam
    new_tunes, label_encoder = label2code(tunes)
    sequence_len = 30
    X, y = prepDataSeq(new_tunes, sequence_len)
    print "Found total %d sequences"%(len(X))
    x_train, y_train, x_test, y_test = partition(X, y)
    print "%d train sequences, %d test sequences"%(len(x_train), len(x_test))
    print "train data dimension", np.array(x_train).shape
```

```
    print "label data dimension", np.array(x_test).shape
```

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

```
from os.path import isfile, isdir
from os import makedirs

from RNNTrain import getTestModelWithWeights, generateMusic, getTemperature, getResult
from utilities import loadTunes, label2code, savefig

QUESTION_ID = 'Q4a'
MODEL_STORAGE = '../model/'
RESULT_STORAGE = '../result/'
TUNE_STORAGE = '%stunes/tune_%s/'%(RESULT_STORAGE, QUESTION_ID)
MAX_TUNE = 30
TUNE_MAX_LEN = 5000

def reportQ4a():
    if not isdir(MODEL_STORAGE):
        makedirs(MODEL_STORAGE)

    if not isdir(RESULT_STORAGE):
        makedirs(RESULT_STORAGE)

    # Generate folders for tunes storage.
    if not isdir(TUNE_STORAGE):
        makedirs(TUNE_STORAGE)
    model = getTestModelWithWeights()
    if model == None:
        return
    print "Read tunes from input.txt file."
    tunes = loadTunes()
    print "Encode chars to ints."
    encoded_tunes, label_encoder = label2code(tunes)
    # for T in [1, 2, 0.5]: we need to change the TEMPERATURE global variable to theses values
    for num in range(MAX_TUNE):
        final_tune = generateMusic(model, encoded_tunes, label_encoder, 1, TUNE_MAX_LEN)
        filename = '%s%g_%d.txt'%(TUNE_STORAGE, getTemperature(), num)
        outputfile = open(filename, 'w')
        # print type(final_tune), len(final_tune), final_tune[:50]
        outputfile.write("%s" % ''.join(final_tune))
        model.reset_states()

def reportQ4b():
    acc, loss, val_acc, val_loss = getResult()
    savefig([acc, val_acc], 'model accuracy',
            'number of epochs',
            'Accuracy',
            ['Train', 'Validation'],
            '%sq4_b/q4_b_acc.png'%RESULT_STORAGE)
    savefig([loss, val_loss], 'model loss',
            'number of epochs',
            'Loss',
            ['Train', 'Validation'],
            '%sq4_b/q4_b_loss.png'%RESULT_STORAGE)

if __name__ == '__main__':
    reportQ4a()
    reportQ4b()
```

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

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

```
from keras.layers.core import Dense
from keras.models import Sequential
from keras.optimizers import RMSprop, Adagrad
from keras.callbacks import EarlyStopping, Callback
from utilities import loadTunes, partition, label2code, prepDataSeq, processSample, savePkl, l
import time
from tqdm import *
import numpy as np
import random
from keras.utils import np_utils
from keras import backend as K
LR = 0.001
QUESTION_ID = 'Q4e-RMSprop'
N_HIDDEN_UNIT = 100
INCREMENT_TRAINING = True
BATCH_SIZE = 64 # How many samples will be processed simultaneously.
PRIME_LEN = 30 # Prime the music generation with PRIME_LEN characters.
MAX_EPOCH = 50 # How many epoch will be run.
EARLY_STOPPING = 6 # Stop training if validation set's loss stop decreasing.
DATA_STORAGE = '../data/'
# For results and models
MODEL_STORAGE = '../model/'
RESULT_STORAGE = '../result/'
MODEL_FILE = '%smyRNN_%s.h5'%(MODEL_STORAGE, QUESTION_ID)
TEMP_MODEL_FILE = '%stemp_test_model_%s.h5'%(MODEL_STORAGE, QUESTION_ID)
WEIGHT_FILE = '%sweight_me%d_lr%g_%s.h5'%(RESULT_STORAGE, MAX_EPOCH, LR, QUESTION_ID)

RESULT_FILE = '%sresult_%g_%s.pkl'%(RESULT_STORAGE, LR, QUESTION_ID)
ACC_FILE = '%saccuracy_%g_%s.png'%(RESULT_STORAGE, LR, QUESTION_ID)
LOSS_FILE = '%sloss_%g_%s.png'%(RESULT_STORAGE, LR, QUESTION_ID)
# For generateMusic
TUNE_STORAGE = '%stunes/tune_%s/'%(RESULT_STORAGE, QUESTION_ID)

END_TOKEN ="<end>"
TUNE_MAX_LEN = 5000 # maximum length of the tune is around 4900, this ensures that we will sto

TEMPERATURE = 1
MAX_SEQ_LEN = 110
def getTemperature():
    return TEMPERATURE

def temperature_activation(a):
    T = TEMPERATURE
    return K.softmax(a/T)

def getTestModel(output_dim = 94, lr=LR, n_hidden_units =N_HIDDEN_UNIT, modelfile = MODEL_FILE
    # as the first layer in a Sequential model
    model = Sequential()
    input_length = 1 # number of timesteps.
    input_dim = output_dim # number of features after one-hot encoding.
    model.add(SimpleRNN(n_hidden_units,
                        batch_input_shape = (BATCH_SIZE, input_length, input_dim),
                        return_sequences=False, # return last output in the output sequence fo
                        stateful=True, # last state for every sample at index i in a batch wi
                        unroll = True)) # network will be unrolled, speedup TF.
    model.add(Dense(output_dim, activation=temperature_activation))
    opt = RMSprop(lr=lr)
    #opt = Adagrad(lr=lr)
    model.compile(optimizer=opt,
                  loss='categorical_crossentropy',
                  metrics=['accuracy'])
    model.summary()
    model.save(modelfile)
    return model

```

```
def getModel(output_dim = 94, lr=LR, n_hidden_units =N_HIDDEN_UNIT, modelfile = MODEL_FILE):
    # as the first layer in a Sequential model
    model = Sequential()
    input_length = 1 # number of timesteps.
    input_dim = output_dim # number of features after one-hot encoding.
    model.add(SimpleRNN(n_hidden_units,
                        batch_input_shape = (BATCH_SIZE, input_length, input_dim),
                        return_sequences=False, # return last output in the output sequence fo
                        stateful=True, # last state for every sample at index i in a batch wi
                        unroll = True)) # network will be unrolled, speedup TF.
    model.add(Dense(output_dim, activation='softmax'))
    opt = RMSprop(lr=lr)
    #opt = Adagrad(lr=lr)
    model.compile(optimizer=opt,
                  loss='categorical_crossentropy',
                  metrics=['accuracy'])
    model.summary()
    model.save(modelfile)
    return model

def getTestModelWithWeights():
    # Get default model with the final weights loaded.

    if not isfile(WEIGHT_FILE):
        print "WARNING: Can not load %s, please run RNNTrain.py first."%(WEIGHT_FILE)
        return None
    else:
        model = getTestModel()
        model.load_weights(WEIGHT_FILE)
        return model

def trainModel(model,seqLen, Xtrain, ytrain, Xvalid, yvalid, nClasses, encodedTunes, le, incre
    print np.array(Xtrain).shape, np.array(ytrain).shape, np.array(Xvalid).shape, np.array(yva
    WEIGHT_FILE_STORE = '../result/weight_me%d_lr%g_seq%d_%s.h5'%(MAX_EPOCH, LR, seqLen, QUES
    print "Training with learning rate %g with sequence %d"%(LR, seqLen)
    mean_tr_accs =[]
    mean_tr_losses =[]
    mean_te_accs =[]
    mean_te_losses =[]
    previous_epoch = 0
    # Load weights if incrementW is true, and previous weights exist.
    if incrementalTraining and isfile(WEIGHT_FILE):
        model.load_weights(WEIGHT_FILE)
        print "Loaded weights from %s"%(WEIGHT_FILE)
        mean_tr_accs, mean_tr_losses, mean_te_accs, mean_te_losses = loadPkl(RESULT_FILE)
        print "Loaded history pkl from %s"%(RESULT_FILE)
        previous_epoch = len(mean_tr_accs)
    elif incrementalTraining and not isfile(WEIGHT_FILE):
        print "Did not find previous weights. Cannot load previous weights."
    elif isfile(WEIGHT_FILE):
        print "Over-writting previous weights %s"%(WEIGHT_FILE)
    else:
        print "Creating %s for the first time."%(WEIGHT_FILE)

    pre_test_loss = float("inf")
    incre = 0
    model_pre = model
    optimal_model = None
    for epoch in range(MAX_EPOCH):
        print "train part Epoch:%d/%d"%(previous_epoch+epoch, previous_epoch+MAX_EPOCH)
        tr_accs = []
        tr_losses = []
        n_batch = len(Xtrain)/BATCH_SIZE
        print '%d samples %d batches with batch size of %d'%(len(Xtrain),
                                                              n_batch,
```

```
                                                                   BATCH_SIZE)
        for i in tqdm(range(n_batch)):
            x_seqs = np.array(Xtrain[i*BATCH_SIZE:(i+1)*BATCH_SIZE])
            y_seqs = np.array(ytrain[i*BATCH_SIZE:(i+1)*BATCH_SIZE])
            for j in range(seqLen):
                # conver every feature to (numsample, lenth, dim) format.
                # here x_hot is in (batch_size, 1, nClasses) format
                # y_hot is in (batch_size, nClasses) format
                x_hot, y_hot = processSample(x_seqs[:, [j]],
                                             y_seqs[:, [j]],
                                             nClasses)
                tr_loss, tr_acc = model.train_on_batch(x_hot, y_hot)
                tr_accs.append(tr_acc)
                tr_losses.append(tr_loss)
            model.reset_states()
        mean_tr_acc = np.mean(tr_accs)
        mean_tr_accs.append(mean_tr_acc)
        mean_tr_loss = np.mean(tr_losses)
        mean_tr_losses.append(mean_tr_loss)
        print ('accuracy trianing = {}'.format(mean_tr_acc))
        print ('loss training = {}'.format(mean_tr_loss))
        print ('___________________________________')

        te_accs = []
        te_losses = []
        n_batch = len(Xvalid)/BATCH_SIZE
        for i in tqdm(range(n_batch)):
            x_seqs = np.array(Xvalid[i*BATCH_SIZE:(i+1)*BATCH_SIZE])
            y_seqs = np.array(yvalid[i*BATCH_SIZE:(i+1)*BATCH_SIZE])
            for j in range(seqLen):
                x_hot, y_hot = processSample(x_seqs[:, [j]],
                                             y_seqs[:, [j]],
                                             nClasses)
                te_loss, te_acc = model.test_on_batch(x_hot,
                                                      y_hot)
                te_accs.append(te_acc)
                te_losses.append(te_loss)
            model.reset_states()

        mean_te_acc = np.mean(te_accs)
        mean_te_accs.append(mean_te_acc)
        mean_te_loss = np.mean(te_losses)
        mean_te_losses.append(mean_te_loss)
        print('accuracy testing = %g'%(mean_te_acc))
        print('loss testing = %g'%(mean_te_loss))
        print('___________________________________')

        # Early Stopping
        if pre_test_loss <= mean_te_loss:
            incre += 1
            if incre == 1:
                optimal_model = model_pre
        else:
            incre = 0
            model_pre = model
            optimal_model = None

        if incre >= EARLY_STOPPING:
            print "Early stpping at %d - %d steps"%(epoch, EARLY_STOPPING)
            break

        pre_test_loss = mean_te_loss

        # Check music generation every 20 epoch.
        if epoch%20 ==0:
```

```
                model.save_weights(TEMP_MODEL_FILE)
                te_model = getTestModel()
                te_model.load_weights(TEMP_MODEL_FILE)
                final_tune = generateMusic(te_model, encodedTunes, le)
                filename = '%s%d.txt'%(TUNE_STORAGE, epoch + previous_epoch)
                outputfile = open(filename, 'w')
                outputfile.write("%s" % ''.join(final_tune))

    # Save weights.
    if optimal_model != None:
        model = optimal_model
        data = mean_tr_accs[:-EARLY_STOPPING], mean_tr_losses[:-EARLY_STOPPING], mean_te_accs[
    else:
        data = mean_tr_accs, mean_tr_losses, mean_te_accs, mean_te_losses

    model.save_weights(WEIGHT_FILE)
    model.save_weights(WEIGHT_FILE_STORE)

    savePkl(data, RESULT_FILE)
    mean_tr_accs, mean_tr_losses, mean_te_accs, mean_te_losses = loadPkl(RESULT_FILE)
    savefig([mean_tr_accs, mean_te_accs], 'model accuracy',
            'number of epochs',
            'Accuracy',
            ['Train', 'Test'],
            ACC_FILE)
    savefig([mean_tr_losses, mean_te_losses], 'model loss',
            'number of epochs',
            'Loss',
            ['Train', 'Test'],
            LOSS_FILE)
    print "Done Saving"

def generateMusic(model, encodedTunes, le, maxTunes = 6, maxTuneLen = TUNE_MAX_LEN):
    nClasses =len(list(le.classes_))
    # prime the network with a sequence randomly selected from tunes.
    random.shuffle(encodedTunes)
    start_seqs = []
    chosen_tune = random.randint(0, len(encodedTunes)-1)
    # Copy the same selected tune for Batch_Size
    for i in range(BATCH_SIZE):
        start_seqs.append(encodedTunes[chosen_tune][:PRIME_LEN])
    y_prime = le.inverse_transform(start_seqs[0])
    print "Prime the sequence with [0]x%d: \n--------\n%s\n--------"%(BATCH_SIZE, ''.join(y_prime)
    start_X = np.array(start_seqs)
    prime_pred = []
    for j in range(PRIME_LEN):
        # convert every feature to (numsample, lenth, dim) format.
        # here x_hot is in (batch_size, 1, nClasses) format
        # y_hot is in (batch_size, nClasses) format
        x_hot = np_utils.to_categorical(start_X[:, [j]], nClasses)
        x_hot = np.expand_dims(x_hot, axis=1)
        y_pred_prob = model.predict_on_batch(x_hot)
        y_pred_code = np.argmax(y_pred_prob[0])
        # take the prediction for the char j in first prime sequence.
        prime_pred.append(le.inverse_transform(y_pred_code))

    print "prime_pred info", len(prime_pred)
    # convert last prob prime output to label to display and check.
    print "Prime result:\n--------\n%s\n--------"%(''.join(prime_pred))

    lastToken = []
    # Take the last predicted character, continue feeding it to model as input.
    x_next_hot = prob2input(y_pred_prob)
    more_preds = [prime_pred[-1]]
    tunes_count = 0
```

```
    while((''.join(lastToken) != END_TOKEN or tunes_count < maxTunes) and len(more_preds) < ma
        y_pred_prob = model.predict_on_batch(x_next_hot)
        x_next_hot = prob2input(y_pred_prob)
        y_pred_code = np.argmax(y_pred_prob[0])
        pred_char = le.inverse_transform(y_pred_code)
        lastToken.append(pred_char)
        if len(lastToken) ==  6:
            lastToken = lastToken[1:]
        more_preds.append(pred_char)
        if ''.join(lastToken) == END_TOKEN:
            tunes_count +=1
    result = [y_prime.tolist()[0]] # Add first character
    result.extend(prime_pred) # Add rest to result.
    result.extend(more_preds)
    return result

def runRNN(seqLen):
    # Generate result folders for result and models.
    if not isdir(MODEL_STORAGE):
        makedirs(MODEL_STORAGE)

    if not isdir(RESULT_STORAGE):
        makedirs(RESULT_STORAGE)

    # Generate folders for tunes storage.
    if not isdir(TUNE_STORAGE):
        makedirs(TUNE_STORAGE)

    PRE_PROCESS_STORE = '%spre_proecessed_seq%d_%s'%(DATA_STORAGE, MAX_SEQ_LEN, QUESTION_ID)
    preprocesspkl = '%s.pkl'%(PRE_PROCESS_STORE)
    preprocesstxt = '%s.txt'%(PRE_PROCESS_STORE)

    if not isfile(preprocesspkl):
        print "Read data from file %s."%(preprocesspkl)
        tunes = loadTunes()
        print "Encode chars to ints."
        encoded_tunes, label_encoder = label2code(tunes)
        print "Chop txt to sequences of %d length" %(MAX_SEQ_LEN)
        X, y = prepDataSeq(encoded_tunes, MAX_SEQ_LEN)
        print "Found total %d sequences"%(len(X))
        x_train, y_train, x_valid, y_valid = partition(X, y)
        print "Done partition."
        data = x_train, y_train, x_valid, y_valid, encoded_tunes, label_encoder
        savePkl(data, preprocesspkl)
    else:
        x_train, y_train, x_valid, y_valid, encoded_tunes, label_encoder = loadPkl(preprocessp

    f = open(preprocesstxt, 'w')
    for seq in x_train:
        f.write(''.join(label_encoder.inverse_transform(seq)))
        f.write('\n---------\n')
    print "Done saving."

    nClasses =len(list(label_encoder.classes_))
    print "There are %d tunes. %d classes."%(len(encoded_tunes), nClasses)
    print "%d sequences for training, %d sequences for testing"%(len(x_train), len(x_valid))
    #print label_encoder.classes_

    model = getModel(nClasses)
    trainModel(model, seqLen, x_train, y_train, x_valid, y_valid, nClasses, encoded_tunes, lab

if __name__ == "__main__":
    for seq_len in [20, 50, 70, 90, 110]:
        runRNN(seq_len)
```

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

```python
from os.path import isfile, isdir
from os import makedirs
from keras.layers.recurrent import SimpleRNN
from keras.layers.core import Dense
from keras.models import Sequential
from keras.optimizers import RMSprop, Adagrad
from keras.callbacks import EarlyStopping, Callback
from utilities import loadTunes, partition, label2code, prepDataSeq, processSample, savePkl, l
import time
from tqdm import *
import numpy as np
import matplotlib.pyplot as plt
from matplotlib import cm
import random
from keras.utils import np_utils
from keras import backend as K
from keras.models import Model
from itertools import izip

LR = 0.01
QUESTION_ID = 'Q4e'
N_HIDDEN_UNIT = 100
INCREMENT_TRAINING = True
BATCH_SIZE = 64 # How many samples will be processed simultaneously.
PRIME_LEN = 30 # Prime the music generation with PRIME_LEN characters.
MAX_EPOCH = 50 # How many epoch will be run.
EARLY_STOPPING = 6 # Stop training if validation set's loss stop decreasing.
DATA_STORAGE = '../data/'
# For results and models
MODEL_STORAGE = '../model/'
RESULT_STORAGE = '../result/'
MODEL_FILE = '%smyRNN_%s.h5'%(MODEL_STORAGE, QUESTION_ID)
TEMP_MODEL_FILE = '%stemp_test_model_%s.h5'%(MODEL_STORAGE, QUESTION_ID)
WEIGHT_FILE = '%sweight_me%d_lr%g_%s.h5'%(RESULT_STORAGE, MAX_EPOCH, LR, QUESTION_ID)

RESULT_FILE = '%sresult_%g_%s.pkl'%(RESULT_STORAGE, LR, QUESTION_ID)
ACC_FILE = '%saccuracy_%g_%s.png'%(RESULT_STORAGE, LR, QUESTION_ID)
LOSS_FILE = '%sloss_%g_%s.png'%(RESULT_STORAGE, LR, QUESTION_ID)
# For generateMusic
TUNE_STORAGE = '%stunes/tune_%s/'%(RESULT_STORAGE, QUESTION_ID)

END_TOKEN ="<end>"
TUNE_MAX_LEN = 5000 # maximum length of the tune is around 4900, this ensures that we will sto

TEMPERATURE = 1
MAX_SEQ_LEN = 110
def getTemperature():
    return TEMPERATURE

def temperature_activation(a):
    T = TEMPERATURE
    return K.softmax(a/T)

def getTestModel(output_dim = 94, lr=LR, n_hidden_units =N_HIDDEN_UNIT, modelfile = MODEL_FILE
    # as the first layer in a Sequential model
    model = Sequential()
    input_length = 1 # number of timesteps.
    input_dim = output_dim # number of features after one-hot encoding.
    model.add(SimpleRNN(n_hidden_units,
                        batch_input_shape = (BATCH_SIZE, input_length, input_dim),
                        return_sequences=False, # return last output in the output sequence fo
                        stateful=True, # last state for every sample at index i in a batch wi
                        unroll = True)) # network will be unrolled, speedup TF.
```

```
    model.add(Dense(output_dim, activation=temperature_activation))
    #opt = RMSprop(lr=lr)
    opt = Adagrad(lr=lr)
    model.compile(optimizer=opt,
                  loss='categorical_crossentropy',
                  metrics=['accuracy'])
    model.summary()
    model.save(modelfile)
    return model

def getModel(output_dim = 94, lr=LR, n_hidden_units =N_HIDDEN_UNIT, modelfile = MODEL_FILE):
    # as the first layer in a Sequential model
    model = Sequential()
    input_length = 1 # number of timesteps.
    input_dim = output_dim # number of features after one-hot encoding.
    model.add(SimpleRNN(n_hidden_units,
                        batch_input_shape = (BATCH_SIZE, input_length, input_dim),
                        return_sequences=False, # return last output in the output sequence fo
                        stateful=True, # last state for every sample at index i in a batch wi
                        unroll = True)) # network will be unrolled, speedup TF.
    model.add(Dense(output_dim, activation='softmax'))
    #opt = RMSprop(lr=lr)
    opt = Adagrad(lr=lr)
    model.compile(optimizer=opt,
                  loss='categorical_crossentropy',
                  metrics=['accuracy'])
    model.summary()
    model.save(modelfile)
    return model

def getTestModelWithWeights():
    # Get default model with the final weights loaded.

    if not isfile(WEIGHT_FILE):
        print "WARNING: Can not load %s, please run RNNTrain.py first."%(WEIGHT_FILE)
        return None
    else:
        model = getTestModel()
        model.load_weights(WEIGHT_FILE)
        return model

def trainModel(model,seqLen, Xtrain, ytrain, Xvalid, yvalid, nClasses, encodedTunes, le, incre
    print np.array(Xtrain).shape, np.array(ytrain).shape, np.array(Xvalid).shape, np.array(yva
    WEIGHT_FILE_STORE = '../result/weight_me%d_lr%g_seq%d_%s.h5'%(MAX_EPOCH, LR, seqLen, QUES
    print "Training with learning rate %g with sequence %d"%(LR, seqLen)
    mean_tr_accs =[]
    mean_tr_losses =[]
    mean_te_accs =[]
    mean_te_losses =[]
    previous_epoch = 0
    # Load weights if incrementW is true, and previous weights exist.
    if incrementalTraining and isfile(WEIGHT_FILE):
        model.load_weights(WEIGHT_FILE)
        print "Loaded weights from %s"%(WEIGHT_FILE)
        mean_tr_accs, mean_tr_losses, mean_te_accs, mean_te_losses = loadPkl(RESULT_FILE)
        print "Loaded history pkl from %s"%(RESULT_FILE)
        previous_epoch = len(mean_tr_accs)
    elif incrementalTraining and not isfile(WEIGHT_FILE):
        print "Did not find previous weights. Cannot load previous weights."
    elif isfile(WEIGHT_FILE):
        print "Over-writting previous weights %s"%(WEIGHT_FILE)
    else:
        print "Creating %s for the first time."%(WEIGHT_FILE)

    pre_test_loss = float("inf")
```

```
    incre = 0
    model_pre = model
    optimal_model = None
    for epoch in range(MAX_EPOCH):
        print "train part Epoch:%d/%d"%(previous_epoch+epoch, previous_epoch+MAX_EPOCH)
        tr_accs = []
        tr_losses = []
        n_batch = len(Xtrain)/BATCH_SIZE
        print '%d samples %d batches with batch size of %d'%(len(Xtrain),
                                                              n_batch,
                                                              BATCH_SIZE)
        for i in tqdm(range(n_batch)):
            x_seqs = np.array(Xtrain[i*BATCH_SIZE:(i+1)*BATCH_SIZE])
            y_seqs = np.array(ytrain[i*BATCH_SIZE:(i+1)*BATCH_SIZE])
            for j in range(seqLen):
                # conver every feature to (numsample, lenth, dim) format.
                # here x_hot is in (batch_size, 1, nClasses) format
                # y_hot is in (batch_size, nClasses) format
                x_hot, y_hot = processSample(x_seqs[:, [j]],
                                             y_seqs[:, [j]],
                                             nClasses)
                tr_loss, tr_acc = model.train_on_batch(x_hot, y_hot)
                tr_accs.append(tr_acc)
                tr_losses.append(tr_loss)
            model.reset_states()
        mean_tr_acc = np.mean(tr_accs)
        mean_tr_accs.append(mean_tr_acc)
        mean_tr_loss = np.mean(tr_losses)
        mean_tr_losses.append(mean_tr_loss)
        print ('accuracy trianing = {}'.format(mean_tr_acc))
        print ('loss training = {}'.format(mean_tr_loss))
        print ('__________________________________')

        te_accs = []
        te_losses = []
        n_batch = len(Xvalid)/BATCH_SIZE
        for i in tqdm(range(n_batch)):
            x_seqs = np.array(Xvalid[i*BATCH_SIZE:(i+1)*BATCH_SIZE])
            y_seqs = np.array(yvalid[i*BATCH_SIZE:(i+1)*BATCH_SIZE])
            for j in range(seqLen):
                x_hot, y_hot = processSample(x_seqs[:, [j]],
                                             y_seqs[:, [j]],
                                             nClasses)
                te_loss, te_acc = model.test_on_batch(x_hot,
                                                      y_hot)
                te_accs.append(te_acc)
                te_losses.append(te_loss)
            model.reset_states()

        mean_te_acc = np.mean(te_accs)
        mean_te_accs.append(mean_te_acc)
        mean_te_loss = np.mean(te_losses)
        mean_te_losses.append(mean_te_loss)
        print('accuracy testing = %g'%(mean_te_acc))
        print('loss testing = %g'%(mean_te_loss))
        print('__________________________________')

        # Early Stopping
        if pre_test_loss <= mean_te_loss:
            incre += 1
            if incre == 1:
                optimal_model = model_pre
        else:
            incre = 0
            model_pre = model
```

```
            optimal_model = None

        if incre >= EARLY_STOPPING:
            print "Early stpping at %d - %d steps"%(epoch, EARLY_STOPPING)
            break

        pre_test_loss = mean_te_loss

        # Check music generation every 20 epoch.
        if epoch%20 ==0:
            model.save_weights(TEMP_MODEL_FILE)
            te_model = getTestModel()
            te_model.load_weights(TEMP_MODEL_FILE)
            final_tune = generateMusic(te_model, encodedTunes, le)
            filename = '%s%d.txt'%(TUNE_STORAGE, epoch + previous_epoch)
            outputfile = open(filename, 'w')
            outputfile.write("%s" % ''.join(final_tune))

    # Save weights.
    if optimal_model != None:
        model = optimal_model
        data = mean_tr_accs[:-EARLY_STOPPING], mean_tr_losses[:-EARLY_STOPPING], mean_te_accs[
    else:
        data = mean_tr_accs, mean_tr_losses, mean_te_accs, mean_te_losses

    model.save_weights(WEIGHT_FILE)
    model.save_weights(WEIGHT_FILE_STORE)

    savePkl(data, RESULT_FILE)
    mean_tr_accs, mean_tr_losses, mean_te_accs, mean_te_losses = loadPkl(RESULT_FILE)
    savefig([mean_tr_accs, mean_te_accs], 'model accuracy',
            'number of epochs',
            'Accuracy',
            ['Train', 'Test'],
            ACC_FILE)
    savefig([mean_tr_losses, mean_te_losses], 'model loss',
            'number of epochs',
            'Loss',
            ['Train', 'Test'],
            LOSS_FILE)
    print "Done Saving"

def generateMusic(model, encodedTunes, le, maxTunes = 6, maxTuneLen = TUNE_MAX_LEN):
    nClasses =len(list(le.classes_))
    # prime the network with a sequence randomly selected from tunes.
    random.shuffle(encodedTunes)
    start_seqs = []
    chosen_tune = random.randint(0, len(encodedTunes)-1)
    # Copy the same selected tune for Batch_Size
    for i in range(BATCH_SIZE):
        start_seqs.append(encodedTunes[chosen_tune][:PRIME_LEN])
    y_prime = le.inverse_transform(start_seqs[0])
    print "Prime the sequence with [0]x%d: \n--------\n%s\n--------"%(BATCH_SIZE, ''.join(y_prime)
    start_X = np.array(start_seqs)
    prime_pred = []
    for j in range(PRIME_LEN):
        # convert every feature to (numsample, lenth, dim) format.
        # here x_hot is in (batch_size, 1, nClasses) format
        # y_hot is in (batch_size, nClasses) format
        x_hot = np_utils.to_categorical(start_X[:, [j]], nClasses)
        x_hot = np.expand_dims(x_hot, axis=1)
        y_pred_prob = model.predict_on_batch(x_hot)
        y_pred_code = np.argmax(y_pred_prob[0])
        # take the prediction for the char j in first prime sequence.
        prime_pred.append(le.inverse_transform(y_pred_code))
```

```
    print "prime_pred_info", len(prime_pred)
    # convert last prob prime output to label to display and check.
    print "Prime result:\n--------\n%s\n--------"%(''.join(prime_pred))

    lastToken = []
    # Take the last predicted character, continue feeding it to model as input.
    x_next_hot = prob2input(y_pred_prob)
    more_preds = [prime_pred[-1]]
    tunes_count = 0
    while((''.join(lastToken) != END_TOKEN or tunes_count < maxTunes) and len(more_preds) < ma
        y_pred_prob = model.predict_on_batch(x_next_hot)
        x_next_hot = prob2input(y_pred_prob)
        y_pred_code = np.argmax(y_pred_prob[0])
        pred_char = le.inverse_transform(y_pred_code)
        lastToken.append(pred_char)
        if len(lastToken) ==  6:
            lastToken = lastToken[1:]
        more_preds.append(pred_char)
        if ''.join(lastToken) == END_TOKEN:
            tunes_count +=1
    result = [y_prime.tolist()[0]] # Add first character
    result.extend(prime_pred) # Add rest to result.
    result.extend(more_preds)
    return result

def runRNN(seqLen):
    # Generate result folders for result and models.
    if not isdir(MODEL_STORAGE):
        makedirs(MODEL_STORAGE)

    if not isdir(RESULT_STORAGE):
        makedirs(RESULT_STORAGE)

    # Generate folders for tunes storage.
    if not isdir(TUNE_STORAGE):
        makedirs(TUNE_STORAGE)

    PRE_PROCESS_STORE = '%spre_proecessed_seq%d_%s'%(DATA_STORAGE, MAX_SEQ_LEN, QUESTION_ID)
    preprocesspkl = '%s.pkl'%(PRE_PROCESS_STORE)
    preprocesstxt = '%s.txt'%(PRE_PROCESS_STORE)

    if not isfile(preprocesspkl):
        print "Read data from file %s."%(preprocesspkl)
        tunes = loadTunes()
        print "Encode chars to ints."
        encoded_tunes, label_encoder = label2code(tunes)
        print "Chop txt to sequences of %d length" %(MAX_SEQ_LEN)
        X, y = prepDataSeq(encoded_tunes, MAX_SEQ_LEN)
        print "Found total %d sequences"%(len(X))
        x_train, y_train, x_valid, y_valid = partition(X, y)
        print "Done partition."
        data = x_train, y_train, x_valid, y_valid, encoded_tunes, label_encoder
        savePkl(data, preprocesspkl)
    else:
        x_train, y_train, x_valid, y_valid, encoded_tunes, label_encoder = loadPkl(preprocessp

    f = open(preprocesstxt, 'w')
    for seq in x_train:
        f.write(''.join(label_encoder.inverse_transform(seq)))
        f.write('\n--------\n')
    print "Done saving."

    nClasses =len(list(label_encoder.classes_))
    print "There are %d tunes. %d classes."%(len(encoded_tunes), nClasses)
```

```python
    print "%d sequences for training, %d sequences for testing"%(len(x_train), len(x_valid))
    #print label_encoder.classes_

    model = getModel(nClasses)
    trainModel(model, seqLen, x_train, y_train, x_valid, y_valid, nClasses, encoded_tunes, lab

def show_values(pc, ax, dchar, fmt="%s", **kw):
    unvisible_char = dict()
    unvisible_char['\t'] = 'bt'
    unvisible_char['\n'] = 'bn'
    unvisible_char['\r'] = 'br'
    unvisible_char[' '] = 'sp'
    unvisible_char[chr(127)] = ''

    pc.update_scalarmappable()
    for p, color, char in izip(pc.get_paths(), pc.get_facecolors(), dchar):
        x, y = p.vertices[:-2, :].mean(0)
        if np.all(color[:3] > 0.5):
            color = (0.0, 0.0, 0.0)
        else:
            color = (1.0, 1.0, 1.0)

        if char in unvisible_char:
            char = unvisible_char[char]
        ax.text(x, y, fmt % char, ha="center", va="center", color=color, **kw)

if __name__ == "__main__":
    model = getModel()
    model.load_weights(WEIGHT_FILE)

    nmodel = Model(input=model.input, output=model.layers[-2].output)
    nmodel.summary()

    PRE_PROCESS_STORE = '%spre_proecessed_seq%d_%s'%(DATA_STORAGE, MAX_SEQ_LEN, QUESTION_ID)
    preprocesspkl = '%s.pkl'%(PRE_PROCESS_STORE)
    print "Read data from file %s."%(preprocesspkl)
    tunes = loadTunes()
    print "Encode chars to ints."
    encoded_tunes, label_encoder = label2code(tunes)

    nClasses = len(list(label_encoder.classes_))

    print "Read data from file generated music file."
    tunes = loadTunes('../result/tunes/tune_Q4e/240.txt')
    print "Encode chars to ints."
    encoded_tunes = map(label_encoder.transform, tunes)

    X = encoded_tunes[5]
    n = len(X)
    print 'Length of selected tune: %d'%(n)

    for neuronID in range(100):
        print 'Choosing NeuronID: %d'%(neuronID)

        echar = []
        value = []
        x_seqs = np.array([X for i in xrange(BATCH_SIZE)])
        for i in range(n):
            # conver every feature to (numsample, lenth, dim) format.
            # here x_hot is in (batch_size, 1, nClasses) format
            # y_hot is in (batch_size, nClasses) format
            x_hot = np_utils.to_categorical(x_seqs[:, [i]], nClasses)
            x_hot = np.expand_dims(x_hot, axis=1)

            a_pred = nmodel.predict_on_batch(x_hot)
```

```
                echar.append(x_seqs[0, i])
                value.append(a_pred[0, neuronID])

        dchar = label_encoder.inverse_transform(echar)
        dchar = np.hstack((dchar, np.array([chr(127) for i in xrange(256-n)])))
        value = np.hstack((value, np.array([0 for i in xrange(256-n)])))
        value = np.reshape(value, (16,16))

        fig, ax = plt.subplots()
        heatmap = ax.pcolor(value, edgecolors='k', linestyle= 'dashed', linewidths=0.2, cmap='
        plt.gca().invert_yaxis()

        plt.colorbar(heatmap)
        show_values(heatmap, ax, dchar)
        plt.savefig('../result/heatmaps/heatmap_' + str(neuronID) + '.png', dpi=200)
```