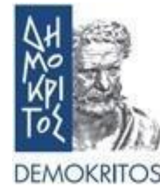


Deep Learning Methods for Cover Song Identification

by

Petros Mitseas

Submitted
in partial fulfilment of the requirements for the degree of

Master of Artificial Intelligence

at the

UNIVERSITY OF PIRAEUS

April 2023

Author: Petros Mitseas

II-MSc “Artificial Intelligence”

April, 2023

Certified by: Theodoros Giannakopoulos

Theodoros Giannakopoulos, Researcher B’, Thesis Supervisor

Certified by: Georgios Vouros

Georgios Vouros, Professor, Member of Examination Committee

Certified by: Iraklis Klampanos

Iraklis Klampanos, Research Associate, Member of Examination Committee

Deep Learning Methods for Cover Song Identification

By

Petros Mitseas

Submitted to the II-MSc "Artificial Intelligence" on March 20, 2023, in
partial fulfilment of the
requirements for the MSc degree

Abstract

Cover song identification (CSI) is the task of determining whether a given recording of a song is a new performance other than the original version. Automatically detecting cover versions has plenty of applications in the music industry as well as copyright law. In this Thesis we present a methodology for CSI based on Convolutional Neural Networks (CNN) and Metric Learning. The model is trained on medium-size datasets of cover songs using a variation of the Triplet Loss, called Angular Loss. The experiments showcase the performance of the proposed CNN model on English and Greek sets of cover songs, as well as other approaches based on deep learning. Our findings demonstrate that the proposed method exhibits viable performance for the specific use case, achieving high scores on the classification and ranking tasks. This, along with the fact that the model can run with minimal hardware requirements, make our method an ideal candidate for real-world applications. To further illustrate this point, we designed a proof of concept of such a system. Finally, as part of this Thesis, we created two new open-source datasets for CSI, that can be used for training or evaluation.

Thesis Supervisor: Theodoros Giannakopoulos
Title: Researcher B' (Demokritos)

Acknowledgments

I would like to acknowledge and express my gratitude to my supervisor Theodoros Giannakopoulos, for his feedback and general guidance above and beyond this Thesis. I would also like to thank my family and dear friends for their unwavering support and encouragement throughout the years. Finally I express my sincere thanks to Varvara and Eleftheria Konstantinidou, Eleni Mitsea, Dimitris Ventouris and the Multimedia Analysis Group of the Computational Intelligence Lab (MagCIL) of the National Center for Scientific Research “Demokritos" for their invaluable assistance in creating the dataset required for the completion of this Thesis.

Table of Contents

1.Introduction **9**
1.1. Motivation 9
1.2. Related Work 9
1.3. Proposed methodology 10
1.4. Next sections 11
2.Audio Representations **12**
2.1. Raw signal representation 12
2.2. Time-based Features 12
2.2.1. Zero crossing rate 12
2.2.2. Statistical measures 13
2.2.3. Energy 13
2.3. Frequency-based Features 13
2.3.1. Fast Fourier Transform 13
2.3.2. Mel Spectrogram 14
2.3.3. Mel Frequency Spectral Coefficients 16
2.3.4. Harmonic Pitch Class Profiles 17
3.Deep Learning **20**
3.1. Introduction 20
3.2. Convolutional Neural Networks 20
3.2.1. Introduction 20
3.2.2. Structure 20
3.2.2.1. Filters 20
3.2.2.2. Convolutional Layers 22
3.2.2.3. Pooling Layers 23
3.2.2.4. Fully Connected Layers 23
3.2.2.5. Activation Layers 23
3.2.2.6. Dropout Layers 24
3.2.2.7. Batch-Normalization Layers 24
3.2.3. Common architectures 24
3.2.3.1. LeNet 24
3.2.3.2. AlexNet 25
3.2.3.3. VGGNet 25
3.2.3.4. ResNet 25
3.3. Recurrent Neural Networks 26
3.3.1. Vanilla RNN 27
3.3.2. Long Short-Term Memory 28
3.3.3. Gated Recurrent Unit 29

3.4. Transformers 29
3.4.1. Self-Attention 30
3.4.2. Multi-head attention 30
3.4.3. Encoder-Decoder 31
3.4.4. Positional Encoding 32
3.4.5. Audio Applications 32

4. Methodology **34**
4.1. Overview 34
4.3. Feature Extraction 35
4.4. Model Architecture 36
4.5. Loss Function 37
4.5.1. Contrastive Loss 38
4.5.2. Triplet Loss 38
4.5.3. Angular Loss 41
4.6. Evaluation 42
4.6.1. Accuracy, Precision, Recall, F1 Score 42
4.6.2. ROC Curve 43
4.6.3. Precision-Recall Curve. 43
4.6.4. Mean Average Precision 44
4.6.5. Mean Reciprocal Rank 44
4.6.5. Precision at 10 45
4.6.6. Mean Rank of 1st identified cover 45
4.7. Implementation Details 45

5. Experiments **46**
5.1. Datasets 46
5.1.1. Covers80 46
5.1.2. Covers1000 46
5.1.3. Custom Dataset 47
5.2. Results on Covers80 test set 48
5.3. Results on Custom test set 50
5.4. Results on Greek covers 52
5.5. LSTM results 54
5.6. Visual Transformer results 55
5.7. t-SNE Visualization 57
5.8. Inference time 59
5.9. Observations 62

6. Web Application **65**
6.1. Platform overview 65
6.1.1. Upload songs 65
6.1.2. Manage song database 65

6.1.3. Cover check 65
6.1.4. Rank songs 65
6.2.Architecture 67
7.Conclusions **69**

List of Figures

Figure 2.1 Visualization of the FFT. The combination stage is also known as "Butterfly". 14

Figure 2.2 Mel scale vs Hertz scale 15

Figure 2.3 The Mel Spectrogram 16

Figure 2.4 Visualization of MFCCs 16

Figure 2.5. Visualization of the chroma profiles w.r.t. the original signal and the music notes. 19

Figure 3.1. Visualization of 2D-convolution 21

Figure 3.2. Illustration of a layer consisting of two filters. Notice the input/output channel size. 22

Figure 3.3 The different ResNet sizes. 26

Figure 3.4 A schematic representation of the recurrent neural network, unfolding per timestep. 26

Figure 3.5. The internal components of an LSTM cell. All operations in orange boxes are pointwise operations. 29

Figure 3.6. The GRU components. 29

Figure 4.1 Visualization of the cover song identification pipeline 34

Figure 4.2 Initial and final representation after minimizing the loss function. Notice that the white points are not pushed further away, since they already exceed the margin m. 38

Figure 4.3 Visualization of triplet loss minimization 39

Figure 4.4 Visualization of the different types of triplets (determined by the location of the negative sample). 40

Figure 4.5 The ROC curve. 43

Figure 4.6 The Precision-Recall curve. 44

Figure 5.1 Training and Validation loss per epoch. 48

Figure 5.2 Confusion matrices (left: unbalanced dataset, right: balanced dataset) 49

Figure 5.3 Covers80 (balanced) - ROC and Precision-Recall Curves 49

Figure 5.4 Covers80 (unbalanced) - ROC and Precision-Recall Curves 50

Figure 5.5 Confusion matrices (left: unbalanced dataset, right: balanced dataset) 51

Figure 5.6 Custom Dataset (balanced) - ROC and Precision-Recall Curves 51

Figure 5.7 Custom Dataset (unbalanced) - ROC and Precision-Recall Curves 52

Figure 5.8 Confusion matrices (left: unbalanced, right: balanced) 53

Figure 5.9 Greek Dataset (balanced) - ROC and Precision-Recall Curves 53

Figure 5.10 Custom Dataset (Full songs) - ROC and Precision-Recall Curves 54

Figure 5.11. The results of the LSTM model on the Covers80 test set. 55

Figure 5.12. The results of the ViT model on the Covers80 test set. 56

Figure 5.13 t-SNE plot of the test dataset 58

Figure 5.14 t-SNE plot of the train dataset 58

Figure 5.15 Acoustic or calm songs, seem to be grouped closer together. 63

Figure 5.16 Rock songs with more complex instrumentation are placed closer together. 64

Figure 6.2 The user selects a song and retrieves a sorted list based on similarity. The icon next to each result indicates a potential cover song (green is positive, red is negative). 66

List of Equations

Equation 2.1. Energy of the signal, calculated in time-domain. 13

Equation 2.2 The Discrete Fourier Transform 13

Equation 2.3 Hertz to mels conversion. 15

Equation 2.4 The elements of the N-length HPCP vector. 18

Equation 3.1. Convolution operation between a filter F and an image I 21

Equation 3.2. Calculating the output size after convolution 21

Equation 3.3 The output of a convolutional layer, where is the output of the convolution layer, is the input, is the filter, is the bias, and is the size of the filter (omitting the depth dimension). 23

Equation 3.4 Max pooling formula 23

Equation 3.5 Average pooling formula 23

Equation 3.6 ReLU activation function 24

Equation 4.1. Relationship between number of HPCP windows and the resulting segment size 35

Equation 4.2 The Contrastive Loss formula 38

Equation 4.3 Requirement regarding the anchor, positive and negative examples in the embedding space. 39

Equation 4.4 The Triplet Loss 39

Equation 4.5 The Angular Loss 41

Equation 4.6 Classification scores 42

Equation 4.7 The Average Precision metric 44

List of Tables

Table 4.1 Optimal model architecture 37

Table 5.1 Classification scores on the Covers80 dataset. 48

Table 5.2 Ranking scores on the Covers80 dataset. 50

Table 5.3 Classification scores on the Custom dataset. 50

Table 5.4 Ranking scores on the Custom dataset. 52

Table 5.5 Classification scores on the Greek dataset. 52

Table 5.6 Ranking scores on the Greek dataset. 54

Table 5.7. Classification scores for LSTM on the Covers80 test set. 55

Table 5.8. Ranking scores for LSTM on the Covers80 test set. 55

Table 5.9. Classification scores for ViT on Covers80 set. 56

Table 5.10. Ranking scores for ViT on the Covers80 test set. 56

Table 5.11 Inference time benchmarking. 60

Table 5.12 Accuracy vs segment size for CNN model. 61

Table 5.13. Comparison between different methods on the test sets. 61

1.Introduction

"Music Streaming Hits Major Milestone as 100,000 Songs are Uploaded Daily to Spotify and Other DSPs" - was the title of an article posted in *Variety* by the end of 2022. This number was estimated around 60,000 in 2021 and roughly 40,000 in 2019. Eventually, keeping track of new song releases, as well as cover songs, remixes and different versions is an increasingly hard challenge. As covers we define performances of the original song by other artists, that may have different instrumentation, rhythm, style or sung in a different language. These can be found in a variety of contexts, including tribute albums, live performances, and online video platforms.

1.1.Motivation

Cover song identification (CSI) is the task of determining whether a given recording of a song is a new performance other than the original version. This can be important for a number of reasons. For example, regarding copyright law, it is necessary to determine whether a cover song requires permission or licensing in order to be distributed or performed publicly. Similarly, in the context of plagiarism, it is important to identify cover songs in order to properly attribute credit to the original artist. Applications can also be found in the music industry, to improve user experience. For instance, music platforms can improve recommendations by taking into account the different versions of a song. In addition, it can be useful for music historians and researchers, as it can help to trace the evolution and spread of particular songs over time.

1.2.Related Work

One of the most common approaches to song identification is audio fingerprinting [1] . This involves extracting a compact numerical representation of the audio signal, called a fingerprint, and comparing it to a database of known fingerprints. If a match is found, the recording is identified.

Audio fingerprinting has been successful in many applications, especially mobile apps, since it is easy to implement and run, but it has some limitations that make it less than ideal for cover song identification. One of the main limitations is that it is sensitive to any deviations from the original audio signal. This can make it difficult to identify covers of low-quality recordings or recordings that have been significantly altered. In addition, it can be unreliable for recordings with changes in arrangement or instrumentation. If the song is performed by a different artist, with varying pitch or the verses do not align completely, then the fingerprint calculated by the main frequencies will be different and the cover won't be identified.

Another method, described in [2], is based on aligning audio sequences and evaluating similarity. The audio of the songs under measurement is segmented into windows and a representation is extracted for each window, which is related to the melody of the audio. Then these sequences undergo transformations to achieve key invariance, and finally the distance between them is calculated using the Smith-Waterman alignment algorithm. While the method above produces adequate results, that quadratic time complexity that arises from Smith-Waterman algorithm, makes it hard to implement in low-hardware resource cases.

There have been various works that leverage deep learning, to solve the task of CSI. In [3] Xiaoshuo Xu et al used a carefully structured convolutional neural network with HPCP (Harmonic Pitch Chroma Profiles) input, to create a key-invariant model for classification. CNNs have also been used with CQT (Constant-Q Transform). Zhesong Yu et al [4] used these features to train a custom model using cross-entropy loss. In [5] the authors applied Temporal Pyramid Pooling, to capture output features at different scales. Triplet loss and classification loss have also been used in conjunction, to train a modified ResNet50 model with instance normalization and batch normalization blocks, achieving remarkable results [6].

1.3.Proposed methodology

In our work, as opposed to pure algorithmic or heuristic solutions, we investigate the use of deep learning techniques to solve this problem. Our method follows a supervised

learning approach: We collect a large number of annotated cover songs which are used as training data for an AI model. The model learns the underlying associations and structure of the audio that makes up a cover song, and is able to determine whether a pair of unseen songs are covers or not. At first sight, our approach demonstrates the following benefits:

- The model yields a solid performance across several different genres of music, as shown in the experiments.
- No custom rules or heuristics are used. The model learns directly from data.
- We can scale the training data and model depending on the available resources.
- The inference time is low enough, even using CPU, making it suitable for real-world applications.

1.4. Next sections

The next chapters are organized in the following way:

In chapter 2 we make an introduction to the most commonly used audio representations. These include time and frequency domain features, which make up the inputs of the model.

In chapter 3 we describe the theoretical background behind the most common deep learning models used in audio applications, and reference some notable architectures found in literature.

The methodology that we followed is extendedly discussed in chapter 4. We analyze the datasets used, the choice of model architecture and the training process.

The experiments are presented in chapter 5, along with the resulting observations.

Finally, chapter 6 contains the implementation procedure and technology stack of the platform that is used for a real-life application of the trained model.

2. Audio Representations

2.1. Raw signal representation

In computers, raw audio signals are typically represented as a sequence of numerical values, often stored in a digital audio format such as a WAV or AIFF file. These numerical values represent the amplitude of the audio signal at discrete points in time, known as samples. The sampling rate is the number of samples taken per second, and is typically measured in hertz (Hz). Common sampling rates are 44.1 kHz, 48 kHz, and 96 kHz, which correspond to 44100, 48000, and 96000 samples per second, respectively. The numerical values of the samples are typically stored as 16-bit or 24-bit integers, depending on the desired audio quality. 16-bit audio is commonly used for consumer audio applications, while 24-bit audio is often used for professional audio applications.

Let's assume a 4-minute long song. With a sample rate of 16 kHz, the resulting array would be a sequence of 3840000 values. The downside of using this sequence directly as input to a machine learning model is that it is computationally expensive and can take a long time to process. Furthermore this representation is very sparse, which can make it difficult for the model to learn useful patterns at reasonable amounts of data. To address these issues, higher-level features such as time and frequency based representations can be extracted.

2.2. Time-based Features

Time-based features are extracted from the amplitude values of the audio signal over time.

2.2.1. Zero crossing rate

Zero crossing rate is a measure of the number of times a signal changes from positive to negative or vice versa in a given time period.

2.2.2.Statistical measures

Standard statistical measures, such as mean, variance, skewness and kurtosis can be used as features in the time domain.

2.2.3.Energy

The energy of the signal for a given segment is defined by the following equation.

$$Energy = \sum_{n=-N}^{N} x^2(n)$$

Equation 2.1. Energy of the signal, calculated in time-domain.

2.3. Frequency-based Features

These features are better suited for audio applications. They are extracted from the frequency components of the audio signal, using the Fourier transform and can give us insight into the spectral content of an audio clip. The most commonly used frequency-based representations, for music information retrieval (MIR) tasks, are the Fast Fourier Transform (FFT), Mel Spectrogram, MFCC (Mel-Frequency Cepstral Coefficients) and Harmonic Pitch Class Profiles (HPCP).

2.3.1. Fast Fourier Transform

The Fast Fourier Transform (FFT) [7] is an efficient algorithm for calculating the Discrete Fourier Transform, a method for analyzing a signal in the frequency domain by decomposing it into its constituent sinusoidal components.

The DFT of a signal (sequence) is defined by the following formula:

$$X[k] = \sum_{n=0}^{N-1} x[n] \cdot e^{-j \cdot 2\pi \cdot \frac{k \cdot n}{N}} = \sum_{n=0}^{N-1} x[n] \cdot (cos(2\pi \cdot \frac{k \cdot n}{N}) - jsin(2\pi \cdot \frac{k \cdot n}{N}))$$

Equation 2.2 The Discrete Fourier Transform

The time complexity of the standard DFT is $O(n^2)$. FFT reduces the number of calculations required, bringing the complexity down to $O(n \log n)$. It does so by splitting the original sequence into an odd and an even part, each of size N/2, calculating the DFT of each and combining the output to get the result for the full sequence. The DFT of the subsequences can be calculated with the same manner, recursively until the length of each sequence becomes equal to 1 (base case of the recursion). This is known as the Cooley-Tukey FFT algorithm.

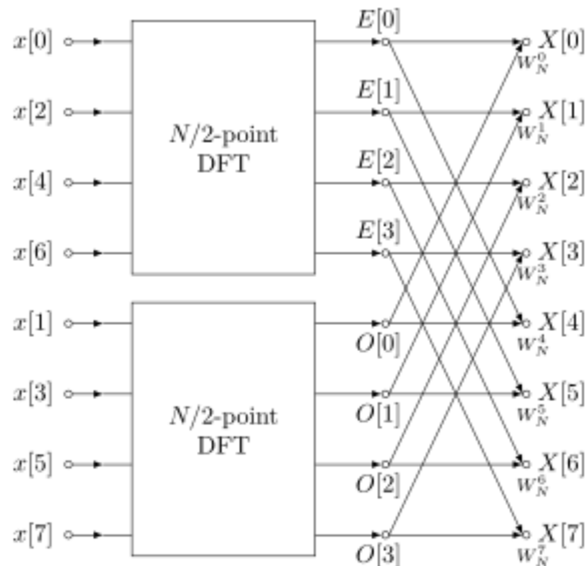


Figure 2.1 Visualization of the FFT. The combination stage is also known as "Butterfly".

Calculating the FFT is the starting point for extracting the next features.

2.3.2. Mel Spectrogram

The Mel Spectrogram is a time-frequency representation of a signal, expressed in the Mel frequency scale. The first step for creating this representation is to generate the Short Time Fourier Transform of the audio signal, by splitting the signal into segments using an arbitrary window size (usually tens or hundreds of milliseconds). The windows may be overlapping. With a small enough window length, we can assume that the frequencies are constant and calculate the FFT. A small window introduces better time

resolution but worse frequency resolution and vice versa, due to the fixed-resolution of the STFT (the product between the time and frequency deviation is bounded). Finally we concatenate all windows to generate the time-frequency representation known as spectrogram.

The next step involves converting the spectrogram from the hertz scale of the signal to the Mel frequency scale. The human perception of audio frequencies is not linear across the audible range. Through psychoacoustic experiments, it is shown that we can better differentiate sounds at lower frequencies rather than higher ones. We can define a scale at which increasing the interval between two frequencies, also leads to the same increase in perceivable pitch. Therefore, the Mel scale is defined by the following:

$$m = 2595 \log_{10} \frac{f}{700}$$

Equation 2.3 Hertz to mels conversion.

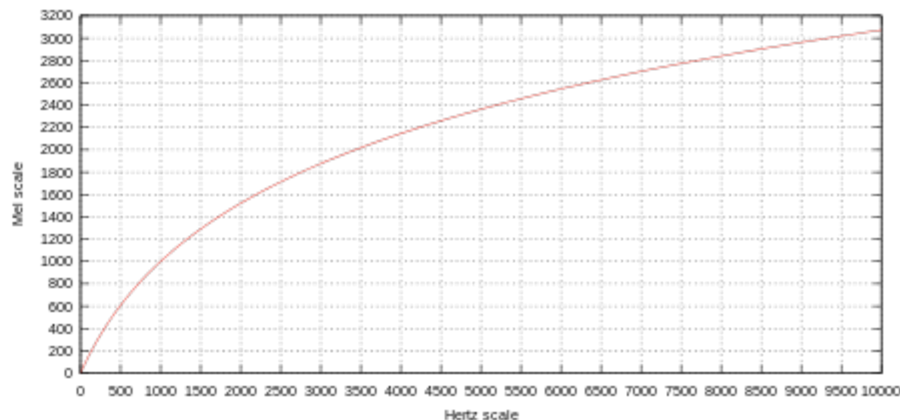


Figure 2.2 Mel scale vs Hertz scale

Next, a filterbank is created, which is a set of overlapping triangular filters that cover the entire mel range (typically 0-8000 Hz). A common choice for the number of filters is 128. Finally for each STFT window, the amplitude is multiplied by the filterbank.

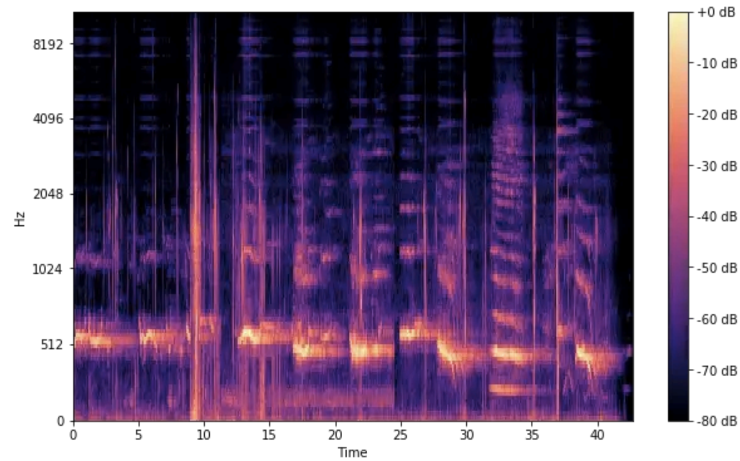


Figure 2.3 The Mel Spectrogram

The Mel Spectrogram is the most common representation of audio, used in conjunction with convolutional neural networks, since this 2D input can be naturally handled by CNNs. This representation is generic enough and can be used for various audio tasks such as speech processing, music information retrieval, and automatic music transcription.

2.3.3. Mel Frequency Spectral Coefficients

The Mel Frequency Spectral Coefficients (MFCCs) are derived from the Mel spectrogram by taking the logarithm of the energy in each frequency bin and then performing a Discrete Cosine Transform (DCT) on the resulting array. The resulting coefficients are then used as features, usually together with the first and second order differences (delta). MFCCs are commonly used in speaker recognition tasks.

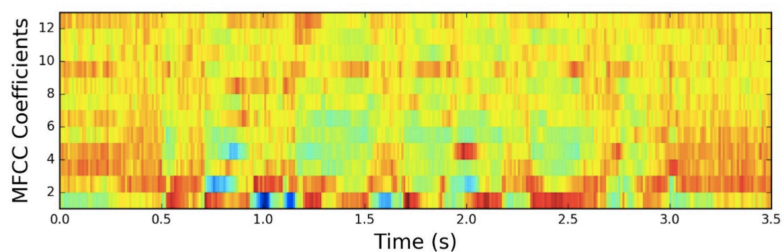


Figure 2.4 Visualization of MFCCs

2.3.4. Harmonic Pitch Class Profiles

Harmonic Pitch Class Profiles is a type of spectral representation of a signal that's based on the intensity of the twelve different pitch classes. This representation is best suited for Music Information Retrieval tasks, since it's compact, closely related to melody and provides characteristics like instrumentation and timbre indifference. In order to further explain how these features are derived, it's necessary to start with some definitions.

In music, an octave is the interval between two pitch sounds, where one has double the frequency of the other. Sounds that are a number of octaves apart, belong to the same pitch class. In psychoacoustics the quality of the pitch is also referred to as "chroma", and sounds that share the same chroma are perceived as similar in color. In western music we consider twelve chroma values represented by the set

{C, C♯, D, D♯, E , F, F♯, G, G♯, A, A♯, B}

A440 is the pitch corresponding to a frequency of 440Hz ("A" music note). It serves as a reference frequency for tuning musical instruments.

HPCP is essentially a vector, measuring the intensity of each of the 12 pitch classes in a given time frame. This feature is calculated in successive short windows, resulting in a sequence of vectors that represent the whole signal. The procedure for calculating the HPCP is the following:

1. Perform FFT on the input signal
2. Keep frequencies in the range of 100-5000 Hz
3. Perform peak detection and keep only the local maximum frequency values
4. Estimate the frequencies of each pitch class using the reference frequency (usually A440)
5. Perform the mapping between frequencies and N pitch classes (usually 12). For each peak frequency f_i we calculate the distance from the reference frequency of each pitch class f_n, with the following formula:

$$d(n, f_i) = 12 \log_2 (\frac{f_i}{f_n}) + 12m ,$$

where m is an integer that minimizes d. The reference frequencies are defined as:

$$f_n = 2^{\frac{n}{N}} f_{ref}$$

The weight of f_i for frequency bin n is given by:

$$w(n, f_i) = \cos^2\left(\frac{\pi d(n, f_i)}{l}\right) \text{ if } d < 0.5l \text{ else } 0,$$

where l is the chosen width of the weight window.

The elements of the HPCP vector are defined as below:

$$C_{HPCP}(n) = \sum_{i=1}^{N_{peaks}} w(n, f_i) a_i^2,\ n = 1, 2, ...N$$

Equation 2.4 The elements of the N-length HPCP vector.

, where a_i is the magnitude of the associated frequency. The vector is also normalized by the max element.

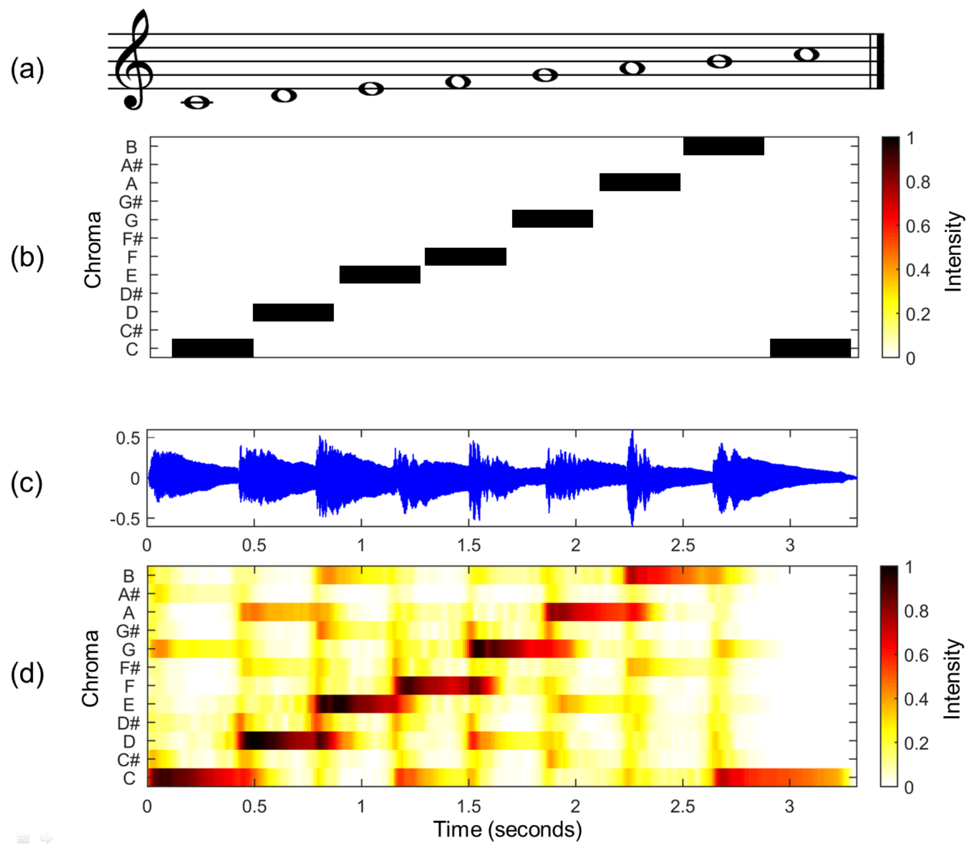


Figure 2.5. Visualization of the chroma profiles w.r.t. the original signal and the music notes.

3. Deep Learning

3.1. Introduction

With the abundance of data and progress in computing power during the past decade, deep learning has been very successful in a wide range of applications. Architectures like convolutional neural networks, recurrent neural networks and most recently Transformers, have achieved major breakthroughs in image and speech recognition, natural language processing, music classification and other tasks, pushing the limit on what AI and machine learning can achieve. In this section we describe the most commonly used deep learning model types for audio applications.

3.2. Convolutional Neural Networks

3.2.1. Introduction

Convolutional Neural Networks (CNN)[8] are a special type of Neural Networks inspired by "neocognitron", a computational model for visual pattern recognition introduced by Dr. Kunihiko Fukushima in 1980. The first successful application of modern CNNs occurred in the 1990's by Yann LeCun et al., who trained the model on the MNIST dataset of handwritten digits. Given example images, the model - running in live mode- was then able to predict the drawn digits. Throughout the decade of 2010-2020 major advancements were made in the field of CNNs, which are widely used up to this day for the tasks of image and audio classification.

3.2.2. Structure

CNNs rely on convolutions between the input and filters with learnable parameters that are used to extract features. In this section, we perform a deeper dive into the components that make up a CNN.

3.2.2.1. Filters

In general, a filter -or kernel- is a matrix consisting of weights which can be applied to an image input through 2D-convolution. By selecting the values of these weights, various results can be achieved such as blurring the original image, or edge detection in different

orientations. In other words, applying different filters on the image produces different features that can be used for further processing. In CNNs the filters' weights are not predefined, but instead adjusted during the network's training, using backpropagation.

$$(F * I)[n, m] = \sum_{i=-k_1}^{k_1} \sum_{j=-k_2}^{k_2} F[i, j] \cdot I[n - i, m - j]$$

Equation 3.1. Convolution operation between a filter F and an image I

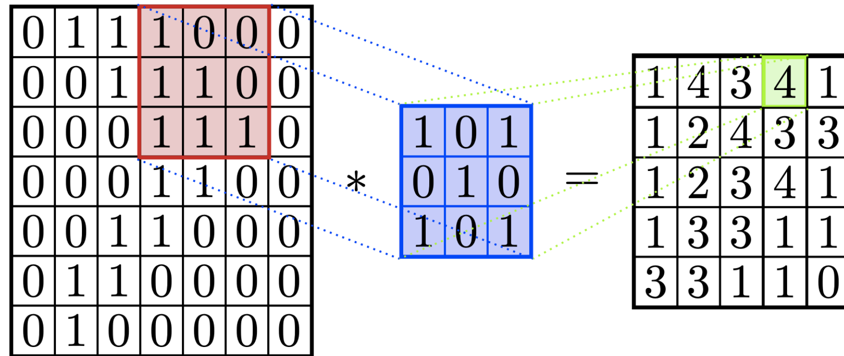


Figure 3.1. Visualization of 2D-convolution

Some notable parameters of convolution operations in CNNs include padding and stride. Suppose an image of size $N \times N$ and a filter of size $K \times K$. The output size of the convoluted image is $(N - K + 1) \times (N - K + 1)$. After successive convolutions the size decreases by the same factor each time. To avoid this, zero padding can be introduced along the edges of the image so that the dimensions stay consistent across convolutions. On the other hand, the stride parameter determines the pixels to skip when performing the convolution, along a dimension. For example with stride = 2, the convolution will only be calculated for elements 0, 2, 4, etc. instead of each element of the dimension. This helps reduce the size of the output. In general the output size for a square image and kernel is given by the following formula:

$$O = \frac{(W + 2P - K)}{S} + 1$$

Equation 3.2. Calculating the output size after convolution

where O is the output size, W is the input size, P is the amount of padding, K is the size of the kernel, and S is the stride.

3.2.2.2. Convolutional Layers

A convolutional layer consists of multiple filters, each with its own set of weights, that are applied to the input and produce an output. The filters in each layer detect features of increasing complexity and abstraction as we move deeper into the network. For example, the filters in the first layer might detect edges, while the filters in the last layer might detect the presence of a certain type of object.

Suppose an initial image of dimensions 224 x 224. Images are usually stored in RGB format where each pixel is a triplet of numbers, each representing the intensity of the respective color (Red, Green, Blue). We refer to each color dimension as a channel. Therefore the image can be represented as a tensor of dimensions (224 x 224 x 3). This is the input to the first layer of the CNN. The filters are usually of size 3 x 3 or 5 x 5 with also a depth dimension, equal to the number of the previous layer's filters. The first layer's kernel depth dimension is equal to the size of the input channels (in this case 3). Suppose a kernel size of 3 x 3, The output of this layer will be a tensor of dimensions (222 x 222 x 32) (without padding), where 32 is the number of filters used in the convolutional layer. In this case, the output channel size is 32, which is equal to the input channel size of the next layer.

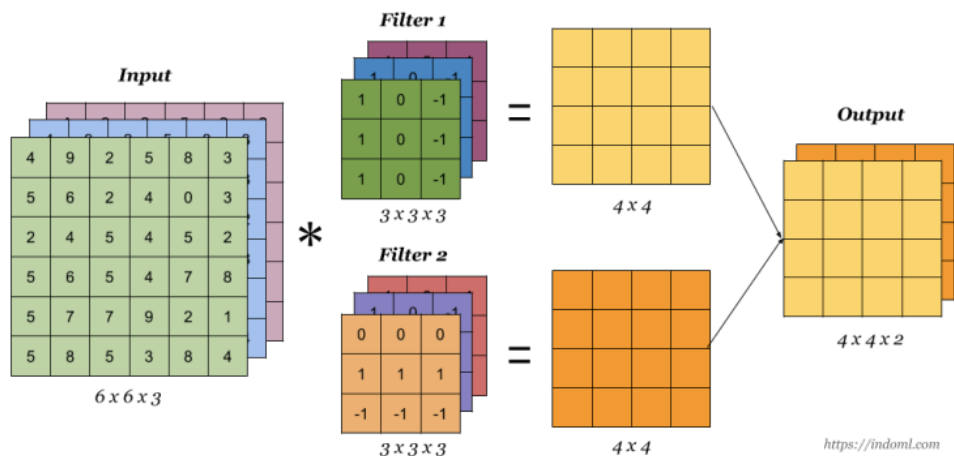


Figure 3.2. Illustration of a layer consisting of two filters. Notice the input/output channel size.

$$y(i,j) = \sum_{m=0}^{k-1} \sum_{n=0}^{k-1} w(m,n) x(i-m, j-n) + b$$

Equation 3.3 The output of a convolutional layer, where y is the output of the convolution layer, x is the input, w is the filter, b is the bias, and k is the size of the filter (omitting the depth dimension).

3.2.2.3. Pooling Layers

Pooling layers are used to reduce the dimensionality of the input. This helps to reduce the amount of computations required to process the input and avoid overfitting. There are two commonly used types of pooling: max pooling and average pooling. In max pooling, the maximum value of a certain region of the input is chosen as the output. In average pooling, we take the average of the values in the region.

The formula to calculate the output of a pooling layer of region size k and stride s is given by:

$$O_{i,j} = \max_{m=0}^{k-1} \max_{n=0}^{k-1} I_{i \cdot s+m, j \cdot s+n}$$

Equation 3.4 Max pooling formula

$$O_{i,j} = \frac{1}{k^2} \sum_{m=0}^{k-1} \sum_{n=0}^{k-1} I_{i \cdot s+m, j \cdot s+n}$$

Equation 3.5 Average pooling formula

3.2.2.4. Fully Connected Layers

Fully connected layers are densely connected layers with every neuron in one layer connected to every neuron in the next layer. They are typically added at the end of the network after the convolutional layers. These are used to combine the output features coming from convolutional layers, detect patterns across the input space (in contrast with local features, detected by convolutional layers) and to generate the final classification or regression output.

3.2.2.5. Activation Layers

The output of convolutional or fully connected layers passes through an activation function. This layer introduces the necessary non-linearity to the output, enabling the

model to learn more complex functions. The most commonly used activation function in CNNs is the *rectified linear activation unit* or *ReLU*. Others include the Sigmoid and Tanh functions.

$$f(x) = \begin{cases} 0, & \text{if } x < 0 \\ x, & \text{if } x \geq 0 \end{cases}$$

Equation 3.6 ReLU activation function

3.2.2.6. Dropout Layers

Dropout layers work by randomly ignoring a number of layer outputs. This helps to reduce overfitting and make the model more robust, since it reduces the phenomenon of layers co-adapting to fix previous layers' mistakes, leading to poor generalization on actual data. The dropping happens only during the training phase.

3.2.2.7. Batch-Normalization Layers

Batch normalization is done by calculating the mean and variance of each layer's inputs, for each mini-batch during the training process. This information is then used to normalize the inputs, so that they have a mean of 0 and a standard deviation of 1. The resulting vector is then scaled and shifted using learnable parameters, that are adjusted during the training process. This normalization helps to reduce the effects of the internal covariate shift, as well as improve the overall performance of the network.

3.2.3.Common architectures

In this section we mention some of the most known CNN architectures in literature.

3.2.3.1.LeNet

LeNet is an early convolutional neural network (CNN) created by Yann LeCun in 1998 [9]. It is a shallow network consisting of 7 layers, including 3 convolutional, 2 average pooling and 2 fully connected layers. It was designed to recognize handwritten digits, such as those seen on bank checks. LeNet was one of the first successful applications of deep learning.

3.2.3.2.AlexNet

AlexNet is a CNN created by Alex Krizhevsky, Ilya Sutskever, and Geoffrey Hinton in 2012 [10]. It consists of 8 layers (5 convolutional and 3 fully connected) and uses techniques like the ReLU activation function and dropout regularization. It was the first CNN to win the ImageNet Large Scale Visual Recognition Challenge (ILSVRC).

3.2.3.3.VGGNet

VGGNet is a convolutional neural network developed by the Visual Geometry Group (VGG) at the University of Oxford in 2014 [11]. It was designed to detect objects in images and classify them into different categories. VGGNet is a deep, feed-forward neural network that consists of 16 layers of convolutional and fully connected layers, with a total of 138 million parameters. The network was trained on ImageNet, a large dataset of images with 1000 different classes.

The network consists of five convolutional blocks followed by three fully connected layers. Each convolutional block contains two or three convolutional layers, followed by a max-pooling layer. The convolutional layers use a 3x3 filter size and a stride of 1, while the max-pooling layers use a 2x2 filter size and a stride of 2. The first two convolutional blocks use 64 filters, while the remaining three blocks use 128 filters.

VGGNet was the first network to achieve top-5 accuracy of 92.7% on the ImageNet dataset. It is still one of the most popular networks for image classification tasks.

3.2.3.4.ResNet

ResNet is a deep residual neural network created by Kaiming He, et al. in 2015 [12]. The largest variation reaches 152 layers and is one of the deepest networks ever created. ResNet was designed to address the problem of vanishing gradients in deep neural networks. It uses skip connections, which allow the network to learn from earlier layers and helps to reduce the amount of computation required. ResNet comes in different sizes of 18, 34, 50, 101 and 152 layers, according to the complexity of the use case.

layer name	output size	18-layer	34-layer	50-layer	101-layer	152-layer
conv1	112×112	7×7, 64, stride 2				
conv2_x	56×56	3×3 max pool, stride 2				
		$\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$
conv3_x	28×28	$\begin{bmatrix} 3\times3, 128 \\ 3\times3, 128 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 128 \\ 3\times3, 128 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times8$
conv4_x	14×14	$\begin{bmatrix} 3\times3, 256 \\ 3\times3, 256 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 256 \\ 3\times3, 256 \end{bmatrix}\times6$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times6$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times23$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times36$
conv5_x	7×7	$\begin{bmatrix} 3\times3, 512 \\ 3\times3, 512 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 512 \\ 3\times3, 512 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$
	1×1	average pool, 1000-d fc, softmax				
FLOPs		1.8×10^9	3.6×10^9	3.8×10^9	7.6×10^9	11.3×10^9

Figure 3.3 The different ResNet sizes.

3.3. Recurrent Neural Networks

A recurrent neural network (RNN) is a type of network that's best suited for sequential data. Until recent years it was considered state of the art in speech recognition, time-series forecasting and language modelling.

RNNs use an internal state mechanism, allowing the network to have a form of memory. This means that the output relies not only on the current input, but also on past predictions, much like a closed loop system (figure 3.4).

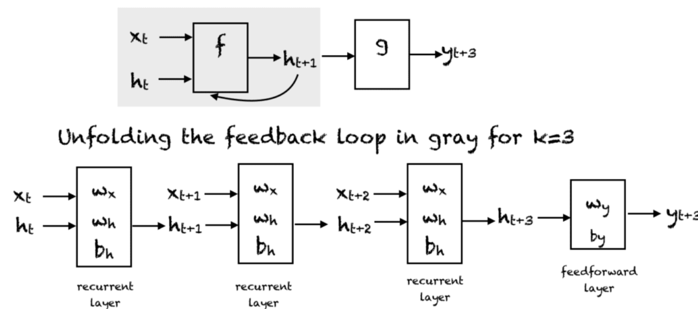


Figure 3.4 A schematic representation of the recurrent neural network, unfolding per timestep.

3.3.1. Vanilla RNN

In the simplest form, RNNs can be described by the following equations:

$$h_t = \sigma(W_{hx}x_t + W_{hh}h_{t-1} + b_h)$$
$$y_t = \sigma(W_{hy}h_t + b_y)$$

where:

h_t - hidden state at time step t

x_t - input vector at time step t

b_h, b_y - bias vectors

y_t - output vector at time step t

σ - activation function, like ReLU, sigmoid or tanh

W_{hx} - weight matrix connecting the input vector at time step t to the hidden state at time step t

W_{hh} - weight matrix connecting the hidden state at time step t-1 to the hidden state at time step t

W_{hy} - weight matrix connecting the hidden state at time step t to the output vector at time step t

Vanilla RNNs suffer from a problem known as vanishing/exploding gradient. Updating the model's weights requires back-propagation of the loss, both through the network layers and also through time. As the sequence length increases, the quantities that contribute to the weights' update either increase or decrease exponentially, leading to instability. For example, in the case of an RNN with linear activation without bias, the gradient of the loss wrt. the hidden state at time t (where 1<t<T), is given by:

$$\frac{\partial L}{\partial h_t} = \sum_{i=t}^{T} (W_{hh}^{\intercal})^{T-i} W_{qh}^{\intercal} \frac{\partial L}{\partial y_{T+t-i}}$$

The gradients of the loss wrt. the weight matrices are given by:

$$\frac{\partial L}{\partial W_{hx}} = \sum_{t=1}^{T} \frac{\partial L}{\partial h_t} x_t^{\intercal}$$

$$\frac{\partial L}{\partial W_{hh}} = \sum_{t=1}^{T} \frac{\partial L}{\partial h_t} h_{t-1}^{\intercal}$$

The large powers of $(W_{hh}^{\intercal})^{T-i}$ may lead to numerical instabilities as eigenvalues smaller than 1 vanish and eigenvalues larger than 1 diverge.

For this reason, two other types of recurrent networks are used in almost all practical cases, called Long Short-Term Memory and Gated Recurrent Units.

3.3.2.Long Short-Term Memory

LSTM networks introduce the notion of a cell that maintains an internal state, retaining information across longer sequences [13]. This state is modified through components known as "gates", as shown in figure 3.5.

The forget gate regulates how much information is discarded from the cell state at each step. Its value is given by:

$$f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f)$$

The input gate is related to new information, entering the cell state. The input to the gate is the concatenated vector of the network's input and the previous hidden state.

$$i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i)$$

$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] + b_C)$$

The cell state's update is regulated by the above gates as:

$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$

Finally the output gate controls the information that is outputted from the cell (hidden state).

$$o_t = \sigma(W_o \cdot [h_{t-1}, x_t] + b_o)$$

$$h_t = o_t * tanh(C_t)$$

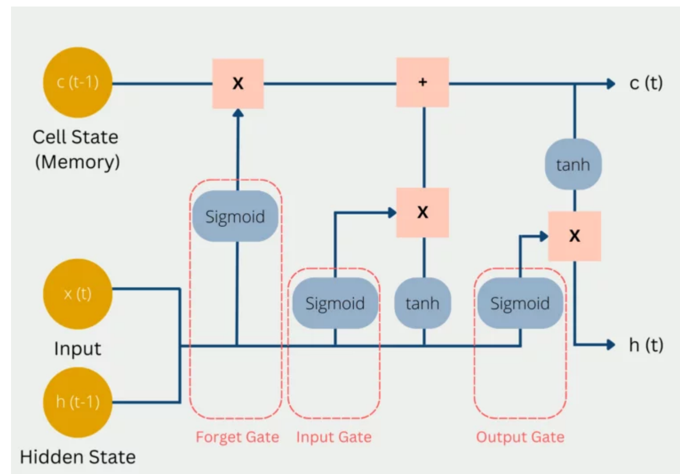


Figure 3.5. The internal components of an LSTM cell. All operations in orange boxes are pointwise operations.

3.3.3.Gated Recurrent Unit

The Gated Recurrent Unit (GRU) [14] is a modified version of the LSTM, which combines the input and forget gate into a single update gate and introduces other structural changes that result in a simpler model.

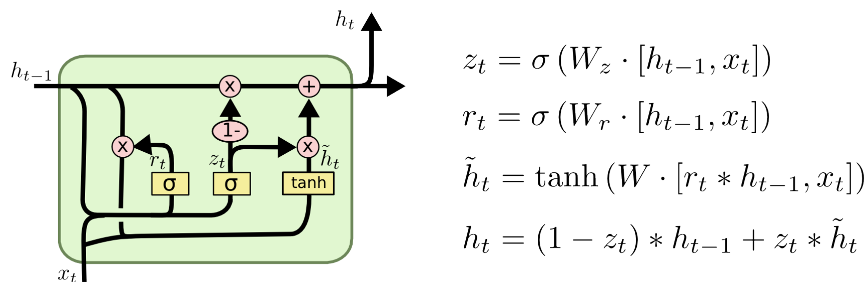


Figure 3.6. The GRU components.

3.4. Transformers

Introduced in 2017, Transformers [15] rely on the mechanism of Attention [16], being able to model very large sequences of data. These networks revolutionized natural language processing and led to the rise of large language models that are widely used

today to perform text-to-text or text-to-image generation, with GPT-3 [17] and DALL-E [18] being the most notable examples.

Transformers consist of two main components: the encoder and the decoder part. Different architectures may include both or only one of these components. For example BERT (BiDirectional Encoder Representations from Transformers) [19] uses only the encoder part with 12 encoder layers, while GPT's architecture is a 12-layer decoder-only transformer.

3.4.1.Self-Attention

Suppose a sequence of vectors x_1, x_2, ..., x_n that represent the input to the transform model. These could be the embeddings of the words in a sentence. For each input vector, three vectors are created: the query, key and value:

$$Q = [x_1, x_2, , x_n]^\intercal W^Q$$

$$K = [x_1, x_2, , x_n]^\intercal W^K$$

$$V = [x_1, x_2, , x_n]^\intercal W^V$$

The weight matrices are learned during training. Then for each $q \in Q$ we calculate the dot product with each $k \in K$ and divide by the square root of n. The output vector for each q, is then passed through a softmax function.

The next step is to multiply each value vector $v \in V$ by the corresponding softmax score and sum the vectors, to retrieve the output. These operations are highly parallelizable, and can be written in a compact matrix form as:

$$Z = \text{softmax}(\frac{QK^\intercal}{\sqrt{n}})V$$

This matrix is the output of a single attention head.

3.4.2.Multi-head attention

We can repeat the above operations with different matrices W_i^K, W_i^Q, W_i^V and retrieve different outputs Z_i. We refer to these as heads, where each head can learn a different

representation. The outputs of each head are concatenated and multiplied by a combination matrix W^O.

$$Z_{out} = [Z_0, Z_1, , Z_{nheads}]W^O$$

The resulting matrix, which captures information from all attention heads, is sent to a Feed Forward neural network. Also residual connections and layer normalization operations are introduced at the output of the self-attention and feed-forward layers.

3.4.3. Encoder-Decoder

Figure 3.7 contains an overview of the architecture. As shown, multiple stacked encoder and decoder blocks are used. The decoder part is similar in structure. At each time step t, the output of the decoder is used as input to make the prediction at the next time step t+1. The Encoder-Decoder Attention is the module used to transfer the context derived from the encoder to the decoder, in order to influence the predictions. It works in a similar fashion as the Self-Attention module except that it creates its Queries matrix from the layer below it, and takes the Keys and Values matrix from the output of the encoder stack.

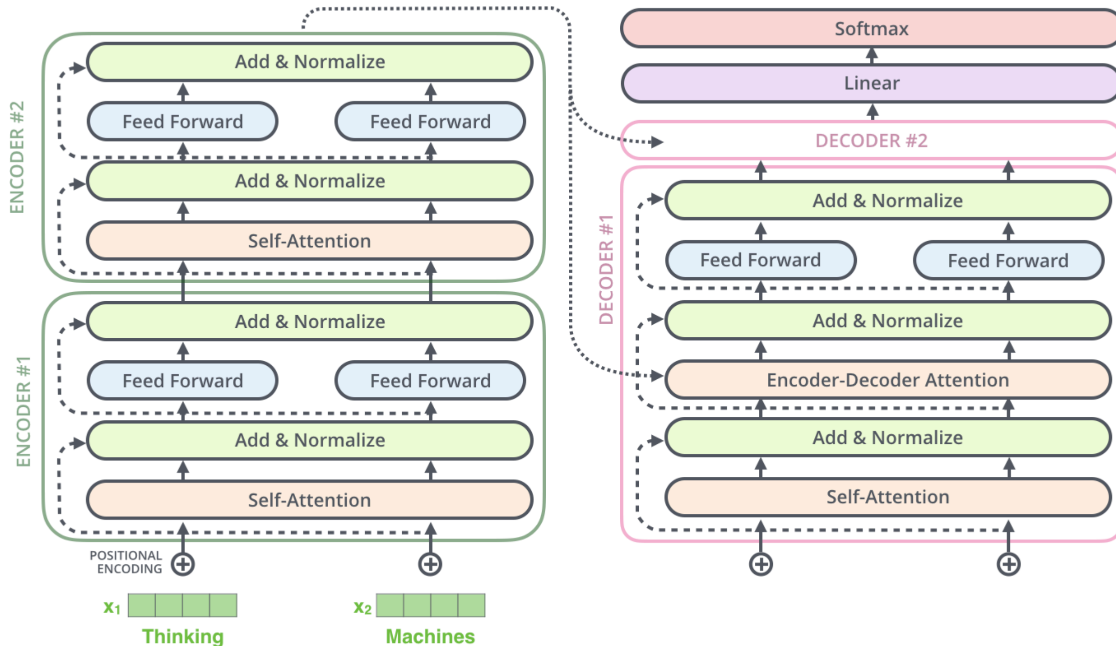


Figure 3.7 The encoder-decoder transformer architecture.

3.4.4.Positional Encoding

Since the order and distance of input tokens matter, the model needs a method to understand the position of the tokens. This is why positional encoding was introduced, which is essentially a vector added to each input. Given a token at position k, the elements of the positional vector P_k of dimension D are defined by:

$$P_k[2i] = \sin(\frac{k}{n^{2i/d}})$$

$$P_k[2i+1] = \cos(\frac{k}{n^{2i/d}})$$

, where $i \in [0, d/2)$ and n a constant scalar.

3.4.5.Audio Applications

While Transformers have been used mostly on NLP applications, there have been successful attempts of extending their functionality in other domains, like audio and image processing.

Vision Transformer (ViT) [20] in an encoder architecture used for image classification. The core idea is that the images are split into 16X16 regions (patches) that are converted into sequential patch embeddings using a learnable matrix projection. These are then used in a similar fashion to the original transformer.

Audio Spectrogram Transformer (AST) [21] is an encoder-only transformer that takes as input the Mel Spectrogram of an audio signal. It closely resembles the structure of ViT, with the differences being the input channels of the image (single channel vs 3-channel used in ViT) and the ability to handle variable lengths. AST outperforms the state-of-the-art CNN models in the test datasets.

Transformers have also been used recently for Automatic Speech Recognition (ASR) with OpenAI's Whisper [22] encoder-decoder transformer model. The input to the model is a 80-channel log-magnitude Mel spectrogram representation of the audio, which is passed through 2 convolutional layers with a filter width of 3 and the GELU activation function. Then sinusoidal position embeddings are then added and the result is fed into the transformer blocks. The model was trained at 680,000 hours of multilingual audio under different tasks: transcription, translation, voice activity

detection, alignment, and language identification. The performance of Whisper's ASR is close to professional human transcribers with only a fraction of a percentage point worse, in word error rate (WER).

4.Methodology

4.1.Overview

Suppose two songs that we wish to identify as cover/non-cover pairs. The first step is to extract the feature representations for both tracks. This will transform the audio tracks into numeric representations that can be used as input to our model. Then, the two inputs are fed separately to the same model, resulting in two outputs. These are essentially vectors representing the tracks in the embedding space of the model, also simply known as embeddings. We calculate the euclidean distance between the two vectors and if the distance is below a set threshold, the songs are considered a cover pair. A visualization is given in the following figure:

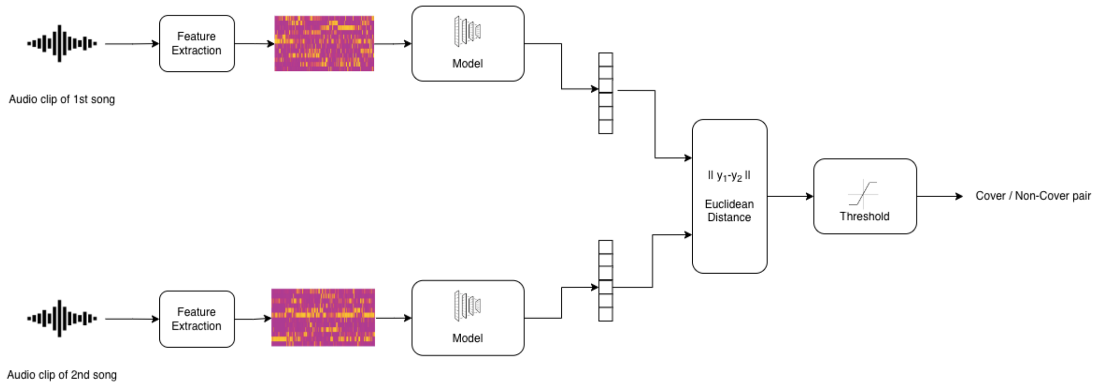


Figure 4.1 Visualization of the cover song identification pipeline

In the embedding space, two songs that are similar will be mapped to points that are closer to each other. This property is also helpful if we intended to rank the songs based on similarity, rather than just outputting a binary prediction.

The distance threshold is a parameter that can be tuned according to the application's needs. A low threshold increases the precision of the decision but lowers the recall, meaning that fewer songs will be identified as cover pairs, but with higher confidence. This could be used in situations where we are not interested in detecting every cover pair, but we require high confidence in those that are actually detected. On

the other hand, if the requirement was to identify as many cover pairs as possible (for example to filter out the obvious non-cover pairs as part of a more sophisticated process) one could opt to use a higher threshold.

4.3. Feature Extraction

As mentioned in previous chapters, HPCPs perform relatively better in music retrieval tasks, since they are able to capture the melody and tonality of the tracks. During the experiments, we verified indeed that these features demonstrated better performance over others. The input of the model is a $12 \times n$ matrix, where 12 is the number of chroma bins and n is the number of HPCP windows. In our application the HPCPs were calculated with a sample rate of 22050 samples per second, and each window containing 2048 samples. We also introduced a 512 sample overlap (hop size) between successive windows. The relationship between the number of windows and the segment size in seconds is given by:

$$t_{segments} = \frac{512 \times n}{22050}$$

Equation 4.1. Relationship between number of HPCP windows and the resulting segment size

For example for $n = 3600$, the total length of the segmented audio is about 85 seconds per segment. Tuning this parameter has a major impact in the performance of the model. Too short length and the model may not be able to capture the features that make up a cover song. That is because the dataset tracks -and cover songs in general- are not perfectly aligned in the time direction. Which means that when the model looks for a pattern between two segments of audio (the original and the cover), these segments need to have at least some overlap. Therefore the segments have to be large enough to account for differences in synchronization. On the other hand if the segments are too large, the inference time increases as the model has to process a bigger input. Also it makes real-time applications more limited.

Furthermore we normalize the representation using z-normalization and scale along the time axis with a factor of 0.1, in order to reduce the size and speed-up the training.

We also perform data augmentation at two different levels: We split the HPCP chromagrams into segments, and train the model with variable segment lengths ranging from 140 to 440 frames, in order to augment the train set and avoid overfitting to specific segment lengths. Moreover, we randomly shift the chromagrams across the octave dimension by 1 to 12 positions, to simulate key transpositions.

4.4. Model Architecture

Convolutional Neural Networks demonstrated the best performance across other choices, for this kind of task. The optimal model consists of 6 convolutional layers + 1 linear layer, as shown in Table 4.1. After each convolutional layer, a dropout and batch normalization layer are added.

Furthermore, at the last convolutional layer we perform what is known as Global Average Pooling [25]. Instead of flattening all the elements of the last layer into a single vector and using this as input to the linear layer, we take the average value at each channel dimension, leading to a fixed-size vector of 256 elements. This technique has several benefits:

- The model can handle images of variable sizes, since the output size, after the global average pooling, is always equal to the number of the output channels of the last layer.
- The model is more robust towards spatial translations of the input.
- The following fully connected layer needs fewer parameters to optimize.

Finally a threshold classifier is used, which takes as input the 256-dimensional embeddings of two songs, calculates the euclidean distance and compares the result with a set threshold. This threshold is adjusted separately, after training the model.

At this point we've established the structure of the network. Next we will describe the loss function used for training.

Table 4.1 Optimal model architecture

Layer no.	**Type**	**Output Channels**	**Kernel Size**	**Padding**	**Stride**	**Activation**
conv_1	conv2d	64	3x3	1	1	ReLU
conv_2	conv2d	64	3x3	1	1	ReLU
max_pool_1	max pooling	-	1x2	1	1	-
conv_3	conv2d	128	3x3	1	1	ReLU
conv_4	conv2d	128	3x7	1	1	ReLU
max_pool_2	max pooling	-	1x2	1	1	-
conv_5	conv2d	256	5x7	1	1	ReLU
conv_6	conv2d	256	5x7	1	1	ReLU
glob_avg_pool_1	avg pooling	-	input_h x input_w	-	-	-
lin_1	linear	256	-	-	-	Linear
Total number of parameters: 3,964,352						

4.5. Loss Function

The model's output is a representation of a song as a fixed size vector in the model's embedding space. We require this representation to have the following property: In this embedding space, covers of the same song must be closer to one another, than any other non related song. This requirement must be represented as a loss function, that the model is trained to minimize. In general the approach that aims to establish a similarity or dissimilarity between inputs, based on a distance metric is called Metric Learning.

4.5.1. Contrastive Loss

Introduced by S. Chopra et al [26], Contrastive loss works on positive and negative examples of similar or dissimilar samples. The model is given pairs of songs (as in figure 4.1) and the distance between them is calculated. Then the loss is given by the following formula:

$$L_{contrastive} = \frac{1}{2N} \sum_{n=1}^{N} \left(y_n {d_n}^2 + (1 - y_n) \max \left(m - d_n, 0 \right)^2 \right)$$

Equation 4.2 The Contrastive Loss formula

where, y_n is 1 if the samples are similar or 0 otherwise, d_n is the distance between the samples, and α is a constant. It is easy to verify that the loss is low when the distance is small between positive samples, and large between negative. The m parameter helps the stability and convergence of the training procedure.

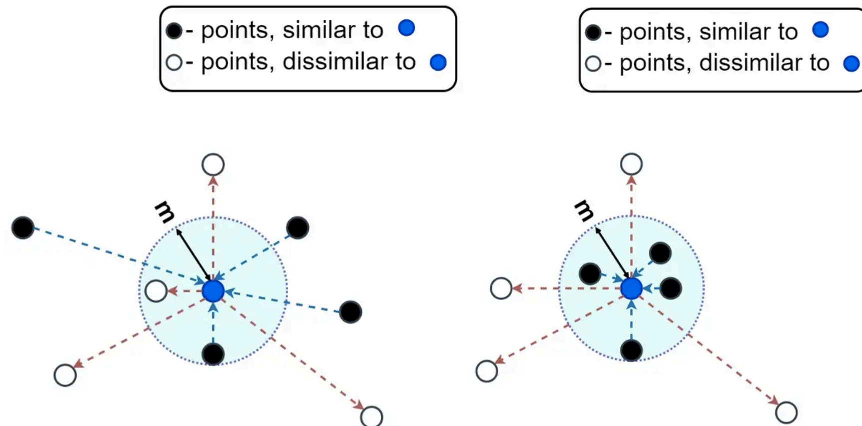


Figure 4.2 Initial and final representation after minimizing the loss function. Notice that the white points are not pushed further away, since they already exceed the margin m.

4.5.2. Triplet Loss

Triplet loss [27] is another metric learning loss, that is also based on the distance between samples. The difference with contrastive loss is that it uses a triplet of data points, consisting of an anchor, a positive example, and a negative example. The model is

trained to ensure that the anchor is closer to the positive example than to the negative example.

Suppose a is an anchor song, p a cover of this song and n any other non-cover song. In the model's embedding space, we'd require the following condition to be true:

$$||x_a - x_n||^2 \geq ||x_a - x_p||^2 + m$$

Equation 4.3 Requirement regarding the anchor, positive and negative examples in the embedding space.

, where x_a, x_p, x_n the embeddings of the songs respectively. Based on this condition the triplet loss is formulated as such:

$$\mathscr{L}_{triplet}(a, p, n) = \max\left(||x_a - x_p||^2 - ||x_a - x_n||^2 + m, 0\right)$$

Equation 4.4 The Triplet Loss

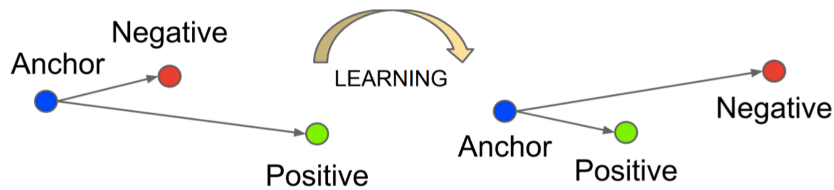


Figure 4.3 Visualization of triplet loss minimization

An intuitive explanation of equation 4.3 is that, for a given triplet, the negative sample must be mapped further away from the anchor point than the positive sample, at least by a quantity m. This is shown in figure 4.3.

The triplet loss differs from contrastive as it takes into account the relative distances of anchor-positive and anchor-negative points, rather than treating them separately. This allows for triplet loss to be less greedy, as it doesn't modify the positive samples distance, when the condition in 4.3 holds true. This allows for higher inter-class variance, in contrast with contrastive loss which tries to map anchor and positive samples to the same point, regardless of the negative samples.

Regarding the choice of triplets, we need to take into account the following facts:

- The possible combinations of anchor-positive-negative song triplets is cubically increased with the number of available songs.
- Not all triplets contribute are beneficial to the training process. In some combinations, the condition 4.3 already holds true, hence the loss is 0 and the model's weights are not modified.
- As the model is trained, triplets that initially violated the condition 4.3 and contributed to the initial training stages, will be sorted out and won't be helpful to the training process anymore.

Essentially as the training goes, the triplets fall into one of the following categories:

- Easy triplets: Sorted triplets that don't violate condition 4.3.
- Semi-hard triplets: Triplets where the negative sample is further away from the anchor than the positive sample, but still below the margin m.
- Hard triplets: Triplets where the negative sample is closer to the anchor than the positive.

A visualization of the above is given in figure 4.4.

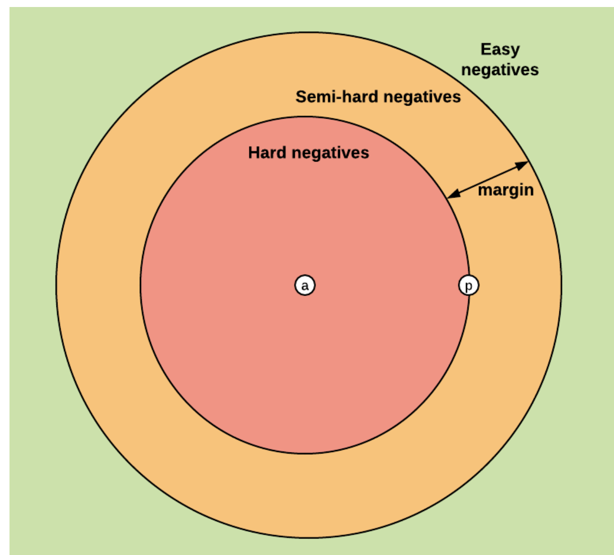


Figure 4.4 Visualization of the different types of triplets (determined by the location of the negative sample).

In our first experiments we used Batch-All online triplet mining. The method's steps are the following:

1. At the beginning of each batch, run the model for all the songs of the batch and output the embeddings.
2. Calculate the distance matrix between all point embeddings ($batchsize \times batchsize$ matrix).
3. Calculate the triplet loss for each anchor-positive-negative pair ($batchsize \times batchsize \times batchsize$ tensor).
4. Apply masking to filter out invalid triplets.
5. Aggregate the losses into a single value.
6. Use this value for updating the model's weights.

The key here is that all valid triplets that violate the condition are used (semi-hard & hard triplets).

We also used Batch-Hard mining, during the last stages of training, which takes into account only hard-triplets when calculating the loss. It's recommended not to start training directly with batch-hard triplets, because the model may converge to a local minimum, mapping the all inputs to a single point.

4.5.3. Angular Loss

The final model was trained using a variation of Triplet Loss called Angular Loss [28]. This loss aims to improve robustness against feature variance and provide better convergence properties. One issue of the standard triplet loss is that the parameter m is a global margin used to separate the clusters. This may pose a problem since the intra-class distance can vary in real life applications. Angular Loss consists of minimizing the following hinge loss:

$$\mathscr{L}_{angular}(a,p,n) = \max\left(||x_a - x_p||^2 - 4\tan^2 a ||x_n - x_c||^2, 0 \right)$$

$$x_c = \frac{x_a + x_p}{2}$$

Equation 4.5 The Angular Loss

The parameter a is tuned between 36° and 55°. The authors of the paper argue that the gradients of the loss w.r.t. x_a, x_p, x_n depend on all three points simultaneously, leading to more robust results.

4.6. Evaluation

The evaluation metrics are divided into two categories: ranking and classification. Ranking metrics are used to evaluate the performance of the model at sorting songs based on similarity. In an actual scenario, a user would provide a query song against a database, to retrieve a list of most relevant (cover) songs. On the other hand, classification metrics are used to evaluate the ability of the model to classify pairs of songs as cover/not-covers, and can be used to tune the classifier according to the application needs.

The evaluation is performed on the test dataset described in 5.1.

4.6.1. Accuracy, Precision, Recall, F1 Score

We can evaluate the performance of our model in classifying cover/non-cover pairs, using the standard classification metrics:

$$\text{Accuracy} = \frac{\text{True Positives} + \text{True Negatives}}{\text{Total}}$$

$$\text{Precision} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Positives}}$$

$$\text{Recall} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Negatives}}$$

$$\text{F1} = 2 \cdot \frac{\text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}}$$

Equation 4.6 Classification scores

4.6.2.ROC Curve

The ROC curve is a visualization of how the true positive rate (TPR) and false positive rate (FPR) change, depending on the value of the classifier's threshold. A common behavior of this curve is given in the following figure:

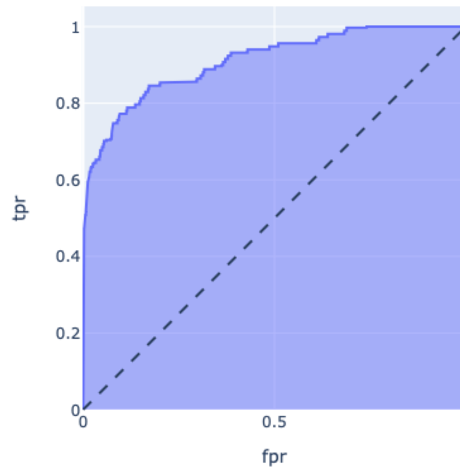


Figure 4.5 The ROC curve.

Minimizing the distance (threshold) in the embedding space, at which two songs are considered covers, results in a smaller TPR but also smaller FPR, making the model more precise at the cost of recall. Increasing the value relaxes the constraint, allowing the model to identify more songs as covers, increasing also the false positives. We can use the ROC curve to set the threshold value according to the application needs.

4.6.3. Precision-Recall Curve.

To generate the Precision-Recall curve, we calculate these quantities on different cutoffs. The result is a graph like the following:

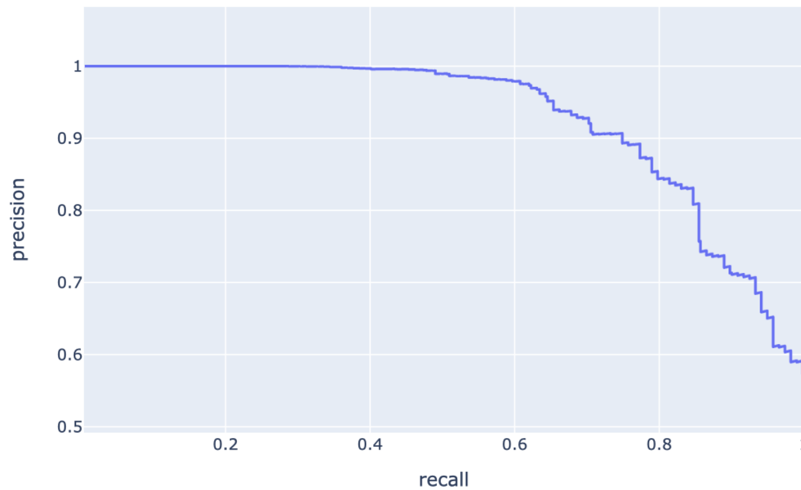


Figure 4.6 The Precision-Recall curve.

4.6.4. Mean Average Precision

Supposed a set of query songs Q. Mean Average Precision is defined in the following manner. For each song $i \in Q$ we calculate the distance in the embedding space from every other song $j \in Q - i$, and sort by ascending order. The Average precision is calculated as:

$$\mathrm{AP}(i) = \frac{\sum_j \frac{\mathrm{C}(i,j)}{\mathrm{rank}(j)}}{\sum_j \mathrm{C}(i,j)}$$

Equation 4.7 The Average Precision metric

Where C(i,j) = 1 if j is a cover version of i, else 0, and rank(j) the position of j in the sorted vector. The MAP is defined as the mean value of AP(i) for all songs $\in Q$.

4.6.5. Mean Reciprocal Rank

The Mean Reciprocal Rank is a ranking metric that takes into account the order of the results returned by the model, for a number of given queries. To calculate the MRR, we use the following process:

- For each song i in the dataset, use it as a query and calculate the relative distances with the other songs.
- Sort the songs by ascending distance.
- rank_i is the position of the first cover song in the sorted results. For example 1 if the closest song is a cover, 2 if the second closest is a cover, and so on.
- The Reciprocal Rank for the query song i is given by:

$$\frac{1}{\mathrm{rank}_i}.$$

- Finally we calculate the mean value for all query songs with:

$$\frac{1}{|Q|}\sum_{i=1}^{|i|}\frac{1}{\mathrm{rank}_i}$$

  where Q is the size of the query song set.

4.6.5. Precision at 10

For P@10 we calculate the distance between a song i ∈ Q and every j ∈ Q − i. We sort the distance vector in ascending order, and keep only the first 10 elements. Then we count the percentage of cover songs and average for all songs i ∈ Q.

4.6.6. Mean Rank of 1st identified cover

Mean Rank of 1sr identified cover (MR1) is defined as the average position of the 1st correctly identified cover song over all query songs.

4.7. Implementation Details

The proposed methodology was implemented in Python using the following libraries:

- The chroma features were extracted from audio using the Essentia Python library [29].
- The models were developed using PyTorch [30] and PyTorch Metric Learning [31].
- For visualization we used Plotly [32] and Matplotlib [33] .

The models were trained using a Tesla A100 graphics card.

5. Experiments

In this chapter we discuss the experimental results of our methodology. We evaluated the model in two different datasets. Since the number of non-cover pairs greatly exceeds the number of cover pairs, we demonstrate the results both on balanced and unbalanced versions of the datasets.

The unbalanced dataset is created by considering all possible pairs in the dataset. The balanced dataset is created by randomly sampling anchor-positive-negative triplets from the dataset.

In chapters 5.5 and 5.6 we present, for comparison, the results of an LSTM and Visual Transformer respectively. The models were trained and evaluated using the same data, input and loss function.

5.1. Datasets

Various cover song datasets were used for training and evaluation of the models.

5.1.1. Covers80

The *Covers80* dataset [23], developed by the Laboratory for the Recognition and Organization of Speech and Audio of Columbia University, consists of 80 songs, each performed by two different artists. The songs are available in wav format, offering the flexibility to extract any representation. This dataset is often used for benchmarking in relevant works, therefore we decided to use it as such, for evaluation. This also offers a baseline for comparing our approach to other published works.

5.1.2. Covers1000

The *Covers1000* dataset [24], published by Chris Tralie et al, is another curated dataset that contains 395 groups of cover songs. The dataset doesn't provide the raw audios, but the following higher-level spectral representations instead:

- MFCCs
- HPCPs
- CREMA
- CENS
- Beats

These features have been extracted with *Librosa* and *Essentia* libraries. We used this dataset for training the model.

5.1.3. Custom Dataset

To increase the training samples for our model, we proceeded to create a custom dataset of covers songs. The choice of songs was assigned to 10 annotators with the following guidelines:

- A cover song is a different performance of the original song. It may be slower/ faster, have different instrumentation, be live/unplugged or be performed by a different artist of different gender.
- The dataset should consist of various genres of music. We try not to focus over a specific genre in order to avoid overfitting the model.
- The quality of the included songs should be as good as possible. Therefore avoiding songs with unrelated intros or blank spaces with speeches within the song (as is usually done in some video clips).
- Attention should be given to the synchronization and duration of the original and cover songs. For better results, the difference in synchronization should not exceed 60 seconds.P

This dataset consists of 208 groups, each containing the original song and up to 4 cover versions. The majority of the dataset was used during the training phase, and a part of it for evaluation.

We also evaluate on a Greek dataset consisting of 26 unique songs with up to 5 cover versions each. The total number of tracks is 62. The dataset consists of greek songs, as well as cover versions of english songs performed by greek artists.

5.2. Results on Covers80 test set

Our model achieved an Accuracy score of 80.9% (balanced) and 93.2% (unbalanced) on the Covers80 dataset, when evaluating segments of 90 seconds, and full songs respectively. The complete scores are shown in table 5.1.

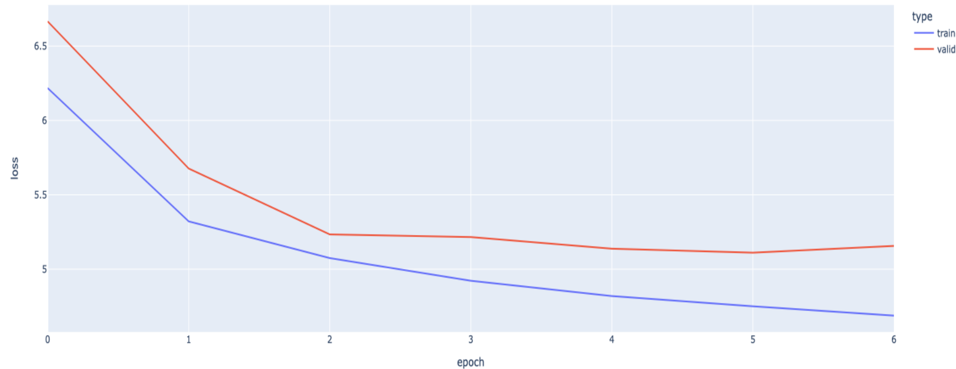


Figure 5.1 Training and Validation loss per epoch.

Table 5.1 Classification scores on the Covers80 dataset.

Class	Precision	Recall	F1	Support
Non-covers (balanced)	75.03%	92.85%	82.99%	16400
Covers (balanced)	90.67%	69.09%	78.41%	16400
Non-covers (unbalanced)	99.78%	93.33%	96.48%	26554
Covers (unbalanced)	6.6%	69.66%	12.06%	178

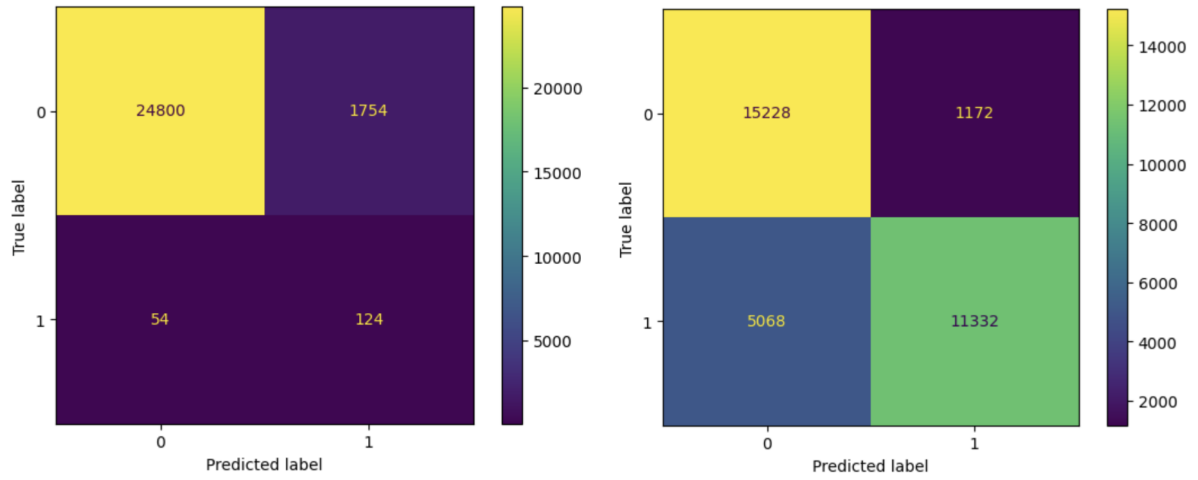
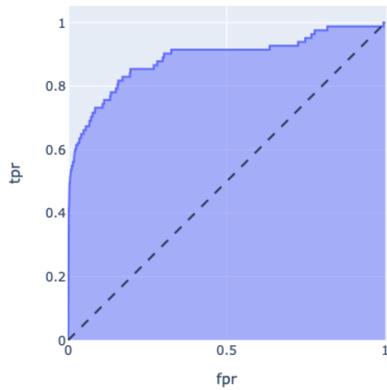


Figure 5.2 Confusion matrices (left: unbalanced dataset, right: balanced dataset)



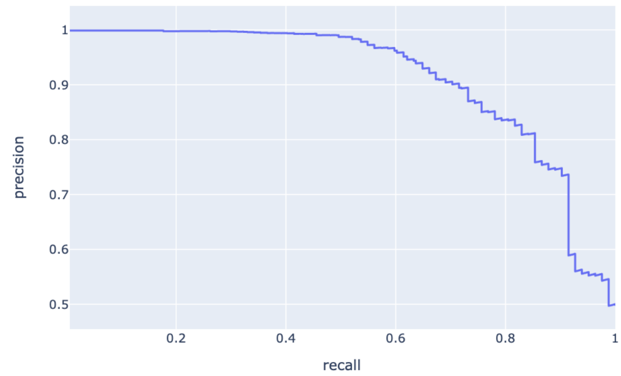
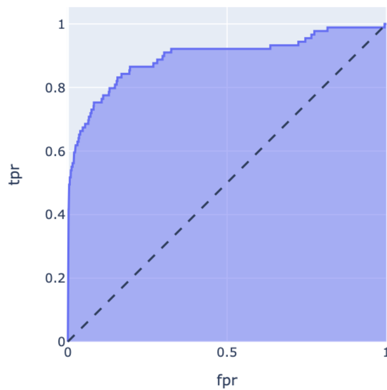


Figure 5.3 Covers80 (balanced) - ROC and Precision-Recall Curves



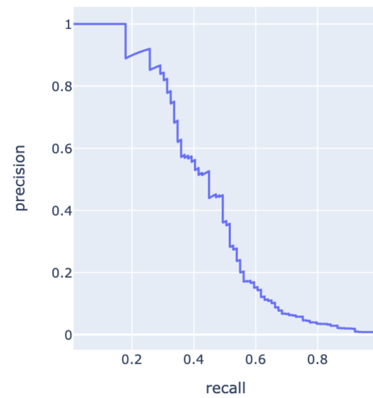


Figure 5.4 Covers80 (unbalanced) - ROC and Precision-Recall Curves

Table 5.2 Ranking scores on the Covers80 dataset.

Mean Reciprocal Rank	Mean Average Precision	MR1	P@10
0.5908	0.5823	17.46	0.076

5.3. Results on Custom test set

The model performed better on the Custom test dataset, with an Accuracy score of 90% (balanced) and 92% (unbalanced).

Table 5.3 Classification scores on the Custom dataset.

Class	Precision	Recall	F1	Support
Non-covers (balanced)	85.61%	96.04%	90.52%	7300
Covers (balanced)	95.49%	83.86%	89.3%	7300
Non-covers (unbalanced)	99.7%	92.96%	96.21%	5172
Covers (unbalanced)	16.12%	83.33%	27.02%	84

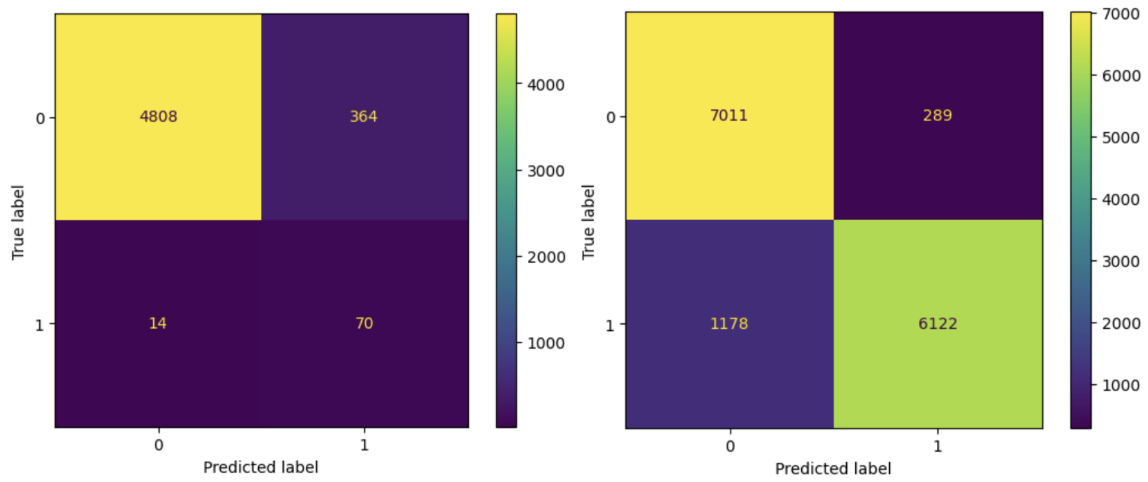
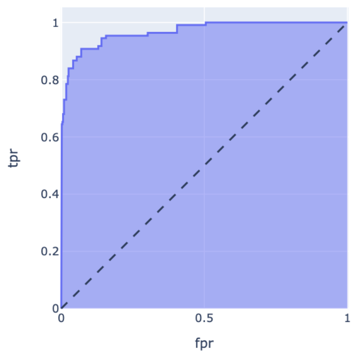


Figure 5.5 Confusion matrices (left: unbalanced dataset, right: balanced dataset)



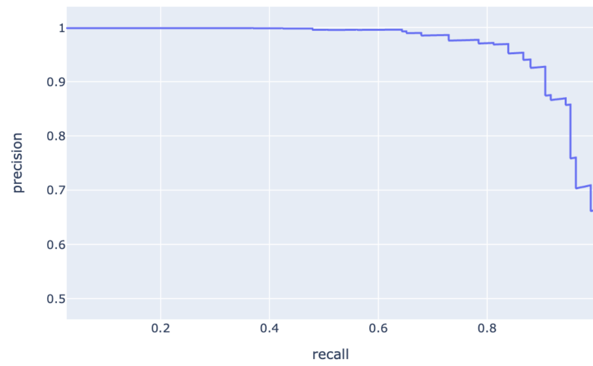
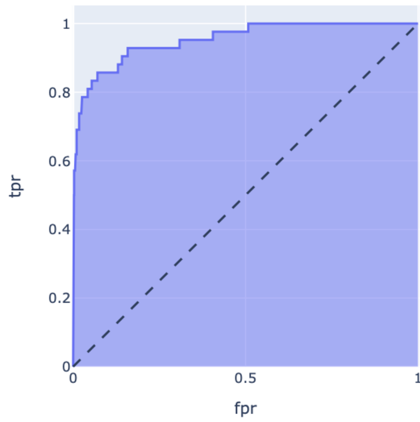


Figure 5.6 Custom Dataset (balanced) - ROC and Precision-Recall Curves



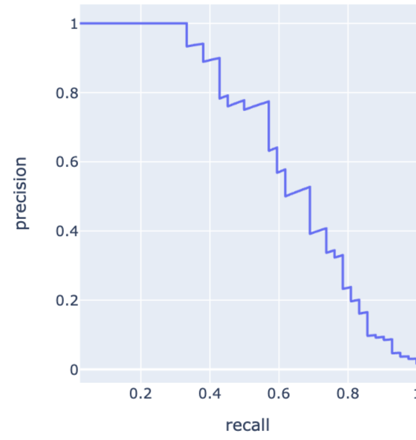


Figure 5.7 Custom Dataset (unbalanced) - ROC and Precision-Recall Curves

Table 5.4 Ranking scores on the Custom dataset.

Mean Reciprocal Rank	Mean Average Precision	MR1	P@10
0.8474	0.8365	2.17	0.101

5.4. Results on Greek covers

For this experiment, we include cover songs that are performed by greek artists. The covers in Greek are based on original Greek or foreign songs. The model, despite not being trained on any Greek song, seems to perform well on this dataset as well, since it relies mostly on the song's melodies rather than the actual lyrics.

Table 5.5 Classification scores on the Greek dataset.

Class	Precision	Recall	F1	Support
Non-covers (balanced)	93.04%	89.16%	91.06%	6000
Covers (balanced)	89.6%	93.33%	91.42%	6000
Non-covers (unbalanced)	99.8%	90.39%	94.86%	3436
Covers (unbalanced)	22.89%	94.23%	36.84%	104

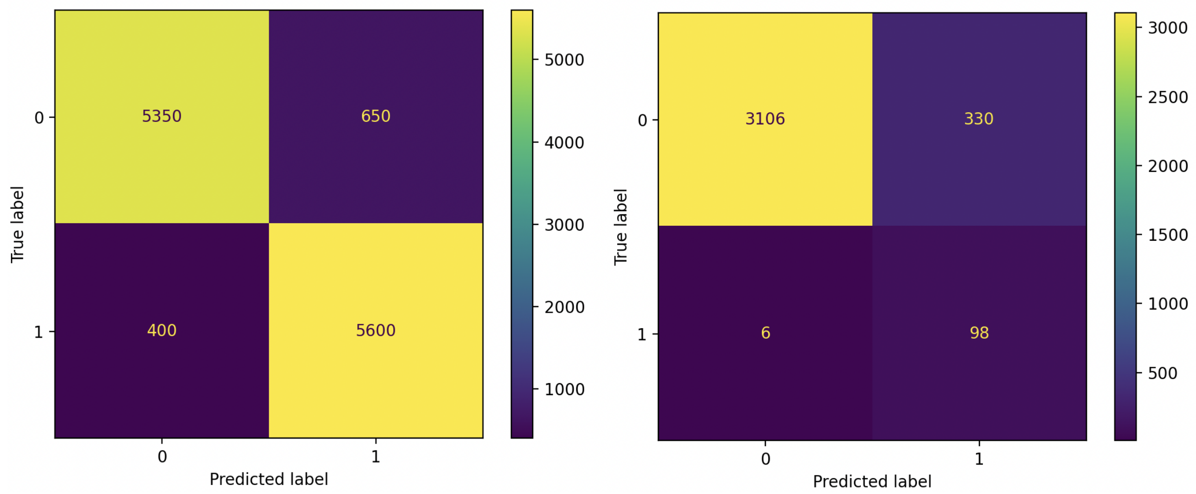


Figure 5.8 Confusion matrices (left: unbalanced, right: balanced)

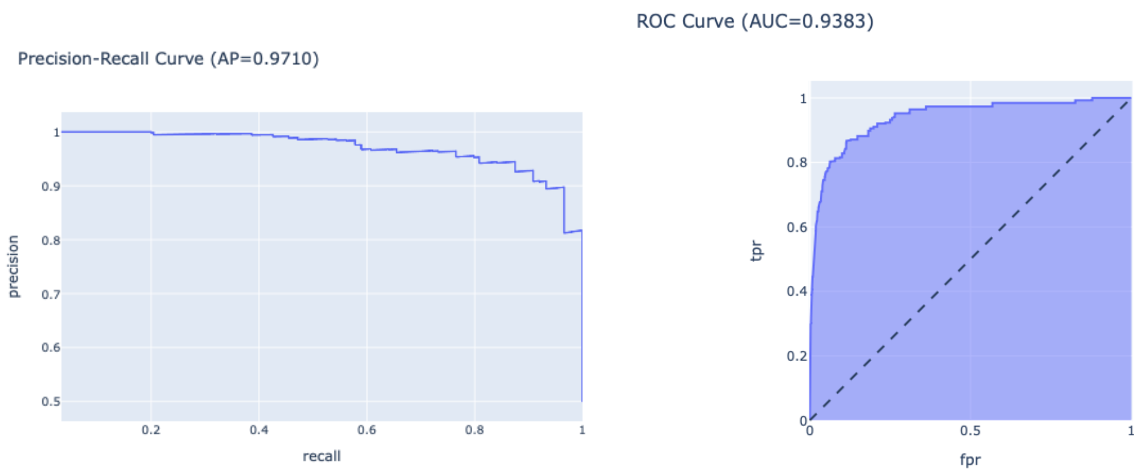
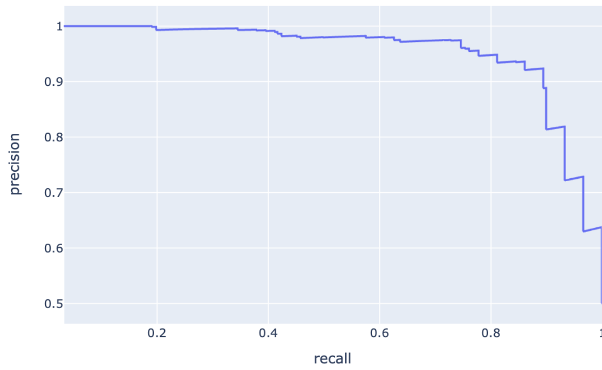


Figure 5.9 Greek Dataset (balanced) - ROC and Precision-Recall Curves



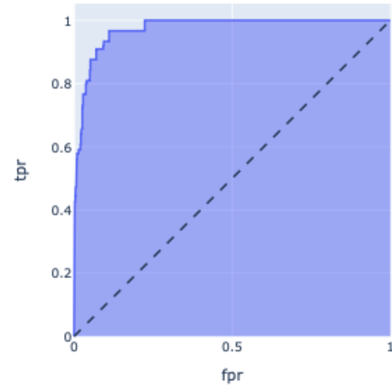


Figure 5.10 Custom Dataset (Full songs) - ROC and Precision-Recall Curves

Table 5.6 Ranking scores on the Greek dataset.

Mean Reciprocal Rank	**Mean Average Precision**	**MR1**	**P@10**
0.8209	0.7958	2.11	0.169

5.5. LSTM results

The input to the LSTM model is a sequence of 12-dimensional vectors, where each vector is essentially the HPCPs.

The network consists of 3 hidden layers, with a hidden state size of 256. The outputs of the last LSTM layer at every timestep are aggregated using average pooling. The output is sent to a 2-layer FF network with ReLU activation.

The model performed poorly, compared to the CNN model, achieving an F1 score of 60.5% on the Covers80 dataset (balanced). The results are demonstrated below.

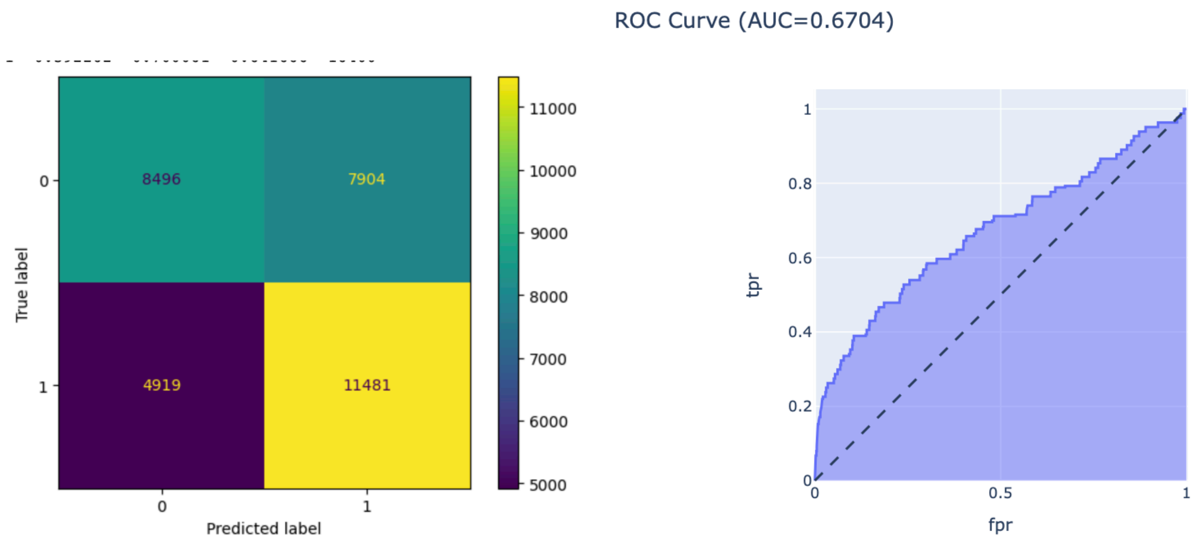


Figure 5.11. The results of the LSTM model on the Covers80 test set.

Table 5.7. Classification scores for LSTM on the Covers80 test set.

Precision	Recall	F1	Support
63.33%	51.8%	56.99%	16400
59.22%	70%	64.16%	16400

Table 5.8. Ranking scores for LSTM on the Covers80 test set.

Mean Reciprocal Rank	Mean Average Precision	MR1	P@10
0.2698	0.2683	49.75	0.041

5.6. Visual Transformer results

Due to the hardware limitations as well as the large data required to train a transformer, the best accuracy achieved for the covers80 balanced dataset was around 66.62%. The full results are available in the tables below. Note that the transformer was not pre-

trained on any dataset. In general using pre-trained models tends to yield better results. The model could potentially improve if trained with more data, using a lower level representation like the Mel-Spectrogram.

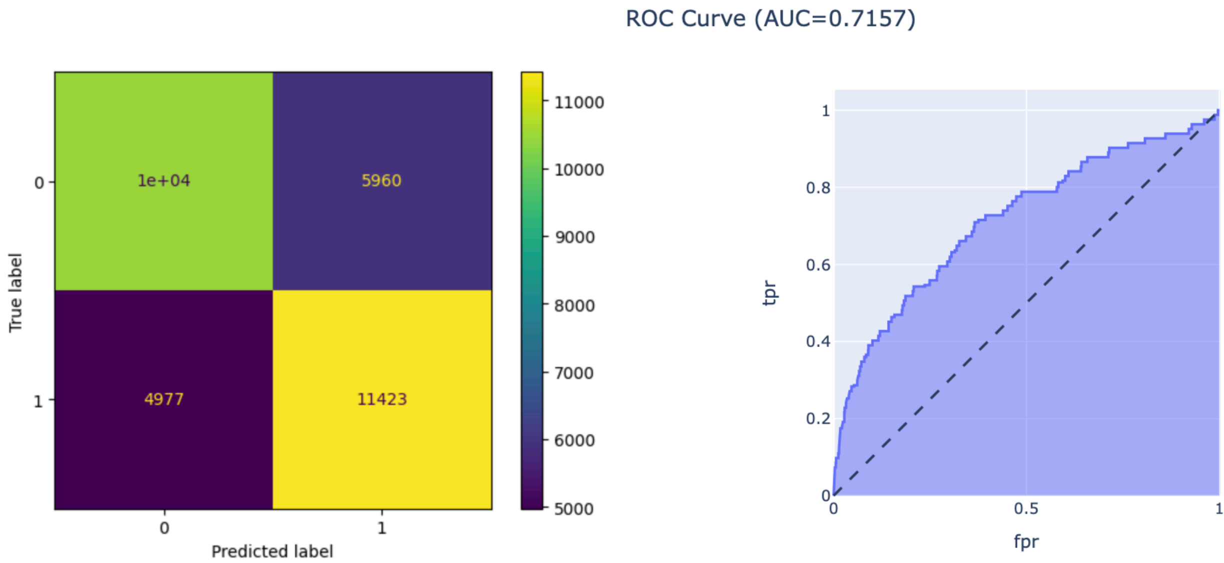


Figure 5.12. The results of the ViT model on the Covers80 test set.

Table 5.9. Classification scores for ViT on Covers80 set.

Precision	Recall	F1	Support
67.77%	63.65%	65.62%	16400
65.71%	69.65%	67.62%	16400

Table 5.10. Ranking scores for ViT on the Covers80 test set.

Mean Reciprocal Rank	Mean Average Precision	MR1	P@10
0.1589	0.1531	45.44	0.038

5.7. t-SNE Visualization

t-distributed stochastic neighbour embedding (t-SNE) [34] is a non-linear dimensionality reduction technique for visualizing high dimensional data. The algorithm works by projecting the original data onto a lower-dimensional space, such that similar data points are mapped to nearby points. It does so by creating two probability distributions in the high and lower dimensions. In the high-dimensional space, t-SNE calculates the probability that a data point will be chosen as the "neighbour" of another data point. This probability is based on the similarity between the two data points. In the low-dimensional space, t-SNE defines a similar probability distribution, where the probability of two data points being neighbours is based on their distance in the low-dimensional space. The t-SNE algorithm then adjusts the positions of the data points in the low-dimensional space iteratively, in an attempt to minimize the divergence between the two distributions.

t-SNE is able to preserve the distance between neighbour points, and can be used to visualize cover songs. However the dimensions themselves are not interpretable, and absolute distances are not correlated with inter-song similarity.

In the following diagrams, points that have the same color represent covers of the same song, and ideally should be closer together. These have been generated by creating the embeddings for the test and train dataset, using the best CNN model.

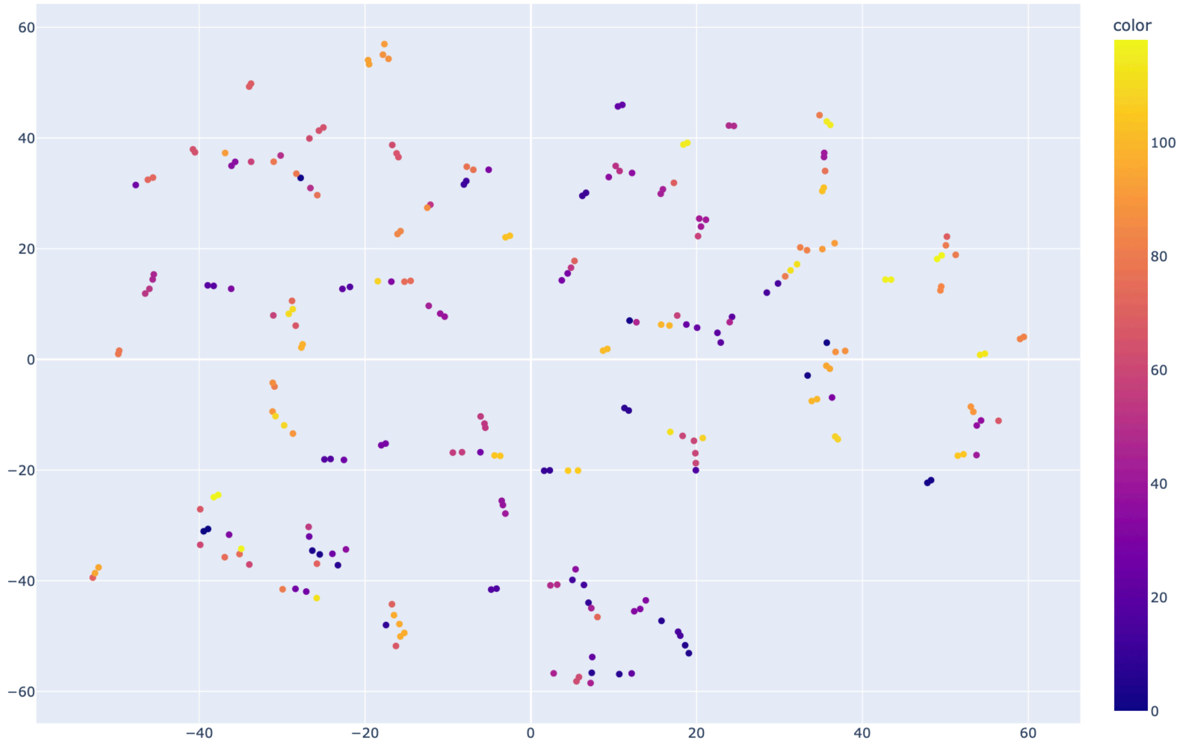


Figure 5.13 t-SNE plot of the test dataset

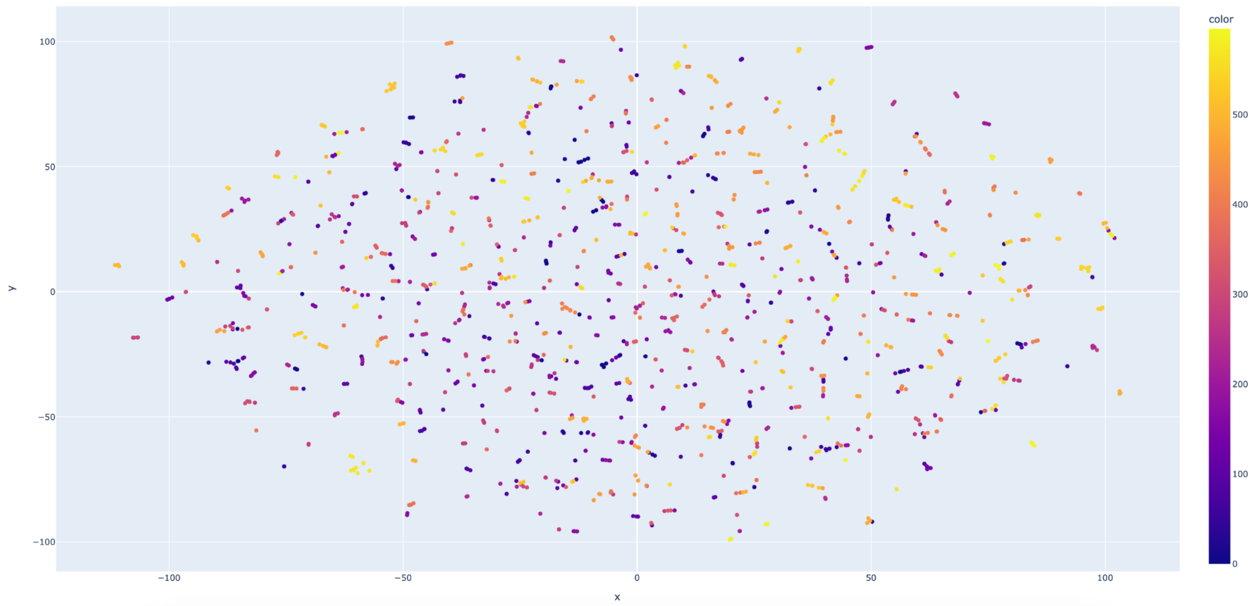


Figure 5.14 t-SNE plot of the train dataset

5.8. Inference time

The amount of time it takes for the model to make predictions on new data is a crucial factor to consider when using the model in real-life applications. If the inference time is too long, it can slow down the performance of the system or application in which the model is being used. On the other hand, if the inference time is short, it can enable the model to make predictions or decisions quickly and efficiently, which can be important for applications that require fast response times or real-time processing, such as cover song identification from a live recording.

Another factor which affects inference time, is the device on which the model runs. In general GPU's allow for parallelization of the operations that are made during the prediction calculation, resulting in a speedup of several magnitudes compared to CPU inference. This comes at a cost, since GPUs are way more expensive than CPUs and require more power.

We tested our model on various song lengths using either CPU or GPU for inferencing, to determine the feasibility of the model for real-life applications. The CNN/LSTM models were tested using an Apple M1 Pro 10-core CPU/16-core GPU. The Visual Transformer was benchmarked on a high-end Tesla A100 GPU. It is shown that the runtime of the proposed CNN model is quite acceptable, since a batch of 512 3-minute songs would require about 30ms to process on a low-end GPU. This benchmark does not include the time required to generate the chroma features from the raw audio.

Table 5.11 Inference time benchmarking.

Device	**Segment size (seconds)**	**Segment size (input image dimensions)**	**Avg. time per sample (milliseconds)**			**Total time for 512 samples (milliseconds)**		
			CNN	**LSTM**	**ViT**	**CNN**	**LSTM**	**ViT**
CPU	15	12 x 65	**1.2136**	1.1661	-	**621.36**	597.04	-
	20	12 x 90	**1.8997**	1.6250	-	**972.64**	832	-
	30	12 x 130	**2.7294**	2.3839	-	**1397.45**	1220.55	-
	60	12 x 260	**5.5028**	5.3672	-	**2817.43**	2748	-
	120	12 x 400	**9.3708**	8.1955	-	**4797.84**	4196.09	-
	180	12 x 780	**19.636**	18.2107	-	**10053.63**	9323.87	-
GPU	15	12 x 65	**0.0208**	0.0224	0.1226	**10.64**	11.47	62.77
	20	12 x 90	**0.0199**	0.0228	0.1237	**10.18**	11.67	63.33
	30	12 x 130	**0.0245**	0.0350	0.1253	**12.54**	17.92	64.15
	60	12 x 260	**0.0252**	0.0311	0.1277	**12.90**	15.92	65.38
	120	12 x 400	**0.0382**	0.0483	0.1276	**19.55**	24.73	65.33
	180	12 x 780	**0.0572**	0.1283	0.1942	**29.28**	65.69	99.43

Table 5.12 Accuracy vs segment size for CNN model.

Segment size (seconds)	**Accuracy (Covers80 - balanced)**
15	58.03%
20	63.20%
30	67.58%
60	75.51%
120	78.79%
180	79.25%

Table 5.13. Comparison between different methods on the test sets.

Model	**MAP**	**MRR**	**MR1**	**P@10**	**F1**
Covers80					
CNN	**0.5823**	**0.5908**	**17.46**	**0.076**	**80.70%**
LSTM	0.2683	0.2683	49.75	0.041	60.05%
ViT	0.1531	0.1589	45.44	0.038	66.62%
Custom Dataset					
CNN	**0.8365**	**0.8474**	**2.17**	**0.101**	**89.91%**
LSTM	0.4177	0.4289	20.34	0.055	63.48%
ViT	0.2582	0.2710	18.90	0.045	60.41%
Greek Dataset					
CNN	**0.7958**	**0.8209**	**2.11**	**0.169**	**91.24%**
LSTM	0.4031	0.4773	12.06	0.084	67.32%
ViT	0.2994	0.3542	12.35	0.088	66.90%

5.9. Observations

The above experiments show that our model demonstrates good performance in identifying similar/dissimilar songs. The generated embeddings do capture said similarities and can be used in conjunction with a simple threshold classifier to identify cover pairs. The distance is also easily tuneable, to aim either for high precision or high recall, depending on the application.

Regarding the training data, the model seems to achieve this performance with a reasonable number of samples. This is possible due to the nature of the chroma features, which are already a compact representation of audio with characteristics that are suited for this task. More samples could potentially boost the performance further, or be used with a different, more generic representation, such as the Mel-Spectrogram.

Another interesting property of our methodology has to do with the audio input. The fully convolutional network architecture eliminates the limitation for fixed size inputs, and therefore the model can handle variable lengths at inference. This is extremely useful in real-life applications for live recordings that may last a few seconds, or in cases where the audio tracks are not synchronized.

Synchronization of audio tracks is a potential issue that arises when checking similarity between songs. For example a cover song may include a longer intro than the original, different tempo or crowd cheers in case of live performances. This causes the tracks to be misaligned, and may lead to performance degradation. Fortunately the nature of fully convolutional networks is such that the discovered features are translation invariant. This means that the network searches for local features across the image, irrespectively of the absolute position of these features in the input space, thus reducing the need for the tracks to be synchronized.

The model seems to be invariant of the lyrics or language used, as shown in the greek dataset experiment, since it focuses primarily on melody.

Furthermore, we attempted to rank the songs based on distances from other songs and determine if the ranking can be used as a sorting mechanism to search for similar songs. We found out that songs that are in general more soft, melodic and contain primarily piano, are placed further apart than aggressive rock songs with

multiple loud instruments. However we did not find a solid pattern, as there are cases where completely different songs are placed close enough than others, without an obvious explanation. This is because the embedding dimensions themselves are hard to interpret and we cannot be sure about the exact way the model associates covers songs. In songs that have a characteristic, distinct melody, the model seems to achieve higher precision. In the following figures, we choose a soft-piano song (gangsta's paradise piano cover) and a live rock performance (Killing in the name) and sort all the other database songs, based on ascending distance. We can see that similar styles are more likely to be placed closer in the embedding space.

Song	Similarity
Coolio - Gangsta's Paradise [piano cover by Georgiana (GlowGirls)] [PHKSP-1nfxw]	100.0%
Gangsta's Paradise - Coolio ｜ Robert vs	77.0%
Coolio - Gangsta's Paradise ｜ Piano cover [yXvouHaGa1w]	67.5%
Coolio - Gangsta's Paradise (feat	58.4%
Alec Benjamin - Let Me Down Slowly (Lyrics) [1gvOv-W_6Go]	54.8%
Gangsta's Paradise - UMC feat	54.4%
Alec Benjamin - Let Me Down Slowly (cover by J	52.7%
Σύννεφα με Παντελόνια - Σύννεφα με Παντελόνια (official video clip) [ocRgOk3wjCo]	51.8%
Nightwish - Ever Dream [uuf3cVS_y9c]	51.1%
Ever Dream Cover - Nightwish (MoonSun) [M3HhOOVGV6U]	51.0%
I Ain't Worried (ACAPELLA) From ＂Top Gun： Maverick＂ - OneRepublic [nbYVLRLs1v4]	50.9%
Lena papadopoulou-Thelo trella [qaNOdlbAaGo]	47.5%
Oliver Tree - Life Goes On (Lyrics) [LnavzVJctAw]	47.5%
Toquel-Pistoli(cover by daplawr) [lj2qy7SD16Q]	46.5%
The Cars - Heartbeat City [ewt2ybYEI08]	45.1%
HIM - It's All Tears (Unplugged) [mR-7HxwiWhc]	44.3%
Oliver Tree - Life Goes On (Female Cover) [YJf65k9O-LU]	43.9%
Shakira, Ozuna - Monotonía (Cover Mileva) [YIo_p2fsuOs]	42.2%
Abba - Gimme Gimme Gimme (Live cover by Bandix) [Jz2jyVSgBS0]	42.0%
Billie Eilish, Khalid - lovely (Lyrics) [ZrsIQCLtNpE]	42.0%

Figure 5.15 Acoustic or calm songs, seem to be grouped closer together.

Rage Against The Machine - Killing In The Name - 1993 [8de2W3rtZsA] 100.0%
Audioslave - Killing In The Name Of (Live 8 2005) [sQFXKOPJOYM] 80.0%
Rage Against The Machine - Killing In the Name (Official HD Video) [bWXazVhlyxQ] 73.2%
Rage Against The Machine - Bombtrack (Live Soundstage performance - 1992) [kfbwbwXNenw]68.3%
Disturbed - Killing in the Name of (Rage Against the Machine Cover) [BvmPtrQOSmQ] 66.0%
Brass Against - Bombtrack (Rage Against the Machine Cover) [_XOkQlebapw] 63.7%
Jacoby Shaddix of Papa Roach Joined Disturbed On Stage At Rock On The Range [B1pNddYo3uw]
63.7%
Rage Against The Machine - Bombtrack (Audio) [hVck6DkOi38] 62.0%
Disturbed - Facade (Russian Cover by Alex_PV) [hvC0YXnoLSI] 61.8%
GLEE - Ice Ice Baby (Full Performance) HD [33vZFEpy4AU] 60.4%
Μάκης Δημάκης - Τι Τι, Ε (Christaf & Dim Xatzis Production) [frt7udx9LjU] 59.8%
Metallica - Enter Sandman (Live in Mexico City) [Orgullo, Pasión, y Gloria] [87by1DjfxLw] 57.6%
Rage Against The Machine - Bombtrack (Acoustic Cover by United Rage) [7BFwMbWPz4I] 57.0%
CAN'T STOP THE FEELING! (from DreamWorks Animation's " TROLLS ") (Official Video) [ru0K8uYEZWw]
56.6%
GIMS - Mi Gna ft 56.5%
Coyot - Devil In Disguise (Official Audio) [AoKPzUvwugE] 56.3%
Till I Collapse [Obim8BYGnOE] 56.0%

Figure 5.16 Rock songs with more complex instrumentation are placed closer together.

6. Web Application

The model was deployed as part of a web application that allows the user to upload songs and identify potential cover songs.

6.1. Platform overview

The platform allows the user to upload songs using the UI, and compare them with others present in the database. The comparison can be either to check directly if 2 songs are a cover pair, or rank songs based on similarity.

6.1.1. Upload songs

The user submits the YouTube URL of the desired song using a form in the Home page. The platform then downloads the song from YouTube, extracts the chroma features and calculates the embeddings using the model. The embeddings are saved to the database along with the song's metadata for future reference.

6.1.2. Manage song database

The platform offers the standard CRUD (create-read-update-delete) functionalities to the user, allowing them to manage existing songs and view them in a table format along with their metadata.

6.1.3. Cover check

The user can select two songs in the database and check whether they are covers of the same song, using the pre-trained model. The platform returns the binary output and the relative distance between the songs in the embedding space.

6.1.4. Rank songs

The user can select a query song and retrieve a sorted list of all the other songs, based on the euclidean distance in the embedding space. Songs that are closer to the query song will appear on top.

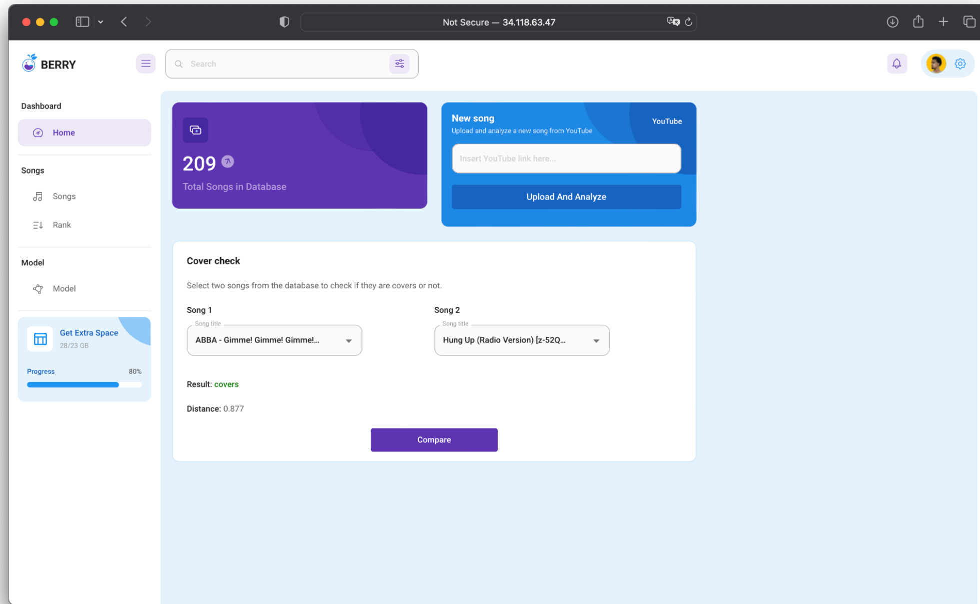


Figure 6.1 The application's home page. The user can upload songs from YouTube or check whether two songs are covers or not.

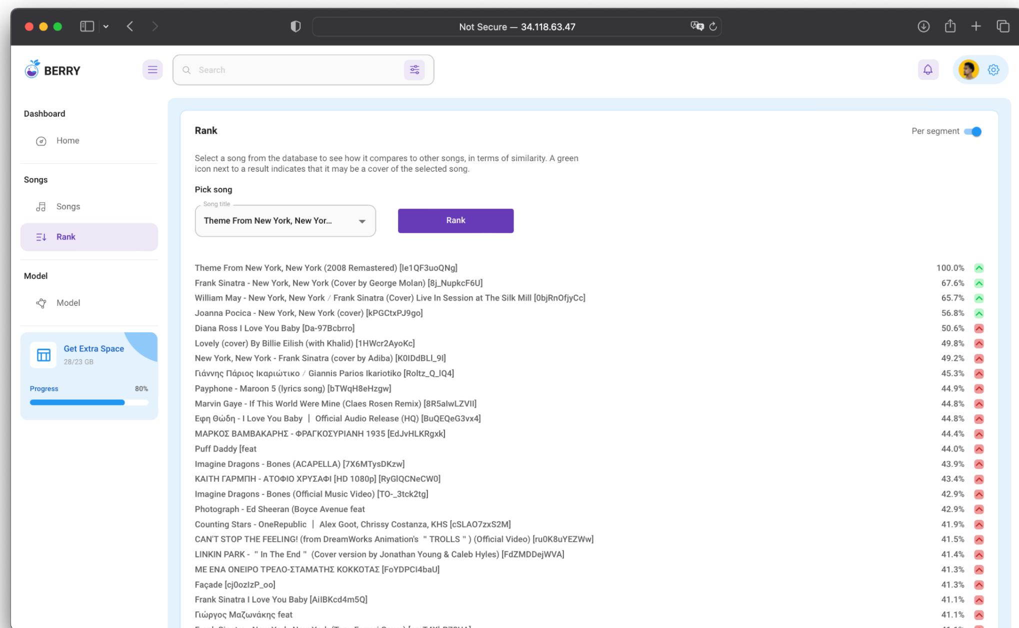


Figure 6.2 The user selects a song and retrieves a sorted list based on similarity. The icon next to each result indicates a potential cover song (green is positive, red is negative).

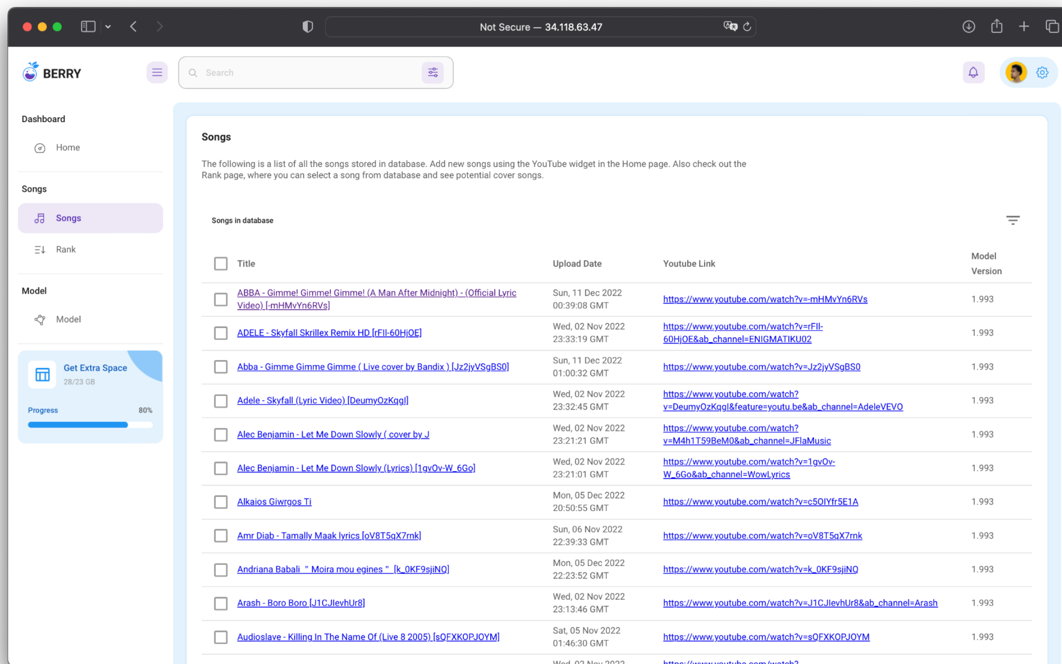


Figure 6.3 Song management page.

6.2. Architecture

The application's frontend is built using Javascript and React, and served using an embedded Node server.

The backend is built using Python and Flask and the model is served using Pytorch (CPU inference). Regarding storage, the song's metadata and embeddings are saved in a MongoDB database, while the song's chroma features are stored in a Google Cloud Bucket due to the amount of space required.

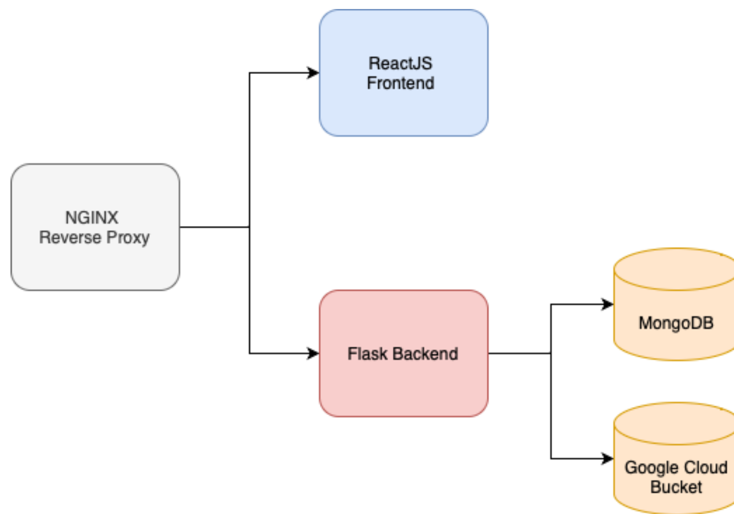


Figure 6.4 Platform architecture.

7. Conclusions

Due to major advancements in the recent years, Deep Learning has been wildly adopted for a variety of applications, including Speech Recognition, Speech Synthesis and Music Information Retrieval. In this thesis we presented a complete framework for automatically identifying whether two recordings are covers of the same song, using Deep Neural Networks. This task is known as Cover Song Identification (CSI).

We've experimented with various models and architectures, with Convolutional Neural Networks being the model of choice for the given task, since it outperforms the LSTM and Vision Transformer models for the given amounts of data and available hardware.

Commonly used in Music Information Retrieval tasks, Harmonic Pitch Class Profile features turned out to be a good selection for the task of CSI as well. These features are closely related to melody, and can be naturally used as an input to the CNN, allowing for the model to learn with moderate amounts of data.

To tackle the problem of CSI, we trained the model to generate embeddings of songs, that have the property of mapping similar tracks (covers) closer together. The loss function based on similarity is expressed using Angular Loss - a variation of the common Triplet Loss, used in metric learning. We've shown that the proposed architecture demonstrates adequate performance on the test datasets, achieving an accuracy of over 80%, and is also resilient to different languages, lyrics and instrumentation. The model is also tunable to aim either for precision or recall and can be used with little hardware requirements, making it affordable to implement in real-life applications. For this we implemented a proof of concept application, where the users upload songs to the platform and can use the trained model to identify similar songs.

Finally we also created two new public datasets consisting of english and greek cover songs, that can be used for training or evaluation purposes.

Future work could include training and using a state-of-the-art model for extracting embeddings for audio, instead of a CNN, such as the Audio Spectrogram Transformer. This method requires far more data and hardware resources to be

implemented, and using a pre-trained transformer is a common practice. A multimodal approach can also be beneficial to further improve the method. OpenAI's Whisper model is capable of accurately extracting transcripts even from audio with music background. Therefore a fusion model that combines audio and lyrical similarities could push the performance even higher.

Bibliography

[1]"How shazam works," *Free Won't*, Jan. 10, 2009. http://laplacian.wordpress.com/2009/01/10/how-shazam-works/ (accessed Mar. 15, 2023).

[2]J. Serra, E. Gomez, P. Herrera, and X. Serra, "Chroma Binary Similarity and Local Alignment Applied to Cover Song Identification," *IEEE Transactions on Audio, Speech, and Language Processing*, vol. 16, no. 6, pp. 1138–1151, Aug. 2008, doi: 10.1109/tasl.2008.924595.

[3]X. Xu, X. Chen, and D. Yang, "Key-Invariant Convolutional Neural Network Toward Efficient Cover Song Identification," in *2018 IEEE International Conference on Multimedia and Expo (ICME)*, Jul. 2018. Accessed: Mar. 15, 2023. [Online]. Available: http://dx.doi.org/10.1109/icme.2018.8486531

[4]Z. Yu, X. Xu, X. Chen, and D. Yang, "Learning a Representation for Cover Song Identification Using Convolutional Neural Network," in *ICASSP 2020 - 2020 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, May 2020. Accessed: Mar. 15, 2023. [Online]. Available: http://dx.doi.org/10.1109/icassp40776.2020.9053839

[5]Z. Yu, X. Xu, X. Chen, and D. Yang, "Temporal Pyramid Pooling Convolutional Neural Network for Cover Song Identification," in *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence*, Aug. 2019. Accessed: Mar. 15, 2023. [Online]. Available: http://dx.doi.org/10.24963/ijcai.2019/673

[6]X. Du, Z. Yu, B. Zhu, X. Chen, and Z. Ma, "Bytecover: Cover Song Identification Via Multi-Loss Training," in *ICASSP 2021 - 2021 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, Jun. 2021. Accessed: Mar. 15, 2023. [Online]. Available: http://dx.doi.org/10.1109/icassp39728.2021.9414128

[7]E. O. Brigham and R. E. Morrow, "The fast Fourier transform," *IEEE Spectrum*, vol. 4, no. 12, pp. 63–70, Dec. 1967, doi: 10.1109/mspec.1967.5217220.

[8]Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *Nature*, vol. 521, no. 7553, pp.

436–444, May 2015, doi: 10.1038/nature14539.

[9]Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner, “Gradient-based learning applied to document recognition,” *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998, doi: 10.1109/5.726791.

[10]A. Krizhevsky, I. Sutskever, and G. E. Hinton, “ImageNet classification with deep convolutional neural networks,” *Communications of the ACM*, vol. 60, no. 6, pp. 84–90, May 2017, doi: 10.1145/3065386.

[11]“Figure 1: VGG16 architecture.”, doi: 10.7717/peerj-cs.557/fig-1.

[12]K. He, X. Zhang, S. Ren, and J. Sun, “Deep Residual Learning for Image Recognition,” in *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, Jun. 2016. Accessed: Mar. 15, 2023. [Online]. Available: http://dx.doi.org/10.1109/cvpr.2016.90

[13]S. Hochreiter and J. Schmidhuber, “Long Short-Term Memory,” *Neural Computation*, vol. 9, no. 8, pp. 1735–1780, Nov. 1997, doi: 10.1162/neco.1997.9.8.1735.

[14]K. Cho, B. van Merrienboer, D. Bahdanau, and Y. Bengio, “On the Properties of Neural Machine Translation: Encoder–Decoder Approaches,” in *Proceedings of SSST-8, Eighth Workshop on Syntax, Semantics and Structure in Statistical Translation*, 2014. Accessed: Mar. 15, 2023. [Online]. Available: http://dx.doi.org/10.3115/v1/w14-4012

[15]A. Vaswani *et al.*, “Attention Is All You Need,” *arXiv.org*, Jun. 12, 2017. https://arxiv.org/abs/1706.03762 (accessed Mar. 15, 2023).

[16]D. Bahdanau, K. Cho, and Y. Bengio, “Neural Machine Translation by Jointly Learning to Align and Translate,” *arXiv.org*, Sep. 01, 2014. https://arxiv.org/abs/1409.0473 (accessed Mar. 15, 2023).

[17]T. B. Brown *et al.*, “Language Models are Few-Shot Learners,” *arXiv.org*, May 28, 2020. https://arxiv.org/abs/2005.14165 (accessed Mar. 15, 2023).

[18]A. Ramesh *et al.*, “Zero-Shot Text-to-Image Generation,” *arXiv.org*, Feb. 24, 2021. https://arxiv.org/abs/2102.12092 (accessed Mar. 15, 2023).

[19]J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, “BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding,” *arXiv.org*, Oct. 11, 2018. https://arxiv.org/abs/1810.04805 (accessed Mar. 15, 2023).

[20]A. Dosovitskiy *et al.*, “An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale,” *arXiv.org*, Oct. 22, 2020. https://arxiv.org/abs/2010.11929 (accessed Mar. 15, 2023).

[21]Y. Gong, Y.-A. Chung, and J. Glass, “AST: Audio Spectrogram Transformer,” *arXiv.org*, Apr. 05, 2021. https://arxiv.org/abs/2104.01778

[22]A. Radford, J. W. Kim, T. Xu, G. Brockman, C. McLeavey, and I. Sutskever, “Robust Speech Recognition via Large-Scale Weak Supervision,” *arXiv.org*, Dec. 06, 2022. https://arxiv.org/abs/2212.04356 (accessed Mar. 15, 2023).

[23]“The covers80 cover song data set.” http://labrosa.ee.columbia.edu/projects/coversongs/covers80/ (accessed Mar. 15, 2023).

[24]“Covers 1000.” https://www.covers1000.net (accessed Mar. 15, 2023).

[25]M. Lin, Q. Chen, and S. Yan, “Network In Network,” *arXiv.org*, Dec. 16, 2013. https://arxiv.org/abs/1312.4400 (accessed Mar. 15, 2023).

[26]S. Chopra, R. Hadsell, and Y. LeCun, “Learning a Similarity Metric Discriminatively, with Application to Face Verification,” in *2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR’05)*. Accessed: Mar. 15, 2023. [Online]. Available: http://dx.doi.org/10.1109/cvpr.2005.202

[27]F. Schroff, D. Kalenichenko, and J. Philbin, “FaceNet: A Unified Embedding for Face Recognition and Clustering,” *arXiv.org*, Mar. 12, 2015. https://arxiv.org/abs/1503.03832 (accessed Mar. 15, 2023).

[28]J. Wang, F. Zhou, S. Wen, X. Liu, and Y. Lin, “Deep Metric Learning with Angular Loss,” *arXiv.org*, Aug. 04, 2017. https://arxiv.org/abs/1708.01682 (accessed Mar. 15, 2023).

[29]“Homepage — Essentia 2.1-beta6-dev documentation.” https://essentia.upf.edu (accessed Mar. 15, 2023).

[30]A. Paszke *et al.*, “Automatic differentiation in PyTorch,” *OpenReview*. https://openreview.net/forum?id=BJJsrmfCZ (accessed Mar. 15, 2023).

[31]K. Musgrave, S. Belongie, and S.-N. Lim, “PyTorch Metric Learning,” *arXiv.org*, Aug. 20, 2020. https://arxiv.org/abs/2008.09164 (accessed Mar. 15, 2023).

[32]“Plotly.” https://plotly.com/?_ga=2.206329427.1510403054.1678910908-1611882241.1678910908 (accessed Mar. 15, 2023).

[33]“Matplotlib — Visualization with Python.” https://matplotlib.org (accessed Mar. 15, 2023).

[34]L. van der Maaten and G. E. Hinton, “Viualizing data using t-SNE,” *unknown*, Nov. 01, 2008. https://www.researchgate.net/publication/228339739_Viualizing_data_using_t-SNE (accessed Mar. 15, 2023).