Neural Network Project

April 2019

1 Topic

For this project, I trained a neural network to classify music genres. I used the GTZAN dataset, which is 1,000 songs distributed evenly between 10 genres. I used keras on tensorflow to do training and used Tkinter for the app. Several people online have tried to use DNNs to solve this problem, prepossessing data with spectograms and then using 2d-convolutions for the networks. All of these projects have terrible data leakage. They take transforms of the data before they split the data into testing and training data. Since the transforms split the songs into windows of overlapping regions, part of the data from a window in the training data can also be in the testing data, allowing for inaccurate results and over-fitting. Another problem that these projects had with the data was in splitting of the songs. Even if you remove the overlapping data, the projects allowed for part of a song to be in the training data and part of the song to be in the testing data. While this is technically not allowing the same data into both testing and training, songs are very repetitive by nature. This lets the nn to simply learn the pattern of each song and therefore over-fit to our data (songs) which will give us results that are better than reality. All this is to say, while my results may not be as impressive as I would have liked, they fixed many problems that I found online. Before I realized I was cheating the data, I was getting 94% test accuracy (NOT ACCURATE RESULTS), much better then anything done before. Once I fixed the explicit data corruption, I was still getting around 85% accuracy (AGAIN, NOT ACCURATE RESULTS). When I finally realized the extent of the data cheating, I made sure the testing data was separated from training data immediately after loading in raw data and to never touch the testing data again until testing. With this fixed data separation, I was able to get 74.5% classification accuracy of songs. Since I could not find anything that compares to these results online, I think this is a reasonable starting point for genre classification. This accuracy even beats most results that were based on testing data with severe data leakage (68% and 72%). My DNN also does no data prepossessing (FFT or any other transform) but simply uses raw data. I do split up the song into chunks of 3 seconds and then apply the DNN to each of these 3s blocks. To predict the song genre, I simply take the genre predicted the most out of these 3s blocks. My DNN is based on the WaveNet architecture which was developed by Google to analyze audio data. This is the first implementation of WaveNet to predict music genre (or at least I did not find any implementations online).

2 Dataset

GTZAN (http://opihi.cs.uvic.ca/sound/genres.tar.gz or better yet https://drive.google.com/open?id=1X33sLOPQohzrVaThHvZFuqF_PfCqY4Ai which is faster to downloaded and hosted by my google account) is a dataset of 1,000 songs, all of length 30 seconds and 1 of 10 genres. The genres include: blues, classical, country, disco, hiphop, jazz, metal, pop, reggae, and rock. Each song is given in .au format and have a playback frequency of 22.05kHz.

3 CNN Model

3.1 Architecture

My nn uses raw data broken into 3 second intervals. I then apply a series of 6 1d convolutions, maxpoolings, and dropouts to this 3 second block of the song. At each level I increase the dilation by a factor of 2 as per the WaveNet architecture. The model looks like the following:

```
input = Input(shape=input_shape)
model = BatchNormalization()(input)
model = Conv1D(filters=128, kernel_size=9, activation='relu', dilation_rate=1)(model)
model = MaxPooling1D(pool_size=3)(model)
model = Dropout(0.25)(model)
model = Conv1D(filters=128, kernel_size=9, activation='relu', dilation_rate=2)(model)
model = MaxPooling1D(pool_size=3)(model)
model = Dropout(0.25)(model)
model = Conv1D(filters=64, kernel_size=9, activation='relu', dilation_rate=4)(model)
model = MaxPooling1D(pool_size=3)(model)
model = Dropout(0.25)(model)
model = Conv1D(filters=64, kernel_size=9, activation='relu', dilation_rate=8)(model)
model = MaxPooling1D(pool_size=3)(model)
model = Dropout(0.25)(model)
model = Conv1D(filters=32, kernel_size=9, activation='relu', dilation_rate=16)(model)
model = MaxPooling1D(pool_size=3)(model)
model = Dropout(0.25)(model)
model = Conv1D(filters=32, kernel_size=7, activation='relu', dilation_rate=32)(model)
model = MaxPooling1D(pool_size=3)(model)
model = Flatten()(model)
model = Dropout(0.25)(model)
output = Dense(num_genres, activation='softmax')(model)
model = Model(input,output)
```

The first thing the nn does is normalize the data. Then it starts the convolutions and maxpooling. The number of filters of the nn also decrease by a factor of 2 (mainly because of memory constraint problems) at each level. All activation functions are ReLu. Maxpooling is added after each convolution to help manage the number of parameters to a reasonable number. Dropout is added after each maxpooling to help curb over fitting. Alot of the parameters are tuned to allow for as deep a network as possible. I tried to get a network with 7 convolution layers to work, but it either ran out of memory or was very inaccurate.

3.2 Input: Shape of Tensor

X train shape is (8000, 60150, 1)

X test shape is (2000, 60150, 1)

3.3 Output: Shape of Tensor

Y train shape is (8000,)

Y test shape is (2000,)

3.4 Shape of Output Tensor for Each Convolution Layer

(Maxpooling layer simply reduces the 2nd dimension by 1/3 and dropout doesn’t affect size)

Output of first convolution layer: (8000, 66142, 128)

Output of second convolution layer: (8000, 22031, 128)

Output of third convolution layer: (8000, 7311, 64)

Output of fourth convolution layer: (8000, 2373, 64)

Output of fifth convolution layer: (8000, 663, 32)

Output of sixth convolution layer: (8000, 29, 32)

4 Hyperparameters

4.1 List of Hyperparameters

The main hyperparameter that I am tuning in this project is the stride (directly related to the dilation rate which is discussed in WaveNet). Although this less tuning and more using WaveNet approch. I also am tuning the number of filters for each CNN, the kernel size, and the dropout rate. Batch size may also be changed and epochs given the validation data results. I also played around with batch normalization, but found applying it just once at the beginning gave the best results.

4.2 Range of Values of Hyperparameters Tried

Kernel size: 3 to 11

Number of filters: 8 to 256

Droupout rate: .0 to .3

Batch size: 16 to 64

Epochs: 0 to 40

4.2 Optimal Hyperparameters Found

Kernel size: 9 (7 for last layer)

Number of filters: 128,64,32 (for each 2 layer)

Droupout rate: .25

Batch size: 32

Epochs: 22

5 Annotated Code

```
import os
import numpy as np
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from pandas import get_dummies
from load import load_data
from transform_data import transform_song, transform_songs
from nn import genre_model, test_model

## parameters:
# location of this file (main.py)
file_path = os.path.abspath(os.path.dirname(__file__))
# file giving the order of genres (used to make sense of nn output)
genre_file = file_path+'/data/genres.txt'
# where we save model
model_file = file_path+'/model.h5'
# where we save weights in case of error
weights_file = file_path+'/temp_weights.h5'
# what sample rate to resample songs to
sr = 22050
# time length to cut the song into
time_length = 3

## for hyperparameter tunig
# use validation set
validate = False
# number of epochs
epochs = 22
# batch size
batch_size = 32
##

# load data
songs, genres = load_data(sr)
# break data into training and testing data
x_train, x_test, y_train, y_test = train_test_split(songs, genres, stratify=genres, test_size=0.2, random_state=1)
# break into training and validation data
if(validate):
  x_train, x_val, y_train, y_val = train_test_split(x_train, y_train, stratify=y_train, test_size=0.2, random_state=1)
  y_val = get_dummies(y_val)
  y_val = y_val.values
  x_val, y_val = transform_songs(x_val, y_val, sr, time_length)
  validation_data = (x_val,y_val)
  x_val = 0
  y_val = 0
else:
  validation_data = None
# clear some memory
songs = 0
genres = 0
# one hot encode genres
y_train = get_dummies(y_train)
y_test_transform = get_dummies(y_test)
# get number of genres and save genre order to file
genres = y_train.columns
num_genres = len(genres)
np.savetxt(genre_file, genres, delimiter=' ', fmt='%s')
# get the data from pandas
y_train = y_train.values
y_test_transform = y_test_transform.values
# transform song into more usable data
x_train, y_train = transform_songs(x_train, y_train, sr, time_length)
x_test_transform, y_test_transform = transform_songs(x_test, y_test_transform, sr, time_length)
# shuffle training data to mix genres
```

```
np.random.seed(1)
p = np.random.permutation(range(y_train.shape[0]))
x_train = x_train[p]
y_train = y_train[p]
# get model
model = genre_model(num_genres, x_train[0].shape)
# compile model
model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
# train model
try:
  history = model.fit(x_train, y_train, epochs=epochs, batch_size=batch_size, validation_data=validation_data)
  historyDict = history.history
  # plot the accuracy and loss of training and validation data
  for key in historyDict.keys():
    plt.plot(historyDict[key],'.-')
  plt.legend(historyDict.keys())
  # save model to file
  model.save(model_file)
  # show plots
  plt.show()
# deal with errors
except Exception as e:
  print(e.args)
  try:
    model.save_weights(weights_file)
    print('error occurred, saving weights to '+weights_file)
  except:
    print('model not defined, no weights saved')
# test model on 3s interval
loss, accuracy = model.evaluate(x_test_transform, y_test_transform)
print('test 3s intervals accuracy:', accuracy)
print('test 3s intervals loss:', loss)
# test model on whole songs
test_model(x_test,y_test,genres,transform_song,sr,time_length,model)
```

```
from __future__ import print_function
import sys
import numpy as np
from scipy import stats
from keras import Input
from keras.models import Model
from keras.layers import Dense, Flatten, Dropout, BatchNormalization
from keras.layers.convolutional import Conv1D, MaxPooling1D, AveragePooling1D
# my wavenet model
def genre_model(num_genres, input_shape):
  input = Input(shape=input_shape)
  model = BatchNormalization()(input)
  model = Conv1D(filters=128, kernel_size=9, activation='relu', dilation_rate=1)(model)
  model = MaxPooling1D(pool_size=3)(model)
  model = Dropout(0.25)(model)
  model = Conv1D(filters=128, kernel_size=9, activation='relu', dilation_rate=2)(model)
  model = MaxPooling1D(pool_size=3)(model)
  model = Dropout(0.25)(model)
  model = Conv1D(filters=64, kernel_size=9, activation='relu', dilation_rate=4)(model)
  model = MaxPooling1D(pool_size=3)(model)
  model = Dropout(0.25)(model)
  model = Conv1D(filters=64, kernel_size=9, activation='relu', dilation_rate=8)(model)
  model = MaxPooling1D(pool_size=3)(model)
  model = Dropout(0.25)(model)
  model = Conv1D(filters=32, kernel_size=9, activation='relu', dilation_rate=16)(model)
  model = MaxPooling1D(pool_size=3)(model)
  model = Dropout(0.25)(model)
  model = Conv1D(filters=32, kernel_size=7, activation='relu', dilation_rate=32)(model)
  model = MaxPooling1D(pool_size=3)(model)
  model = Flatten()(model)
  model = Dropout(0.25)(model)
  output = Dense(num_genres, activation='softmax')(model)
  model = Model(input,output)
  return(model)
# get index integer value that nn predicts
def predict_song_index(song,model):
  pred = model.predict(song)
  pred = np.argmax(pred,axis =1)
  return stats.mode(pred)[0]
# predict genre for a single song (genre_list is possible genres to pick from)
def predict_song(song,genres_list,transform_song,sr,time_length,model):
  song = transform_song(song,sr,time_length)
  index = predict_song_index(song,model)
  return genres_list[index][0]
# determine the accuracy of model given test data songs, genres
def test_model(songs,genres,genres_list,transform_song,sr,time_length,model):
  print('testing model ', end='')
  correct = 0
  i = 0
  for song in songs:
    genre = predict_song(song,genres_list,transform_song,sr,time_length,model)
    if genre == genres[i]:
      correct = correct+1
    i = i+1
    print('.', end='')
    sys.stdout.flush()
  print('\ntest song accuracy: ', correct/genres.shape[0])
```

```python
import numpy as np

# split song into sections of time length given
def splitsong(song, sr, time_length):
  x = []
  step = sr*time_length
  for s in [song[i:i+step] for i in range(0, song.shape[0], step)]:
    x.append(s)
  return np.asarray(x)

# make song usable to nn
def transform_song(song, sr, time_length):
  song = splitsong(song, sr, time_length)
  song = np.expand_dims(song, axis=3)
  return song

# make all songs in train/test data usable to nn
def transform_songs(songs, genres, sr, time_length):
  print('transforming songs')
  x = []
  y = []
  for i in range(songs.shape[0]):
    song = transform_song(songs[i], sr, time_length)
    x.extend(song)
    y.extend(song.shape[0]*[genres[i]])
  print('finished transforming songs')
  return np.asarray(x), np.asarray(y)
```

6 Training and Testing Performance

6.1 Train/Validate:

Train on 6400 samples, validate on 1600 samples

Epoch 1/40

6400/6400 [==============================] - 75s 12ms/step - loss: 2.1094 - acc: 0.2028 - val_loss: 1.9968 - val_acc: 0.3006

Epoch 2/40

6400/6400 [==============================] - 69s 11ms/step - loss: 1.7302 - acc: 0.3700 - val_loss: 1.7965 - val_acc: 0.3887

Epoch 3/40

6400/6400 [==============================] - 63s 10ms/step - loss: 1.5286 - acc: 0.4453 - val_loss: 1.7196 - val_acc: 0.4744

Epoch 4/40

6400/6400 [==============================] - 64s 10ms/step - loss: 1.3370 - acc: 0.5233 - val_loss: 1.6781 - val_acc: 0.4631

Epoch 5/40

6400/6400 [==============================] - 68s 11ms/step - loss: 1.2087 - acc: 0.5673 - val_loss: 1.6166 - val_acc: 0.4700

Epoch 6/40

6400/6400 [==============================] - 65s 10ms/step - loss: 1.1067 - acc: 0.5973 - val_loss: 1.6449 - val_acc: 0.4631

Epoch 7/40

6400/6400 [==============================] - 65s 10ms/step - loss: 1.0057 - acc: 0.6400 - val_loss: 1.5202 - val_acc: 0.5425

Epoch 8/40

6400/6400 [==============================] - 65s 10ms/step - loss: 0.9326 - acc: 0.6684 - val_loss: 1.3886 - val_acc: 0.5463

Epoch 9/40

6400/6400 [==============================] - 69s 11ms/step - loss: 0.8433 - acc: 0.6981 - val_loss: 1.4619 - val_acc: 0.5337

Epoch 10/40

6400/6400 [==============================] - 63s 10ms/step - loss: 0.7825 - acc: 0.7203 - val_loss: 1.4928 - val_acc: 0.5494

Epoch 11/40

6400/6400 [==============================] - 62s 10ms/step - loss: 0.7117 - acc: 0.7534 - val_loss: 1.4800 - val_acc: 0.5556

Epoch 12/40

6400/6400 [==============================] - 62s 10ms/step - loss: 0.6708 - acc: 0.7612 - val_loss: 1.2536 - val_acc: 0.5881

Epoch 13/40

6400/6400 [==============================] - 62s 10ms/step - loss: 0.6114 - acc: 0.7837 - val_loss: 1.3641 - val_acc: 0.5813

Epoch 14/40

6400/6400 [==============================] - 62s 10ms/step - loss: 0.5537 - acc: 0.8069 - val_loss: 1.4254 - val_acc: 0.5863

Epoch 15/40

6400/6400 [==============================] - 67s 10ms/step - loss: 0.5276 - acc: 0.8139 - val_loss: 1.2589 - val_acc: 0.5913

Epoch 16/40

6400/6400 [==============================] - 64s 10ms/step - loss: 0.5022 - acc: 0.8237 - val_loss: 1.2799 - val_acc: 0.6025

Epoch 17/40

6400/6400 [==============================] - 61s 10ms/step - loss: 0.4549 - acc: 0.8413 - val_loss: 1.3060 - val_acc: 0.6119

Epoch 18/40

6400/6400 [==============================] - 61s 10ms/step - loss: 0.4309 - acc: 0.8519 - val_loss: 1.3329 - val_acc: 0.5956

Epoch 19/40

6400/6400 [==============================] - 61s 10ms/step - loss: 0.3937 - acc: 0.8633 - val_loss: 1.2333 - val_acc: 0.6425

Epoch 20/40

6400/6400 [==============================] - 61s 10ms/step - loss: 0.3853 - acc: 0.8645 - val_loss: 1.3713 - val_acc: 0.6362

Epoch 21/40

6400/6400 [==============================] - 61s 10ms/step - loss: 0.3461 - acc: 0.8798 - val_loss: 1.2374 - val_acc: 0.6375

Epoch 22/40

6400/6400 [==============================] - 62s 10ms/step - loss: 0.3721 - acc: 0.8698 - val_loss: 1.2059 - val_acc: 0.6331

Epoch 23/40

6400/6400 [==============================] - 61s 10ms/step - loss: 0.3321 - acc: 0.8856 - val_loss: 1.2906 - val_acc: 0.6175

Epoch 24/40

6400/6400 [==============================] - 61s 10ms/step - loss: 0.3000 - acc: 0.8969 - val_loss: 1.3813 - val_acc: 0.6056

Epoch 25/40

6400/6400 [==============================] - 61s 10ms/step - loss: 0.2934 - acc: 0.8991 - val_loss: 1.5493 - val_acc: 0.6138

Epoch 26/40

6400/6400 [==============================] - 64s 10ms/step - loss: 0.2724 - acc: 0.9045 - val_loss: 1.4265 - val_acc: 0.6206

Epoch 27/40

6400/6400 [==============================] - 60s 9ms/step - loss: 0.2607 - acc: 0.9114 - val_loss: 1.2946 - val_acc: 0.6469

Epoch 28/40

6400/6400 [==============================] - 59s 9ms/step - loss: 0.2809 - acc: 0.9036 - val_loss: 1.2318 - val_acc: 0.6544

Epoch 29/40

6400/6400 [==============================] - 59s 9ms/step - loss: 0.2457 - acc: 0.9141 - val_loss: 1.3156 - val_acc: 0.6325

Epoch 30/40

6400/6400 [==============================] - 60s 9ms/step - loss: 0.2363 - acc: 0.9181 - val_loss: 1.4089 - val_acc: 0.6256

Epoch 31/40

6400/6400 [==============================] - 60s 9ms/step - loss: 0.2255 - acc: 0.9192 - val_loss: 1.3816 - val_acc: 0.6200

Epoch 32/40

6400/6400 [==============================] - 59s 9ms/step - loss: 0.2430 - acc: 0.9187 - val_loss: 1.2853 - val_acc: 0.6469

Epoch 33/40

6400/6400 [==============================] - 60s 9ms/step - loss: 0.2125 - acc: 0.9281 - val_loss: 1.4080 - val_acc: 0.6119

Epoch 34/40

6400/6400 [==============================] - 59s 9ms/step - loss: 0.2648 - acc: 0.9087 - val_loss: 1.4305 - val_acc: 0.6469

Epoch 35/40

6400/6400 [==============================] - 59s 9ms/step - loss: 0.2037 - acc: 0.9295 - val_loss: 1.5683 - val_acc: 0.6019

Epoch 36/40

6400/6400 [==============================] - 59s 9ms/step - loss: 0.1745 - acc: 0.9383 - val_loss: 1.4817 - val_acc: 0.6500

Epoch 37/40

6400/6400 [==============================] - 59s 9ms/step - loss: 0.1815 - acc: 0.9383 - val_loss: 1.5604 - val_acc: 0.6219

Epoch 38/40

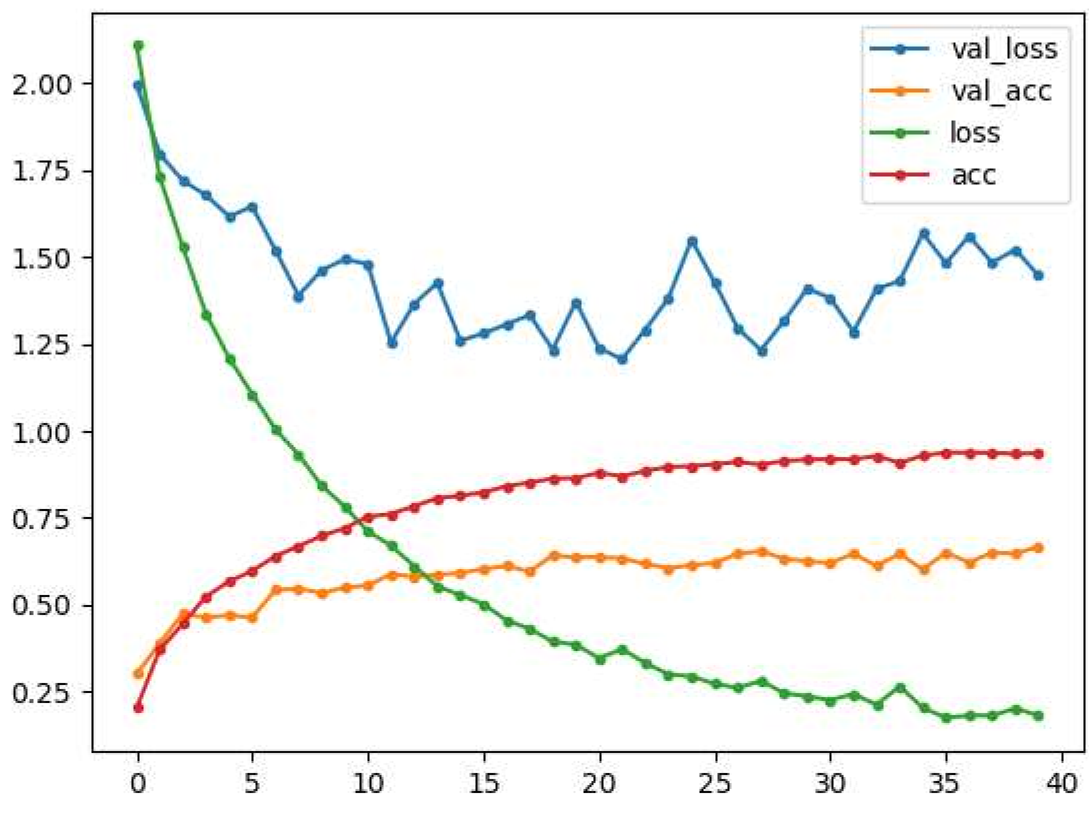
6400/6400 [==============================] - 59s 9ms/step - loss: 0.1810 - acc: 0.9375 - val_loss: 1.4835 - val_acc: 0.6500

Epoch 39/40

6400/6400 [==============================] - 59s 9ms/step - loss: 0.2018 - acc: 0.9344 - val_loss: 1.5213 - val_acc: 0.6475

Epoch 40/40

6400/6400 [==============================] - 59s 9ms/step - loss: 0.1812 - acc: 0.9366 - val_loss: 1.4470 - val_acc: 0.6675



6.2 Train/Test:

8000/8000 [==============================] - 77s 10ms/step - loss: 2.0658 - acc: 0.2251

Epoch 2/22

8000/8000 [==============================] - 68s 9ms/step - loss: 1.6278 - acc: 0.4048

Epoch 3/22

8000/8000 [==============================] - 68s 9ms/step - loss: 1.4241 - acc: 0.4835

Epoch 4/22

8000/8000 [==============================] - 69s 9ms/step - loss: 1.2598 - acc: 0.5484

Epoch 5/22

8000/8000 [==============================] - 69s 9ms/step - loss: 1.1453 - acc: 0.5952

Epoch 6/22

8000/8000 [==============================] - 68s 9ms/step - loss: 1.0283 - acc: 0.6374

Epoch 7/22

8000/8000 [==============================] - 68s 9ms/step - loss: 0.9199 - acc: 0.6729

Epoch 8/22

8000/8000 [==============================] - 69s 9ms/step - loss: 0.8359 - acc: 0.7074

Epoch 9/22

8000/8000 [==============================] - 69s 9ms/step - loss: 0.7592 - acc: 0.7382

Epoch 10/22

8000/8000 [==============================] - 68s 9ms/step - loss: 0.6937 - acc: 0.7615

Epoch 11/22

8000/8000 [==============================] - 68s 9ms/step - loss: 0.6602 - acc: 0.7732

Epoch 12/22

8000/8000 [==============================] - 70s 9ms/step - loss: 0.5989 - acc: 0.7916

Epoch 13/22

8000/8000 [==============================] - 68s 9ms/step - loss: 0.5485 - acc: 0.8119

Epoch 14/22

8000/8000 [==============================] - 68s 9ms/step - loss: 0.4881 - acc: 0.8309

Epoch 15/22

8000/8000 [==============================] - 69s 9ms/step - loss: 0.4734 - acc: 0.8350

Epoch 16/22

8000/8000 [==============================] - 71s 9ms/step - loss: 0.4444 - acc: 0.8441

Epoch 17/22

8000/8000 [==============================] - 68s 9ms/step - loss: 0.4078 - acc: 0.8606

Epoch 18/22

8000/8000 [==============================] - 68s 9ms/step - loss: 0.3864 - acc: 0.8625

Epoch 19/22

8000/8000 [==============================] - 68s 9ms/step - loss: 0.3813 - acc: 0.8685

Epoch 20/22

8000/8000 [==============================] - 70s 9ms/step - loss: 0.3662 - acc: 0.8720

Epoch 21/22

8000/8000 [==============================] - 69s 9ms/step - loss: 0.3386 - acc: 0.8830

Epoch 22/22

8000/8000 [==============================] - 69s 9ms/step - loss: 0.3217 - acc: 0.8861

2000/2000 [==============================] - 6s 3ms/step

test 3s intervals accuracy: 0.701

test 3s intervals loss: 1.0057121869325638

testing model ………………..………………..………………..………………..……………….. ………………..………………..………………..………………..………………..

test song accuracy: 0.745

7 Instructions on how to test the trained CNN and how to use the GUI

7.1 Install Dependencies

- Python 2 or 3
- Tkinter
- Keras
- Tensorflow
- Numpy. Scipy
- sklearn
- librosa (may need ffmpeg installed too)
- pandas
- sounddevice

7.2 Execution

To train/test CNN, go to model directory and run python main.py. To edit CNN, go to nn.py. To run app, run python gui.py

7.3 Code

Code on Github: https://github.com/jcroc32/music-genre-classification

7.4 Video Link

Video for gui demo: https://youtu.be/OvO67VXRK7s