Authors: Joshua Patterson & Alex Richardson

November 6, 2019

CMSI 401

SDD: Software Design Document (Architecture + Detailed)

6.1. Introduction

This document serves as a detailed guide to understanding the functionality and design of our Software Project: *yung gan*. Our software provides a command line application to both generate music from pretrained models and to train a new model from a .wav or .mp3 music collection. Our application that will fabricate authentic-sounding hip hop beats through its unique implementation of a GAN, or Generative Adversarial Network. A vector of random noise is given as the only input to a deep network, known as the generator, where it funnels information through both recurrent and convolutional layers to create a computer generated beat in an attempt to fool another deep network, known as the discriminator. We will curate a playlist of real human-made beats from sources: Soundcloud, Youtube, and our own music. Those files will be preprocessed and stored in an SQL database along with its pertinent information. When training, those songs feed into the discriminator in batches that are combined with fake beats the generator has constructed. Over many epochs, the generator and discriminator loss should converge, indicating the training has finished. If our neural architecture is well constructed, the generator should produce beats that sound authentic to both the discriminator and to the human ear. Our system will include both a training command-line application written in Python and a generator command-line application written in Python that uses pre-trained generators included with the software to produce beats on quick command. A front-end web application is projected to be created for our command-line software. This will be done by using React.JS, CSS, and HTML. By having our command-line software run on through AWS GPU Server, an EC2 Virtual Machine Instance could be created to allow the python program to reach our front-end. Pointing Amazon API Gateway to a http endpoint on EC2 would allow this to happen.

6.1.1 System Objectives

The objectives of our system are to fulfill the task of replicating a hip-hop beat efficiently enough to our standards, and to continue to train/build on what we will have developed. As an

application itself, an objective is to provide the user with a beat of their choosing (genre, sub-genre, bpm..) and to be able to interact with a front end that stores the command-line program's functionality.

6.1.2 Hardware, Software, and Human Interfaces

Hardware:

- Macbook
- Mac Computer
- Wifi
- For CPU:

  Minimum Requirements:
  - CPU: 2.4GHz dual core processor
  - RAM: 16 GB
  - Storage: 64 GB

  Suggested:
  - CPU: 2.4GHz quad core processor
  - GPU: CUA enabled NVIDIA
  - RAM: 64 GB
  - Storage: 128 GB

Software:

- Text Editor
  - Visual Studio Code Version 1.38.1
- OS
  - Mac OS Mojave Ver. 10.14.6
  - Windows OS
- Python Ver. 3.6 or above
- Keras Ver. 2.2.4
- TensorFlow Ver. 1.14
- Javascript ES6
- React.JS 16.8
- HTML5
- CSS3
- Other
  - Amazon Web Services

- Terminal
- Github
- Google Docs
- Soundcloud
- Youtube
- Soundcloud/Youtube Downloader

6.2 Architectural Design

The *yung gan* system is designed as a generative adversarial network, two deep learning networks minimizing loss adversarially. The networks are designed as a combination of recurrent and convolutional networks. The convolutional networks model the spectral element of music while recurrent networks model the time aspect. The discriminator runs through 2 recurrent layers, known as Long Short Term Memory, then through 8 convolutional blocks to determine song validity. The generator does essentially the reverse process by beginning with 100 random values and building that up through convolution then the LSTM layers.

6.2.1 Major Software Components

- Back-End
  - Generator
    - 2 LSTM layers
    - 10 Convolutional layers
  - Discriminator
    - 2 LSTM layers
    - 10 Convolutional layers
- Server
  - AWS GPU
    - Tensorflow Backend with CUDA
    - Keras
- Front End
  - React.JS

6.2.2 Major Software Interactions

- Back-End and Front-End Interactions
  - By having our command-line software run on through AWS GPU Server, an EC2 Virtual Machine Instance could be created to allow the python program to reach our front-end. Pointing Amazon API Gateway to a http endpoint on EC2 would allow this to happen.

- Front End
    - Front Page
        - On-Click Button
            - Downloads .mp3 file
        - Header
            - Explanation of software

6.2.3 Architectural Design Diagrams

Figure 1: Discriminator/Generator

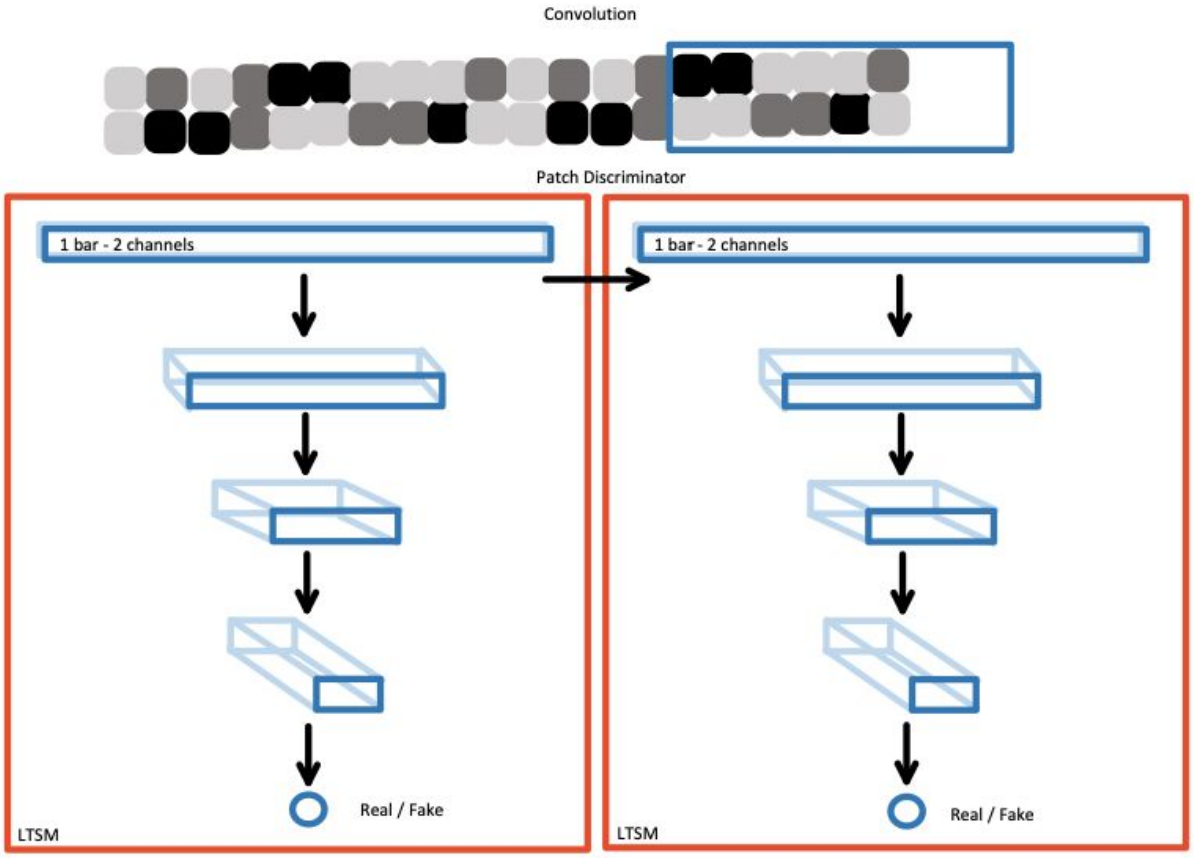


Figure 2: Scaled Down Back-End/Front-End Design

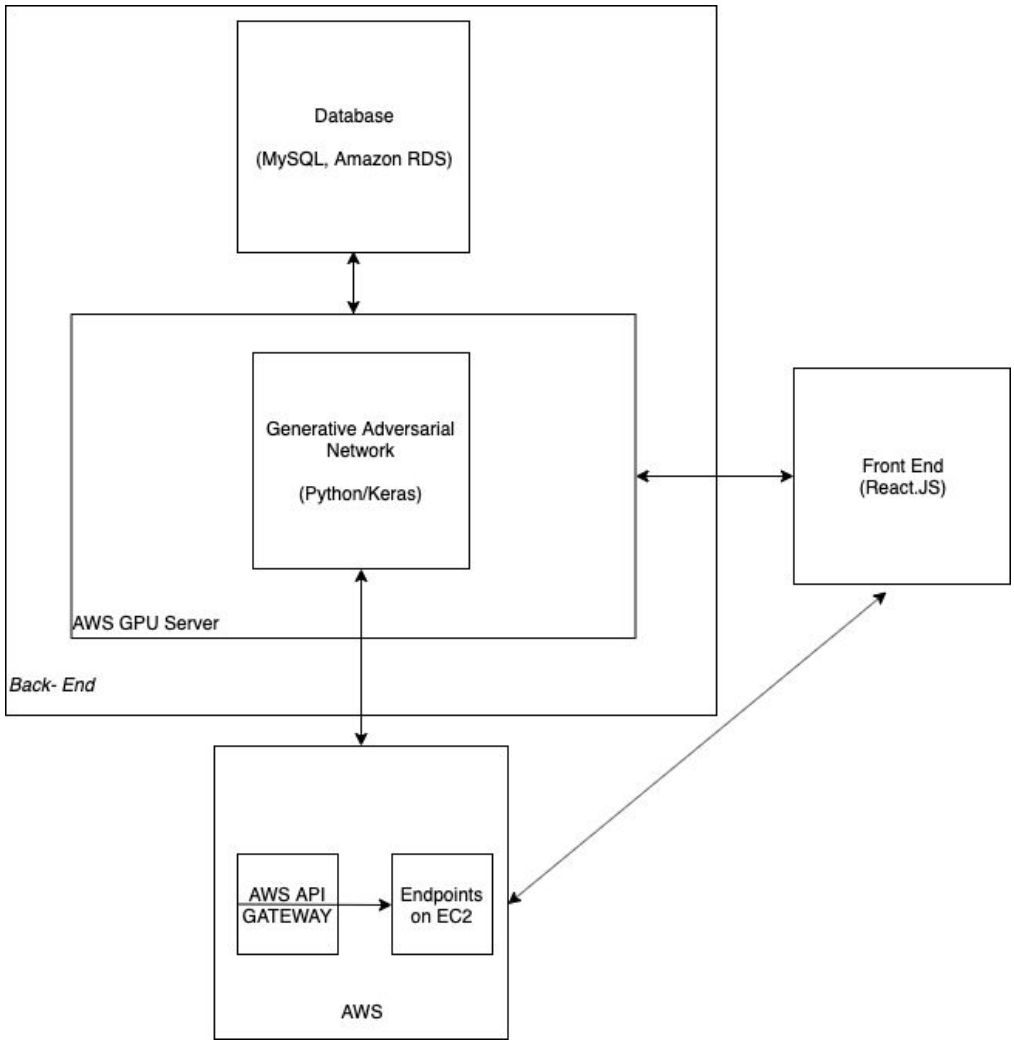
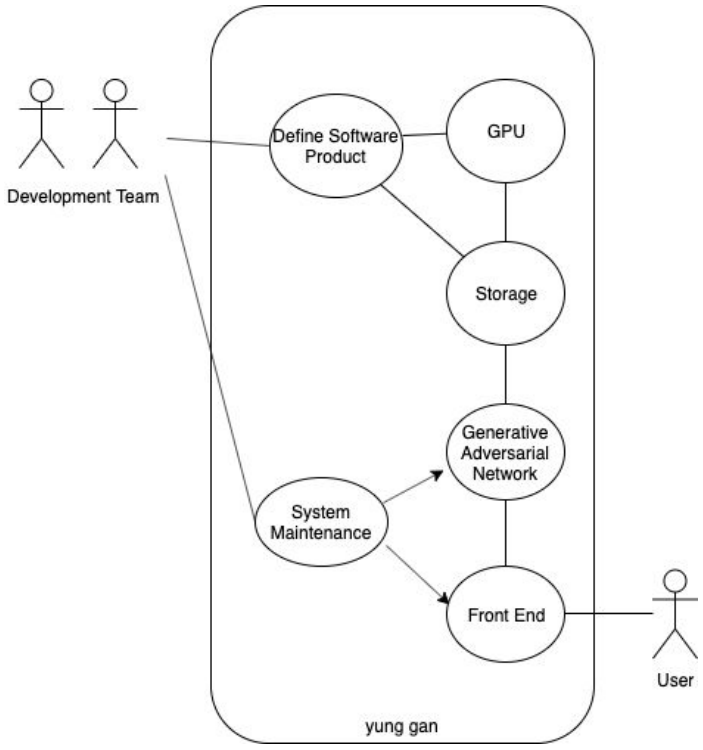


Figure 3: Use Case Diagram



6.3. CSC and CSU Descriptions

6.3.1 Class Descriptions

CSCI yung gan is composed of the following CSCs:

6.3.1 Server CSC -- Amazon EC2 with GPU
- 6.3.1.1 Generator
  - 6.3.1.1.1 Number of Songs
  - 6.3.1.1.2 Output Directory
  - 6.3.1.1.3 Genre
  - 6.3.1.1.4 Subgenre
  - 6.3.1.1.5 Generator Filepath
- 6.3.1.2 Train CSC
  - 6.3.1.2.1 Number of Epochs
  - 6.3.1.2.2 Training Directory
  - 6.3.1.2.3 Data Tempo
  - 6.3.1.2.4 Batch Size
  - 6.3.1.2.5 Save Interval

6.3.1.2.6 Preprocessing
6.3.1.3 Front End CSC -- React Native
6.3.1.3.1 Front Page
6.3.1.3.2 Text Field Parameters
6.3.1.3.2.1 BPM
6.3.1.3.2.2 Genre
6.3.1.3.2.3 Sub-Genre
6.3.1.3.2.4 Key
6.3.1.3.3 Button
6.3.1.3.3.1 API Call
6.3.3.1.3.3.1.1 Download MP3 File
6.3.3.1.3.3.2 Download WAV File
6.3.1.3.4 API CSC - Amazon EC2 with GPU
6.3.1.3.4.1 APIGW
6.3.1.3.4.1.1 HTTP Endpoint

6.3.3 The generator application shall include a required input parameter for number of songs to generate

6.3.4 The generator application shall include a required input parameter for the export file path location.

6.3.5 The generator application shall include an option for sub-genre.

6.3.6 The generator application shall include an option for genre.

6.3.7 The generator application shall include an option for giving a filepath to a saved generator in h5 format.

6.3.8 The generator application shall return a help message if the user attempts to provide a generator filepath if genre or subgenre are already provided by the user

6.3.9 The generator application shall return a help message if the user attempts to provide a genre or subgenre if generator filepath is already provided.

6.3.10 The generator application shall include an option for help by showing the allowed options and mandatory parameters.

6.3.11 The generator application shall provide the user with at least one generated song per second.

6.3.12 The generator application shall save the output directly in the output destination if number of songs is given as 1.

6.3.13 The generator application shall save the outputs as a folder in the output destination if number of songs is given as more than 1.

6.3.14 The generator application shall save songs in .wav format with standard 44.1 kHz 16bit audio

6.3.15 The training application shall require the user to provide the number of training epochs for the process to undergo

6.3.16 The training application shall require the user to provide the input directory for the training set

6.3.17 The training application shall require the user to provide the tempo in beats per minute of the training set if preprocess is set to off

6.3.18 The training application shall require the user to provide their desired output tempo in beats per minute if preprocess is set to on

6.3.19 The training application shall include an option for the user to provide their desired batch size

6.3.20 The training application shall include an option for the user to provide their desired save rate

6.3.21 The training application shall include an option for the user to preprocess their dataset

6.3.22 The training application shall save the preprocessed data in its given training directory

6.3.23 The training application shall save preprocessed data in .wav format with standard 44.1 kHz 16bit audio

6.3.24 The training application shall save data samples in accordance to the save rate in .wav format with standard 44.1 kHz 16bit audio

6.3.25 The training application shall only accept training data in .wav format with standard 44.1 kHz 16bit audio

6.3.26 The training application shall provide a summary of songs loaded once loading finishes

6.3.27 The training application shall provide the loss statistics for the generator and discriminator at each epoch

6.3.28 The training application shall provide the elapsed time at each epoch

6.3.2 Detailed Interface Descriptions

6.3.2.0 Introduction

The classes of our two CSU's are primarily helpers of one main class for each CSU. The generators process is simply to load weights for the generator model and pass noise into the network for song outputs to be saved. The user provides the number of songs, output directory, and a genre string. For training, the process involves building a generator that upscales white noise through convolution to match the length of the data. The discriminator does essentially the opposite process by downscaling real and generated songs to a single validity score. After building the underlying network structure, we must compile two trainable networks

such they learn adversarially of each other. Finally, given user input for batch-size, epochs, and dataset directory, the training process is run to completion outputting samples at a given save-rate until outputting and saving the final model for use in the Generator CSU.

6.3.2.1 Generate

6.3.2.1.1 LoadWeights

Loads the keras network weights from a h5 file in given directory.

6.3.2.1.2 SaveSongToDir

Saves the song as 44100 hZ 16-bit wav's in given directory.

6.3.2.1.2 GenerateSong

Loads weights, performs n evaluations, and saves the results as wav files.

6.3.2.2 Train

6.3.2.2.1 BuildGenerator

Builds the multi-layered network to upscale noise into the desired format

6.3.2.2.1 BuildDiscriminator

Builds the multi-layered network to downscale songs into validity

6.3.2.2.3 CompileModels

Compiles the two models to properly train each other adversarially

6.3.2.2.4 Train

Trains model on given dataset over n epochs and saves weights as h5

6.3.3 Detailed Data Structure Descriptions

6.3.3.1 Keras Neural Network

The Keras Networks used are built with a collection of layers. In a Sequential model, as often used, layers are arranged in the same order that they were added to the model. Each layer

may contain a collection of both trainable and untrainable parameters stored as weight matrices. Upon compilation, these weight matrices are adjusted each time its train function is called.

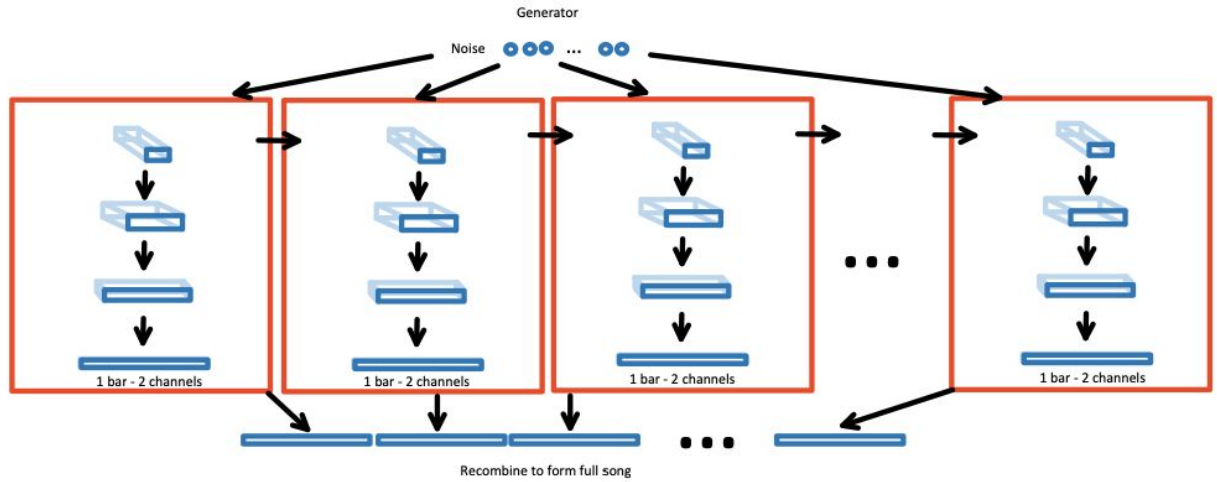
6.3.3.2 Scipy Wavfile

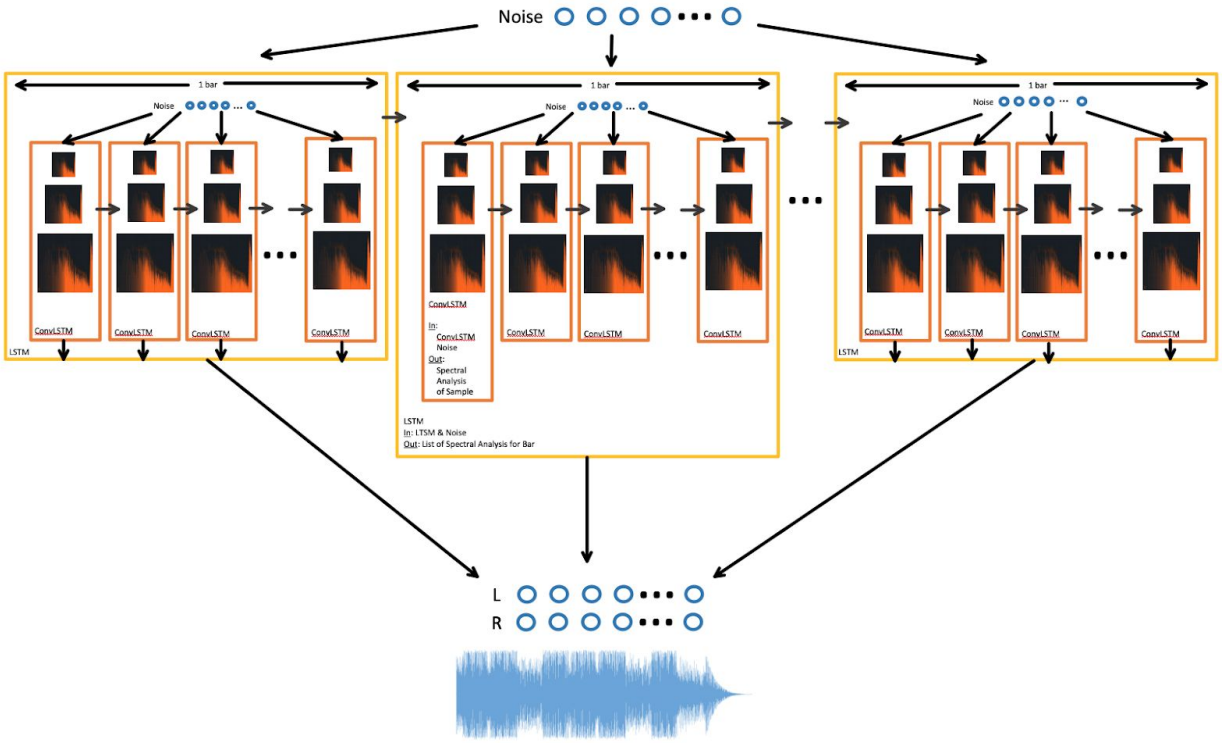
Scipy provides the most versatile format for encoding wav files. For each numpy datatype, wavfile can interpret wav music data based on the datatype's bounding limits. Therefore, wavfile can work well within the bounds of Kera's activation functions such that wav data can be interpreted from tanh activation, which activates within the range -1 to 1. Scipy is also helpful in loading songs and preparing them for processing with the discriminator network.

6.3.2.3 Numpy Array

Even within the larger context of Keras and Scipy, the underlying data structure implemented is the Numpy Array. The Numpy Array is Python's answer to their default memory-hogging lists because they are constructed as a fixed array size with a given datatype. This works well within the context of deep learning because deep networks have predetermined sizes for inputs, outputs, and weights. While the weights may change, the size of the weight matrix does not change.

6.3.4 Detailed Design Diagrams





6.4 Database Design and Description

"Not Applicable"

6.4.1 Database Design ER Diagram

"Not Applicable"

6.4.2 Database Access

"Not Applicable"

6.4.3 Database Security

"Not Applicable"