CLASSIFYING HINDUSTANI INDIAN RAGAS WITH CONVOLUTIONAL NEURAL NETWORKS

BY: RISHOV S. CHATTERJEE

May 15, 2019

Introduction

I have been exposed to Hindustani classical music ever since I was small. I was raised in an Indian household to immigrant parents and my mom is a vocalist and an exponent of Indian classical music. Indian classical music is categorized into two distinct forms: Hindustani and Carnatic, which are practiced in North and Southern India. Unlike western classical music, Indian classical music is very old form and typically doesn't have clear structures but largely depends on the performers or instrument players own elaboration of a melody. Indian classical music is defined by two basic elements – it must follow a Raga (classical mode), and a specific rhythm, the Taal (Sharma and Bali, 2015). Most compositions follow a Raga and I have noticed that even experts sometimes have difficulty of telling which Raga a particular song or composition is based on. This is particularly challenging for novices or beginners. Being a data science major, I quickly became attracted to this problem of "Raga detection". My intuition said that machine learning algorithms and techniques could help classify a composition into a main Raga on which it is based. Thus begins my journey to explore and hence this senior thesis.

I will mainly focus on North Indian form which is referred to as the Hindustani classical music. Compositions in Hindustani classical music also are based on a drone, i.e., a continual pitch that sounds throughout the concert, which is tonic (Gulati and Rao, 2011). This drone acts as a point of reference as the performer is expected to come back to this home base after a flight of improvisation. The variations and complexity in Hindustani music stems from its use of notes that comprise a Raga. There are seven main musical notes (also called swaras) – Sa, Re, Ga, Ma, Pa, Dha and Ni – along with five intermediate notes (flats and sharps) referred to as "vikrit swaras". The seven notes are referred to as Shuddha and belongs to the saptak (a scale). The flat notes are called "komal" and the sharp notes are called "teevra". A raga consists of at least five notes, and each raga provides the musician with a musical framework within which to improvise (Chordia and Ray, 2007; Shetty and Achary, 2009; Avtar, 1999). The specific notes within a raga can be reordered and improvised by the musician. Ragas range from small ragas like Bahar and Shahana that are not much more than songs to big ragas like Malkauns, Darbari and Yaman, which have great scope for improvisation and for which performances can last over an hour. Each raga traditionally has an emotional significance and symbolic associations such as with season, time and mood (Raaga, 2019). The raga is considered a means in Indian musical tradition to evoke certain feelings in an audience. Hundreds of raga are recognized in the classical tradition, of which about 30 are common (Raaga, 2019).

The swaras in a raga can be played in three octaves, the first or lower octave starting from 130 Hz, then middle octave starting at 260 Hz; and upper octave from 520 Hz. The artists are allowed to improvise over the definitions of raga to create their own renditions. If you listen to two performance of the same raga, they may sound strikingly different to novice ears, though they still retain the rules and defining qualities of ragas.

The rest of the thesis is organized as follows. In Chapter 2, we take a closer look at ragas to understand certain nuances and patterns they exhibit. In Chapter 3, we discuss Librosa, a python package for audio and music signal processing. In Chapter 4, we cover background and related work done on identifying Indian ragas using machine learning and other methods. Chapter 5 will go over the concepts associated with convolutional neural networks. Chapter 6 represents the methodology for raga classification using Convolutional Neural Networks (CNN). This chapter also discusses the dataset I have used and the image data generation preprocessing steps prior to CNN training. In Chapter 7, I present the results and analysis of this project. Finally, I conclude in Chapter 8 with future work for this proof of concept.

Indian Classical Ragas

Raga can be identified by various parameters. The particular choice of notes, Ascending and Descending sequences (known as arohana and avarohana pattern), nature of inflexion on different notes (gamaka/meend), characteristic phrases (pakad) all can be helpful to classify a raga (Raaga, 2019). These are further described below:

1. Choice of Notes

   A rāga has a given set of notes (swaras), on a scale, ordered in melodies with musical motifs. The Indian tradition suggests a certain sequencing of how the musician moves from note to note for each rāga, in order for the performance to create a rasa (mood, atmosphere, essence, inner feeling) that is unique to each rāga. Theoretically, thousands of rāga are possible given 5 or more notes, but in practical use, the classical tradition has refined and typically relies on several hundred. For most artists, their basic perfected repertoire has some forty to fifty rāgas (Hindustani, 2019; Chordia and Ray, 2007). Each raga has a different set of swaras that constitutes it. There must be the notes of the rag.

They are the allowd swar. This concept is similar to the Western solfege. There must also be a modal structure. This is called that in North Indian music and mela in Carnatic music. There is also the jati. Jati is the number of notes used in the rag. Rāga in Indian classic music is intimately related to tala or guidance about "division of time", with each unit called a matra (beat, and duration between beats) (Raaga, 2019). A rāga is not a tune, because the same rāga can yield an infinite number of tunes (Raaga, 2019). A rāga is not a scale, because many rāgas can be based on the same scale. Each raga tends to have a "Vadi" swara, a king swara on which maximum focus is given in a performance (Sharma and Bali, 2015). It is also known as the most frequently occurring swara in a particular raga. It is followed by Samvadi (next in importance), then Anuvadi. The swaras that are not allowed in a particular raga are known as Vivadi swaras (enemy notes).

2. Arohana/Avarohana

   There must also be the ascending and descending sequence of notes. This is called arohana /avarohana. Arohana and avarohana are the descriptions of how the raga moves. The arohana, also called aroh or arohi, is the pattern in which a raga ascends the scale. The avarohana, also called avaroh or avarohi, describes the way that the raga descends the scale. Both the arohana and avarohana may use certain

characteristic twists and turns.

3. Pakad

   The pakad or swarup, is a defining phrase or a characteristic pattern for a raga. This is often a particular way in which a raga moves; for instance the "Pa M'a Ga Ma Ga" is a tell-tale sign for Raga Bihag, or "Ni Re Ga M'a" is a telltale sign for Yaman. Often the pakad is a natural consequence of the notes of arohana / avarohana (ascending and descending structures). However, sometimes the pakad is unique and not implied by the notes of the arohana /avarohana. It is customary to enfold the pakad into the arohana / avarohana to make the ascending and descending structures more descriptive (Raaga, 2019).

4. Gamakas

   Gamakas are better known as ornamentations used in Hindustani music system. These are inflexions and rapid oscillatory movements taken across swaras (Sharma and Bali, 2015).

We now take one common raga as a running example and explain how the notes behave with respect to the above definitions and terms. Yaman emerged from the parent musical scale of Kalyan. Considered to be one of the most fundamental ragas in Hindustani tradition, it is thus often one of the first ragas taught to students. Yaman is a heptatonic (Sampurna) Indian classical raga of Kalyan Thaat. Yaman's Jati is a Sampurna raga.

Arohana: Sa Re Ga Ma(Kori Ma/tivra Ma i.e. Ma#) Pa Dha Ni Sa'

Avarohana: Sa' Ni Dha Pa Ma ((Kori Ma/tivra Ma i.e. Ma#)) Ga Re Sa

The ascending Aaroha scale and the descending style of the avroha includes all seven notes in the octave (When it is Shadav, the Aroha goes like N,RGmDNS' , where the fifth note is omitted; Pa but the Avaroha is the same complete octave). All the scale notes (called swaras) in the raga are Shuddha, the exception being Teevra Madhyam or prati madhyamam (Hindustani, 2019).

Vadi and Samavadi

Vadi is G (ga) and Samvadi is N (ni).

Pakad or Chalan

Kalyan has no specific phrases or particular features, many musicians avoid Sa and Pa in ascend or treat them very weakly. You will often hear N0 R G M+ D N S' in ascent and S' N D M+ G R S in descent). Sa is avoided in beginning the ascend such as N0 R G M+ P D N S'. To summarize the rules that make a raga:

A raga has at least 5 notes, including the Shadja. More than 5 is definitely ok. We can then add a further complexity, by choosing different swara in the ascent (Aaroha) and the descent (Avaroha). So the number of combinations can then be 5-5, 5-6,5-7,6-5,6-6,6-7,7-5,7-6 and 7-7. Five is

Odhav, six is Shadhav and 7 is Sampoorna. So, an odhav-sampoorna raga will have 5 swar in the aaroha and 7 in the awaroha.

Librosa: Processing Musical Information in Python

The research field of music information retrieval (MIR) has been evolving rapidly. Taken broadly, it covers the area of musicology, digital signal processing, machine learning, information retrieval and library science. As digital music service platforms such as iTunes, Spotify and Pandora have grown, so has been the need for MIR tools which to date has been largely written by programmers as scripts using C++ or MATLAB. In recent years, interest has grown within the MIR community in using (scientific) Python as a viable alternative (McFee et. al. 2015). LibROSA is a python package for music and audio analysis (Librosa, 2019). It provides the building blocks necessary to create music information retrieval systems. Here I provide a brief introduction to basic ideas and elements in libROSA as I have used this package in conducting bulk of the work in my thesis.

In general, librosa's functions tend to expose all relevant parameters to the caller. While this provides a great deal of flexibility to expert users, it can be overwhelming to novice users who simply need a consistent interface to process audio files. To satisfy both needs, they define a set of general

conventions and standardized default parameter values shared across many functions.

An audio signal is represented as a one-dimensional numpy array, denoted as y throughout librosa (McFee et. al. 2015). Typically the signal y is accompanied by the sampling rate (denoted sr) which denotes the frequency (in Hz) at which values of y are sampled. The duration of a signal (d) can then be computed by dividing the number of samples (y) by the sampling rate (sr):

$$d = \frac{y}{sr}$$

```
# duration_seconds.py
from librosa import load
import os
# Getting all mp3 files of raga Bhairav from dataset
fileList = os.listdir("data/Hindustani/mp3/Bhairav")
# Creating dictionary to store number of samples and sampling rate
sampleDict = {}
# y is number of samples, sr is sampling rate of the audio file
for file in fileList:
  y, sr = load(file)
  sampleDict[file].append(y)
  sampleDict[file].append(sr)
  # calculating signal duration
  duration = y / sr
```

```
sampleDict[file].append(duration)
```

By default, when loading stereo audio files, the librosa.load() function down mixes to mono by averaging left- and right-channels, and then resamples the monophonic signal to the default rate $sr = 22050Hz$. Most audio analysis methods operate not at the native sampling rate of the signal, but over small frames of the signal which are spaced by a hop length (in samples). The default frame and hop lengths are set to 2048 and 512 samples, respectively. At the default sampling rate of 22050 Hz, this corresponds to overlapping frames of approximately 93ms spaced by 23ms. Frames are centered by default, so frame index t corresponds to the slice:

```
y[(t*hop_length - frame_length/2):(t*hop_length+frame_length/2)]
```

where boundary conditions are handled by reflection padding the input signal y. For analyses that do not use fixed-width frames (such as the constant-Q transform), the default hop length of 512 is retained to facilitate alignment of results.

The majority of feature analyses implemented by librosa produce two-dimensional outputs stored as numpy.ndarray, e.g., S[f, t] might contain the energy within a particular frequency band f at frame index t. We follow the convention that the final dimension provides the index over time, e.g., S[:, 0], S[:,1] access features at the first and second frames. Feature arrays are organized column-major (Fortran style) in memory, so that common access

patterns benefit from cache locality. By default, all pitch-based analyses are assumed to be relative to a 12-bin equal-tempered chromatic scale with a reference tuning of A440 = 440.0 Hz. Pitch and pitch-class analyses are arranged such that the 0th bin corresponds to C for pitch class or C1 (32.7 Hz) for absolute pitch measurements.

Core Functionality

The librosa.core submodule includes a range of commonly used functions. Broadly, core functionality falls into four categories: audio and time-series operations, spectrogram calculation, time and frequency conversion, and pitch operations (McFee et. al. 2015). For convenience, all functions within the core submodule are aliased at the top level of the package hierarchy, e.g., "librosa.core.load" is aliased to "librosa.load".

Audio and time-series operations include functions such as: reading audio from disk via the audioread package (Librosa, 2019), resampling a signal at a desired rate, stereo to mono conversion, time-domain bounded auto-correlation, and zero-crossing detection. Spectrogram operations include the short-time Fourier transform (stft), inverse STFT (istft), and instantaneous frequency spectrogram (ifgram) (Abe et. al. 1995), which provide much of the core functionality for down-stream feature analysis. Additionally, an efficient constant-Q transform (cqt) implementation based upon the recursive down-sampling method of Schoerkhuber and Klapuri (Schoerkhuber and Klapuri, 2010) is provided, which produces logarithmically-spaced frequency

representations suitable for pitch-based signal analysis. Finally, logamplitude provides a flexible and robust implementation of log-amplitude scaling, which can be used to avoid numerical underflow and set an adaptive noise floor when converting from linear amplitude. Since data may be represented in a variety of time or frequency units, a comprehensive set of convenience functions is provided to map between different time representations: seconds, frames, or samples; and frequency representations: hertz, constant-Q basis index, Fourier basis index, Mel basis index, MIDI note number, or note in scientific pitch notation. Finally, the core submodule provides functionality to estimate the dominant frequency of STFT bins via parabolic interpolation (piptrack) (Smith, 2011), and estimation of tuning deviation (incents) from the frequency reference A440. These functions allow pitch-based analyses (e.g., cqt) to dynamically adapt filter banks to match the global tuning offset of a particular audio signal.

Spectral Features

Spectral representations are the distributions of energy over a set of frequencies which form the basis of many analysis techniques in MIR and digital signal processing in general. Librosa implements a variety of spectral representations, most of which are based upon the short time Fourier transform. The Mel frequency scale is commonly used to represent audio signals, as it provides a rough model of human frequency perception (McFee et. al. 2015). Both a Mel-scale spectrogram and the commonly used Mel-frequency

Cepstral Coefficients (MFCC) are provided. By default, Mel scales are defined to match the implementation provided by Slaney's auditory toolbox (McFee et. al. 2015), but they can be made to match the Hidden Markov Model Toolkit (HTK). While Mel scaled representations are commonly used to capture timbral aspects of music, they provide poor resolution o pitches and pitch classes. Pitch class (or chroma) representations are often used to encode harmony while suppressing variations in octave height, loudness, or timbre. Two flexible chroma implementations are provided: one uses a fixed-window STFT analysis (chroma _ stft) (Librosa, 2019) and the other uses variable-window constant-Q transform analysis (chroma_cqt). An alternative representation of pitch and harmony can be obtained by the tonnetz function, which estimates tonal centroids as coordinates in a six-dimensional interval space using the method of Harte (Harte et al.). Figures 1 and 2 illustrate the difference between STFT, Mel spectrogram, chromagram, and Tonnetz representations.

Display

The display module provides simple interfaces to visually render audio data through matplotlib (McFee, 2015). The first function, display.waveplot simply renders the amplitude envelope of an audio signal y using matplotlib's fill_between function. For efficiency purposes, the signal is dynamically downsampled.

Mono signals are rendered symmetrically about the horizontal axis;

stereo signals are rendered with the left-channel's amplitude above the axis and the right-channel's below. An example of waveplot is depicted in Figure 3. The second function, display.specshow wraps matplotlib's imshow function with default settings (origin and aspect) adapted to the expected defaults for visualizing spectrograms. Additionally, specshow dynamically selects appropriate colormaps (binary, sequential, or diverging) from the data type and range (McFee et. al. 2015). Finally, specshow provides a variety of acoustically relevant axis labeling and scaling parameters. Examples of specshow output are displayed in Figures 1, 2, and 3.

Figure 1: Raga Specshow Output for STFT and Mel-Spectrogram Representations in Librosa

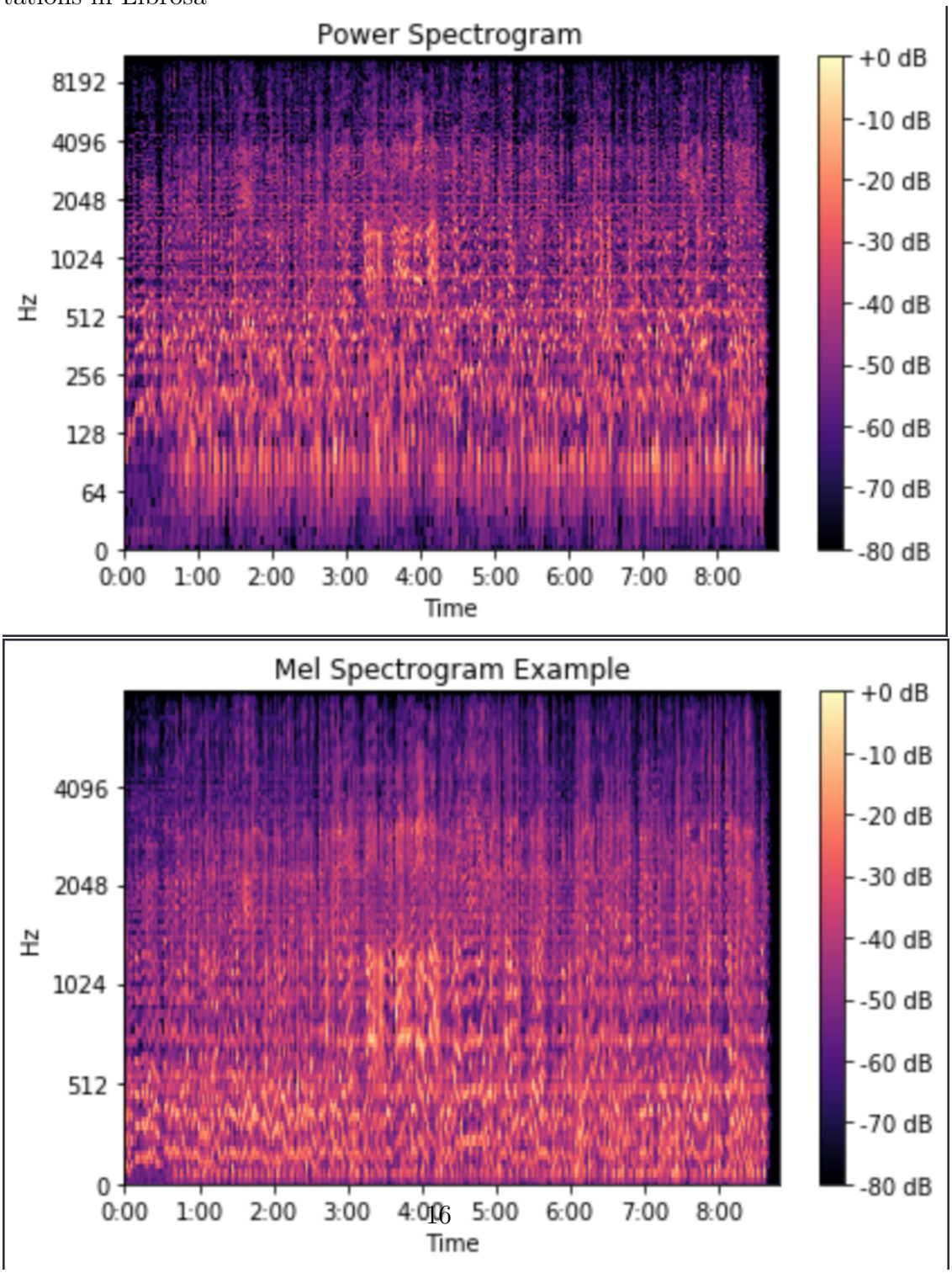
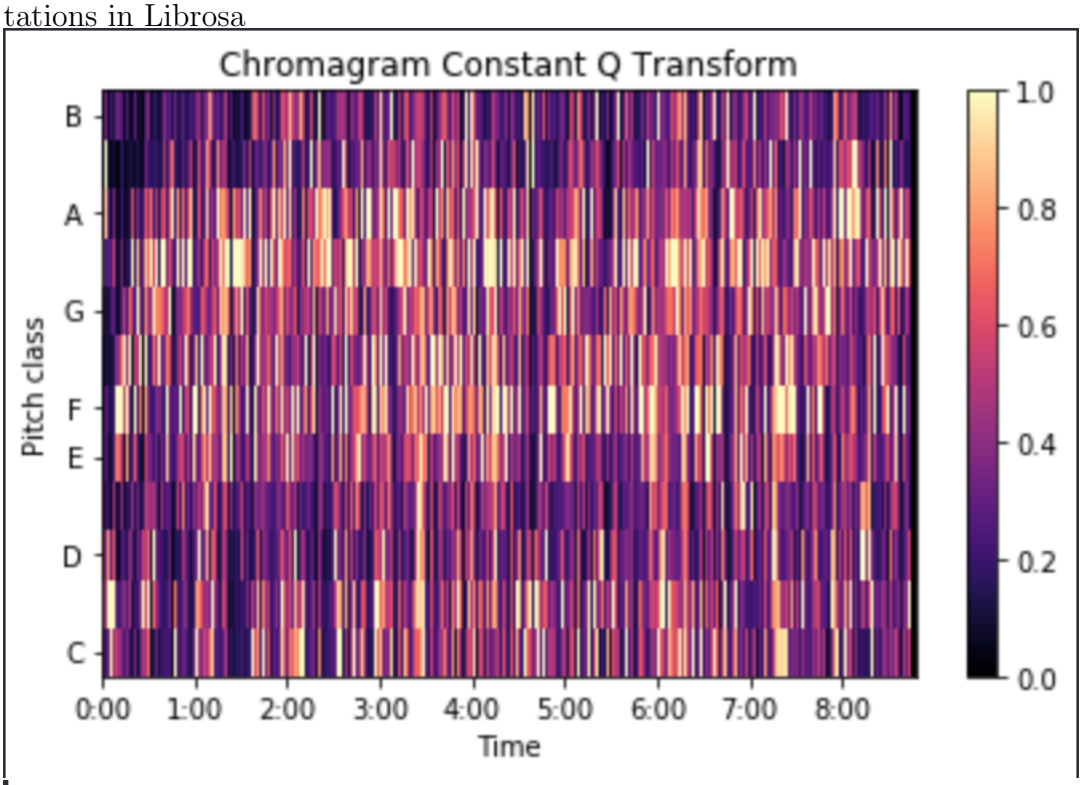


Figure 2: Raga Specshow Output for Chroma_CQT and Tonnetz Representations in Librosa



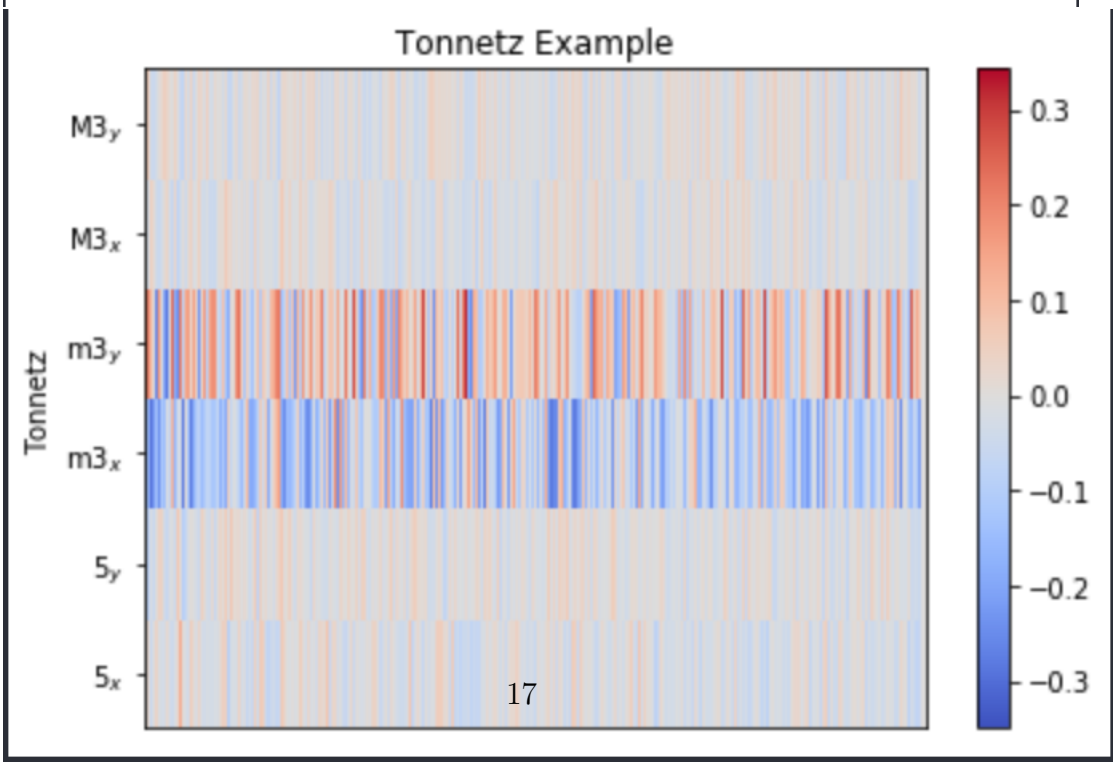
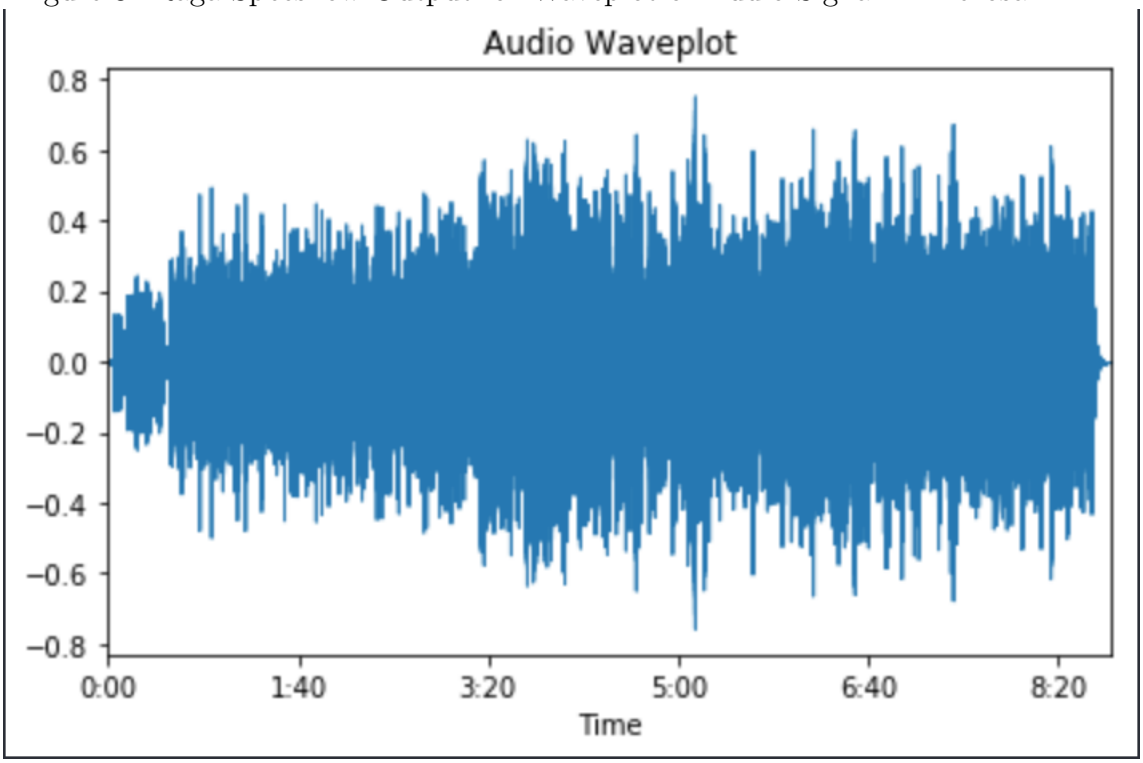


Figure 3: Raga Specshow Output for Waveplot of Audio Signal in Librosa



Background and Related Work

Scholars and researchers have used different mechanisms to identify ragas. In a system called Tansen, Pandey (Pandey, Mishra, and Ipe, 2003) created a system in which a raga is automatically identified based on Hidden markov model. Pendekar (Pendekar et. al. 2013) were able to identify a raga by segmentation of audio signal via spectral flux and thereby identifying raga by using its pitch frequency. Certain other researchers such as Chelpa (Chelpa, 1991) proposed a fuzzy set theory for generation of alap patterns. Sreedhar et. al. (Sreedhar and Gita, 2009), created a database of ragas and used the scale of the raga performance as a similarity metric using nearest neighbors algorithms. Within a scale, notes are matched with the existing sets of notes in the ragas in the database. The closest raga in the database is given as output for the test raga.

Tzanetakis (Tzanetakis and Cook, 2002) has also proposed various schemes in the English music classification based on their moods and styles of the performer as well as songs genre classification. Clustering is suggested as the classifier (Vaska, 2015). Sentiment analysis of movie review based on naïve Bayes and genetic algorithm is suggested in Govindarajan (Govin-

darajan, 2013). Since this methodology depends on the likelihood it can be connected to a wide assortment of spaces and results can be utilized as a part of numerous ways (Sharma, 2018). Shetty et. al., (Shetty and Achary, 2009) has identified raga based upon arohana-avorahana pattern on different ragas using neural network technique.

In a 2015 paper (Sharma and Bali, 2015), Sharma and Bali compared several ML classifiers to dataset of music labeled by 4 ragas: Des, Bhupali, Yaman and Todi. The audio performances are converted into .wav extension and chroma features are extracted using MIR toolbox in Matlab. A hop factor of 0.025 second is selected which gives 4719 frames. Then they were able to extract the Vadi swara, also known as the king swara whose magnitude and pitch are relatively greater than notes (swaras). Then they used the WEKA tool which includes a comprehensive collection of machine learning algorithms. The dataset of different ragas is classified using machine learning classifiers in WEKA. Classifiers they used are Random Forest, C4.5, Bayesian network and K-star. After performing comparison of classifiers on ragas, they observed that K-star gives the largest accuracy of 93.38%, on dataset of ragas followed by the random forest with 92.64%.

In another work, Kumar et. al. (Kumar et. al. 2014), proposed a method to identify the ragas of an Indian Carnatic music signal. They discuss why this problem is hard due to i) the absence of a fixed frequency for a note, ii) relative scale of notes, iii) oscillations around the note, and iv) improvisations. In this work, they framed the raga classification problem

in a non-linear SVM framework using a combination of two kernels that represent the similarities of a music signal using two different features pitch-class profile and n-gram distribution of notes. This differs from the previous pitch-class profile based approaches where the temporal information of notes is ignored. They evaluated the proposed approach on their own raga dataset and CompMusic dataset (CompMusic, 2019) and show an improvement of 10.19% by combining the information from two features relevant to Indian Carnatic music.

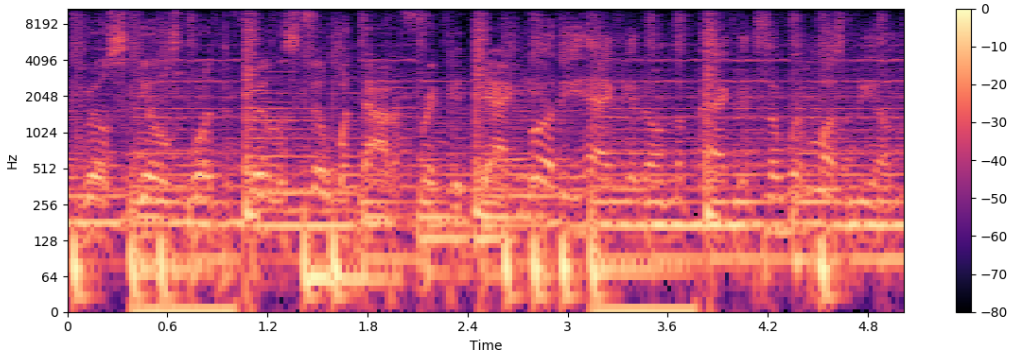
In a recent article, Ale Koretzky (Koretzky, 2019) provides valuable guidance on music signal processing that is helpful for the work that I have to do. Ale addresses the problem of how can we extract vocals out of a mixed track? This is formally known as Audio Source Separation. When we look at a audio source file (,m3 or ,wav), we are visualizing a waveform in the time domain. All we have access to are the amplitude values of the signal over time. While it is possible to extract features like envelopes, RMS values, zero-crossing rate etc., but these features aren't strong discriminators. To extract vocal content from a mix, we must somehow expose the structure of human speech. This is where the Short-Time Fourier Transform (STFT) comes to our rescue.

What the STFT plot shows us are:

i a fundamental frequency (f0) determined by the frequency of the vibration of our vocal cords.

The figure below shows Ariana (Koretzky, 2019) is singing in the 300-500

Figure 4: Short Term Fourier Transform of Vocal Audio Sample



Hz range.

ii A number of harmonics above f0 following some shape or pattern

iii no unvoiced speech, which includes consonants like ‘t’, ‘p,’, ‘k’, ‘s’, breaths, etc.

They manifest as short bursts in the high frequency.

The basic idea now is to figure out some sort of mask that when applied (element-wise multiplication) to the magnitude STFT mix gives us an approximate reconstruction pf the magnitude of STFT of the vocals. If this idea works, then we can begin to look at Convolutional Neural Networks (CNN) and what they have been able to achieve on images. Since STFT exhibits spatial patterns (in the time versus frequency space), CNNs should be able to learn from.

Using CNNs on spectral information extracted from librosa to learn patterns has not been explored much in prior work. This is the approach I

am going to take in the thesis. My research question becomes:

Can CNNs learn and classify ragas based on spectral features represented within chromagrams?

Before answering this question, it is first important to review what a convolutional neural network is as well as its underlying mechanics for recognizing images in the next chapter.

Convolutional Neural Networks

To understand Convolutional Neural Networks, we have to understand the following foundational ideas:

- Convolution
- Pooling
- Jargon: padding, stride, filter, etc

In computer vision, we have used diverse techniques in the past on images to do object detection and image classification. One major problem with computer vision problems is that the input data can get really big. Suppose an image is of the size 68 X 68 X 3. The input feature dimension then becomes 12,288. This will be even bigger if we have larger images (say, of size 720 X 720 X 3). Now, if we feed this big input to a neural network, the number of parameters will swell up to a very large number (depending on the number of hidden layers and hidden units). This will result in more computational and memory requirements – not something most of us can deal with.

Figure 5: Image with Many Edges

We begin by looking at edge detection as a simple example. The early layers of a neural network detect edges from an image (Sharma, 2018). Deeper layers might be able to detect the cause of the objects and even more deeper layers might detect the cause of complete objects (like a person's face).

Edge Detection Problem

In this section, we will focus on how the edges can be detected from an image. The examples provided here are adopted from an online tutorial (Sharma, 2018). Suppose we are given the figure below:

As you can see, there are many vertical and horizontal edges in the

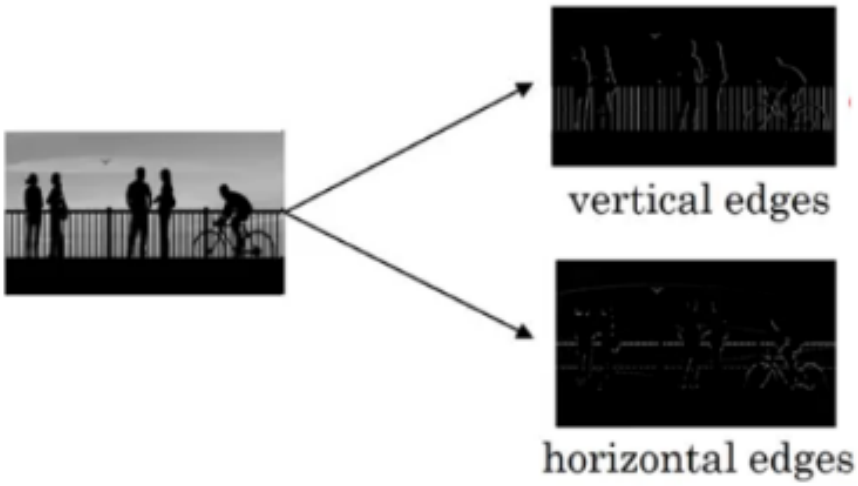


figure above. Therefore, the first thing to be done is to detect these edges.

How do we detect these edges? The first thing to do is assuming the image is BW, we represent it by a pixel map of gray scale values. Assume it is a 6 x 6 matrix.

The values in these cells show pixel values in grayscale. Next, we convolve this 6 X 6 matrix with a 3 X 3 filter. They are also sometime referred to as feature detector or kernel.

We take the feature filter and apply it over the original image. By placing it on the left upper corner, we get:

So, we take the first 3 X 3 matrix from the 6 X 6 image and multiply it with the filter. Now, the first element of the output (which is a 4 X 4 matrix) will be the sum of the element-wise product of these values, i.e. 3*1 + 0 + 1*-1 + 1*1 + 5*0 + 8*-1 + 2*1 + 7*0 + 2*-1 = -5. To calculate the

3	0	1	2	7	4
1	5	8	9	3	1
2	7	2	5	1	3
0	1	3	1	7	8
4	2	1	6	2	8
2	4	5	2	3	9

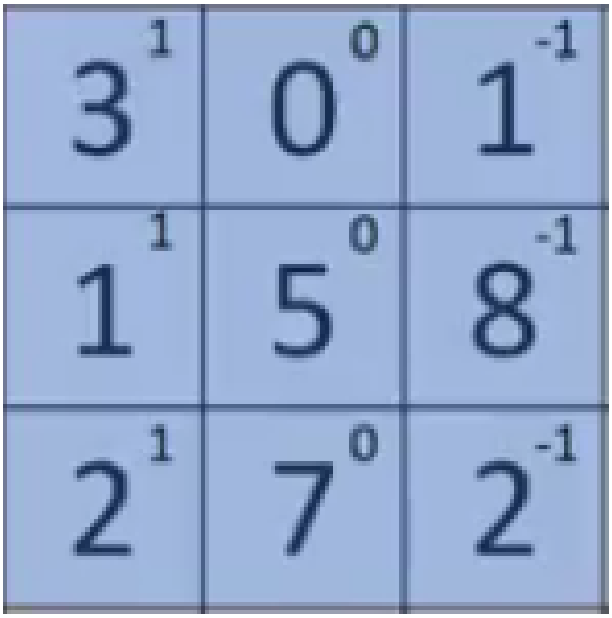
3	0	1	2	7	4
1	5	8	9	3	1
2	7	2	5	**1**	3
0	**1**	3	1	7	8
4	2	1	6	2	8
2	4	5	2	3	9

6 X 6 image

*

1	0	-1
1	0	-1
1	0	-1

3 X 3 filter



second element of the 4 X 4 output, we will shift our filter one step towards the right and again get the sum of the element-wise product:

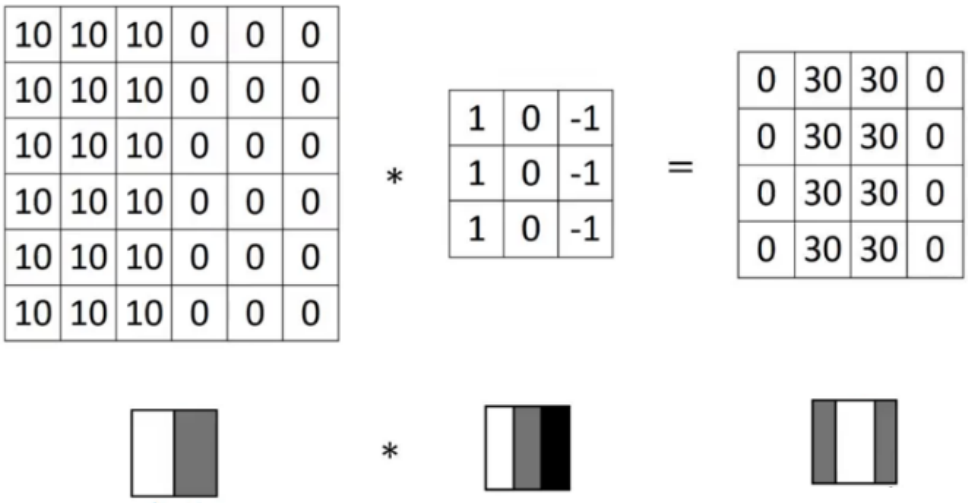
Similarly, we will convolve over the entire image and get a 4 X 4 output:

The shifting of the filter is also called a stride. A stride of 1 shifts it one cell to the right, while a stride of 2 will shift it 2 cells to the right. So, convolving a 6 X 6 input with a 3 X 3 filter gave us an output of 4 X 4. This output is sometimes referred to as feature map. While the above explains what convolve operation does, we will take another example which can clarify how edges can be detected. If high pixel values represent bright areas and low values represent darker areas of an image, then look at this following example.

Higher pixel values represent the brighter portion of the image and the

0^{1}	1^{0}	2^{-1}
5^{1}	8^{0}	9^{-1}
7^{1}	2^{0}	5^{-1}

-5	-4	0	8
-10	-2	2	3
0	-2	-4	-7
-3	-2	-3	-16



lower pixel values represent the darker portions. Looking at the 4 x 4 output matrix, we can detect a vertical edge in an image.

In mathematics, functional analysis, convolution is a mathematical operation on two functions (f and g) to produce a third function that expresses how the shape of one is modified by the other. The convolution of f and g is written f∗g, using an asterisk or star. It is defined as the integral of the product of the two functions after one is reversed and shifted. As such, it is a specific kind of integral transform:

$$[f * g](t) = \int_0^t f(\tau)g(t-\tau)\mathrm{d}\tau$$

Kernel Types

A number of well-known filters or kernels have been used in prior work (Sharma, 2018). Some common ones include Sharpen, Blue, Edge-Detect, or

Edge-Enhance.

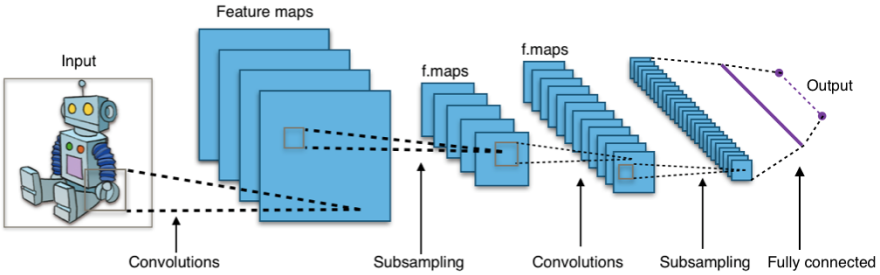
1	0	-1
2	0	-2
1	0	-1

Sobel filter

3	0	-3
10	0	-10
3	0	-3

Scharr filter

The Sobel filter puts a little bit more weight on the central pixels. Instead of using these filters, we can create our own as well and treat them as a parameter which the model will learn using backpropagation. In order to understand the next details of a CNN, let us take a look at the entire series of steps that happens on a single input image.



It is important to note that a single image can generate many feature maps using different kernels. We do this so that different features are extracted out. This first step creates a convolutional layer consisting of many feature maps.

Padding in Convolution

We have seen that convolving an input of 6 X 6 dimension with a 3 X 3 filter results in 4 X 4 feature map. We can generalize it and say that if the input is n X n and the filter size is f X f, then the output size will be (n-f+1) X (n-f+1):

- Input: n X n
- Filter size: f X f
- Output: (n-f+1) X (n-f+1)

There are two disadvantages to this approach:

1. Every time a convolution is applied, the size of the image shrinks.
2. Pixels present in the corner of the image are used only a few number of times during convolution as compared to the central pixels. Hence, not much focus is put on the corners since that can lead to a loss of information.

In order to overcome these issues, we can pad the image with an additional border, i.e., we add one pixel all around the edges. This means that the input will be an 8 X 8 matrix (instead of a 6 X 6 matrix). Applying convolution of 3 X 3 on it will result in a 6 X 6 matrix which is the original shape of the image.

This is where padding is very useful:

- Input: n X n

- Padding: p
- Filter size: f X f
- Output: (n+2p-f+1) X (n+2p-f+1)

There are two common choices for padding.

1. Valid: No padding at all. If we are using valid padding, the output will be (n-f+1) X (n-f+1)
2. Same: Here, we apply padding so that the output size is the same as the input size, i.e., n+2p-f+1 = n ;therefore, p = (f-1)/2

We now know how to use padded convolution. This way we don't lose a lot of information and the image does not shrink either. It is also important to note that the ReLU function (Rectifier Linear Function) is applied during the kernel application step since images are often very non-linear and we want to ensure that convolution does not create something too linear. Hence applying the ReLU rectifier function helps to solve this problem.

Pooling or Subsampling

Pooling layers are generally used to reduce the size of the inputs and hence speed up the computation. Consider a 4 X 4 matrix as shown below:

1	3	2	1
2	9	1	1
1	3	2	3
5	6	1	2

Applying max pooling on this matrix will result in a 2 X 2 output:

It is called max pooling as for every consecutive 2 X 2 block, we take the maximum number. Here, we have applied a filter of size 2 and a stride of 2. These are the hyperparameters for the pooling layer. Apart from max pooling, we can also apply average pooling where, instead of taking the max of the numbers, we take their average. In summary, the hyperparameters for a pooling layer are:

1. Filter size
2. Stride

3. Max or average pooling

If the input of the pooling layer is nh X nw X nc, then the output will be:

$$\frac{(n_h - f)}{s+1} \times \frac{(n_w - f)}{s+1} \times n_c$$

In fact what max pooling does is that it takes features from convolutional layer and tries to pool all the relevant features and discard those that doesn't help us in identification purposes.

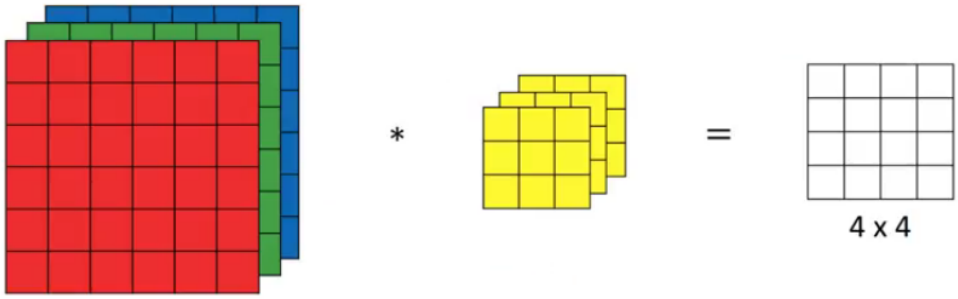
Convolution for Colored Images: A Third Dimension

So far we have taken a single grayscale image as input represented by n x m matrix in two dimensions. However in real world we often get colored or 3-dimensional images. How will we apply convolution on this image? We will use a 3 X 3 X 3 filter instead of a 3 X 3 filter. Let's look at an example:

- 6 X 6 X 3
- 3 X 3 X 3

Since there are three channels in the input, the filter will consequently also have three channels. After convolution, the output shape is a 4 X 4 matrix feature map. So, the first element of the output is the sum of the element-wise product of the first 27 values from the input (9 values from each channel) and the 27 values from the filter. After that we convolve over the entire image, seen in Figure 6.

Figure 6: 3D Convolution with a 3D Filter



As mentioned earlier, instead of using one kernel, we could use different kernels to extract different features. How do we do that? Let's say the first filter will detect vertical edges and the second filter will detect horizontal edges from the image. If we use multiple filters, the output dimension will change. So, instead of having a 4 X 4 output as in the above example, we would have a 4 X 4 X 2 output (if we have used 2 filters).

Generalized dimensions can be given as:

- Input: n X n X n_c
- Filter: f X f X n_c
- Padding: p
- Stride: s

The output of this convolution would be:

$$[\frac{(n+2p-f)}{s+1} \times \frac{(n+2p-f)}{s+1} \times n'_c]$$

In this output, n_c is the number of channels in the input and filter, while n'_c is the number of filters.

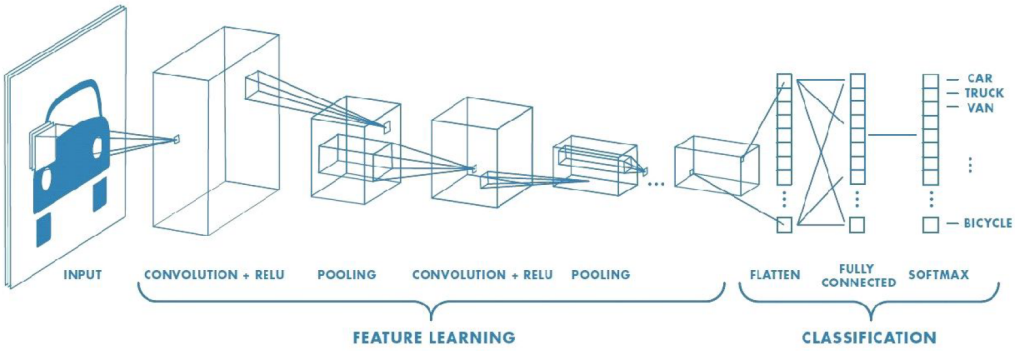
In a convolutional neural network, there are basically three types of layers:

1. Convolution layer
2. Pooling layer
3. Fully connected layer

Once we get an output after convolving over the entire image using a filter, we add a bias term to those outputs and finally apply an activation function to generate activations. This is one layer of a convolutional network . In our case, input image is (6 X 6 X 3) and filters are (3 X 3 X 3). These activations from layer 1 act as the input for layer 2, and so on. Clearly, the number of parameters in case of convolutional neural networks is independent of the size of the image. It essentially depends on the filter size. Suppose we have 10 filters, each of shape 3 X 3 X 3. It is easy to calculate the number of parameters:

- Number of parameters for each filter: $3^3 = 27$
- There will be a bias term for each filter, so total parameters per filter = 28
- Since there are 10 filters, total parameters will be 28*10 = 280

Figure 7: A Complete CNN Example



No matter how big the image is, the parameters only depend on the filter size. This is a powerful feature of CNNs.

From the convolutional layer, we can do max pooling to create subsampling layer and can again do multiple convolutions and max pooling until we get a reasonable number of output numbers which can be fed to an Artificial Neural Network to learn.

Here is what a full CNN would look like based on some of the examples we have used so far. In Figure 7, we also show notation for filter size, padding if used and the stride values.

There are a combination of convolution and pooling layers at the beginning, a few fully connected layers at the end and finally a softmax classifier to classify the input into various categories. There are a lot of hyperparameters in this network which we have to specify as well.

Generally, we take the set of hyperparameters which have been used in

proven research and they end up doing well. As seen in the above example, the height and width of the input shrinks as we go deeper into the network (from 32 X 32 to 5 X 5) and the number of channels increases (from 3 to 10).

There are a number of hyperparameters that we can tweak while building a convolutional network. These include the number of filters, size of filters, stride to be used, padding, etc. We will look at each of these in detail later in this article. Just keep in mind that as we go deeper into the network, the size of the image shrinks whereas the number of channels usually increases.

Methodology

Design of an End to End Raga Data Pipeline

For this thesis, I had to think of a way to get a dataset of images that I can then use for training a convolutional neural network architecture. In order to go about this process, I decided to first understand how I will look at getting audio files in which I know the raga associated with each file. I obtained the COMP-STAT Indian Raga Music Dataset (COMP-MUSIC, 2019) from Dr. Gulati. This contains both samples of Hindustani classical and Carnatic classical music. My preference was to work with the Hindustani classical music dataset. This dataset consisted of over 100 ragas and various mp3 files for each raga. Since my goal in my thesis is to show a proof of concept, I chose 4 ragas from this dataset: Bhairav, Bhup, Des, and Yaman. Within each of these ragas, there were multiple mp3 clips some of which were 8 minutes long and some were even 30 minutes long. After getting this data, I converted each audio file into chromagrams with a short term Fourier transform in order to get spectral information as well as differences in pitch over the duration of the audio file. In order to get multiple chromagrams for each audio file, I decided to generate a chromagram for each minute of the

audio file's duration. To make this possible, I had to use an advanced IO feature in Librosa known as blockwise reading. Since Librosa did not have blockwise reading on its own, I had to make use of another library in tandem with Librosa for making this possible. The name of this second library is known as PySoundFile which is a Python wrapper for SoundFile which makes blockwise-reading possible as well as a functionality for metadata information regarding any input audio file as long as it in a .wav format. This premise presented a challenge for me as it required a conversion from a compressed .mp3 format of medium quality into .wav to input files as arguments into SoundFile's functional interface.

Converting from MP3 to WAV

Since MP3 files are a compressed, digital format of audio data based on the bit rate measured in kilobits per second, I first decided to look into those values surrounding the mp3 quality from my initial dataset of mp3 files. Since the dataset is based on medium-quality mp3 files, I deduced that the bit rate, x, will be a range bounded between $(128 < x < 320)$ kbps depending on the extracted mp3 files from the dataset's architect. Since I do not have control over the variance in bit rates nor time to generate a uniform dataset of mp3 files, I decided to go straight into the conversion of the mp3 format to the wav format. In order to do this, I made use of a Python module known as scikit-sound. Scikit-sound (Scikit-Sound, 2019) is a very useful utility as it makes it possible to read compressed audio file formats and encode them into wav

format by making use of a third party utility known as FFMPEG. FFMPEG has a sampling algorithm to scale various audio formats based on the type of audio conversion that needs to be done. For the case of scaling from mp3 to wav in which the file size is much greater, FFMPEG samples at a standard sampling rate based on a default encoding from mp3 to wav. The encoding is not readily observable by the end user. FFMPEG is considered as the engine within scikit-sound's Sound class. From the software point of view, a simple instantiation of the Sound class with the input being the mp3 file will automatically read a wav file of the same file name while disregarding the .mp3 extension. Fortunately, FFMPEG is available across many operating systems and so I was able to install it using the Homebrew package manager within my Macbook. The uniform sampling rate from FFMPEG becomes a huge convenience when going into block reading which will be discussed more specifically in the next section.

```
# An instance of Scikit-Sound's Sound Class
# Python 3.6+
from sksound.sounds import Sound
import os


fileList = os.listdir("/path to raga audio files in mp3 format")
# dictionary to store each instantiated response
fileDict = {}
# iterating through every file
```

```
for file in fileList:
  fileDict[file] = Sound(f"path/{file}")
```

Each wav file will be in the same directory as the path which contains the mp3 files. To alleviate this challenge, I created a directory structure as follows:

Once the wav files finished generating for each specific raag, I ran the following bash commands in my Terminal to move all the wav files into the wav directory.

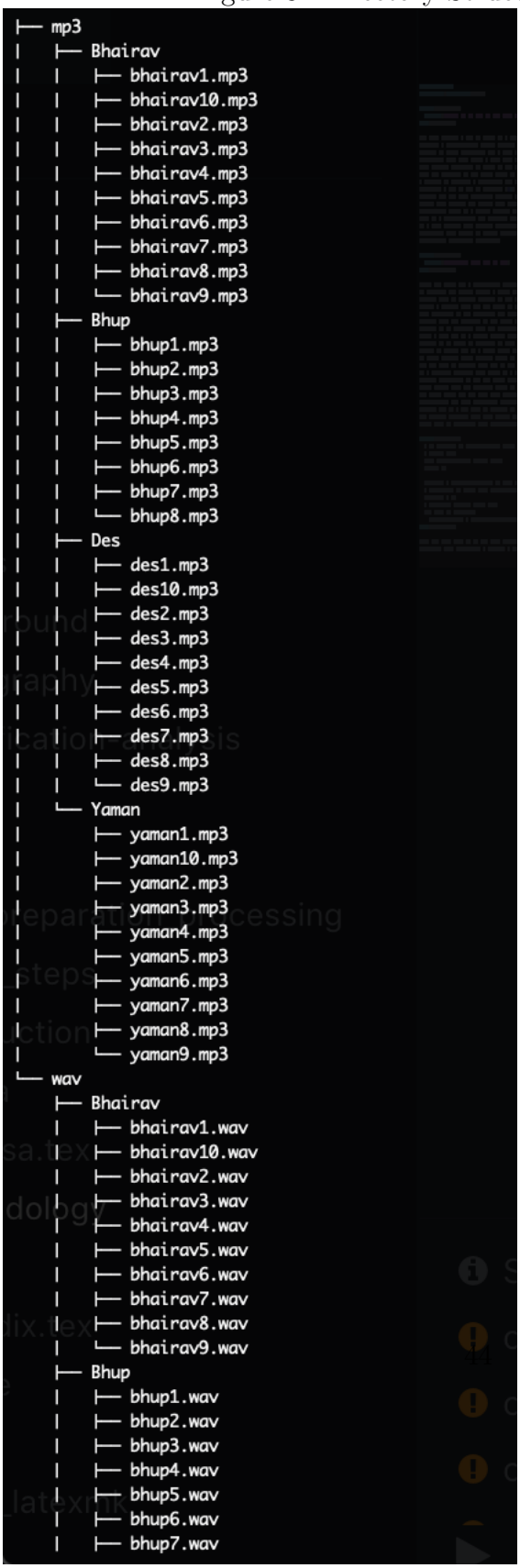
```
cd data/Hindustani/mp3/Bhairav && mv *.wav ../../wav/Bhairav
cd data/Hindustani/mp3/Bhup && mv *.wav ../../wav/Bhup
cd data/Hindustani/mp3/Des && mv *.wav ../../wav/Des
cd data/Hindustani/mp3/Yaman && mv *.wav ../../wav/Yaman
```

I also want to note that in order to perform proper version controlling of all my code commits, I tracked all mp3 and wav files using git-lfs so they would be known to Git and my GitHub repository as large files and will therefore be written to GitHub upon pushing commits with a higher file writing rate than that of normally pushed files.

Block-Wise Reading with PySoundFile

As mentioned before, I wanted to read the audio files as blocks of a certain size in order to generate stft-chromagrams for each minute in every audio file. To calculate the block size, b, I used the following equation in

Figure 8: Directory Structure for Audio Files



which sr refers to the sampling rate per second and n refers to the number of seconds:

$$b = sr * n$$

To get the specific sampling rate which FFMPEG used for sampling, I used an attribute of PySoundFile.

```
# importing PySoundFile
import soundfile as sf
import os
bhairavList = os.listdir("data/Hindustani/wav/Bhairav")
for file in bhairavList:
  rate = sf.info(f"data/Hindustani/wav/Bhairav/{file}").samplerate
  print(rate)
```

The sampling rate that FFMPEG used for converting from mp3 to wav for the audio files corresponding to each raga is 44100 Hz. This means that in order to get a chromagram for each minute in the audio file, n would have to be equal to 60 since there are 60 seconds in a minute.

My uniform blocksize computes to:

$$b = 44100 * 60 = 2646000$$

After getting this value, reading in audio files into Librosa in tandem

with PySoundFile's functionality became a very trivial matter.

```
import numpy as np
# importing soundfile and librosa
!pip install pysoundfile
import soundfile as sf
from librosa.feature import chroma_stft
from librosa.display import specshow
import matplotlib.pyplot as plt
# Bhairav Block Wise Reading


## Bhairav 1


block_gen = sf.blocks('data/Hindustani/wav/Bhairav/bhairav1.wav',
    blocksize=2646000)
rate =
    sf.info("data/Hindustani/wav/Bhairav/bhairav1.wav").samplerate
info = sf.info("data/Hindustani/wav/Bhairav/bhairav1.wav")
print(info)
chromas = []


for bl in block_gen:
    y = np.mean(bl, axis=1)
    chromas.append(chroma_stft(y, sr=rate))
```

```
len(chromas)
for j, chroma in enumerate(chromas):
    specshow(chroma, x_axis="time", y_axis="chroma", vmin=0, vmax=1)
    plt.title(f"Chromagram of Bhairav1_{j}")
    plt.savefig(f"data/chroma_files/bhairav-chromas/bhairav1/bhairav1_{j}.png")
```

The above script allowed me to automate the generation of stft-chromagrams for every one minute block in the first audio recording of raga Bhairav. After thinking about the iterative process, I managed to build a more powerful automation script that would be able to generate chromagrams for every block for every audio file for a specific raga. An example of this process is shown below to generate stft-chromagrams for all the audio files of the dataset I used for raga, Bhup.

```
import os
import numpy as np
import soundfile as sf
from librosa.feature import chroma_stft
from librosa.display import specshow
import matplotlib.pyplot as plt

bhup_files = os.listdir("data/Hindustani/wav/Bhup")
print(bhup_files)

# Generating directories to denote each bhup audio file an
```

```
    iterative path

for h,i in enumerate(bhup_files):
    os.system(f"mkdir data/chroma_files/bhup-chromas/bhup{h+1}")

# Creating an empty dictionary to store

chroma_dict = {}

for j in range(len(bhup_files)):

    # building blocks for each file pertaining to the Bhup raga

    rate =
        sf.info(f"data/Hindustani/wav/Bhup/bhup{j+1}.wav").samplerate
    block_gen =
        sf.blocks(f"data/Hindustani/wav/Bhup/bhup{j+1}.wav",
        blocksize=rate*60)
    chroma_dict[f"bhup{j+1}"] = []

    # Computing the chroma_stft for every block in each Bhup audio
        file

    for bl in block_gen:
```

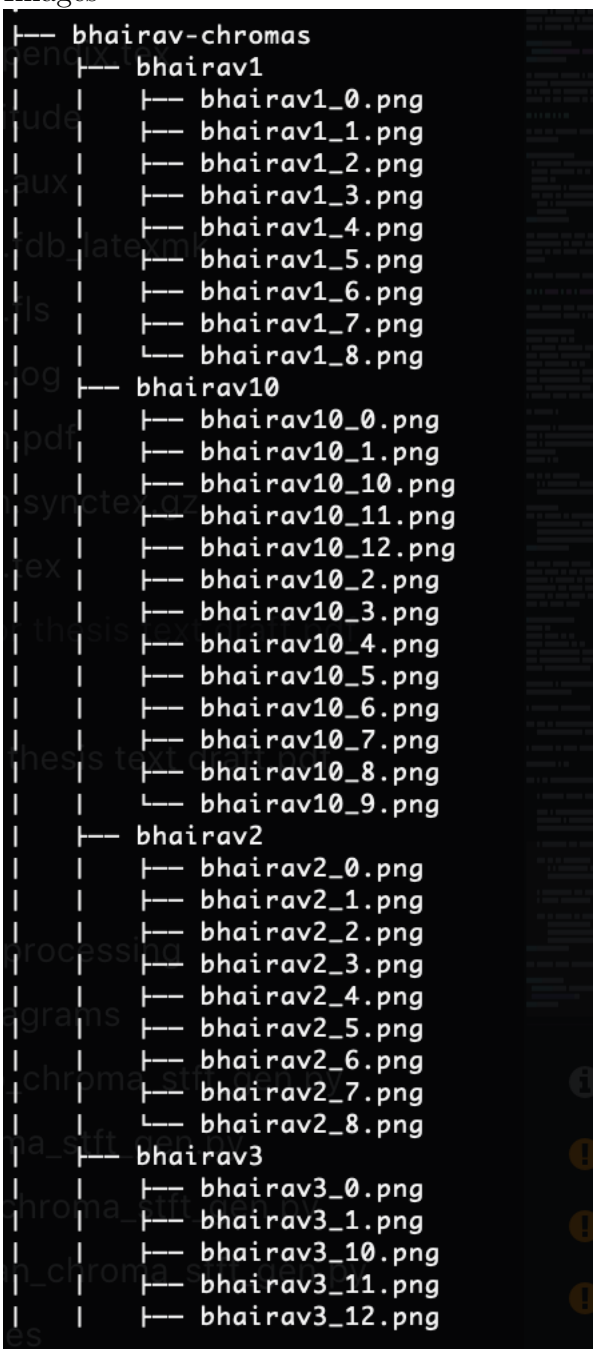
```
        y = np.mean(bl, axis=1)
        chroma_dict[f"bhup{j+1}"].append(chroma_stft(y, sr=rate))

    # Visualizing and saving the stft-chromagram for every block
        for every Bhup audio file
    # naming convention: bhup[file enumeration]_[block number]

    for k, chroma in enumerate(chroma_dict[f"bhup{j+1}"]):
        specshow(chroma, x_axis="time", y_axis="chroma", vmin=0,
            vmax=1)
        plt.title(f"Chromagram of Bhup{j+1}_{k+1}")
        plt.savefig(f"data/chroma_files/bhup-chromas/bhup{j+1}/bhup{j+1}_{k+1}.png")
```

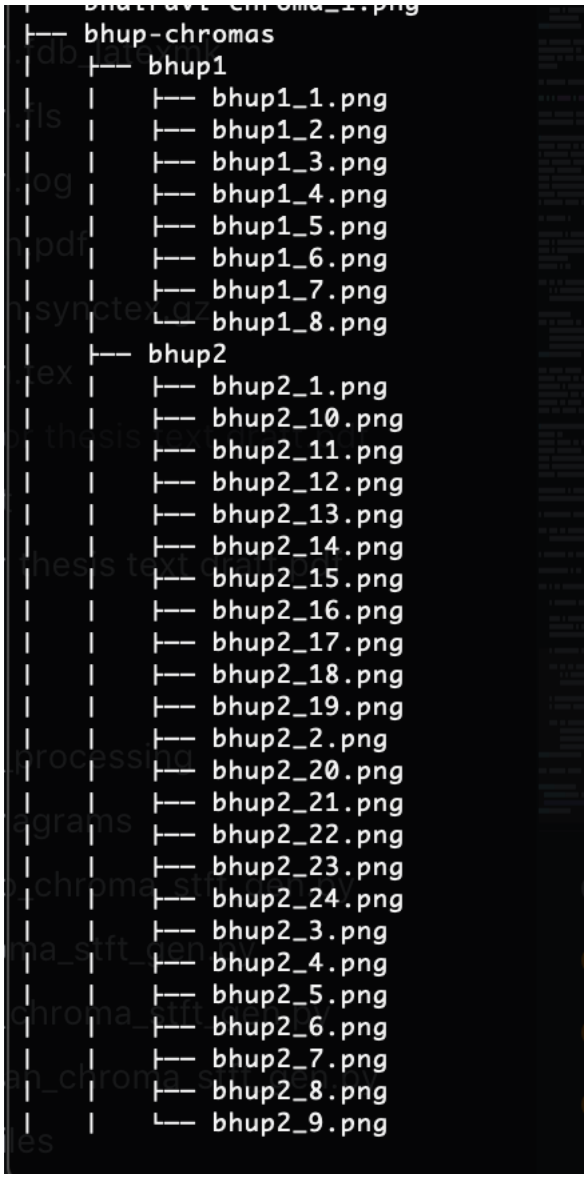
The figure below illustrates the file structure for the generated chromagrams as png files.

Data Preparation for CNN Training

In order to have the images ready for a machine learning experiment with a convolutional neural network, I first needed to generate a csv file of labels corresponding to the raga of each recording. Since I had an original file structure comprising of all the images of all the ragas, building up labels for all the files became very trivial and the ragas were all in alphabetical order. The script below shows the process I used in pandas to build the miml_labels_2 csv file.

Figure 9: File Structure for Stored Bhup and Bhairav STFT-Chromagram Images





```
import pandas as pd
import os

# Empty dictionary

columnDict = {}

# Getting all image files in one Python list

fileList = os.listdir("data")
print(len(fileList))

columnDict['file_name'] = fileList

print(columnDict)

# Empty list to store labels

labels = []

# Range values calculated beforehand

for i in range(202):
  labels.append("bhairav")
```

```
for i in range(183):
  labels.append("bhup")

for i in range(137):
  labels.append("des")

for i in range(258):
  labels.append("yaman")

print(len(labels))

# Generating second column to store labels

columnDict['raga'] = labels

# Converting from a dictionary to a Pandas DataFrame

df = pd.DataFrame.from_dict(columnDict)

# Saving DataFrame as a Comma Separated Value (csv) file

df.to_csv("csv/miml_labels_2.csv")
```

CNN Preprocessing and Training with Tensorflow 2.0 and Keras on Google Colab

Now that I had my CSV file and image data, I moved over from my local environment to Google Colaboratory in order to make use of a free tier graphics processing unit (GPU) optimized for neural network training with ease offered by the Google Compute Engine (GCE) with a memory allocation of 12 GB RAM.

In order to make this transition simple, I created another GitHub repository which contains 780 stft-chromagram images as well as the csv file pertaining to labels that correspond in order of the files. Since Git and several popular Python modules are already pre-installed in Google Colaboratory's Jupyter notebook interface, I easily cloned my repository into Colaboratory and changed the notebook's base directory to the repository's directory for ease of file reading. The next step is to build a convolutional neural network architecture.

The resolution for each stft-chromagram is 432 X 288 pixels therefore I decided to add in more than just two convolutional layers in the architecture. The architecture I decided to use is 2 convolutional layers, a max pooling layer, another set of two convolutional layers, and then another max pooling layer and then a connection from the second max pooling layer to the hidden layer of 512 neurons and then a final connection from the hidden layer to an output layer of 4 neurons where the classification decision is made by the softmax activation function. All other layers have a rectified linear unit

activation function. The loss function used was categorical_crossentropy and optimization was performed using the Adam optimizer. This architecture is shown in Tensorflow's 2.0 Keras module below.

```
import tensorflow
# Importing Tensorflow's keras wrapper
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten,
    Dense, Dropout, Activation, BatchNormalization
from tensorflow.keras import regularizers, optimizers


classifier = Sequential()
classifier.add(Conv2D(32, (3, 3), padding='same', input_shape =
    (432, 288, 3), activation = 'relu'))
classifier.add(Conv2D(32, (3, 3), activation = 'relu'))
classifier.add(MaxPooling2D(pool_size = (2, 2)))
classifier.add(Dropout(rate=1-0.25))
classifier.add(Conv2D(64,(3,3), padding="same", activation="relu"))
classifier.add(Conv2D(64, (3,3), activation="relu"))
classifier.add(MaxPooling2D(pool_size=(2,2)))
classifier.add(Dropout(rate=1-0.25))
classifier.add(Flatten())
classifier.add(Dense(units = 512, activation = 'relu'))
classifier.add(Dropout(rate=1-0.5))
classifier.add(Dense(units = 4, activation = 'softmax'))
```

```
classifier.compile(optimizer = 'adam', loss =
    'categorical_crossentropy', metrics = ['accuracy'])
```

In order to separate the image data into training, testing, and validation sets, I made use of the image submodule in Keras's preprocessing module by instantiating the ImageDataGenerator class for the selection of images into training and testing sets. The first 680 images were selected for training, 50 images after the first 680 were selected for validation, and the last 50 images were chosen for testing. The script below shows this class instantiation with a rescaling of every pixel value.

```
import pandas as pd
df = pd.read_csv("csv/miml_labels_2.csv")
df.drop(list(df.columns)[0], axis=1, inplace=True)
df.head()
df["raga"]=df["raga"].apply(lambda x:x.split(","))
df.head()

train_generator=datagen.flow_from_dataframe(
dataframe=df[:680],
directory="./data",
x_col="file_name",
y_col="raga",
batch_size=32,
seed=42,
```

```
shuffle=True,
class_mode="categorical",
classes=["bhairav", "bhup", "des", "yaman"],
target_size=(432,288))

valid_generator=test_datagen.flow_from_dataframe(
dataframe=df[680:730],
directory="./data",
x_col="file_name",
y_col="raga",
batch_size=32,
seed=42,
shuffle=True,
class_mode="categorical",
classes=["bhairav", "bhup", "des", "yaman"],
target_size=(432,288))

test_generator=test_datagen.flow_from_dataframe(
dataframe=df[730:],
directory="./data",
x_col="file_name",
batch_size=1,
seed=42,
shuffle=False,
```

```
class_mode=None,
target_size=(432,288))
```

The training started after running the script below which uses the fit_generator method of the CNN in Keras with a specified step size for training and validation per epoch for a total of 10 epochs.

```
STEP_SIZE_TRAIN=train_generator.n//train_generator.batch_size
STEP_SIZE_VALID=valid_generator.n//valid_generator.batch_size
STEP_SIZE_TEST=test_generator.n//test_generator.batch_size
classifier.fit_generator(generator=train_generator,
                steps_per_epoch=STEP_SIZE_TRAIN,
                validation_data=valid_generator,
                validation_steps=STEP_SIZE_VALID,
                epochs=10
)
```

Results and an analysis of the machine learning aspect will be discussed in the next chapter.

Results and Analysis

Results

After running the fit function from the previous section, the results for the validation set came out with an accuracy of 94% after 10 epochs.

Figure 10: Validation Accuracy

```
Epoch 7/10
2/2 [==============================] - 0s 200ms/step - loss: 1.3719 - acc: 0.3000
22/22 [==============================] - 10s 449ms/step - loss: 1.1649 - acc: 0.4809 - val_loss:
1.3719 - val_acc: 0.3000
Epoch 8/10
2/2 [==============================] - 0s 202ms/step - loss: 1.0635 - acc: 0.8000
22/22 [==============================] - 10s 453ms/step - loss: 1.0269 - acc: 0.5721 - val_loss:
1.0635 - val_acc: 0.8000
Epoch 9/10
2/2 [==============================] - 0s 214ms/step - loss: 0.9029 - acc: 0.7200
22/22 [==============================] - 10s 454ms/step - loss: 0.9307 - acc: 0.6176 - val_loss:
0.9029 - val_acc: 0.7200
Epoch 10/10
2/2 [==============================] - 0s 202ms/step - loss: 0.7102 - acc: 0.9400
22/22 [==============================] - 10s 452ms/step - loss: 0.7544 - acc: 0.7103 - val_loss:
0.7102 - val_acc: 0.9400
```

Future Steps

Bibliography