Table of Contents

Chapter 1 – Introduction

Chapter 2 – Closer look at raga

Chapter 3 – Librosa – a python package for music and audio signal processing

Chapter 4 – Background and Related Work

Chapter 5 – Convolutional Neural Networks

Chapter 6 – Our Approach

Chapter 7 – The dataset, our CNN algorithm and Classification Work

Chapter 8 – Results and Findings

Chapter 9 – Conclusions and Future Work

References

Chapter 1 – 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 [1]. 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 begun 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 [2]. 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 [3, 4 5]. 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 [6]. 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 [7].

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. In Chapter 5, we present our approach. Chapter 6, discusses advances in Convolutional Neural Network (CNNs). Chapter 7 presents the details of the dataset we use, our CNN algorithms and our raga detection work we have done. In Chapter 8 we present the results and our findings. Finally we conclude in Chapter 9 with a summary of our findings and possible future work.

Chapter 2 – Raga a closer look

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. 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* [8 -10]. 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) [11]. A *rāga* is not a tune, because the same *rāga* can yield an infinite number of tunes [12]. 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 [1]. 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.

4. Gamakas: Gamakas are better known as ornamentations used in Hindustani music system. These are inflexions and rapid oscillatory movements taken across swaras.

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.

Vadi and Samavadi

Vadi is G (ga), 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 often hear N^0 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 N^0 R G M^+ P D N S'

To summarize the rules that make a raga:

Rule: 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.

Chapter 3 – Librosa – a python package for music and audio signal processing

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 [ref]. LibROSA is a python package for music and audio analysis. 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. 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 can then be computed by dividing the number of samples by the sampling rate:

>>> *duration_seconds = float (len(y)) /sr*

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.

3.1 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 [ref]. 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 [ref] (*core.load*), resampling a signal at a desired rate (*core.resample*), stereo to mono conversion (*core.to_mono*), time-domain bounded auto-correlation (*core.autocorrelate*), and zero-crossing detection (*core.zero_crossings*).

Spectrogram operations include the short-time Fourier transform (stft), inverse STFT (istft), and instantaneous frequency spectrogram (ifgram) [Abe95], 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 [ref] 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.

Because data may be represented in a variety of time or frequency units, we provide a comprehensive set of convenience functions 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*) [Smith11], and estimation of tuning deviation (incents) from the 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.

3.2 Spectral features

Spectral representations—the distributions of energy over a set of frequencies—form the basis of many analysis techniques in MIR and digital signal processing in general. I use this representation to convert music samples of Indian ragas into energy images which can then later be used to train CNNs. The *librosa.feature* module 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 [ref]. Both a Mel-scale spectrogram (*librosa.feature.melspectrogram*) and the commonly used Mel-frequency Cepstral Coefficients (MFCC) (*librosa.feature.mfcc*) are provided. By default, Mel scales are defined to match the implementation provided by Slaney's auditory toolbox [ref], but they can be made to match the Hidden Markov Model Toolkit (HTK) by setting the flag *htk=True* [ref].

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*) [ref] 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 etal. [ref]. Figure 1 illustrates the difference between STFT, Mel spectrogram, chromagram, and Tonnetz representations.

3.3 Display

The display module provides simple interfaces to visually render audio data through *matplotlib* [ref]. 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 down-sampled.

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 2 (top).

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 [ref]. Finally, specshow provides a variety of acoustically relevant axis labeling and scaling parameters. Examples of specshow output are displayed in Figures 1 and 2 (middle).

--- insert Figure 1 STFT, Mel spectrogram, chromagram, and Tonnetz here ----

---- insert Figure 2 waveplot here ----

Chapter 4 – Background and Related Work

Scholars and researchers have used different mechanisms to identify ragas. In a system called Tansen, Pandey [ref] created a system in which a raga is automatically identified based on Hidden markov model. Pendekar et. al., [ref] 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 [ref] proposed a fuzzy set theory for generation of alap patterns. In [ref], Sreedhar et. al., 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 [ref] 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 [ref]. Sentiment analysis of movie review based on naïve Bayes and genetic algorithm is suggested in [ref]. 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 [ref]. Shetty et. al., [ref] has identified raga based upon arohana-avorahana pattern on different ragas using neural network technique.

In a 2015 paper [ref], 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., 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 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 [ref] 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.

--- insert a picture of STFT of vocals ---

What the STFT plot shows us are i) a fundamental frequency (f0) determined by the frequency of vibration of our vocal cords. The above sample shows Ariana is singing in the 300-500 Hz range. Ii) A

number of harmonics above f0 following some shape or pattern, iii0 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 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 STFT waveforms or Chroma waveforms to learn patterns hasn’t been explored much in prior work. This is the approach I am going to take in the thesis. My research question becomes:

RQ: Can CNNs learn and classify ragas based on STFT and Chroma waveplots with enough training sample data fed to it? If so what kind of accuracy can we get in raga identification?

In the next chapter, we review what is a CNN and how a CNN works.

Chapter 5 – Convolutional Neural Networks

To understand CNNs, we have to understand the following foundational ideas:

- The convolution operation
- The pooling operation
- Certain vocabulary used in convolutional neural networks (padding, stride, filter, etc.)
- Building a convolutional neural network for multi-class classification in images

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.

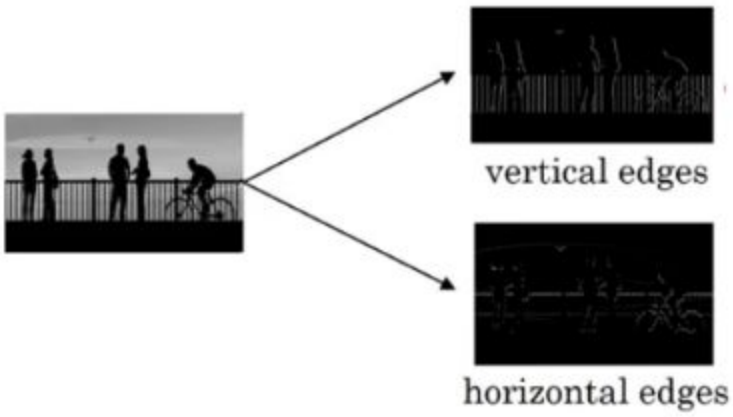
We begin by looking at edge detection as a simple example. The early layers of a neural network detect edges from an image. Deeper layers might be able to detect the cause of the objects and even more deeper layers might detect the cause of complete objects (like a person's face).

Edge detection problem:

In this section, we will focus on how the edges can be detected from an image. Suppose we are given the below image:

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

The first thing to do 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.

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

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.

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

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 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:

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

The shifting of the filter is also called 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.

10	10	10	0	0	0
10	10	10	0	0	0
10	10	10	0	0	0
10	10	10	0	0	0
10	10	10	0	0	0
10	10	10	0	0	0

*

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

=

0	30	30	0
0	30	30	0
0	30	30	0
0	30	30	0

Higher pixel values represent the brighter portion of the image and the 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 (in particular, 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 particular kind of integral transform:

$$[f*g](t) \equiv \int_0^t f(\tau)\, g(t-\tau)\, d\tau,$$

Kernel Types:

A number of well-known filters or kernels have been used in prior work. 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.

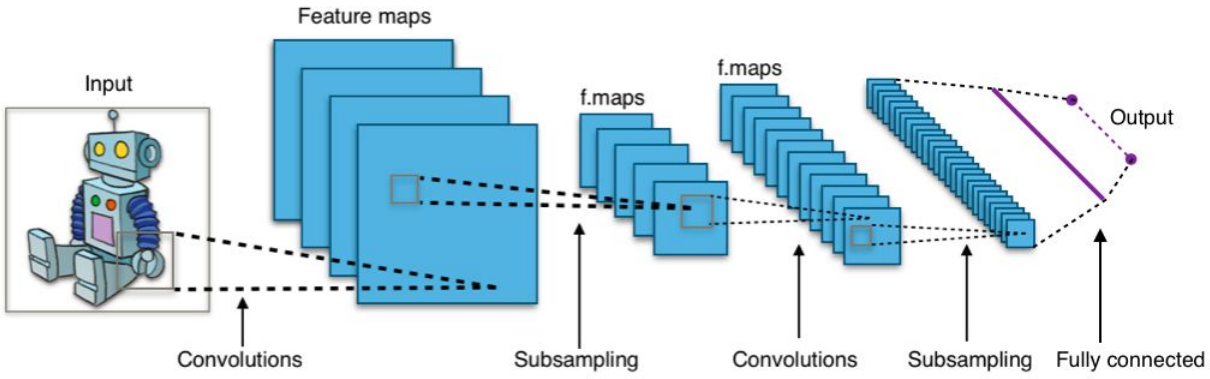


Figure: A Typical CNN

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 Convolutional Step

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 primarily two disadvantages here:

1. Every time we apply a convolutional operation, 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, we do not focus too much on the corners since that can lead to information loss

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 comes to the help:

- **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:** It means no padding. 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
   So, 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.

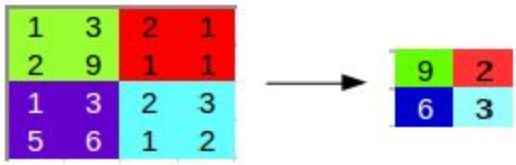
It is also important to note that the ReLU function (Rectifier Linear Function) is applied during the kernel application step since images are often very non-linear and we want to ensure that convolution does not create something too linear. Hence applying the ReLU rectifier function helps to solve this 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 max 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 n_h X n_w X n_c, then the output will be [{(n_h – f) / s + 1} X {(n_w – f) / s + 1} X 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.

How to deal with color or 3D images?

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:

- **Input:** 6 X 6 X 3
- **Filter:** 3 X 3 X 3

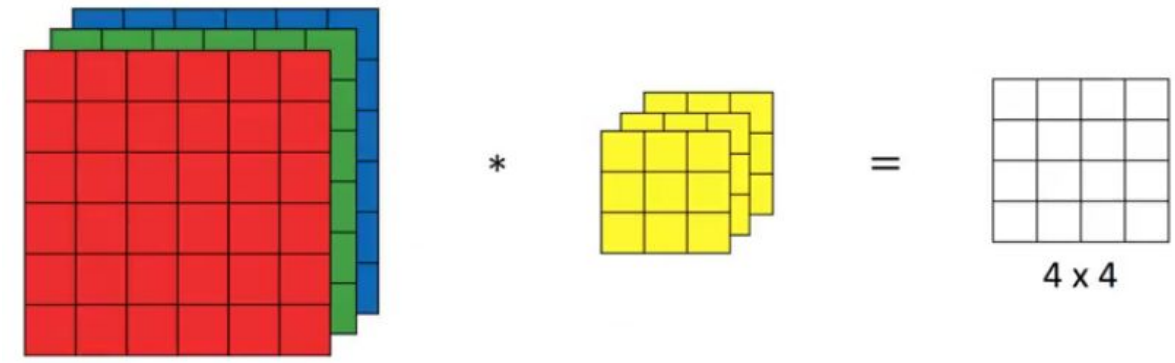


Figure: 3D convolutional step using a 3D filter

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.

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

- **Output:** [(n+2p-f)/s+1] X [(n+2p-f)/s+1] X n_c'

Here, n_c is the number of channels in the input and filter, while n_c' is the number of filters.

In a convolutional 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*3 = 27
- There will be a bias term for each filter, so total parameters per filter = 28
- As there are 10 filters, the total parameters for that layer = 28*10 = 280

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.

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

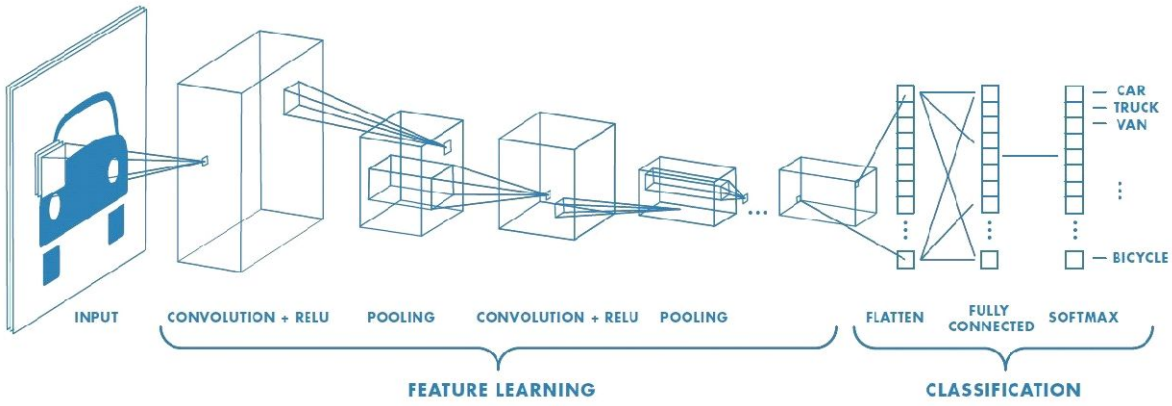


Figure: A complete CNN example

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.

Chapter 6 – Our Approach

Chapter 7 – The dataset, our CNN algorithm and Classification Work

Chapter 8 – Results and Findings

Chapter 9 – Conclusions and Future Work