cadence: Compressing Audio Data by Exploiting Cyclical Elements

Master of Science Thesis

Marc Elias Solèr
18-650-119

August 2023

Supervisor
Prof. Dr.
Anna-Lena Horlemann-Trautmann
University of St. Gallen

Co-Supervisor
Prof. Dr.
Guido Salvaneschi
University of St. Gallen

Abstract

Data compression is the task of reducing the size of data while preserving all or important aspects of it. With digital technologies dominating almost all of data storage and telecommunications, data compression plays an instrumental role in reducing the resources needed by these technologies. In particular, the compression of audio data is crucial to digital communication and streaming music or films. Yet, methods of audio compression have experienced relatively few changes in recent decades, despite significant contributions in computer science.

In this thesis, we evaluate approaches to improve audio compression in two ways. In the first, we investigate how to improve parts of MP3, a classical representative of audio compression standards. Although the direct replacement of the foundations of MP3 does not enhance compression performance, an alternative approach achieves an improvement.

In the second, we explore novel methods for audio compression. Concretely, we suggest and evaluate three prototypical audio compressors, drawing from the topics of topology, tensor factorisation, and vector quantisation. Two of the compressors provide promising results, one compressing audio with almost no loss at the level of a popular lossless audio compression standards, the other in achieving extremely high compression on an audio sample with strong repetitive structure. In addition, we analyse further approaches for compressing audio.

Finally, we discuss several concepts related to data compression and suggest directions for future investigations.

Contents

1 | Introduction

> The purpose of computation is insight, not numbers.
>
> Richard Hamming

In the last forty years, the world's capacity to store, communicate and compute information has increased at annual rates of about 20%, 23% and 58%. In 2007, storage was dominated by digital technologies with a 94% of all data stored, and telecommunications with 99.9% [1] of all data transmitted. Many of todays digital information applications rely on efficient data compression techniques, such as the compression of images with JPEG [2], its related format for videos MPEG [3] and audio compression with MP3 and its successors [4]. Everyday tasks as navigating the web, streaming music or videos were not imaginable without data compression. As an audio format, MP3 has had an impact beyond the technical sphere and was studied as cultural phenomenon [5].

Beyond everyday life, data compression is indispensable for scientific operations, as for the Large Hadron Collider producing measurement data at a rate of more than 3.5 TB per second [6], or for volumetric data from large-scale computer simulations [7]. Despite the wealth of compression techniques available, it is estimated that the amount of digital data stored could be compressed by a factor of 4.5 without losing information [8].

Next to its practical importance, data compression is a research area that interacts with many topics from computer science and has correspondences or similarities with biology, physics and other fields [9, 10]. At its core, data compression is concerned with transforming data into a representation that uses less space. This goal is achieved by identifying structure within data, and to use it to encode the data more efficiently. From a computational view, structure can be a probability model that allows to encode more frequent elements of data using shorter descriptions. In an image of a blue sky, it makes sense to encode blue pixels with shorter codes than red pixels. From a human view, structure can be a seen as the selection of a subset of data that is perceivable by human senses. Sounds above a certain frequency are inaudible and may therefore not be encoded at all. Depending on the kind of structure, there are different data compression methods, and while some are applicable to any kind of data, others specialise in particular kinds of data, such as images, audio, or high-dimensional volumetric data. Therefore, any method from any field that helps finding structure within data may be considered to assist in the task of compressing data. Furthermore, there are methods that discard data (lossy compression) and methods in which compression is exactly reversible (lossless compression).

Typical representatives of data compression on a symbol level (e.g. bits, bytes, numbers, ...) are Huffman and arithmetic coding [11], known as entropy coders. They are part of almost ev-

ery data compression standard such as JPEG, MPEG, or MP3, where they losslessly encode data that is preprocessed by specialised routines. The structure these methods make use of is the the probabilistic distribution of symbols, described by Shannon's information entropy [12].

Dictionary-based techniques such as Lempel-Ziv are extensively used in universal file compression such as ZIP. They exploit structure in form of repeating patterns such as words, and represent each occurrence with a reference to the pattern stored within a dictionary [11]. However, dictionary-based techniques are limited by locality — the scope in which recurrences are detected [8] and the assumption that patterns appear as precise copies of successive symbols.

Structure and compression performance are closely linked — for example, consider a $m \times n$ matrix representing a grayscale image with height m and width n that may be flattened into a 1-d vector for storage by walking the matrix in a zig-zag fashion (see Figure 1.1).

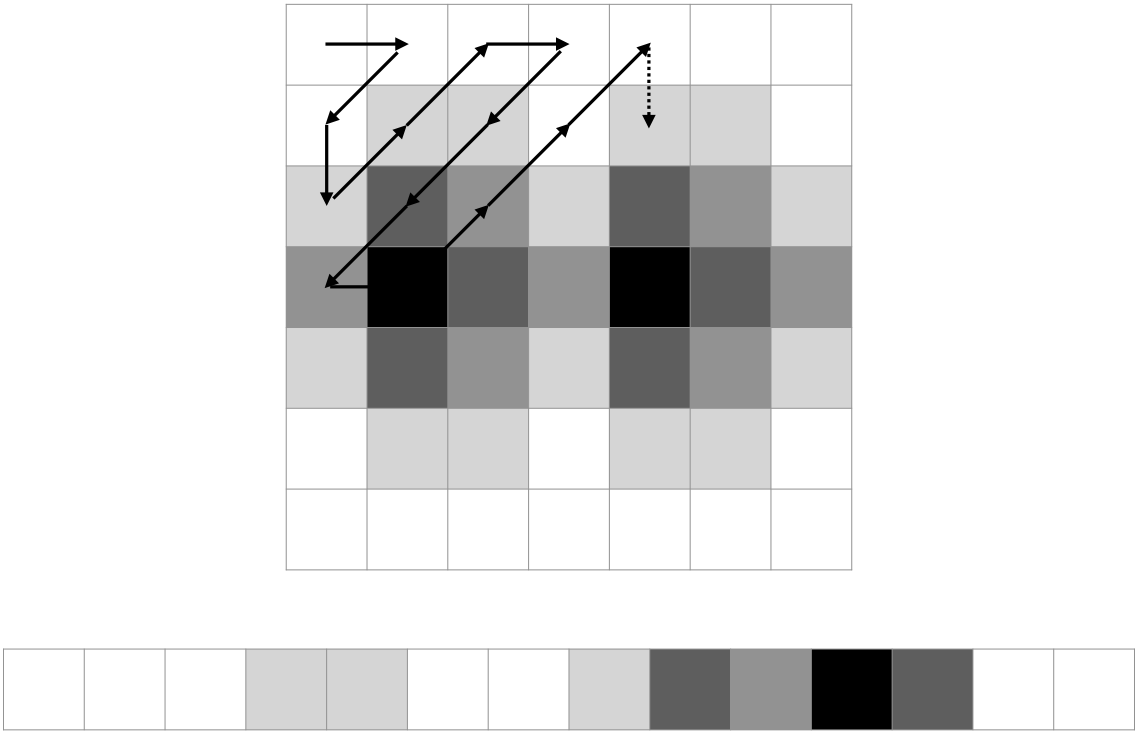

Figure 1.1: An image matrix with the corresponding vector obtained by flattening in a zig-zag manner. Flattening of an image matrix leads to loss of structure.

Such a procedure leads to loss of structure. In Figure 1.1, this happens in two ways: First, the first and last rows consist solely of white pixels, yet due to the zig-zag flattening, this fact is not represented in the vector. Secondly, the columns 4, 5 and 6 are an exact copy of columns 1, 2 and 3. This higher-level structure is lost due to the flattening as well and it becomes more difficult to exploit it for compression in this form. In practice, similar structures exist and there are similar losses of structure for many kinds of data.

Traditionally, structure is exploited heuristically. Examples are the discrete cosine transformation used in JPEG and MP3 assuming that small parts of images or audio may be represented with periodic functions [1]. Still, these techniques focus on a highly local, or microstructure rather than taking larger-scale information, or macrostructure into consideration. More recently, instead of hard-wiring heuristics into compression algorithms, methods from the machine

[1] JPEG uses blocks of 8×8 pixels and MP3 frames of approximately 26 ms.

learning paradigm have been employed for this purpose, often in the form of neural networks [13, 14, 15, 16].

The goal of this thesis is twofold: firstly, it targets to replace parts of the MP3 encoding scheme, targeting a better compression ratio and using modern methods not available during development of MP3. Secondly, it is set to find new methods to compress uncompressed audio. In both subgoals, the micro- and macrostructure, such as repetitions or similarities from music should be made use of and in addition, lossy and lossless methods should be employed or suggested in each subgoal. Table 1.1 links these subgoals to the chapters of this thesis.

	Lossless & Lossy
MP3	Chapter 3
Uncompressed	Chapter 4

Table 1.1: The tasks of this thesis.

The remainder of this thesis is organised as follows. In Chapter 2, we provide a brush up of basic notions of information theory, data compression, the JPEG and MP3 formats and advanced data compression methods. This chapter furthermore serves as a survey section following the explorative spirit of the second subgoal.

Chapter 3 investigates the first subgoal. First, the MP3 standard is examined, and suggestions for replacing the Huffman coding with an arithmetic coding are made. Several extensions are reviewed and results are reported. Then, other lossy and lossless methods are suggested, both achieving compression beyond MP3.

Chapter 4 deals with the second subgoal. First, we propose a simple audio compression method that achieves compression close to MP3. Some issues of audio compression are examined and discussed over two sections. Then, three audio compression methods based on the mathematical theory of topology, tensors and vector quantisation are suggested, some providing advantages towards current lossy and lossless audio compression standards. Finally, attempts using neural networks and fractal compression are made and observations made from their application are discussed.

Chapter 5 concludes this thesis and summarises its contributions. Also, it gives a short introduction to the idea of topological data compression.

Program Code and Notation

Code reference. Accompanying code for in this thesis is available at
`https://github.com/macemoth/audio-compression-msc`.

Note that variable names are not always unique between individual topics or sections. However, the variables are defined before every use and their meaning should be clear from the context.

2 | Fundamentals

In this chapter, we provide a basic introduction on concepts which this thesis constructs on. It is concentrated on breadth rather than depth, and provides references to the sources.

2.1 Entropy

Most data has the property that the symbols contained in a file or stream, such as bits, bytes, numbers, appear with non-uniform probability. Therefore, compression can be achieved by representing frequent symbols with shorter codes than infrequent ones. A typical example is the higher probability to encounter the letter e than the letter z in the English language.

The probability distribution of symbols is conceptually related to the distribution of physical quantities (such as speed) of particles in a system such as a gas, where entropy describes the disorder of the system. A larger entropy describes a more uniform distribution resulting in more randomness. Intuitively this is because in a random system, the a-priori knowledge of a specific particle or symbol is minimal (as they are equally probable) and the knowledge gain when observing it is maximal. The knowledge gain of a symbol s may be interpreted as information content [12] and is defined as

$$i(s) = \log_2(\frac{1}{p(s)})$$

where $p(s)$ describes the symbol's probability. A frequently occurring symbol will have a low information content as the knowledge gain is small. The information per symbol for a set of symbols S, such as a file or stream of symbols can easily be extended by the weighted sum

$$H(S) = \sum_{s \in S} p(s) i(s) = \sum_{s \in S} p(s) \log_2(\frac{1}{p(s)})$$

where $H(S)$ is the information entropy, due to Shannon [12]. Following above's intuition, this quantity is maximal for a uniform distribution. Trivially, if a symbol occurs with absolute certainty, its information content is nullified and the entropy is 0. The a-priori knowledge is complete and by observing symbols, no knowledge is gained (Figure 2.1).

Language and Conditional Entropy For example, the entropy of symbols in the German bible is 4.35 bits, while the entropy of the Roman characters including dots, commas and spaces with uniform probability is around 5.8 bits. The difference of entropy between these sources implies that there is some a-priori knowledge about the symbol distribution in language, namely that certain letters occur more frequently. In this form, entropy considers the symbols to be i.i.d., neglecting contextual structure present within words and sentences of natural language. A natural exten-

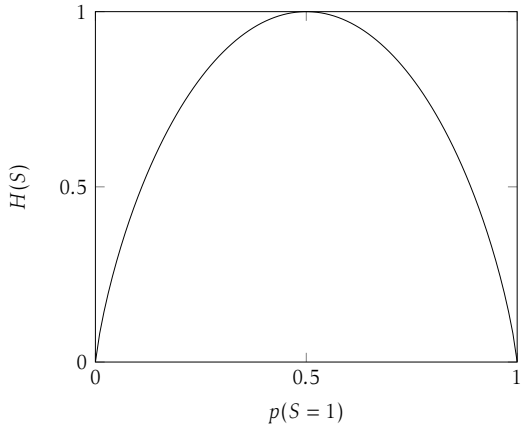

Figure 2.1: Entropy of a binary variable S depending on its distribution. If S is known to only yield 0 or 1 the entropy is zero, but maximal if $0, 1$ are equiprobable.

sion of this model is *conditional entropy* given by $H(S|C)$ quantifying the entropy given a the set of all contexts C, based on conditional probabilities $p(s|c)$ where c is a context. Intuitively, keeping track of the context helps the encoder to gain knowledge about the file's structure. For example, if a image file contains a repeating pattern, a pixel following this pattern is well-expected if the context is known and therefore carries little information. In contrast, if only the unconditional distribution of pixels is observed, that is, not considering the context, none of the pixels within the pattern can be expected. Because it is often computationally infeasible to consider C, one way to model the context for applications is to memorise the k previous symbols, modelled as a Markov chain of k-th order [17]. Compression methods employing such modelling are revisited later. A context model used for natural language is to examine the entropy of words rather than that of single symbols. The entropy becomes smaller as structure increases a-priori knowledge about what words will be encountered [1].

For compression, entropy is relevant as a measure of redundancy. A frequently occurring symbol carries little information while occupying space in a file or stream. Compression techniques exploiting uneven probability distributions are therefore called entropy coders. Information entropy is further relevant for cryptography in important ways. For example, if symbols in a file occur with a non-uniform distribution (and lower entropy), statistical analysis may allow conclusions about the original file before encryption.

Another example is, with a similar argument as in the English language, non-uniformly distributed encryption keys allow an attacker to try these first in a brute-force attack, thus increasing the chance of success. The set of all possible keys is called *key space* and its quality is quantified by the work factor, typically given in bits and equivalent to the entropy. For safe keys, it is therefore vital to maximise the keys' entropy.

Relative Entropy The entropy of a probability distribution (e.g. over a set of symbols) $p(x)$ can be defined relative to the entropy of another probability distribution $q(x)$ using the *Kullback-Leibler divergence*

$$D_{KL}(p|q) = \sum_{s \in S} p(x) \log_2 \left(\frac{p(s)}{q(s)}\right)$$

[1] For example, the word *airplane* is more readily expected than a random set of letters such as *zrogliksa*.

with set of symbols S [11]. D_{KL} is 0 if $p = q$ and quantifies the information that is gained when drawing symbols from p while expecting that they are drawn from q. An important result is that this quantity is always nonnegative, known as Gibbs' inequality

$$D_{KL} \geq 0$$

For example, if a scheme designed to encode symbols distributed with q has to encode symbols distributed with p, then D_{KL} describes the additional bits required in the encoding due to inefficiencies. The divergence is also used in machine learning as quantity of information gained by learning an additional variable [11].

2.1.1 Kolmogorov Complexity

While the Shannon entropy described above is concerned with the information content of a static collection of symbols, Kolmogorov complexity quantifies the information content produced by an algorithm. Concretely, the Kolmogorov complexity of an object (e.g. again a digital file or a stream) is the shortest computer program that produces the object as an output [18]. For example, an infinite stream of alternating zeros and ones 01010101... can be produced by a simple computer program, yet it would require an infinite amount of memory when stored, making the program an extremely efficient compressor. Because the symbols zero and one are equiprobable, the string's symbol-entropy would be maximal, suggesting no possibility for entropic compression[2]. In general, the Kolmogorov complexity cannot be computed, but it is only possible to find upper bounds by discovering programs [19].

Kolmogorov complexity illustrates the problem of finding compressable structure within data well. Consider a 1 GB file consisting of zeros and encrypting it using a block cipher in cypher block chaining (CBC) mode[3] and with a few bytes long encryption key. Although the data could easily be compressed if the encryption procedure and key were known, compressing it without this knowledge would require to break the cipher [19].

2.2 Data Compression Primitives

Data compression can be divided into two general flavours: lossless and lossy. The goal of lossless compression is to recover a precise copy of the original data from the compressed file. In lossy compression, some information, is deliberately lost during compression. The challenge lies in determining which information to loose, and typical choices are noise or parts less relevant in preserving the crucial aspects of the original data. Finding these aspects requires knowledge of the data and of the use scenario — perceptual coding describes the process of removing data humans are less sensitive to and is employed successfully for audio [20], images [2] and video [3].

A further spectrum of compression methods could be made along the level of knowledge about the data's structure. On one end, there are approaches concentrating on the smallest elements, exploiting the probabilistic distribution of the symbols (microstructure). Approaches on the other end provide deep understanding of the data's structure, such as symbol patterns and

[2] However, if two successive symbols 01 are considered as a new symbol, its probability would be one and the entropy would become zero. The definition of a symbol matters.

[3] In CBC mode, the encrypted blocks are linked to each other such that two identical clear text blocks are not encrypted into identical cipher blocks.

context, or even the procedure that generated them (macrostructure). This end has no sharp termination, and bears most promise for more efficient compression and is believed to be closely related to artificial intelligence due to the high degree of understanding about the data required [19, 21]. Methods on that end are related to the concept of *decorrelation transforms*, that have the intention of extracting redundant structure such as the modelling of audio signals in the frequency domain instead of the time domain. Usually, these methods are succeeded by entropy coding because decorrelation reduces the apparent randomness and therefore entropy [8]. The randomness is apparent (rather than intrinsic) because if the structure of data is known, it can be extracted. As an intuition, consider the above situation of the encrypted file containing mere zeros. When encrypted, the data appears random (with high entropy), yet its conditional entropy (on the context of the file content) is zero once the decryption algorithm has been applied. The problem of understanding data's structure can be cast as the problem of finding an optimal decorrelation transform [8]. From a probabilistic view, the understanding of structure can be seen as the reduction of uncertainty about the process generating the data, with symbol probability distribution as poor understanding, and a deterministic program generating data as the strongest.

In the next paragraphs, several compression methods and general phenomena about compression are introduced. We will use the following terminology to describe the methods. An uncompressed file consisting of symbols is given to an encoder that encodes it into a compressed file consisting of codewords. Both symbols and codewords may be binary digits or groups of multiple binary digits. A compressed file may be given to a decoder, decoding it into the original uncompressed file. The literature refers more often to streams rather than files. However, we often assume that the probability distribution is known and static, which is typically not the case with streams. These two properties have further significance in this thesis.

2.2.1 Trade-offs

A high compression rate is often not the only goal of compression. For many applications, compression speed and consumption of computing resources such as memory, processor time and access order of data are relevant, too. In live streaming of audio and video and real-time communication, low latency is desired, which is often obtained by reducing quality or consuming more computing resources. Furthermore, requiring low latency imposes the compression system to operate locally, reducing the units to be compressed to typical durations of several milliseconds. Structure present beyond this scope can usually not be regarded.

2.2.2 Run-length Coding

Beginning at coding methods that require no knowledge about the overall property of a file, run-length coding works by replacing consecutive symbols by a tuple defined by the symbol and the count with which it appears in sequence, as for example with this bitstring:

$$0000011010001111 \longrightarrow (0,5), (1,2), (0,1), (1,1), (0,3), (1,4)$$

If the symbols are binary digits, it is not necessary to represent the symbol and the above bitstring can be encoded as 521134, where a "dummy" zero has been appended to the string. Despite its simplicity, the coding performs well where some symbols appear very frequently and consecutively, examples being black and white images with a large proportion of white pixels.

2.2.3 Entropy Coding

If the probability distribution of symbols of a file is known, it is intuitive to replace frequently occurring symbols with short codewords while accepting long codewords for infrequent symbols. The result is to obtain a file with symbols of similar frequency, or high entropy, which is why these methods are known as entropy coders. They represent a class of coders requiring little knowledge about the process which generated the data.

Prefix Codes An additional requirement is that codewords must be uniquely decodable. For example, the codewords $A = 0$, $B = 1$, $C = 00$ and $D = 01$ do not satisfy this requirement, as the compressed file 001 may represent AAB, CA, or AD. A well-known class of uniquely decodable codes are prefix codes which produce no codewords that are prefixes of other codewords.

Huffman Coding Huffman coding is a classical method to create prefix codes assigning shorter codes to more frequent symbols. The Huffman coding algorithm is as follows. First, every unique symbol (e.g. a byte) in a file is represented as a trivial tree consisting of only one node. The node's weight corresponds to the symbol's frequency in the file. In the example shown in Figure 2.2, the symbols are $\{A, B, C, D\}$ with frequencies $\{1, 1, 2, 3\}$. Then, the two trees with the lowest frequencies are appended to a new tree as children in which their root node has the summed weight of the children. In the example, the first merged nodes are A and B. The algorithm continues merging trees until only one tree remains. The Huffman codewords for the symbols are then obtained from the position of a leaf node from the root, with left branches as 0 and right branches as 1 [17].

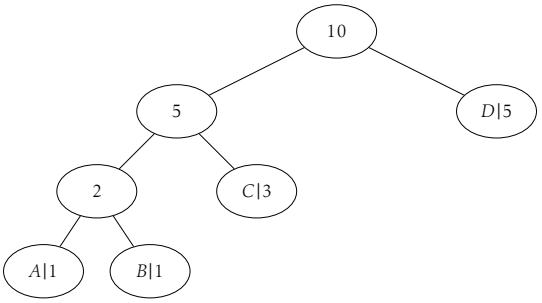

Figure 2.2: Huffman tree for the symbols A, B, C and D with their frequencies.

The codewords from this example are shown in Table 2.1. Note that the codewords are prefix free, they are uniquely decodable.

Symbol	Frequency	Codeword
A	1	000
B	1	001
C	3	01
D	5	1

Table 2.1: Huffman codewords obtained from the tree in Figure 2.2

Although the code produces optimal prefix codewords, this only holds for symbol probabilities that are negative powers of two. When the probability of symbols approaches uniform distribution, each codeword can have up to one bit of overhead. For files with entropy lower than

one bit per symbol, this overhead can become significant. A typical mitigation of this problem is to combine n symbols of the input file into longer symbols, thus reducing the overhead from one bit to $1/n$ bits [17].

Arithmetic Coding In contrast to Huffman coding, arithmetic coding is not optimal, but attacks most of the above mentioned problems by compressing a file in whole instead of symbol-by-symbol. Intuitively, the coder proceeds by representing a sequence of symbols within an interval $[0,1)$ which is successively partitioned after the probability of the current symbol [22]. In Figure 2.3, the encoding procedure for the symbol sequence $CADB$ is represented using the same frequencies as above: $p(A) = p(B) = 0.1, p(C) = 0.3$ and $p(D) = 0.5$. To partition an interval into subintervals, the cumulative probabilities of the symbols are used, being $\{0.1, 0.2, 0.5, 1\}$ in our example. To encode C, its subinterval $[0.2, 0.5)$ is chosen, and partitioned in proportion to the symbol probabilities. From it, a new subinterval $[0.2, 0.23)$ is chosen for A. This process is repeated for the complete symbol sequence. Finally, the symbol sequence $CADB$ is encoded as the interval (0.2165, 0.218).

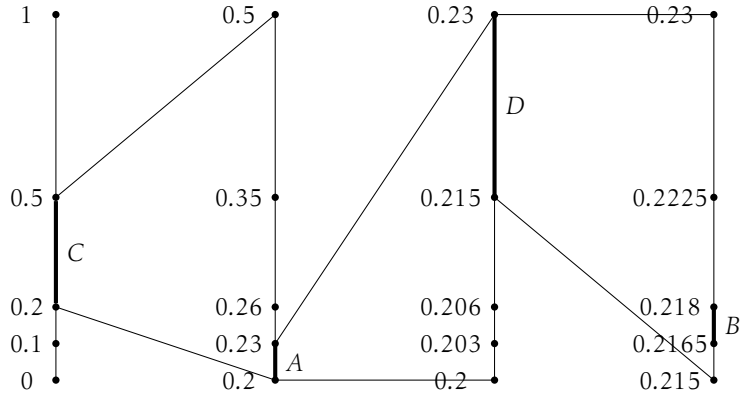

Figure 2.3: Illustration of the interval partitioning of an arithmetic code for the symbol sequence $CADB$. The interval is magnified in every step to full height for better visibility.

Software implementations typically provide two changes to the plain algorithm. Firstly, symbol probabilities are represented in terms of integers instead of floating point numbers in order to maintain precision. This requires the renormalisation of the interval bounds after processing a symbol. Secondly, the interval encoding the original file is represented as a binary fraction in the form $.b_1b_2b_3\ldots$, where a zero indicates the bottom subinterval and one the upper. Similarly as with the binary representation of integers, an interval of size x requires $-\lceil(\log_2(x))\rceil$ binary digits. Thirdly, the above description just yields the codeword once the complete file has been processed, while it is often desired that decoding can be started incrementally. The algorithm can be implemented to yield digits of the binary fraction once the choice of the next interval cannot change the digit, thus allowing incremental processing. However, the resulting algorithm is more complex and is described in Witten, Neal and Cleary, including a proof [22]. Finally, as the number representing the encoded data could specify an infinitesimally small interval and therefore an infinitely long sequence of symbols, an *end-of-file* number is appended to the encoded data to terminate the decoding process. The probability model can either be static or dynamic. Static models assume that the probability distribution of symbols does not change within a file, while a

dynamic model is updated based on the symbols that are encountered while coding or provided externally, e.g. based on the context [9]. The probability model is crucial to the encoding and is revisited throughout this chapter.

Arithmetic codes have several advantages towards Huffman codes. A typical example is that a file containing a symbol with very high probability, a Huffman code has to represent this symbol with at least one bit, although the symbol's information content may be close to zero. Meanwhile, an arithmetic code can represent the file much more efficiently as the same symbol can be represented with less than a bit. Despite the apparent advantages of arithmetic coding, it is not in widely used formats such as MP3 and JPEG as it was patented at the time they were designed. However, it is in use in newer codes such as Dirac [23] and VP8 [24], and has been used as a recompressor for JPEG images, replacing Huffman code [25].

Asymmetric Numeral Systems While arithmetic codes attack several of the limitations of Huffman codes, most notably providing higher efficiency, it is computationally much more intensive. Asymmetric Numeral Systems (ANS) have recently been suggested as an alternative to both, resolving the tradeoff of computational expense and efficiency [26]. Intuitively, it is based on the insight that in general, numeral systems (such as the binary or decimal system) encode sequences of equiprobable digits efficiently. Every new input symbol is trivially appended as a digit to the output. If, however, the input symbols have a non-uniform distribution it should be possible to asymmetrise, i.e. skew the choice of output digits in order to obtain an equiprobable output distribution, and hence, maximal entropy.

In efficient encoding systems, this implies that some frequent input symbols are encoded by less than a symbol in the output. While no fractional bits can be written in a digital system, the information represented by the fractional bit is stored as state. Implicitly, this idea is being used by arithmetic coding as well, where the state is stored in the two interval bounds and requires two numbers. In contrast, ANS stores the state into one number, allowing easier implementations. Furthermore ANS appends new digits of the output at the least significant end of the encoded number. An example is shown in Table 2.2. x denotes the current state, s the binary digit to be appended and x' the new state.

For example, if the current state $x = 3$ and $s = 0$, the new state equals $x' = 5$ and if $s = 1$, then $x' = 10$. If no asymmetrisation is chosen, i.e. the output digits are equiprobable, the same state would lead to $x' = 6$ for $s = 0$ and $x' = 7$ for $s = 1$ because appending 0 to the least significant position of the bitstring of 3 doubles it and appending 1 additionally adds 1, or $x' \leftarrow x + 2^m s$, where m is the number of digits of x.

x'	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
$x\|s=0$		0	1		2	3		4	5	6		7	8		9	10		11	12
$x\|s=1$	0			1			2				3			4			5		

Table 2.2: Example of an asymmetrised binary numeral system coding table where $p(s = 0) = 0.7$ and $p(s = 1) = 0.3$ [27]. Because the probability of appending 1 is lower, the resulting new state is allowed to contain more bits than when appending 0.

The ANS encoder is implemented as a finite state automaton, using the above state table to determine state transitions. The information of fractional bits is stored in the automaton's states, while some transitions yield codewords. The automaton therefore acts as a kind of buffer that with more states can compress data more efficiently. Due to the recent publication of ANS, few implementations exist.

2.2.4 Golomb Coding

Golomb codes is another class of codes that exploit a particular distribution of input symbols and can be seen as entropic coding method. They assume that the symbols are geometrically distributed integers, that is, that small values are more frequent than large ones[4]. Note that if the input values were finite, Huffman or arithmetic coding could be used [28]. The input x is coded as the integer part q and the remainder part r of the result of the division by m, a parameter:

$$q = \lfloor x/m \rfloor \quad \text{and} \quad r = x - qm$$

Then, q is encoded in unary representation (terminated with one if unary is expressed in zeros, or inverted) and r in binary. One may wonder how this encoding scheme can be more efficient than the simple binary encoding after seeing the example in Table 2.3. The resolution to this is that the code must be uniquely decodable, a condition that can obtained by encoding integers using a fixed length code (which is problematic if the size of the numbers are unknown a-priori), or using a prefix code, such as Huffman codes. Golomb codes fulfil this condition by default as they are prefix codes.

Rice codes are a special case of Golomb codes where $m = 2^n$ for $n \in \mathbb{N}$, and pure unary coding has $m = 1$. The choice of m is generally based on the success probability p of the geometric distribution of input data, but computational efficiency is another factor as divisions by powers of 2 (as used for Rice codes) are fast on digital computers.

x (Probability)	Binary	q	r	Golomb code
0 (0.2)	000	0	0	00
1 (0.16)	001	0	1	010
2 (0.128)	010	0	2	011
3 (0.102)	011	1	0	100
4 (0.082)	100	1	1	1010
5 (0.066)	101	1	2	1011
6 (0.052)	110	2	0	1100

Table 2.3: Example of Golomb codes for small integers, $m = 3$ and $p = 0.2$.

2.2.5 Dictionary-based Methods

While entropy coding focuses on fixed-length inputs and variable length outputs (depending on the probability distribution), dictionary-based methods consider variable-length inputs and typically produce fixed-length outputs. The methods are widely used in generic compression programs as ZIP, Gzip and standards as GIF, PDF and others and provide compression and decompression speeds [17]. However, their compression efficiency converges slowly with the data size (because the size of a finite dictionary becomes small with increasing size of compressed data), and the methods expect exact and local recurrences, thus neglecting the macrostructure of data [8].

Lempel-Ziv 77 Lempel-Ziv 77 (LZ77) is a greedy algorithm that passes a symbol sequence with a sliding window from left to right, divided into a lookahead buffer on the right and a dictionary on the left, separated by a cursor in the middle. The window passes the sequence from left to

[4] As shown later, this is a property of many transforms used in compression.

right. It proceeds by seeking the longest match from the cursor into the lookahead buffer to a subsequence that is in the dictionary. Once found, it yields a triple (p, n, c) indicating the relative position p of the occurrence, the length of the match n and the next character found after the match c. For example, the sequence *aabcab* yields the triples $(0,0,a),(-1,1,b),(0,0,c),(-4,2,\varepsilon)$ with ε as empty symbol. Decoding simply iterates through triples, reconstructing the original sequence and using it as dictionary. The algorithm is greedy as it seeks out matches for the lookahead buffer starting at the first position, even if better matches would be possible when skipping them.

Lempel-Ziv 78 is an extension of LZ77, but instead of using an implicit directory as the part of the sliding window, an explicit dictionary is constructed iteratively during en- and decoding, in which previous dictionary entries are recursively linked. A further development of it is the Lempel-Ziv-Welch algorithm.

Lempel-Ziv-Welch In contrast to LZ77, and similar to LZ78, the Lempel-Ziv-Welch (LWZ) algorithm uses a dictionary, which is prefilled with all of the alphabet's symbols. The algorithm iterates the symbol sequence until the read subsequence is not contained in the dictionary completely and therefore consists of a part contained in a dictionary and an additional symbol. The new substring is appended to the dictionary, but the dictionary index of contained in the dictionary before is returned. For example, consider a dictionary initialised with characters $(1,A),(2,B),\ldots,(26,Z)$ and the sequence to be compressed is

$$ABABBBAAA\ldots$$

After AB have been read, a new dictionary entry is appended: $(27, AB)$. However, the codeword 1 (index for A) is written. The algorithm continues to read the symbol that has not been represented by the token, in our case B at the second position. The combination with the next character BA is not contained in the dictionary, and appended as $(28, BA)$, but again, the codeword 2 is written. The next subsequence not contained in the dictionary is ABB, appending $(29, ABB)$ and yielding 27 as codeword, as AB is already in the dictionary. The next dictionary entries are $(30, BB)$ with codeword 2, $(31, BAA)$ with codeword 28 and $(32, AA)$ and codeword 1. For decoding, the dictionary is reconstructed, and because when reading the first codeword, only A of the dictionary entry 27 is known, the next codeword has to be read to complete entry 27.

Typical dictionaries contain approximately a million entries, and are either frozen once this number is reached, or they are discarded and recreated. LZW is regarded as a fast compression algorithm [29, 9].

2.2.6 Probability Modelling

An assumption of the probability models of the entropic coders shown above is that symbols are independently and identically distributed. As this model only considers the probability of one symbol unconditionally, they are also called order-0 models. In contrast, the Lempel-Ziv methods shown above implicitly assume that symbols are correlated and that data has been generated by a Markov model [30], but without modelling it explicitly. In general, the knowledge of the probability distribution within a file is incorporated in a *model*. Consequently, models can be extended with knowledge about the file's structure, which can then inform the coder about the (conditional) probability of encountering a symbol at its current state, thus reducing information

and improving compression. As an extreme example, a probability model can be built to encompass a specific file as context, giving the encoder absolute certainty about the next symbol to be encoded. Each symbol it would encounter would now carry zero information. The compression would be the most efficient possible, and the compressed file would be practically empty as all knowledge about the file were kept in the probability model itself.

Dynamic Markov Compression Dynamic Markov compression uses a Markov chain as a model to estimate the conditional probability based on the $1,\ldots,k$ previous symbols[5]. The probabilities are then used to inform the interval partitioning of an arithmetic coder dynamically, opposed to static models presented before [31, 30]. With higher order k, codewords tend to require fewer bits in correlated data such as text, as shown in Figure 2.4, motivating the estimation of conditional probabilities.

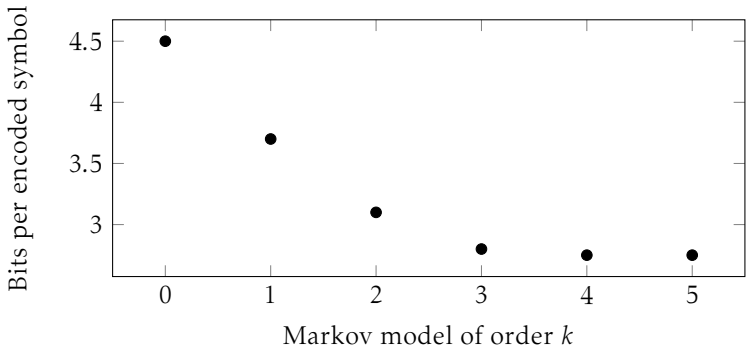

Figure 2.4: Codeword size in dependency of the Markov model's order k when encoding text data. Data from [31].

Concretely, the dynamic Markov model uses a state transition graph with vertices representing the observed bit sequence, or context as state and edges the frequencies of observing bit 0 or 1 next, from which the conditional probabilities can be computed (see Figure 2.5 for an example). The transition table initially contains only one state. New states are inserted with a *cloning* procedure, in which an existing state will be cloned, with the two clones being distinguished by the bit that leads to them. Cloning only takes place if the bit distinction reveals a correlation, i.e. changes the conditional probabilities after the cloned state, based on two thresholds. It is not necessary to store the bit sequences in the states explicitly as they are created when traversing the transition graph. Compression algorithms of this class typically belong to the best for text and generic compression [9]. More generally, utilising the context to compress data allows to discover patterns within structured data and to attain higher compression ratios.

2.2.7 Decorrelation Transforms

Decorrelation transforms have been introduced as ways to extract structure, or correlations from data, allowing entropy coding to operate better. An important class of transforms is to select a set of basis functions that better represent the space that the data occupies. The motivation is to shift intrinsic properties of the signal into the space spanned by the basis functions so that the transformed data contains fewer redundancies, or correlations and therefore a sparser representation.

[5] Note that the dynamic Markov chain model is more flexible than a fixed-order (say k) model as less than k previous symbols can be represented.

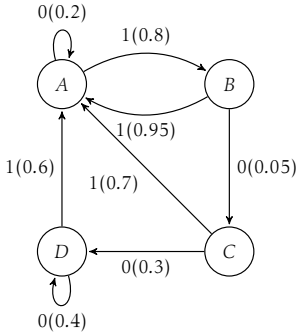

Figure 2.5: Example Markov transition graph with five states. Transitions are denoted with the bit and its probability.

A sparse representation is characterised by a lower entropy than the original representation, thus supporting entropic compression.

Fourier-related Transforms The *Fourier transform* is a classical example of a transform that converts signals from their representation over the time axis into a representation over their frequency, as illustrated in Figure 2.6. On digital systems, the Fourier transform is applied as *Discrete Fourier transform* (DFT) on signals that are stored and processed as a collection of discrete points. As an algorithm, the DFT receives the signal as a series of samples and transforms it into a frequency spectrum. Concretely, the frequency spectrum is represented as a list of complex values, where the position inside the list indicates a frequency and the value at a frequency is called amplitude.

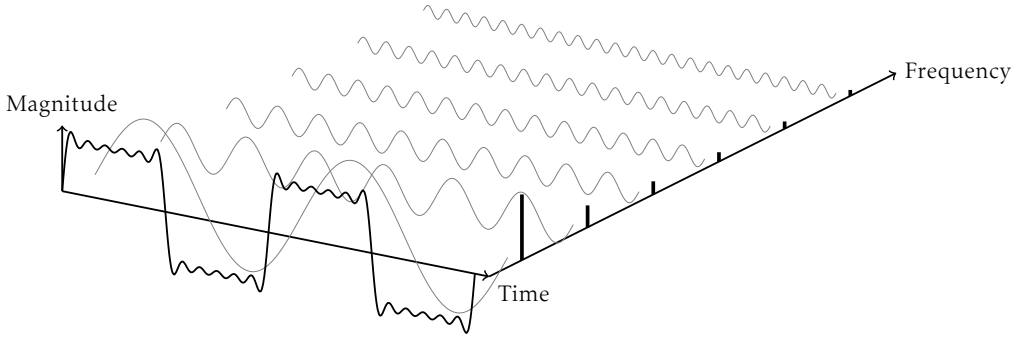

Figure 2.6: The Fourier transform converts an input signal in its time domain (front view) to its frequency domain (side view) as a linear combination of sine waves.

Similar transforms for the time- to frequency domain conversion belong to the family of *Fourier-related transforms*. In compression, the *discrete cosine transform* (DCT) is often used as an alternative to the discrete Fourier transform because the DCT does not expect periodicity in the signal [17]. A frequently used variant of the DCT is defined for a set of N real numbers and yields N coefficients

$$X_k = \sum_{n=0}^{N-1} \cos[\frac{\pi}{N}(n+\frac{1}{2})k] \quad \text{for} \quad k = 0 \ldots N-1$$

Where X_k is the k-th coefficient. The transform is invertible, but looses information in practice, as explained below. The coefficients obtained from the transform are often unevenly distributed with most amplitudes centred around 0 (especially those for high frequencies), allowing entropy

coding to perform well and allowing perceptual coding to discard relevant from less relevant information.

The DFT or DCT assumes that the signal is stationary, i.e. that the amplitudes within the time segments remain the same. Clearly, most audio (or most non-synthesised) data violates this assumption, as it does not consist of a constant set of pitches, or a regular wave pattern. Therefore, the signal is often partitioned into T short and overlapping time segments, and extracting the frequency spectrum via a Fourier-related transform from each of those, obtaining N frequency lines per segment. Although the signal is not constant within this segment too, the approximation is good for practical purposes and often indistinguishable by human perception. The operation yields a $N \times T$ matrix where each entry indicates the amplitude of frequency $n \in N$ in the time slice $t \in T$, often revealing important structure and allowing to remove noise for lossy compression. Such a matrix for audio signals can be visualised as a spectrogram with the horizontal axis as time, the vertical axis as frequency and the colour as the amplitude (see Figure 2.7). In this thesis, we will refer to matrix as well as its visual interpretation as spectrogram. The DCT and related transforms are used in many compression standards, as described in the remainder of this chapter.

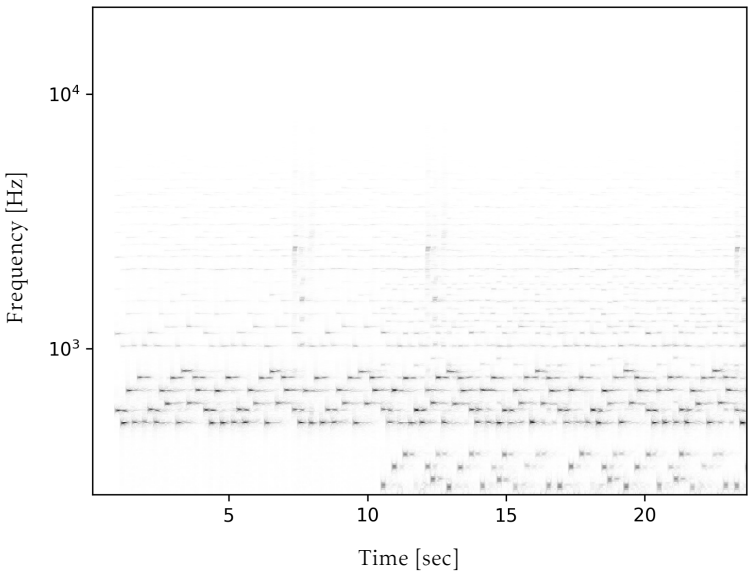

Figure 2.7: A spectrogram obtained from a 24 second audio sample. Here, the frequency axis is logarithmic to better accommodate high frequencies.

Further Fixed-base Transforms Fourier-related transforms that use sine or cosine functions as transform bases can be generalised to other functions that represent the signal in the transformation space. Examples are polynomial-, wavelet- or Bessel-functions, as shown in Figure 2.8 [17].

Kosambi-Karhunen-Loève transform The theoretical upper limit for decorrelating data is given by the *Kosambi-Karhunen-Loève transform* (KLT) [8], in which coefficients are random variables and bases are continuous real-valued functions. Instead of using predefined transform bases as

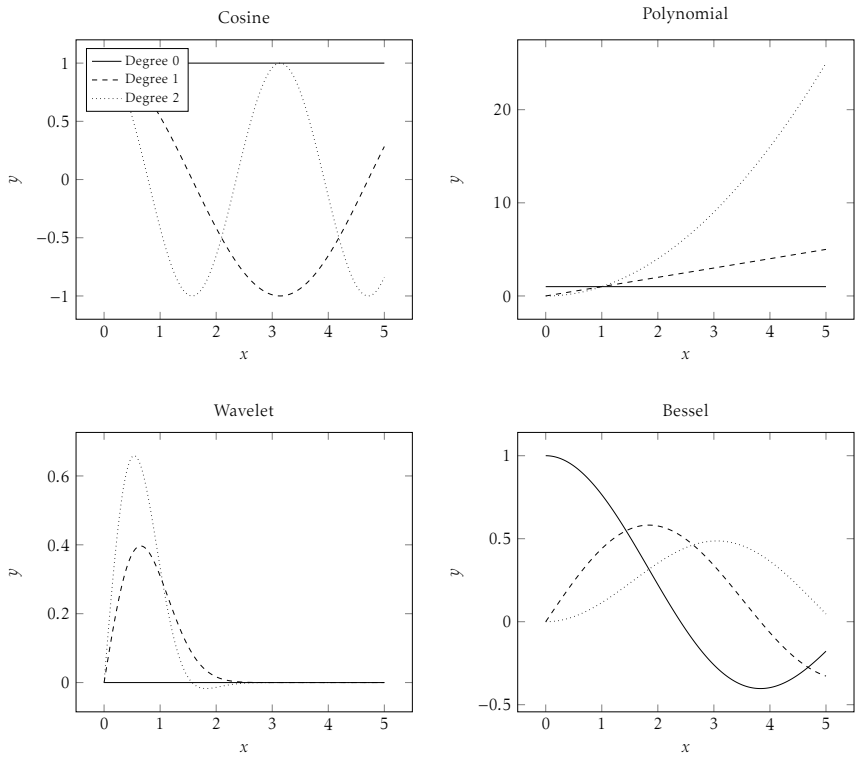

Figure 2.8: Various base functions used in transforms. Wavelets have many variants, here a simplified 1-D Gabor wavelet is shown. Bessel functions are used in the Hankel transform that substitutes the Fourier transform in hyperspherical coordinates.

Fourier-related transforms, the bases of the KLT and related transforms are data-dependent, although asymptotically, the DCT approximates the KLT [32]. In the discrete case, it is equivalent to the *Principal Components Analysis* (PCA) for a vector of data X

$$X = \sum_{i=0}^{N} y_i \phi_i$$

with coefficients y and bases ϕ that are the eigenvalues and -vectors of the covariance matrix of X. Each summand is a principal component [33]. For compression, the top k principal components are retained, with $k < N$, lossy compression is achieved. A frequent application of PCA is to reduce the dimensions of a dataset as the principal components are chosen to explain most of the variance without correlating, intuitively to not occupy surplus dimensions. This can furthermore give clues to the inherent dimension of the data, an aspect that is discussed later. Yet, PCA and KLT assumes the data to be generated by a random process, which is not necessarily the case, as for example images and audio samples.

2.2.8 Tensor Factorisation

One way to obtain the principal components of PCA is by *Singular Value Decomposition* (SVD). The SVD of a data matrix X is

$$X = U_1 S U_2^T$$

where U_1 contain the left-singular and U_2 the right-singular values in their columns, the diagonal elements of S are the singular values, and its off-diagonal elements are zero. The principal

components are given by $U_1 S$ [34]. The SVD can be generalised to *Higher Order Singular Value Decomposition*, or the related *Tucker decomposition*, where a Tensor of dimension d instead of matrices are decomposed. The left- and right-singular values of SVD are substituted by *n factor matrices* (or or *Tucker factors*) that are orthogonal to each other. The factor matrices are, like in KLT, data-dependent bases, in contrast to the fixed bases of Fourier-related transforms. However, this requires that the factor matrices have to be stored for the encoding, too. A typical use of tensor factorisation is to convert tensor problems that often appear in physics and engineering (e.g. elasticity, fluid mechanics or general relativity) into approximately equivalent, but easier to solve problems. A use that is more relevant to data compression is to reveal structure inherent in tensor data, and to separate it [35], providing decorrelation.

The introduction of Tucker decomposition is mostly following the notation used in Van Loan [35]. The element at the i-th row and the j-th column of a matrix M is accessed as $M[i, j]$. This notation extends to tensors, where there are as many indexes as the dimension of the tensor. In the 2-d case, e.g. from the SVD as above, a $r_1 \times r_2$ matrix X can be decomposed into the matrices $S : r_1 \times r_2$ and $U_1 : r_1 \times r_1$ and $U_2 : r_2 \times r_2$ and is written as

$$X[i_1, i_2] = \sum_{j_1=1}^{r_1} \sum_{j_2=1}^{r_2} S[j_1, j_2] \cdot U_1[i_1, j_1] \cdot U_2[i_2, j_2]$$

Note that matrices X and S have the same shape. The matrices U_1, U_2 are square, although this does not need to be the case in general. If the matrix X that is of order two is replaced by a Tensor of order three or more, the decomposition is called Tucker decomposition. Correspondingly, for a third-order tensor $\mathcal{X}$, the decomposition is written as

$$\mathcal{X}[i_1, i_2, i_3] = \sum_{j_1=1}^{r_1} \sum_{j_2=1}^{r_2} \sum_{j_3=1}^{r_3} \mathcal{S}[j_1, j_2, j_3] \cdot U_1[i_1, j_1] \cdot U_2[i_2, j_2] \cdot U_3[i_3, j_3]$$

See Figure 2.9 for a rendition of the 3-d case. The mode-k product $\times_k$ allows to omit the sum over the k-th dimension, so the above expression can be written more compactly as

$$\mathcal{X} = \mathcal{S} \times_1 U_1 \times_2 U_2 \times_3 U_3$$

where U_1, U_2 and U_3 are factor matrices, and are always tensors of second order, and $\mathcal{S}$ is the core tensor. If the tensor $\mathcal{X}$ has d dimensions, the decomposition yields d factor matrices while $\mathcal{S}$ has the same shape as $\mathcal{X}$. The factor matrices can often have an interpretation in the sense of the principal components of PCA — they reveal structure of $\mathcal{X}$ [35]. Meanwhile, the core tensor expresses how and how strong the factor matrices interact with each other [36]. A more thorough account on Tensor factorisation is given in L. De Lathauwer et al. [37], while Rabanser et al. [36] provide a lighter treatment of tensor decompositions and their applications.

Compression is achieved by truncating, i.e. reducing the size of the core tensor, similar to only retaining the largest s principal components of PCA, but along all dimensions of $\mathcal{S}$. Truncation is closely related to the concept of low-rank approximation. Concretely, the truncation is achieved by replacing r_d by $\tilde{r_d} \leq r_d$ along with the corresponding factor matrices

$$\mathcal{X} \approx \tilde{\mathcal{X}} = \sum_{j_1=1}^{\tilde{r_1}} \sum_{j_2=1}^{\tilde{r_2}} \sum_{j_3=1}^{\tilde{r_3}} \tilde{\mathcal{S}}[i_1, j_2] \cdot U_1[i_1, j_1] \cdot U_2[i_2, j_2] \cdot U_3[i_3, j_3]$$

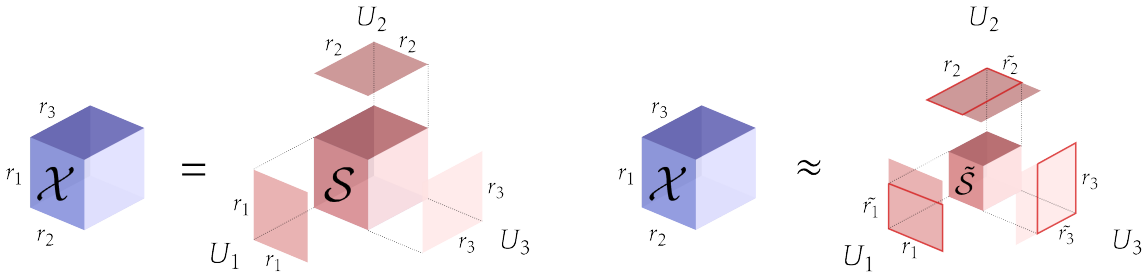

Figure 2.9: A Tucker decomposition for a tensor $\mathcal{X}$ into the core tensor $\mathcal{S}$ and the factor matrices U. The right side shows the full core and the left shows the truncated core. Note that with truncation, only parts of the factor matrices are required. Figure recreated from [38].

From the definition of the tensor product, the new tensor $\tilde{\mathcal{X}}$ retains the same shape, but gains error from the truncation. Remarkably, the error $\|\tilde{\mathcal{X}} - \mathcal{X}\|$ is equivalent to the error induced by truncating the core tensor [38], i.e.

$$\|\tilde{\mathcal{X}} - \mathcal{X}\| = \|\tilde{\mathcal{S}} - \mathcal{S}\|$$

where $\|\cdot\|$ denotes the Euclidean norm of the flattened tensor elements.

Applications of Tensor Factorisation In an abstract reasoning, the Tucker decomposition can be seen as a natural continuation of the time-domain to frequency-domain conversion of discrete Fourier-related transforms, where higher-level patterns are extracted from a tensor. If one axis of the tensors represents time, the Tucker decomposition again attains a similar role to the the Fourier-related transforms. This intuition has been used to extract patterns from audio signals for analysis [39] and for compression [40, 41].

For analysis of the audio signal, Smith and Goto [39] first convert an audio signal into a $N \times T$ spectrogram, with N frequencies and T discrete time steps. They proceed to split the matrix along the time axis into Q submatrices with size $N \times P$, and stack them into a $T \times P \times Q$ tensor, which is then decomposed. P is chosen to match the downbeats of the music sample, such that each $N \times P$ matrix captures a "period" or musical "loop". The resulting factor matrices can be interpreted as templates related to the music, such as notes, activation patterns (*when does a note occur within the submatrix*) or repetition patterns (*when does a pattern repeat within the complete sample*).

One limitation of this method is that a constant period must be assumed, as mirrored in the fixed P and imposed by the fact that tensor factorisation requires a tensor as input. However, music rarely fulfils the assumption of a constant period as there are often pauses in between repetitions. A related issue is that the split has to be fixed at one loop length, P, while repetitions may occur at different periods, depending on the sound, sometimes being separable by frequency[6].

While Wang et al. [40] use tensor factorisation for compression, they do not exploit repetitions in the music. Instead, their tensor is spanned by time, frequency and channel and only a reduced core tensor is transmitted, while the factor matrices are trained for many audio samples and are part of the en- and decoder. Compression is achieved by truncating the core tensor along the axes of frequency and channels. The author's main goal is to exploit similarities between channels in signals that have more than two channels, rather than to exploit repetitions in the time axis. The

[6]Consider that a sound from percussion may repeat every two seconds, and a melodic loop at a higher frequency repeats every five seconds. Similarly, the percussion beat of the percussion may be part of a longer percussion loop that completes a repetition every six seconds. What is the appropriate period length?

method achieves a compression rate of 86% (ratio of 1:7.14) from truncating six to two channels and from 960 frequency lines to 400, while subjective audio tests still score acceptable listening quality. The authors evaluate even higher compression rates.

Tensor factorisation approaches are examined in Chapter 4 of this thesis.

2.2.9 Data Deduplication and Delta Encoding

In data transfer and storage, especially when it is available in several versions, *data deduplication* is an effective way to store duplicated parts within the data more efficiently. Typical examples are backup systems that may store identical e-mails found in multiple mailboxes only once, or store only the changes made in file hierarchies instead of producing full copies on every backup. The idea to store duplicated units of data only once and keeping references to it is similar to dictionaries. The difference could be characterised that data deduplication rather describes a paradigm, while dictionaries are a collection of data structures and algorithms. The latter is also known as *delta encoding* and although it is conceptually similar to data deduplication, it describes the storing of the difference, or delta, rather than the not-storing of identical parts in deduplication. The terminology is essentially about a half-full or a half-empty glass.

For example, the *venti* system observes the data on a block-level, each consisting of about 9 kilobytes, from which 160 bit[7] hashes are produced [42]. The hashes are stored as an index that refers to unique data blocks. Multiply occurring blocks therefore only have to be stored once. venti further uses the insight that with large quantities of data, many blocks are repeated within it, and if they are stored centrally, users can produce extremely short representations of files and directories (similar to zip archives) that refer to the centrally stored blocks.

Delta encoding is mostly used for dynamic rather than static data and assumes that changes occur locally and in small quantities. Examples are the *rsync* protocol, designed to synchronise files between computers. The files are also split in blocks, from which checksums are calculated, with which differences (deltas) can be recognised and efficiently synchronised [43]. Other examples include video compression, where subsequent frames often only differ marginally and only the delta is stored in a method known as *motion compensation*. Video codecs achieve compression rates of 1:20 to 1:200 [44].

More generally, delta encoding reduces the variance of data (when its assumptions are correct) and can be related to the concept of relative entropy introduced above in the sense that only the excess information rather than the complete information has to be stored.

However, data deduplication and delta encoding is not effective when there are no exact duplications, as for example if the colours in an image matrix change slightly. Many codecs account for such effects, as with motion compensation for videos, but there is no general mitigation of the problem of almost-duplications. (Note that this discussion also largely applies to dictionaries due to their similarity with data deduplication and delta encoding.) This problem stems from the architecture of digital computers that, by default, operate with data as exact patterns rather than conceptual representations as biological brains. Textual or numerical data can be handled conveniently on a digital computer, for example using ASCII, or as floating point representation. Meanwhile, data from analog sources, such as images, video or audio data have a less "natural" digital representation, and must be quantised to be digitally represented. In the context compression, in particular of deduplication or delta encoding, small changes in the analog source

[7] Or equivalently 20 bytes, hence the name.

(e.g. a change of brightness, or a slight irregularity of the percussion) will alter the digital representation significantly, and often unpredictably, as rendered in Figure 2.10. This problem will be discussed in a range of contexts in this thesis, and a mitigation of it will be introduced with vector quantisation.

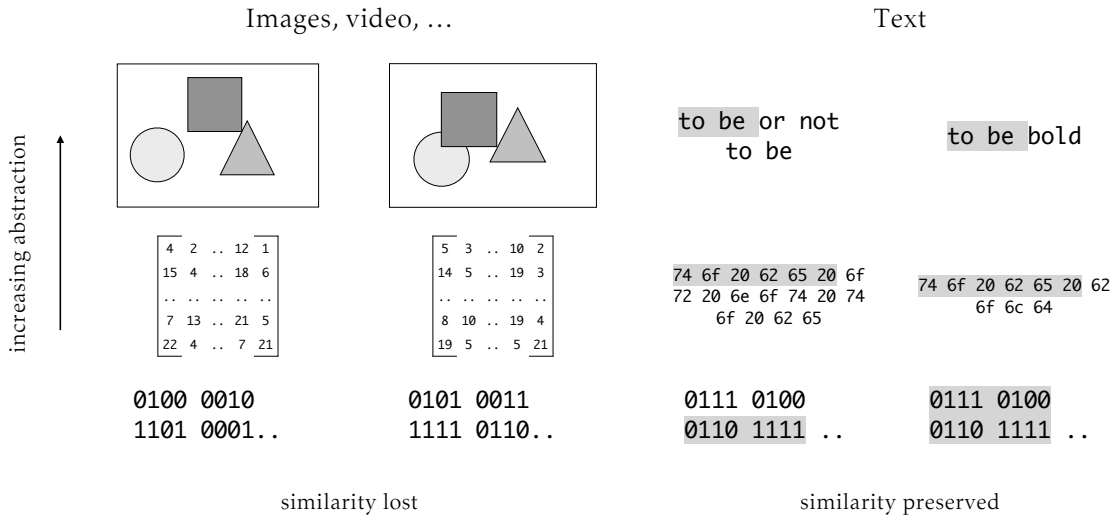

Figure 2.10: While small changes in an image usually have significant influence in the image matrix and the 1-D bit string (left), changes in textual data remain local and more predictable in the digital representation. Identical characters of two texts can be tracked down to the bit string, and similarity is retained. This makes it easier to exploit similarities in textual data for compression.

2.2.10 Neural Network Coders

Approximately since the publication of AlexNet about a decade ago, Deep Learning, spearheaded by Deep Neural Networks, has experienced a surge in almost every scientific discipline [45]. Although the concept of neural networks dates back to the 1950s [46], the increase in computing power has allowed the construction of larger neural networks, drastically improving their performance.

Neural Networks as Context Models In data compression, neural networks integrate well with asymmetric coders by predicting the content based on the previous context. More concretely, the networks inform the arithmetic coder about the probability of the next symbol, e.g. the probability that the next bit is one, such that the interval can be set appropriately. In this setup, the neural networks are said to implement the context model. Recall that if the symbol has been predicted correctly with a high probability, the range representing the current codeword in the interval $[0,1]$ is wide and few codewords are necessary to encode data. This model reduces the problem of data compression to the problem of predicting the next symbol based on the context. It can also be seen as a translation of conditional entropy into used codewords: in the extreme case that the next symbol is known with probability of one, the entropy is zero and no codewords are required, yielding ultimate compression. There is a direct correspondence between uncertainty and codeword length. Furthermore, the model is in some sense tolerant to wrong predictions — if a symbol was mispredicted, the chosen interval is smaller and the codeword becomes larger, but no data is lost.

Using a neural networks for compression was first suggested by Schmidhuber and Heil in 1996 [47]. They use a three-layer network trained by backpropagation, but it requires several days for training on a single text file, making it impractical for use as a compressor. In 2000, Mahoney introduced a faster compressor with a two-layer neural network [16], that serves as basis for the PAQ compressors. It combines the learning and prediction phase in a single step, whereas Schmidhuber and Heil's work requires a separate learning phase. The advantage of the former is that during decoding, the neural network weights do not need to be available as they are updated as decoded symbols become available, making the en- and decoding steps symmetrical. With two phases, the network weights would need to be stored alongside the codewords as they can not be trained from encoded data.

Designed as universal file compression programs for research, the PAQ archivers[8] achieve high compression ratios (compared to ZIP or 7-Zip) by trading higher memory and CPU consumption and computation time. The compressors rank high in compression benchmarks such as the Hutter Prize[9] or the Large Text Compression Benchmark[10]. A conceptual overview of the neural network used in PAQ compressors is shown in 2.11. It is trained to predict the probability that the next bit y_0 is one based on M contexts, or more formally,

$$P(y_0 = 1|x_0, x_1, \ldots, x_M)$$

where contexts $x_i \in \{0, 1\}$ and $i \in \{0, 1, \ldots, M\}$ are assigned to input neurons[11]. The input neurons are connected to the output neuron by the weights w_{ji}, which with one output neuron, is a vector and the inverted indexes are again written to adhere to standard neural network notation [48]. The weights are initialised as 0 and can be positive or negative to increase or inhibit the influence of inputs to outputs. The probability is given by

$$P(y_0 = 1|x_0, x_1, \ldots, x_M) = g(\sum_{i=0}^{M} w_{0i} x_i)$$

where $g(x) = 1/(1 + e^{-x})$ is a sigmoid activation function $g : [-\infty, \infty] \to [0, 1]$, which is suitable to represent probabilities.

$x_0, x_1, \ldots, x_M$ are divided into several context sets (denoted as $\text{contextset}_\text{i}$), such as the last 8 bits with the current position in the byte to be encoded, the last 16 bits, or hashed values of longer bit sequences. The contexts of a context set are the bit patterns possible under the constraints of the context set, and therefore, only every context set has only one active input neuron at a time. For example, the context set of the last 8 bits with the position, encoded with 3 bits entails 2^{11} contexts, or input neurons. In PAQ implementations, 10^6 to millions contexts are used.

The weight update rule is updated on every bit to minimise the error between the predicted bit $P(y = 1)$ and the actual bit. While the details are left out here, it is worth to note that, depending on the variance of the data, short- and long-term learning rates are used.

[8] http://mattmahoney.net/dc/

[9] http://prize.hutter1.net

[10] http://mattmahoney.net/dc/text.html

[11] In Mahoney's implementation, there is only one output neuron y_0, but we write the index to remain more consistent with the neural network learning literature. Many neural network implementations encode output probabilities with multiple output neurons $y_0, y_1, \ldots$.

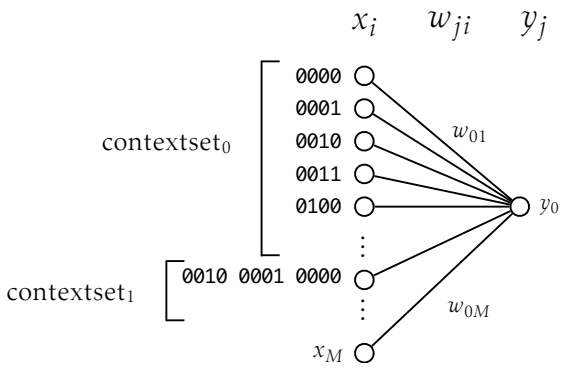

Figure 2.11: The two-layer neural network of newer PAQ compressors. It computes the probability that the next bit is one $y_0 = 1$ given M contexts x_i. The weights w_{ji} are updated on every evaluation.

Relation to Language Models Generally, the task of predicting the next symbol based on the previous context is a highly active research topic in the context of *Large Language Models* (LLMs), such as GPT-3 [49] or Llama2 [50]. Meanwhile, these models are used as personal assistants, able to answer complex questions in natural language and exhibiting human-level performance in exams [51]. This connection has been recognised at least since the 1990s [16] and the techniques of these models have been applied for compression, with focus on text compression. Notable examples are NNCP [15], featuring attention mechanisms that are successful in LLMs, or tensorflow-compress[12]. It is important to note that these models are designed and trained on text data, and that their predictive power is focused on natural language rather than arbitrary byte sequences, especially with long-range dependencies or a higher-dimensional structure not readily seen from a 1-d sequence, similarly as with the example in the introduction (aspects of the relationship between dimensionality and structure are discussed at the end of this chapter).

Neural Networks as Conceptual Compressors In the approaches described above, neural networks are used for their pattern recognition capabilities, but the encoding part is done by a lossless part that works on the symbol level. Parting from the intuition that the neural networks carry information in their weights, analogously to the human brain that stores memories in the synapses [10], Karol at al. have investigated to use them to store an abstract concept from images [52]. For this, the neural network is tasked to separate low-level noise from global conceptual information an image carries, discarding the noise and thus achieving compression. More concretely, two neural networks are set up in an *autoencoder*. The encoder network receives an image on its input neurons and passes it through several subsequently more narrow layers to a central layer, which output is called an *efficient representation* in a lower dimensional space. Then, a symmetrical decoder neural network picks up the efficient representation from the central layer and expands it to a possibly close approximation of the original image. The lower dimensional space is found by training the autoencoder, and is also called *latent space*, while the efficient representation is a set of *latent variables* in latent space. The concept of a latent space is similar to PCA (or KLT, respectively), and autoencoders have been described as nonlinear extension of PCA [14]. In the context of compression, the encoder and decoders are not too different to lossy coders, except that the learned en- and decoding are learned rather than engineered, and the latent space is typically not interpretable, similarly to PCA.

Similarly to the conceptual image compression by Gregor et al., Défossez et al. suggest an

[12] `https://github.com/byronknoll/tensorflow-compress`github.com/byronknoll/tensorflow-compress

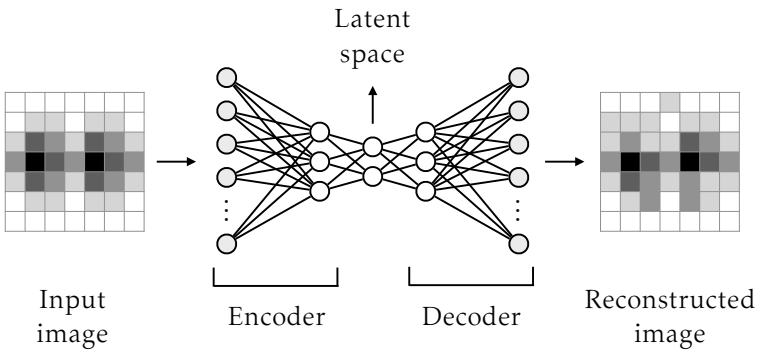

Figure 2.12: Architecture of an autoencoder. The input image is passed through a narrowing neural network (encoder), and then directly though a symmetrical widening neural network (decoder). In the centre, an efficient representation in the latent space is obtained.

autoencoder structure for audio compression [13]. While achieving higher compression ratios than classical audio compression algorithms, the procedure is computationally more expensive, and as typical of latent representations, yields little insight into how compression is achieved. Additionally, Défossez et al.'s approach comprises a language model as context model for arithmetic coding, as described in the previous paragraph. Despite suggesting new directions in audio compression, the model is trained with sequences of 5 seconds and the language model has an attention span of 3.5 seconds, making it unlikely to detect musical structure of an audio file, as intended in this thesis.

Limitations The situation for audio compression, additionally as standard neural networks have difficulty learning periodic data, as investigated by Ziyin et al. [53]. As a mitigation, he authors suggest an alternative activation function, or to treat data using Fourier-related transforms prior to feeding it to a neural network. However, these mitigations operate on exact periodic functions such as sine waves, but do not address cases where repetitions are almost-periodic or complete composite patterns repeat.

As this thesis has no particular focus on deep learning and seeks more interpretable compression systems, neural networks will not form its mainstay, but will be included in the analysis of compressing raw audio from its waveform and have a short appearance in MP3 recompression.

2.2.11 Data Compression using Logic Synthesis

In this chapter, decorrelation has been introduced as a method to extract the inherent structure of data to reduce the apparent randomness and the entropy. A simple example is an alternating pattern of zeros and ones like 01010101, that once discovered, can be represented in a more compact way. Recall that the above pattern cannot be compressed by an entropic coder as the frequency of zeros and ones is identical.

Many of the above decorrelation methods achieve their goal by assuming some kind of structure such as short, contiguous patterns in languages (e.g. Lempel-Ziv or Dynamic Markov Compression), or a superposition of base functions (e.g. Fourier transform).

Amarù et al. [8] suggest an automatic discovery of the function that produced the data, achieving optimal decorrelation if successful. Conceptually, their method partitions a binary string B into substrings of length L and then represents them as a logic circuit (see [54] for an overview of logic synthesis). This circuit can be seen as a (complex) function that can reproduce the binary

string. Furthermore, the circuit can be simplified using Boolean minimisation with the aim to find the logic function from which the binary data originated. This step is called *synthesis*. Experimentally, the method achieves a compression ratios of one to three orders of magnitude and up to two orders of magnitude better than universal compressors as ZIP or 7-Zip. However, the authors note that their data sets are either synthetically generated by a logic circuit, or by precise or even perfect measurements and are all highly correlated and causal. The authors choose such data rather than more standard data as they intend their method to be used on causal and highly correlated data.

The approach of [8] can be more formally written as in Algorithm 1. In difference to above description, a treatment is introduced for binary substrings that cannot be synthesised into a more compact circuit, called residuals R. Here, these residuals are not represented using circuits and entropy coded separately. Furthermore, the M circuits synthesised individually for each substring and collected in K can again be synthesised into one, large circuit, again removing redundancy.

Algorithm 1 Logic Synthesis Compression Algorithm

```
1:  function Main(B)                                   ▷ B is a binary string
2:      R = ∅                                          ▷ Set of residual substrings
3:      K = ∅                                          ▷ Set of logic functions
4:      for S_i ∈ B do                                 ▷ S_i is a substring of length L with 0 < i < M
5:          G_i ← MakeLogicCircuit(S_i)                ▷ G_i is a (complex) logic circuit
6:          G̃_i ← Synthesis(G_i)                       ▷ Simplify G_i
7:          if |G̃_i| > threshold then
8:              R ← R ∪ S_i
9:          else
10:             K ← K ∪ G̃_i
11:         end if
12:     end for
13:     K ← Synthesis(K)
14:     R̂ ← EntropyCoder(R)
15:     return K, R̂, L, M
16: end function
```

The method's program code is unreleased, and reproducing it requires access to a logic synthesis tool. As the method is intended for causal and strongly correlated data it seems unsuitable to the task of audio compression. Yet, its concepts of representing causal data as a logic circuit and attempting to discover inherent structure will be examined in this thesis.

2.2.12 Vector quantisation

Quantisation in general is an important concept of digital computing, in which analog quantities, such as colours in images or pressure of sound waves (see the next section for examples) are cast into discrete values. In vector quantisation, input vectors of dimension d, describing points in d-dimensional space, are approximated by a centroid, each with a similar number of input points assigned to it [55]. Instead of storing the position of the input points, only the centroids and references to them are stored, by which compression is achieved. The more centroids are introduced, the higher the "resolution" and the less quantisation error occurs. The space in which the points reside can be divided into a Voronoi tessellation (Figure 2.13 shows a 2-d version), where each centroid represents the points within its surrounding cell. As each centroid represents approxi-

mately the same number of points, the distribution of the points is represented more accurately than if each point would be quantised to e.g. the nearest point of a regular lattice, introducing less quantisation error. Therefore, vector quantisation can be seen as a learning technique.

A simple algorithm for vector quantisation is to pick a random input point, move the closest quantisation vector (i.e. the winner) closer to the point and repeat this procedure [56]. This learning paradigm is known as *competitive learning* because rather than to adjust parameters to minimise error, as done most conventional neural networks, centroids (or sometimes called neurons) compete to represent a point.

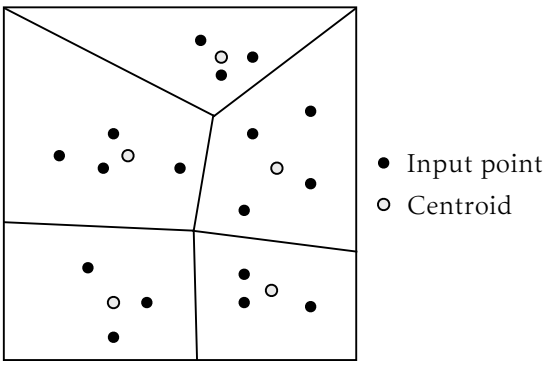

Figure 2.13: The input points are represented by centroids that partition the space into a Voronoi tessellation. Note that the centroids quantise represent the distribution of the points in contrast to if a regular lattice was used.

Self-organising Maps A method closely related to vector quantisation is the *self-organising map* (SOM). SOMs can be seen as a kind of artificial networks, but as with the learning algorithm from vector quantisation, they feature competitive learning [57]. In difference to the algorithm of vector quantisation, centroids (often called neurons in the context of SOM) are arranged neighbourhoods, according to some proximity metric. When the winning neuron is pulled towards the input point, its neighbours are pulled towards it as well, similar to covering the input points using a web of neurons (see Figure 2.14). Despite the simplicity of the learning algorithm, SOMs exhibit complex behaviour [58] and have connections to topology and neuroscience [57].

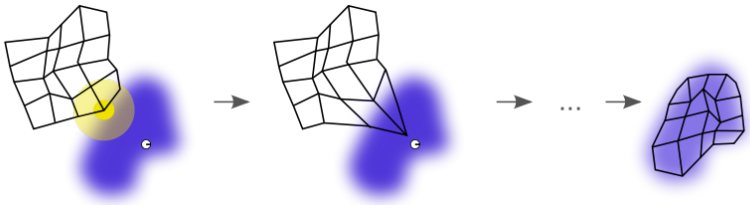

Figure 2.14: The neurons of the SOM cover points from an underlying probability distribution. The inner yellow disc denotes the winning neuron, and the outer disk its neighbourhood. The white dot is the currently selected input point. Training progresses from left to right. Source: `https://commons.wikimedia.org/wiki/File:Somtraining.svg`.

Vector quantisation can be seen as a mitigation to the limitations of dictionaries (and by extension, data deduplication and delta encoding) if one has to compress data in which there are approximate rather than exact duplications. Here, the dictionary entries are the centroids (or neurons). While dictionaries could be extended with a distance metric to match similar input points to dictionary entries, the dictionary entries would not necessarily represent the distribution of input points and lead to higher quantisation error or waste of space.

2.2.13 Fractal Image Compression

Although not directly related to audio compression, fractal image compression is a method to exploit the self-similarity at different scales found in images. Intuitively, it compresses an image by storing only relationships between parts of an image at different scales. The image can then be reconstructed by repeatedly applying the relationship information to a random image matrix such that the original relationships are regained.

Self-similarity is a concept closely related to fractals, and is colloquially known from the structure of Romanesco broccoli, clouds, trees, but is also observable in a range of other fields [59]. It also appears within images at no specific scale, although self-similarity in images is often not exact as in mathematical objects. When rotations and changes in brightness are allowed, more self-similar parts are found (see Figure 2.15 for a Sierpiński triangle and self-similar parts of an image).

(a) (b)

Figure 2.15: (**a**) The Sierpiński triangle is a typical self-similar structure that evolves by stacking triangles. (**b**) Self-similarities at two scales and using rotation and brightness modifications within an image of Claude E. Shannon. Source: https://web.archive.org/web/20050827083636/http://www.bell-labs.com/news/2001/february/26/1.html.

The core of the algorithm lies in the "relationship information" — it is mathematically defined as an affine transformation[13] w_i linking a large (domain block) to a smaller part (range block) within the image using translation, rotation and scaling $(a_i, b_i, c_i, d_i, e_i, f_i)$, contrast (s_i) and brightness (o_i) as

$$w_i \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} a_i & b_i & 0 \\ c_i & d_i & 0 \\ 0 & 0 & s_i \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} + \begin{bmatrix} e_i \\ f_i \\ o_i \end{bmatrix}$$

In practice, simplified transformations may be used. An image matrix is denoted as function $z = f(x, y)$ where x, y indicate the column and row of a pixel and z the intensity of the pixel as a scalar e.g. from 0 to 255. A transformation can be applied to an image as $w_i(f) = w_i(x, y, f(x, y))$ where it "writes" the range block using a domain block of the image [60] during decompression (see Figure 2.16).

For the transformations to approximate the original image when applied repeatedly, they must satisfy the *contractive* property. That is, for two arbitrary points p_1, p_2 and a distance metric

[13] A useful introduction to affine transformations is given in https://www.maa.org/sites/default/files/pdf/pubs/books/meg/meg_ch12.pdf

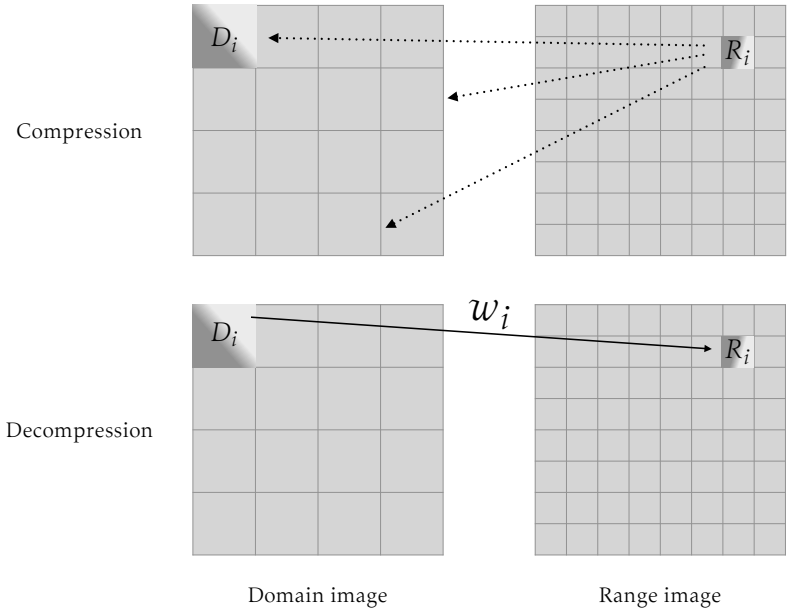

Figure 2.16: During compression, the algorithm searches a domain block D_i that resembles the current range block R_i and finds the parameters of w_i. During decompression, w_i is repeatedly applied. Note that the range and domain image are the same image, and only differ by their block size. D_i may be overlapping and vary in size while R_i have fixed size and are non-overlapping.

d (such as Euclidean distance) and $s < 1$, the transformation w must guarantee that

$$d(w(p_1), w(p_2)) < sd(p_1, p_2)$$

or equivalently, that the transformation brings points closer (i.e. contracts) than some factor $s < 1$. When a contractive transform is applied multiple times to any point, this point converges towards a unique fixed point[14]. The transformation w is said to have a fixed point. Because $s < 1$, it is required that the domain blocks D_i are larger than the range blocks R_i. Each transformation w_i describes one range block R_i in terms of D_i, but for all n blocks, the set of transformations is a map

$$W(f) = \bigcup_{i=1}^{n} w_i(f) = w_0(f) \cup w_1(f) \cup \ldots w_n(f)$$

that maps its input space to itself, in our case $\mathbb{R}^3 \to \mathbb{R}^3$.

Hutchinson's theorem [61] expresses that if the transformations w_i are contractive, the map $W(f)$ is contractive in the space of all f, that is, the space of all possible image matrices. The theorem thus connects the transformations to an complete image matrix and the transformation's fixed points to the original image f_{orig} as a fixed point f^*, or more concretely that

$$W(f_0)^{\circ m} = W(W(\ldots W(f_0)\ldots)) = f^* \approx f_{orig}$$

For decompression, any image f_0 can be used as first input to W. With every application of W, the image obtains better resolution until a sufficient resolution is obtained after m iterations (see Figure 2.17). However, because it is typically impossible to find exact self-similar parts within an image, the transformations w_i will not be able to reproduce a range block R_i perfectly. Thus,

[14] Due to the contractive mapping fixed point theorem, for a proof see [60].

the fixed point f^* will remain an approximation of f_{orig}. For compression, it is sufficient to find for every range block R_i a fitting domain block D_j (under rotation, brightness and contrast) at to store these transformations. The quality of fit of a domain block to a range block is given by the residual sum of squares $e(D_j, R_i) = w_k(D_j) - R_i^2$, which is the quantity that the compression algorithm minimises.

In summary, the algorithm for fractal image compression and decompression are shown in Algorithms 2 and 3.

Algorithm 2 Fractal image compression algorithm

```
1: function Compression(Image z = f_orig(x, y))
2:     D ← Partition(f_orig)                          ▷ Partition image into blocks D_i
3:     R ← Partition(f_orig)                          ▷ Partition image into blocks R_i
4:     Ω ← MakeTransformations(D)   ▷ Constructs k variants of every D_j under rotation,
   brightness and contrast as ω_jk = w_k(D_j)
5:     T ← ∅                                          ▷ Transformations for compression
6:     for i ∈ {0,...|R|} do
7:         T = T ∪ arg min_{ω∈Ω} e(ω, R_i)  ▷ Choose transformed D_j with minimal error to current R_i
8:     end for
9:     return T
10: end function
```

Algorithm 3 Fractal image decompression algorithm

```
1: function Decompression(Transformations T)
2:     f_0 ← RandomImageMatrix()
3:     f ← f_0
4:     for j ∈ {0,...,m} do
5:         f ← W(f)
6:     end for
7:     return f
8: end function
```

2.3 Data Compression Standards

After introducing some methods from data compression, we will concentrate on compression standards for some types of data. Many of them have their origins in the 1990s, and while they have performed well [4, 3, 2], only few improvements have been made to them in the last decades [25].

2.3.1 JPEG

JPEG is a typical example of a format that combines lossy and lossless compression. In the lossless part, it exploits limitations in the human perception. An overview of the JPEG encoding is shown in Figure 2.18 and described in the next paragraph, based on [17].

In the first step, the color space is converted from red, blue and green (RGB), each represented by eight bits, into $Y'C_BC_R$, with Y representing brightness (or luma) and C_B and C_R the difference of blue and red with the luma component, also called chroma components. This separation allows to reduce resolution of the chroma components as they are less perceptible by human vision than brightness. This step, known as chroma subsampling, reduces the file size about a factor of

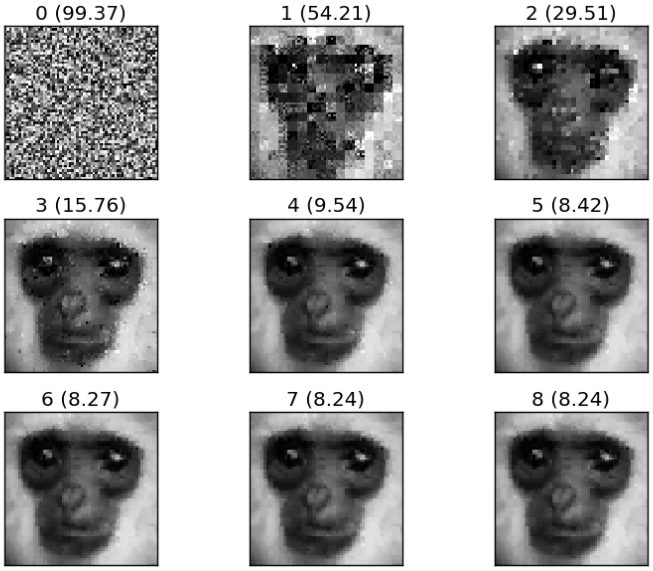

Figure 2.17: Decompression of an image using fractal image compression. In the first iteration, f is a random image matrix and is subsequently improved by applying W. The image labels show the iteration and the total error to the original image. Source: `https://github.com/c00kiemon5ter/Fractal-Image-Compression/`

two. Then, the image is split into blocks of 8×8 pixels and a discrete cosine transform (DCT) is applied on the three colourspace components. This allows the 8×8 block to be described by a superposition of frequencies instead of an explicit pixel representation. This step is based on the intuition that if an image is subdivided into small blocks, these contain correlations in the form of shapes that can be approximated by linear combinations of waves[15]. This step is an example of a decorrelation transform, mentioned at the beginning of this chapter. Because the human eye cannot perceive high-frequency variations as well as low-frequency variations, a nonlinear quantisation is applied. It encodes high frequencies with less precision and fewer bits, while low frequencies are encoded with higher precision. Finally, the quantised values are flattened using a zigzag reordering and then entropy-encoded. Typically, Huffman coding is used because Arithmetic coding was patent-encumbered at the time of standardisation.

Recent works have replaced the Huffman coding with Arithmetic coding, achieving a 23% size reduction on average while maintaining compression performance [25]. Furthermore, JPEG illustrates the compression gains that can be achieved by exploiting the image's structure by exploiting a bias in perception (chroma subsampling and quantisation), but also in representation (DCT and Entropic coding). In total, JPEG achieves size reductions by factors ranging from 1:5 to 1:120 [62], compared with lossless formats such as GIF and PNG that reduce the file size of images by factors of 1:4 to 1:10 [63] while already exploiting some characteristics specific to images. With generic compression tools as ZIP, Gzip and 7-Zip, factors of only 1:1.10 to 1:1.25 are achievable.

2.3.2 MP3

Uncompressed audio signals require significant amounts of storage space, and universal compression algorithms achieve compression ratios of up to 1:1.5 (7-Zip), or less. Specialised lossless

[15] Note however that this assumption does not hold for all kinds of images, such as images containing text or graphics.

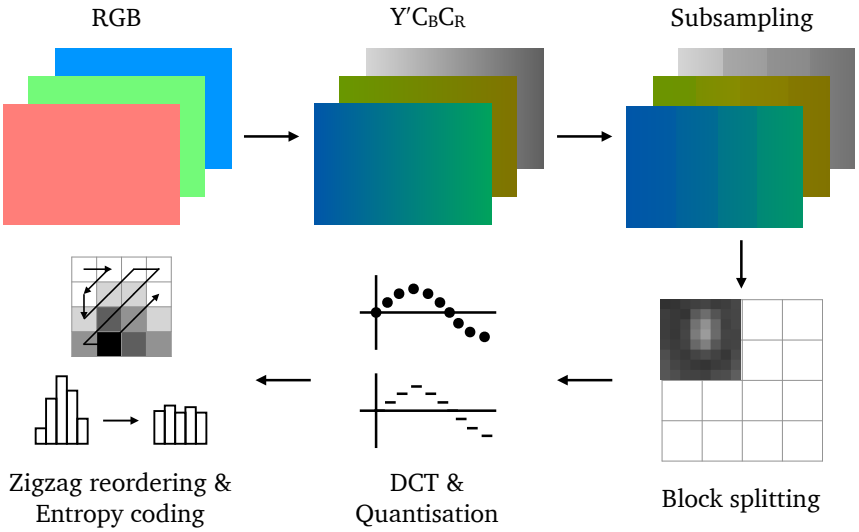

Figure 2.18: Overview of the JPEG encoding.

compressors for raw audio formats achieve compression ratios of 1:1.5 to 1:3.3 [64], while more specialised formats like FLAC achieve higher ratios. Using a similar rationale as JPEG to discard information less relevant to human perception, MP3 employs a series of lossy and lossless compression methods, achieving compression ratios of 1:4 to 1:12 [65]. Although MP3 has been developed as MPEG-1 Layer III in the 1990s, its principles are very similar to more modern codecs while featuring more implementations and descriptions, providing easier analysis and modification [16].

An overview of the MP3 scheme is shown in Figure 2.19. In the following paragraph, a simplified description is made. Audio data is delivered as uncompressed waveform consisting of samples at a sample rate of N samples per second (e.g. 44100 Hz). Each sample is a number representing the audio pressure at sample time. This format is typically obtained as an output of an analog-to-digital converter (ADC), for example when using a microphone on a digital computer.

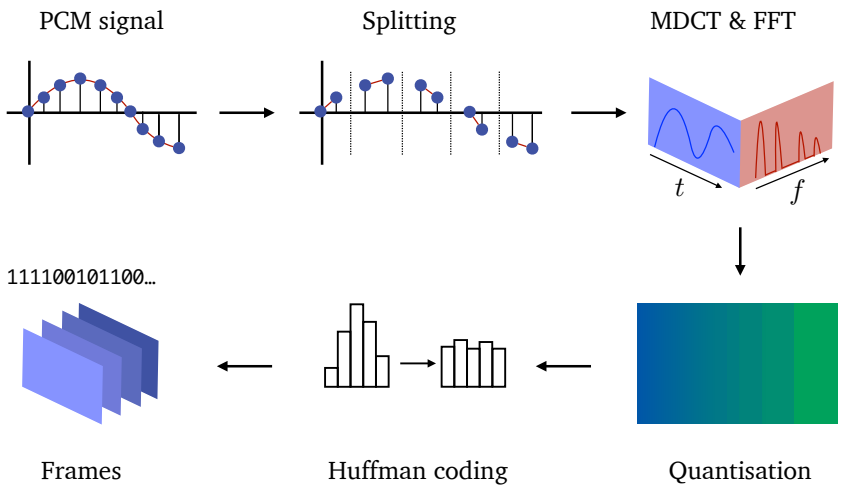

Figure 2.19: Overview of the MP3 encoding.

[16] Furthermore, AAC as the successor of MP3 is still patent-encumbered[66].

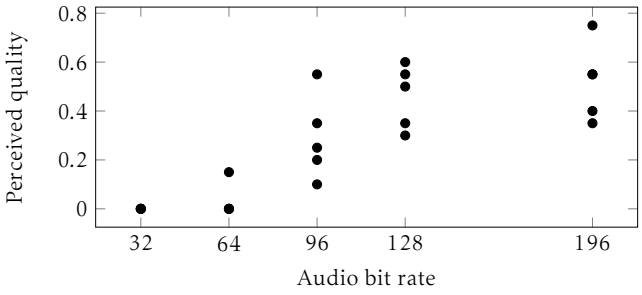

Figure 2.20: Perceived audio quality in relation to the MP3 bit rate [68].

In the first step, the samples are grouped into frames, each consisting of normally 1152 samples. In a second step, each sample is converted by a *Fast Fourier transform* (FFT), converting the time-dependent signal into a frequency spectrum. This spectrum is not used for encoding directly, but to inform the a *psychoacoustic model* that removes information that human perception is less sensitive to. This typically includes discarding information on frequencies human perception is insensitive to and masking, the removal of similar sounds appearing in a short time interval to which the human auditory system is less sensitive[17]. Now, a *modified discrete cosine transform* (MDCT) again converts the wave-like form of the audio signal into a frequency spectrum. In contrast to the previous FFT, MDCT is applied on overlapping blocks. The obtained frequency values are real-valued numbers. For easier encoding, they are quantised as integer values and finally Huffman coded. The Huffman codewords are combined with information about parameters chosen in the encoding and about the audio stream into a MP3 frame that can finally be written as a bitstream.

The MP3 format heavily relies on perceptual coding, where it gains most of its compression effectiveness [65]. This theses' emphasis does not lie on perceptual coding, and therefore *auditory masking*, and the *psychoacoustic model* are not discussed in more detail. An extensive documentation of them is given by Lagerstrom [20] Raissi [65] and the official ISO standard [67]. We conclude this the introduction of MP3 by illustrating the relationship between loss of information (the lower the bit rate the higher the loss) and the perceived quality of MP3 encoded audio in Figure 2.20. Evidently, with small bit rates, the quality is perceived as poor, but the difference from 128 to 196 bits per second is less dramatic, suggesting that the effect of information loss is non-linear.

2.3.3 Other lossy Audio Compression Standards

Next to MP3 there exist various audio compression standards such as the Advanced Audio Codec [66], or Vorbis [69]. Vorbis has been intended as Open-Source alternative to MP3 and AAC, resolving several of MP3's limitations [69], but uses similar concepts as MP3, such as the MDCT and Huffman coding for increasing entropy but makes heavy use of sophisticated vector quantisation [70].

[17] Different psychoacoustic models exist for different types of audio, such as music or speech.

2.3.4 Lossless Audio Compression

With MP3, a lossy audio compression standard was examined. Although MP3 and its successors have been dominant in audio compression, several lossless audio compression standards exist. Clearly, as perceptual coding is ruled out in lossless compression, more generic compression methods have been employed. In the following paragraphs, an overview of theoretical concepts and implementations is given.

At a high level, the task of lossless audio compression is to compress a waveform, often in the form of Pulse-code Modulation (PCM) signal because, as mentioned above, transforms as STFT lead to loss of information. A PCM signal is represented as a sequence of amplitudes obtained at a sample rate f_s (see Figure 2.21). Traditionally, the sample rate is chosen at or above the Nyquist rate

$$f_S \geq 2f_{max}$$

with the a signal that has a maximal frequency of f_{max}. The intuition is that if the signal consists of cycles, each cycle requires two samples to be detected.

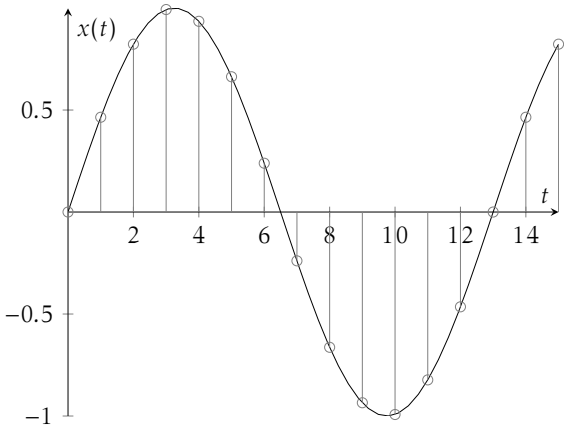

Figure 2.21: The continuous signal $x(t)$ (black) is sampled at discrete intervals (grey).

However, the maximal frequency f_M of typical signals is not present at all times, and is described by the sparsity property. Based on this principle, *Compressive Sensing* (CS) attempts to reconstruct the original signal with fewer samples as suggested by the Nyquist frequency [71]. The WAVE format is a widespread standard to define how waveform signals are organised and stored on computers, but comprises no compression by itself.

FLAC The free lossless audio coded (FLAC) compresses waveform audio data losslessly and experimentally achieves compression ratios of 1:2 to 1:3. It operates completely without transforming the audio signal into the time-frequency domain. Each file starts with a header, and the audio data is encoded separately in frames, as MP3 and its successors [72]. The number of samples per frame is variable, but defaults to 4096 waveform samples, or about four times as many as in a MP3 frame. Correlation of stereo signals is reduced by introducing a middle-channel averaging over left and right channels and storing the difference of the two as a side channel, without losing information. Samples are then approximated by either fitting polynomial functions of different orders to the signal, or by *linear predictive coding* (LPC) depending on the user setting, and trading higher compression gain with more computation cost. More than three quarters of the size reduction is already achieved with first and second degree polynomials, and higher orders

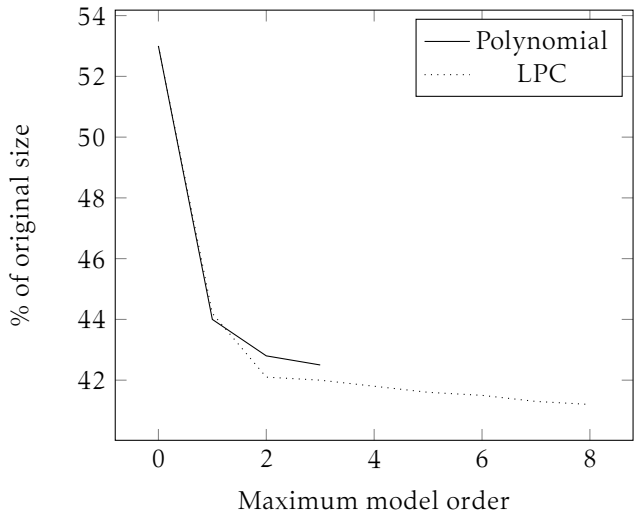

Figure 2.22: Compression ratio achieved by SHORTEN (on which FLAC is based) with respect to the model order [73]. Note that the compression with zeroth order is solely achieved by compressing the residuals.

generally do not improve compression (see Figure 2.22). Meanwhile, LPC can gain slight size reductions with orders higher than 2 [73].

The errors (or residuals) between the approximation and the original signal are kept to preserve information and coded separately separately. As the residuals are roughly distributed by a Laplacian distribution, they are amenable to Golomb coding. LPC assumes that the audio signal is autoregressive and that a sample $s(t)$ can be approximated as a linear combination of previous samples as

$$\hat{s}(t) = \sum_{i=1}^{p} a_i s(t-i)$$

where p is the order of the model and the residual is $r(t) = s(t) - \hat{s}(t)$. Here, the coefficients of the predictor a_i need to be stored along with the residuals. FLAC is built on insights of SHORTEN, which explains linear prediction and coding of the residuals in more detail [73].

Cyclic Pattern Decorrelation The task of audio compression using repetitions over a temporal axis has already been treated by Jehan [74]. The author first applies STFT, and extracts perceptually important 200 ms long segments (time slices of the audio sample) within the spectrogram after several transformations. Perceptual importance is determined using an auditory model. The extracted segments are then used for a dictionary-like compression. Still, the issue remains that segments along the temporal axis contain correlation in the form of superimposed sounds. This issue is revisited as *composite patterns* at the end of this chapter.

Jehan's intuition of music as an "event-synchronous path within a perceptual multidimensional space of audio segments" [74] is an interesting reference to topology, a connection that is discussed below.

2.4 Relation to other Topics

The ability to compress data is related to the structural understanding of it — for example by modelling character sequences as Markov chains that represent structures such as words or sentences as probabilities, and improve text compression significantly. Therefore, fields and their methods that examine the structure of data may have applications in data compression and some potential candidates are reviewed in this section.

2.4.1 Topology

Topology is a subfield of mathematics concerned with the properties of geometric objects invariant to deformations such as stretching or twisting. Informally, it can be imagined as "rubber-sheet geometry", as two objects that can be deformed into each other without creating or closing holes, cutting or connecting, are topologically equivalent [75]. In contrast, geometry is concerned with the properties that change with deformations, such as areas, distances, angles or curvatures [75]. Therefore, two objects that are topologically equivalent must not be geometrically equivalent (an example of this is the chaotic attractor at the end of this chapter). Another example is given in Figure 2.23.

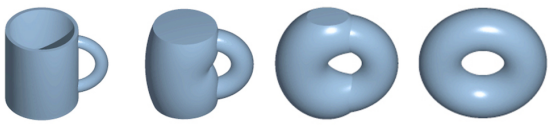

Figure 2.23: The homeotopy between a mug and a torus is a classic example from topology. While the geometry (distances, angles, ...) change, the topological properties are preserved. Source: `https://en.wikipedia.org/wiki/User:LucasVB/Gallery`

Topology has applications in biology [76], chemistry [77] and materials science (e.g. [78]), physics (e.g. [79]) and computer science, where it is known as computational topology. Computational topology is traditionally focused on topological data analysis [80], computer graphics, and in connection to Biology. However, as tools from computational topology are concerned about understanding the "shape" of data, they should be applicable to data compression as well because the structural understanding of data is directly related to the ability to compress it [9].

Topology has found few applications in compression yet. Edelsbrunner et al. have suggested an an algorithm to simplify functions based on tools from topology [81], but do not describe it as data compression method, but as a method to remove topological noise. The *Topological Signal Compression* (TSC) library[18] is based on that work. A lossy compression algorithm for volumetric data due to Soler et al. uses persistent homology to control the quantisation error [7]. In contrast to data compression, data analysis [80], signal processing [82, 83] and machine learning [84, 85], methods from topology have obtained more attention in recent years, especially for time series analysis.

Before reviewing topology for data compression, we give an overview to a few relevant concepts from topology. A introductory text to computational topology containing algorithms for the below concepts is due to Edelsbrunner and Harer [86].

[18] `https://geomdata.gitlab.io/topological-signal-compression/`

Morse-Smale Complex Informally speaking, the Morse-Smale complex is a set of cells on a manifold (such as a surface or a vector field) with the property that points within a cell have a similar behaviour with respect to the gradient. In case of a vector field, all points within a cell have uniform flow. As data collections from physics and engineering can be placed on such a manifolds, they can be analysed using the Morse-Smale complex [87]. A manifold could be an optimisation landscape, and a point within a cell would lead to the same (local) minimum or maximum when following the gradient.

Although the Morse-Smale complex finds only an indirect application in this thesis, understanding some of its concepts provides intuition for the following topics in topology. Furthermore, the Morse-Smale can be seen as a simplified representation of the manifold, achieving a kind of compression. Several technicalities are left out to make the following explanation more crisp. A more rigorous account is given in Edelsbrunner et al. [87].

For illustration, consider the manifold M in Figure 2.24. Maxima, saddles and minima are called *critical points* of M, and are connected via *integral lines* (a saddle is defined as a point on which the derivative is zero, although it is neither a minimum nor a maximum). An integral line l has its origin $orig(l)$ at the maximal point (or the higher of the two points) and its destination $dest(l)$ at the minimal point (or the lower). Integral lines cover all non-critical points of the manifold (not only the shortest lines between critical points).

Using integral lines, the manifold can be decomposed into regions that either flow to or away from critical points, and these regions are again manifolds. Concretely, the stable manifold $S(p)$ and an unstable manifold $U(p)$ are are defined as

$$S(p) = \{y \in M | y \in l, dest(l) = p\} \cup \{p\} \quad \text{and} \quad U(p) = \{y \in M | y \in l, orig(l) = p\} \cap \{p\}$$

This is, a stable manifold $S(p)$ of critical point p consists of all the points y that lie on an integral line l which leads to p (or that flow to p). The unstable manifold conversely consists of all the points on integral lines that originate from p (or flow away from p). For a maximum a, the stable manifold is only a and conversely, the unstable manifold of a minimum b is b. A cell is the intersection of the stable and unstable manifolds $S(p) \cap U(q)$ of any two critical points p and q. This means that when following an integral line from any of the points within a cell, the same destination and the same origin will be reached.

The Morse-Smale complex on the manifold M can now be defined as the collection of all cells of a manifold M and any two critical two points p and q

$$Morse - Smale(M) = \{S(p) \cap U(q)\}$$

Persistent Homology Persistent homology is a method to examine how topological features (described using homology) persist while varying geometric features such as scale. Although the concepts are traditionally used for mathematical objects, they can be applied on data [80]. There, persistence has valuable properties because the source of data has no intrinsic scale or metric, such as music that revolves around a contextual rhythm rather than exact timestamps, or that data may be missing or be noisy.

For example, the persistence of a 1-d data sequence can be characterised by a persistence plot. It gives clues about how persistent maxima and minima are in relation to each other and allows simplification of the data [88, 81]. More generally, it can help to identify e.g. holes, voids, or

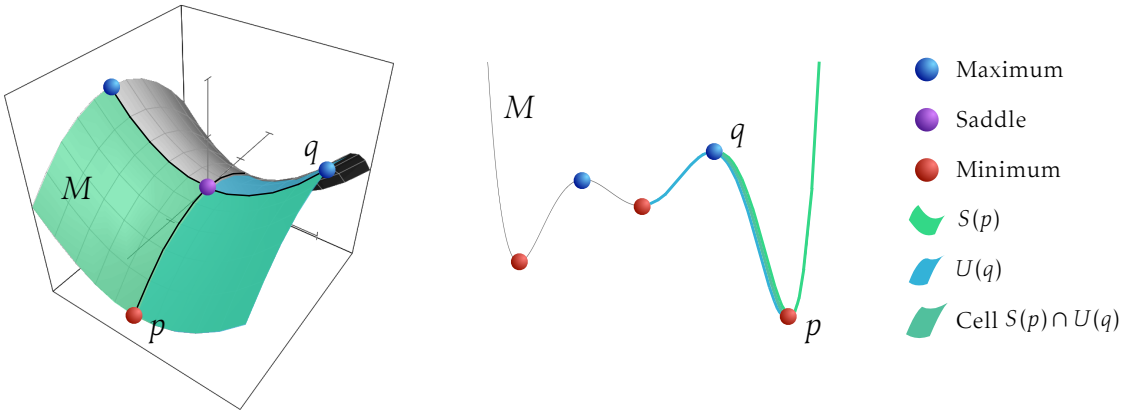

Figure 2.24: Examples of 2-d and 1-d manifolds. Stable and unstable manifolds $S(p)$ and $U(q)$ of two points p and q are highlighted in green and blue, and their intersection is highlighted in turquoise (in the 1-d case, these are adjacent for visibility). When following an integral line from any point on the cell, the destination is p and the origin is q.

loops independently how these structures are scaled [89].

As a mathematical concept, persistence is often defined algebraically, building on the formalism of topology. Instead, we will give a description by example. Consider the plot shown in the left panel in Figure 2.25. When sweeping a line from the bottom of the plot to the top, the minima encountered become the first element in a cluster of connected points. In the example, the rightmost point (13.5, 1) will create the first cluster and the point (2.5, 2.5) creates the second cluster. The y-component of the first point will be denoted as birth on the persistence diagram. If the line encounters a non-critical point, it will be assigned to the it is connected to (one can imagine that the lines connecting the points as belonging to the clusters, too). For example, the non-critical point (1, 4) will be assigned to the second cluster. When a local maximum is encountered, it will be connected to two clusters. The clusters merge and the younger cluster dies ("elder's rule"), terminating the path of the founding point on the persistence diagram. This occurs the first time when the local maximum at (4.5, 4.5) is encountered. From this point two clusters survive and eventually merge at the global maximum (11.5, 11), where it terminates the paths of the founding points of the merged clusters.

Persistence also be applied to data of higher dimension (such as 3-d models of molecules), and there exist efficient algorithms to compute it [88].

With persistence, critical points can be ordered by their lifetime, points with longer lives are more important. The easiest simplification (or compression) is given by only preserving the critical points, but additional simplification be achieved by eliminating critical points whose lifetime falls below a certain threshold. Alternatively, it is possible to define a number of points to keep, making this parameter independent from the data. It cannot be determined which points have been discarded only by the compressed data. This requires that the indexes of the discarded data are stored separately. The data is then reconstructed by interpolating the missing points. Depending on the type of data, different interpolation methods (polynomial, cubic, spline) may be chosen. The compression ratio depends, again, on the data and the threshold for discarding critical points. When compressing audio data from MP3 in the next chapter, compression ratios of up to 1:2 are achieved.

This behaviour can be summarised in the PHSimplification subroutine. It is based on the persistence diagram, which can be obtained by Algorithm 4 due to Edelsbrunner and Harer [86]. As

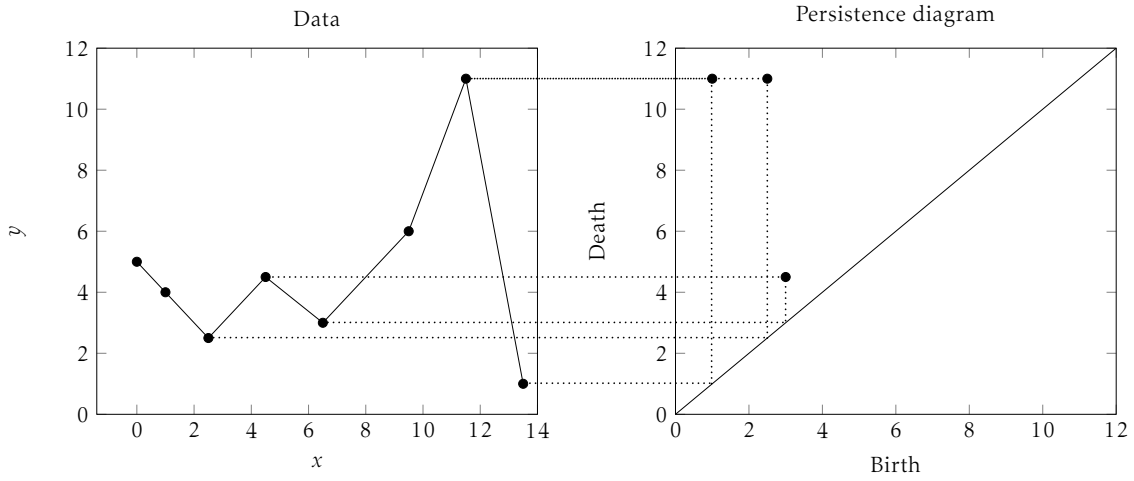

Figure 2.25: In the left panel, signal data is shown, on the right panel its associated persistence diagram. Data points are taken from the TSC library.

its input, the algorithm receives the 1-d data sequence such as a time series as graph (V, E) with $N+1$ vertices $v_j \in V$ and $j \in \{0,\dots,N+1\}$ being the data points with their height and time (which is only the index if the points are equidistant) and the edges $e_i \in E$, $i \in \{0,\dots,N\}$ the connections between successive points. In the algorithm, sets such as the ones of vertices and edges be accessed using their index starting from zero as for example $E[3]$ to access the fourth edge. The height and index of a vertex v can be accessed using $v.height$ and $v.idx$. The height and index of an edge between (v_0, v_1) are defined as $max(v_0.height, v_1.height)$ and $\arg\max_i(v_0.height, v_1.height)$. C is an ordered set of indexes corresponding to vertices. The i-th index is a reference to the root of the vertex of position i. For example, if $C[5]$ is 9, then the root of vertex v_5 is v_9. The root of a vertex can be found by resolving references $i \leftarrow C[i]$ until a i is found that fulfils $C[i] = i$. The subset of vertices that have the same root is called component. The complexity of the algorithm is $\mathcal{O}(N^2)$.

Algorithm 4 provides the the persistence diagram which allows to discard points based on their persistence (discarding the ones with least persistence). If, however, only the critical points were to be used, an easier method, drawing from the definition of critical points on continuous functions can be used. The definition is as follows. On a curve defined by the equation $y = f(x)$ a point x_c is critical if $f'(x_c) = 0$, where f' is the first derivative of f. For a sequence of N data points, the method iterates it e.g. with incrementing an index $0 < i < N$ and compares values at $i-1, i, i+1$. If the value at i is larger or smaller than both of the two, i describes a minimum or a maximum, respectively. If it is equal to one or both, one of them is a saddle point. It is sufficient for the method to pass the data sequence once, making its time complexity $\mathcal{O}(n)$ and thus faster than Algorithm 4.

Graph-based topology In the same vein as data analysis applies concepts from topology, Barbarossa and Sardelliti suggest applying topology to signal processing [83] This connection is promising for data compression as it has its roots in signal processing, and standards as MP3 and JPEG make heavy use of classical signal processing methods. Topological signal processing introduces graphs to incorporate relationships between data points, and extend graph analysis with topological structures such simplicial complexes. With this extensions, topological signal processing can reveal structure beyond pairs of nodes, which is the focus of traditional graph analysis, to groups of nodes and edges.

Algorithm 4 Method to compute the persistence diagram

```
1: function Main(1-d sequence as graph (V, E))
2:     N ← |E|
3:     C ← V                                  ▷ Initially, every vertex has itself as root
4:     n_c ← |C|                              ▷ Number of components
5:     T ← ∅                                  ▷ Mergetree
6:     B ← {arg min_i(V)}                     ▷ Births, first element is index of global minimum
7:     D ← {arg max_i(V)}                     ▷ Deaths, first element is index of global maximum
8:     for i ∈ {0, ..., N} do
9:         if n_c ≤ 0 then
10:            break
11:        end if
12:        src, dest ← E[i]                   ▷ Get pair of successive vertices
13:        a, b ← root(src), root(dst)        ▷ r gives the root component of an edge
14:        if a.height < b.height then
15:            C[root(b)] = root(a)           ▷ b's root becomes part of a's root (elder's rule)
16:            if b.height = e.height then    ▷ dst is a non-critical point
17:                continue                   ▷ Skip this iteration i
18:            end if
19:            B ← B ∪ (b.idx, b.height)
20:            D ← D ∪ (e.idx, e.height)
21:            T ← T ∪ (e.idx, (root(a), root(b))   ▷ Insert e.idx as point where roots of a and b
    merged
22:        end if
23:        n_c ← n_c − 1                      ▷ With the merge, a component has been lost
24:        Symmetrical case for a.height > b.height is omitted
25:        if a.height = b.height then
26:            Proceed as above, but depending on the index instead the height
27:        end if
28:    end for
29:    return (B, D, T)                       ▷ The persistence diagram can be obtained by B and D
30: end function
```

Next to persistent homology, Munch notes the mapper graph as applicable to data analysis [89]. It consists of a filter function that assigns data points to nodes of a graph. With it, data points can be represented as a graph, that if successful, captures the structure and relationships of the underlying data. Although she introduces it as a tool for data analysis, it can be seen as a kind of compression, and if the node maintains information about the distribution of the points it represents, the graph may be used to approximately reproduce the original data.

2.4.2 Micro-Macro Problem

Without being mentioned explicitly, many of the methods introduced in this chapter attempt to solve the problem of extracting larger-scale, abstract information such as pitch from a waveform, or character or word patterns in natural language from the smaller-scale, concrete byte representation, which we will call the *micro-macro problem*. The problem could be stated as: how is the structure of a source (e.g. an image, an audio file, or a text) as it is perceived by humans related to the byte-level representation?

Figure 2.26 shows a spectrum of abstraction levels and several examples of data compression methods. Order-0 entropic coding has no macro-level understanding beyond the probability distribution of symbols. Meanwhile, data compression standards as MP3 and JPEG employ some

structural insight (e.g. that high-frequency components are less important for perception). The highest abstraction level may be given by conceptual representations, which may be achieved with neural networks [10, 52]. Although neural networks can find ways to relate these macro-level concepts to micro-level representations (e.g. autoencoders, see Figure 2.12), these ways are latent and difficult or impossible to interpret [52].

The micro-macro problem is already discussed in the context of data deduplication and delta encoding (and is shown in Figure 2.10), where it is related to the kind of data. Solving this problem for some kind of data and some macro level would indicate that data of that kind can be compressed to the macro level. The micro-macro problem defines the representation part of the artificial intelligence problem of compression suggested by Mahoney and Hutter [19, 21].

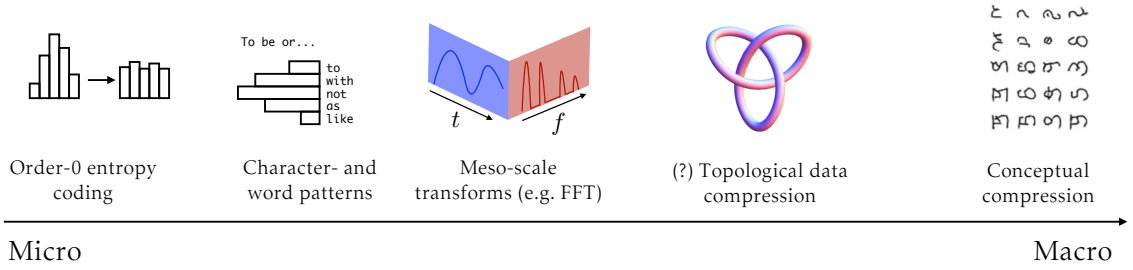

Figure 2.26: Several data compression methods with respect to their highest abstraction level they can relate to the micro-level information (every method has a byte-level output). Higher abstraction are usually more lossy, but achieve higher compression ratios. Thus, a higher abstraction level is not necessarily better if data needs to be compressed without loss. Topological data compression methods are rare, but they may achieve a relatively high abstraction level. Source of torus knot image: `https://mathworld.wolfram.com/TorusKnot.html`. Source of conceptual compression symbols: [52].

2.4.3 Decorrelation, Composite Patterns and Subdivided Neural Networks

The task of decorrelating data is related to the task of finding patterns within data that appear independently. For example, consider images containing shapes as triangles, circles and squares (see Figure 2.27). The images are composite patterns one way to encode them is to use the name of the shape and its position, such as *circle top-left* or *triangle bottom-right*. A more efficient approach were to subdivide the encoding operation: one component of the encoder is responsible to recognise the shapes (component 1, similar to a dictionary), another part is concerned with the position of the shape within the image (component 2). This is possible because the shapes and their positions are uncorrelated, and the subdivided components can operate independently. In the first approach, there are $\#shapes \times \#positions$ possible terms to be encoded, while in the second there are $\#shapes + \#positions$.

However, such easy subdivision this is often not possible with real-world data because there may not be a clear separation from properties such as position and shape, and properties cannot be classified into discrete categories.

This problem is also relevant in audio compression, and illustrated by Figure 2.28. If, for example, an audio sequence is composed of precise copies of a sound occurring after each other, these copies are independent from each other and can be compressed using a simple dictionary. A repetition in music may be given by a bass, but may be accompanied by a melody, breaking the independence between successive repetitions, as soon as the bass and melody have not precisely the same period. This new correlation extends along the temporal axis, but if the melody and bass

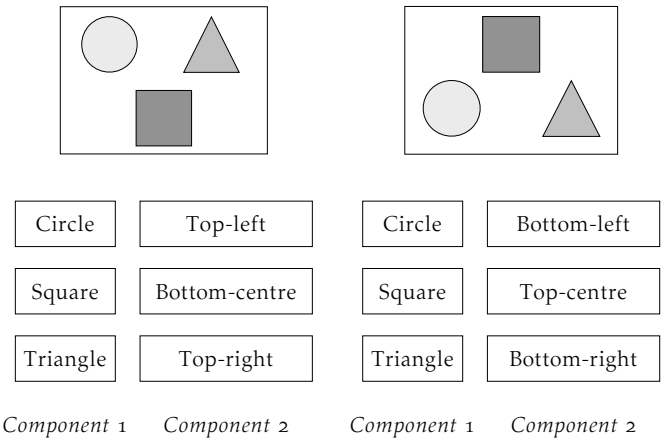

Figure 2.27: Because shapes and their positions are uncorrelated, they can be processed independently. Theoretically, the position description could be subdivided another time by separating *top* and *bottom* from *left*, *right* and *centre*.

exclusively occupy different frequency bands, they can be subdivided and again and decorrelated on the frequency axis. In practice, there are several issues with this notion that will be discussed in later chapters.

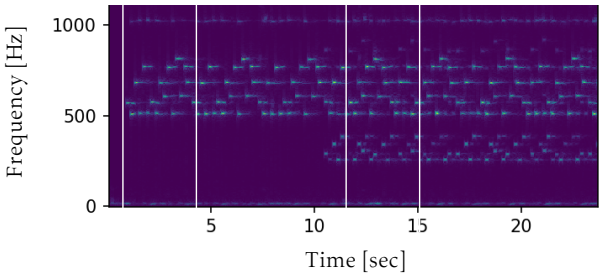

Figure 2.28: The repetition marked by white lines on the left is an independent repetition. The one on the right is however correlated with the bass and may be decorrelated by separating the bass in the frequency spectrum.

Bar-Yam [10] discusses subdivided neural networks that can, in principle, target properties such as shape and position, or time and frequency with a subdivision while being able to untangle some degree of correlation. In practice, untangling such composite patterns has remained a problem for neural networks where it is known as *binding problem* [90, 91]. The binding problem is subject of current research and is addressed as a weakness of the neural network paradigm [91].

2.4.4 Dimension and Structure

Dimensionality reduction is used frequently in fields such as data science and data visualisation [92]. The motivation for this is that the performance of statistical and computational methods often degrades when the dimensionality of input data is high, a phenomenon known as the *curse of dimensionality* [93]. The goal of dimensionality reduction is therefore to represent data within fewer dimensions without discarding aspects that are important for the task. A popular method is the principal component analysis already encountered above, collapsing the original features $X_1, \dots X_p$ as linear combinations into principal components, the first one being

$$Z_1 = \phi_{11} X_1 + \phi_{21} X_2 + \cdots + \phi_{p1} X_p$$

and with the largest variance. Each subsequent principal component is statistically uncorrelated to the prior ones and have decreasing variance [92]. Dimensionality reduction is achieved by only using s principal components, usually with the highest variance, and discarding the others.

For example, an ecological dataset of trees of a forest may contain many features, such as the height, circumference, geographical coordinates, and date of measurement. Some of the features may even be empty, or contain similar values in most samples. For visualisation and exploration of the data, one may want to obtain a two-dimensional representation, while maintaining natural clustering or spread of the data. Using PCA, the two largest principal components, containing most of the variance, may be plotted in a diagram, showing the statistically most varying features.

The task of dimensionality reduction can be seen as an example of a the more general task of *embedding*, in which the goal is to fit data into a an appropriate dimension that is specific to the task. In natural language processing, word embeddings have the goal of assigning vectors to words under the constraint that semantically similar words have similar vectors, and that similar difference vectors again carry similar semantics [94]. In nonlinear dynamics, embedding dimensions are sought that resolve the dynamics contained in lower-dimensional data.

For example, consider the Lorenz system in Figure 2.29. In the left panel a chaotic attractor is shown in three dimensional phase space (the time dimension is not explicitly shown), where the state of the system denoted by the line, oscillating in the butterfly-like shape. The right panel shows the system's projection of the x-axis with respect to time. The projection looks periodic (although it is not, as the repetition is not exact), from which the attractor can be reconstructed by *Takens's embedding* (or Delay embedding) [95]. The reconstruction is made by choosing a delay τ and a number of lags k and the projected values $x(t)$ are sampled at $t+\tau,\ldots,t+k\tau$ where the value at each lag $x(t+\tau),\ldots,x(t+k\tau)$ is projected back into k-dimensional phase space [96]. Intuitively, one can imagine to comb $k = 3$ evenly spaced vertical lines through the diagram in Figure 2.29 (b) from left to right. While combing through the diagram, the three values where the vertical lines intersect with the projection produce a dot in a three-dimensional space and reconstruct the attractor. While the topology of the attractor is preserved with the embedding if the conditions of Takens's theorem are fulfilled, the geometry is not, meaning that the "shape" can be reproduced, but not the exact size.

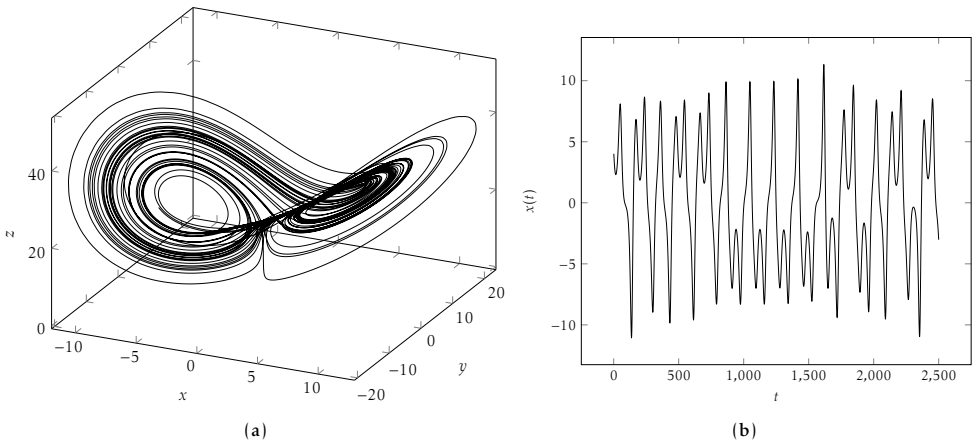

Figure 2.29: (**a**) The original three-dimensional Lorenz system, (**b**) The projection of the x-axis as a function of time.

The example further shows the connection of an appropriate dimension with compression and the example shown in Chapter 1 and Figure 1.1. Recall that in the introductory example, the

flattening of the 2-d image matrix into a vector obfuscated a structure that was clearly apparent in the image matrix. Similarly, the time-dependent projection obfuscates the dynamics of the Lorenz attractor in three-dimensional phase space. Both have in common that points that are close with respect to some semantical or structural assumption should be close in embedding space, allowing some sort of minimum description [96], which again relates to the Kolmogorov complexity. In general, the embedding dimension k is not known, and can sometimes be estimated empirically, or by an analytical understanding of the system.

Skrabe et al. propose a framework combining Takens's theorem with topological analysis [97]. They examine periodic, quasi-periodic and recurrent dynamical systems and reconstruct higher-dimensional topological structures from 1-d data. In contrast to the example in Figure 2.29 that uses one delay τ, the authors are able to extract multiple cycles with different delays — for example, a signal with two periods can be embedded on to a two-dimensional torus, yielding a kind of "topological Fourier transform" [97]. Due to the approaches' capability to identify periods at different time scales, it will be briefly discussed in Chapter 4 for the use on music signals.

With the discrete cosine transform and the discrete Fourier transform, functions were already encountered that add dimensions, generally known as *Zak transforms* [98]. Adding dimensions to find structure in data is also the concept of kernel methods used with Support Vector Machines, allowing them to separate data points using a linear decision boundary [92].

3 | MP3 Recompression

> We can only see a short distance ahead, but we can see plenty there that needs to be done.
>
> Alan Turing

Code reference. Code used for this chapter can be found in the `recompressor` subdirectory. Most of the MP3 decoding logic can be found in the `Frame.py` file, and compression is done in files `NaiveArithmeticBitCoder.py` and `ArithmeticBitCoder.py`.

This chapter addresses the first subgoal of this thesis — the compression of MP3 files. The motivation for this is threefold. Firstly, MP3 recompression is inspired by the success of Lepton, a system to recompress JPEG files by a factor of 1:1.25 on average [25], mainly by replacing Huffman coding by arithmetic on the entropic compression layer. As MP3 and AAC also employ Huffman coding, this seems a promising avenue.

Secondly, reducing the file size of music files could be used as an alternative to traditional streaming methods. Music streaming services such as Spotify have garnered significant popularity in the last decade and make heavy use on audio compression and aggressively optimising playback latency to around 270 ms [99]. Yet, Internet infrastructure has been designed for bursts of data rather than streams that induce significant overhead for buffer management and congestion control [100]. Although playback can start without fetching a complete audio track, establishing the stream is time-consuming and afterwards highly dependent on a reliable network connection as frequent requests for packets are made. As an alternative to streaming, the complete audio file could be transferred, but with current file sizes, latency would be in the order of seconds to minutes. However, with a significantly smaller file size, up front transfer may become viable, reducing dependency on network connectivity and the overhead of streaming management.

Thirdly, audio compression standards as MP3 have been designed for restricted computational resources in mind. For example, the sequential reading and decoding of MP3 files prevents that the complete file must be loaded wholly into the main memory and the processor-intensive decoding is distributed over the length of the audio track [1]. As mobile devices have gained computational resources since the inception of MP3, it is worth of seeking alternatives beyond the approach of MP3, profiting from larger-scale structure within an audio file with negligible additional load on computational resources. Although this idea is mainly pursued in the next chapter, it can already be applied *within* MP3 files.

[1] The requirements of the MP3 standards included that the length of addressable units were less than 1/30 s, and decoding delay be less than 80 ms [101], precluding efforts that would compress audio files as a whole.

3.1 Anatomy of a MP3 frame

An MP3 stream consists of MP3 frames, each typically encompassing around 26 ms of audio (see the MP3 subsection in Chapter 2 for an outline of the MP3 compression standard). A frame consists of a header, an optional CRC checksum, side information, main data and optionally, ancillary data appended at the end (see Figure 3.1).

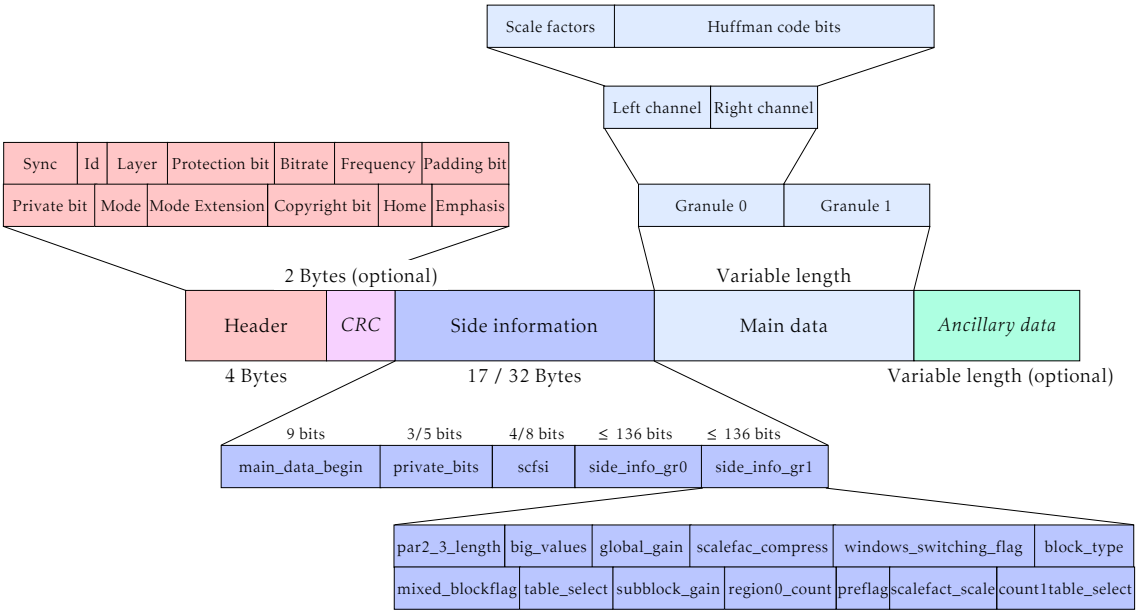

Figure 3.1: Layout of a single MP3 frame. The bulk of data is stored in the Main data section as Huffman coded bits. A typical frame is 417 bytes long. Detailed descriptions of the individual fields are given in Raissi [65] or the official ISO standard [67]

The byte length of a frame is determined by the bitrate r_{bit}, the number of samples $n_{samples}$ of the chosen protocol (typically this is 1152, but possible are 576 or 384) and the sample rate r_{sample} as

$$l_{Frame} = \frac{r_{bit}}{8}\frac{n_{samples}}{r_{sample}} = r_{byte}t_{Frame}$$

If the padding bit above is set, an additional byte is appended. For the typical case of $n_{samples}$ =1152, r_{sample}=44100 Hz, and r_{bit}=128000 this is 417 bytes, or 418 with the set padding bit. Consequently, such a frame spans about 26 ms.

The header is always 4 bytes long and starts with a *Sync* codeword of 12 set bits. The remaining contents define the MP3 frame's version (*Id*) and layer, the *Protection bit* determines whether the CRC block exists, *Bitrate* codes rates from 32 to 448 bits per second and *frequency* from 8000 Hz to 44100 Hz (typical). The *Padding bit* indicates that an additional byte is appended to the frame to fit bitrate requirements and the *Private bit* is application specific. *Mode* specifies whether the audio is stereo, joint stereo, dual or single channel, *mode extensions* allow further stereo settings for the joint stereo mode. The *Copyright bit* determines whether the audio contents may be legally copied, and the *Home bit* whether it is located on its original media. Finally, *Emphasis* bits indicate whether the decoder has to revert noise suppression previously applied.

The *Side information* section gives information about the location, encoding and quantisation of the *Main data* section. For example, the *big_values* and *region0_count* fields indicate regions where larger values or values close to zero are expected after Huffman decoding. Fields

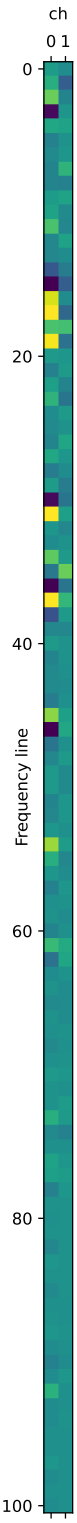

Figure 3.2: The amplitudes of the first 100 frequency lines of an MP3 frame.

table_select and *count1table_select* indicate which Huffman code table has been used for encoding — an interesting property of entropy coding in MP3 which will be examined below.

Main data contains is the largest section in an MP3 frame and consists of *Scale factors*, aimed at masking quantisation noise and *Huffman code bits*, containing the actual data. Decoding yields a $n_{channels} \times 2 \times l_{granule}$ array, with $n_{channels}$=2 for stereo audio, two granules and a granule length $l_{granule} = n_{Samples}/2$ of 576, as simplified in Figure 3.3. The granule length $l_{granule}$ is also the number of frequency lines obtained from the MCDT and is fixed and covers frequency bands from zero to the Nyquist frequency. The array values are MDCT amplitudes, and is divided into five regions along the frequency lines, characterising the quickly diminishing amplitudes along the axis. For example, *region0* contains the largest values (in the range {−8206,…,8206}) while *rzero* contains only zero. When decoding, these amplitudes are dequantised and sent to an inverse MDCT to recreate the waveform signal.

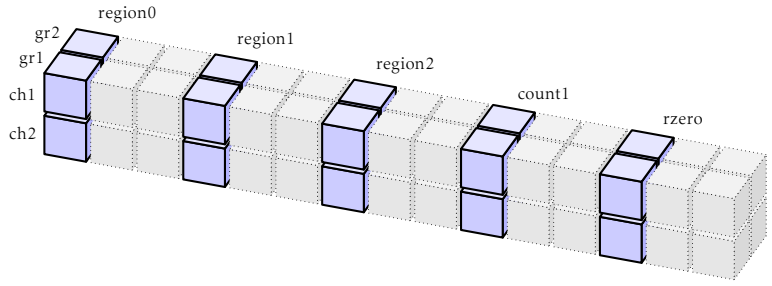

Figure 3.3: Array representation of main data without Huffman compression. Amplitudes are split by channel *ch*1 and *ch*2 and granule *gr*1 and *gr*2. The length of the array is 576, and contains 2304 values with two channels and two granules per frame.

Finally, *Ancillary data* allows appending user-defined data and is not standardised. More detailed descriptions of the MP3 protocol are given in Raissi [65] and Lagerstrom [20].

3.1.1 Entropy Coding in MP3

The non-uniformly distributed amplitudes along the frequency spectrum, as recognised by the regions described above can be seen as correlations, or structure appearing among all frames. Figure 3.4 illustrates this structure along several dimensions using MP3 frames from 33 seconds of classical music. Figure 3.5 shows the distribution of amplitudes over the 33 seconds of music and the frequency spectrum, revealing further structures over the time axis.

While most entropic coders generally have weak assumptions about the structure of data, i.e. only the probabilities of symbols as described in Chapter 2, its MP3 implementation entails many insights into the frame's structure, improving coding efficiency. In particular, it employs several Huffman coding tables based on the region (determining the maximum amplitude to be coded), structural properties and statistical properties of the region. See Figure 3.5 for an example frame excerpt, with colour-coded amplitudes. The rationale of this decision is that using an encoding table that offers more symbols than what appear in the data wastes codewords. By encoding two digits into one codeword in the *big_values* region, further efficiency is gained. The insight here is that not all possible integer pairs appear within that region. Another method is to encode the *count1* region, consisting only of values -1, 0 or 1 in quadruples, and to run-length code the *rzero* region as it only contains zeros. Correlations between the channels for joint stereo audio are also

exploited for more efficient coding. Additionally, the Huffman coder informs the quantisation step to discard data to meet the bitrate requirements, although Huffman coding itself is not lossy.

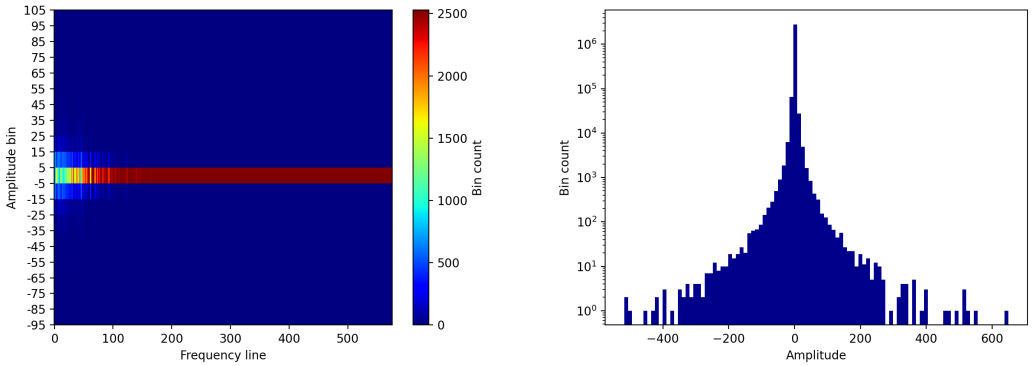

Figure 3.4: (**a**) Distribution of the amplitude / value of the uncompressed frequency lines. The colormap represents the count of amplitudes within the bin by increasing frequency lines on the x-axis. The upper two thirds of the frequencies are almost exclusively zeros or ones. (**b**) Distribution of the amplitudes summed over all frequency lines. This distribution can be obtained by viewing the diagram in (a) from the left, with the colourmap in (a) as the y-axis and the range of amplitudes extended.

Using these and further insights, how well can the coder compress the amplitudes in the *Main data* section? Assume an MP3 frame of 417 bytes, a four byte header and 32 byte side information, leaving 383 bytes, or 3064 bits for the main data. To be encoded are 576 amplitudes, two granules and two channels (2304 amplitudes) and scalefactors. Ignoring the scalefactors and dividing the bits by the amplitudes, we obtain 1.33 bits per amplitude. Table 3.1 displays the information content or bits required by the compression methods reviewed in this chapter.

The knowledge about a frame's structure is incorporated as explicit heuristics, such as the choice of different Huffman tables withing the MP3 encoding algorithm. In the remainder of this chapter, the concentration will be on finding alternative compression techniques that discover and exploit such structure, within single frames, but also on groups of frames.

3.2 Replacing Huffman Coding

The first approach is inspired by Lepton that replaces the Huffman coding in JPEG by arithmetic coding [25]. In general, we encode the 2304 amplitude samples of an MP3 frame, each represented as a 16-bit integer as a total of 4608 bytes. First, an arithmetic coder following [22] is used, encoding data bytewise. It encodes the frame to 768 bytes on average, yielding 2.67 bits per symbol and a compression ratio of 1:6. The probability model for the arithmetic coder is derived empirically using the same 33 seconds of music used for the diagrams above. Here, the probability model only considers the unconditional probability distribution of bytes, or order-0 probability. Although the amplitudes are represented as 16-bit integers, byte frequencies are modelled. This decision has two reasons: firstly, storing a probability table with all 16-bit values would require in the order of 2^8 more space. Secondly, the frequency table would be more sparsely filled, either requiring large samples to collect, or providing no probabilities for some values, worsening performance of the encoder.

In a second approach, we replace the byte-wise by a bit-wise arithmetic coder using a dynamically updated probability distribution of bytes, the compression ratio can be increased to around

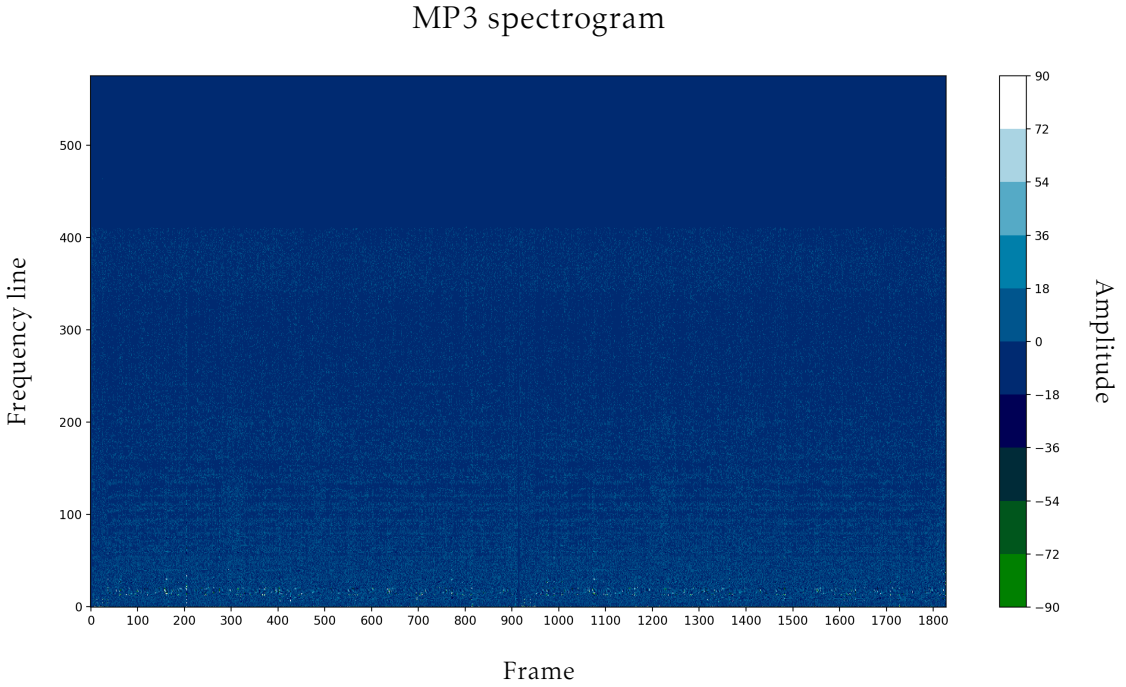

Figure 3.5: Heatmap of the amplitudes (colour) by frequency lines and throughout the first 33 seconds, or 2500 frames. Note that 0 is binned into $(-18, 0]$ and therefore in a different bin as small positive values. From frequency lines 400 up only zeros appear. Also, patterns along the time axis are visible. Notable are the repetitions around frames 500 and 1800 (magnification required).

1:6.45. Although the compression ratio would be satisfying for text, or general data, the encoding requires slightly more than double the space per amplitude value as the native MP3 coder. However, a naive Huffman encoder, only using one encoding table for all symbols, achieves 3.33 bits per symbol (or 959 bytes per frame), a factor of 2.5 more than MP3 encoding. Although the performance of the Arithmetic coder varies by the probability model selected, where a more precise model yields better results, the compression ratio never comes close to MP3 encoding.

3.2.1 Model extensions

Arithmetic coding already performs one of the heuristics employed by native MP3 coding by being able to encode the long sequences of zeros efficiently, while this is done via a separate run-length coding in MP3. This is done by assigning an extremely high probability to zeros, allowing to encode input bits with fractional bits[2]. This highlights an advantage of the dynamic probability model — it can adapt to the non-stationary probability of symbols such as long sequences of zeros. Similarly, other knowledge about the frame structure can be included into the probability model of the arithmetic coder, bearing the risk of shifting the MP3 hand-crafted heuristics into the probabilistic model. Still, several model extensions are reviewed in the next paragraphs.

2-d Probability Distribution A natural extension of unconditional arithmetic coding is to extend the probability model with the frequency of the current byte as context. This extension is used e.g. in the PAQ compressors and is similar to the regions used in MP3 encoding, incorporating the knowledge that high amplitudes are more likely to be found in low frequencies. To

[2] Recall that the smallest output of a Huffman coder is a bit, making the group-wise encoding a workaround for fractional bits.

evaluate this approach, a 2-d frequency table is defined as a $255 \times r \times 2$ matrix, where $1 < r \leq 576$ is the number of regions. The last dimension denotes the number of bits (zero or one). Within this approach, the order-0 case represents the extreme case of $r = 1$ and with $r = 576$, every frequency line is modelled by a separate byte distribution. The region sizes were evaluated empirically, with good results using 5-20 regions, or region sizes of 30-100 frequency lines. With the 2-d distribution approach, frames can be compressed to about 550 bytes on average, or a 1:8.2 compression ratio.

	Bits per symbol	**Compression ratio**
MP3 coding	1.33	1:12.03
Order-0 arithmetic coding (bytewise)	2.67	1:5.99
Order-0 arithmetic coding (bitwise)	2.48	1:6.45
Order-0 Huffman coding	3.33	1:4.80
2-d arithmetic coding (bitwise)	1.94	1:8.23
Topological compression with entropy coding	1.22	1:13.2

Table 3.1: Comparison of bits used to represent a single symbol. MP3 coding is able to encode data with fewer bits than the Shannon limit, using correlations in the data. All practical results assume that every amplitude is represented as a 16-bit integer.

An additional attempt was made to replace the dynamical byte-level by a 16-bit probability table, matching the size of the data type of amplitudes. However, no improvement was detected while incurring the drawbacks of larger sample sizes to counter sparsity and more space consumption.

Short-Term Biased Probability Distribution In this attempt, the probability model was modified to keep a shorter memory, emphasising the recent probability distribution more than the long-term distribution. The intuition is that sounds in music often persist over short intervals in time, spanning several frames and remain in similar frequencies, i.e. they are local in time and frequency. The model was implemented by limiting the bit counts to the last 128 to 2048 bits (steps in powers of two). However, this model does not improve coding performance.

An analysis of correlation among frames confirms that subsequent frames seldom correlate, possibly explaining the ineffectiveness of the short-term probability model. In Figures 3.6 and 3.7, autocorrelation plots of unpacked MP3 frames of classical music and electronic music are shown. As a measure of autocorrelation, the Pearson correlation coefficient is used. It should further be noted that the correlation has strong emphasis on lower frequencies as they appear with larger amplitudes. This observation is in strong contrast to video, where encoders rely on the similarity of subsequent frames, only storing the difference in a procedure using motion compensation [3]. In video, this similarity is expressed in a high autocorrelation that decays with time. In audio, the generally low correlation of subsequent frames preclude audio compression using difference-based compression techniques that apply for video.

3.3 Transforms

As alternative to further extend the probability model, transforms of the frame data before entropic encoding are reviewed now. The challenge to this task it not merely to find transforms that compress the data, but rather ones that decorrelate the data (and increasing the entropy) and allowing the entropic coder to compress more efficiently. If the transform achieves compression

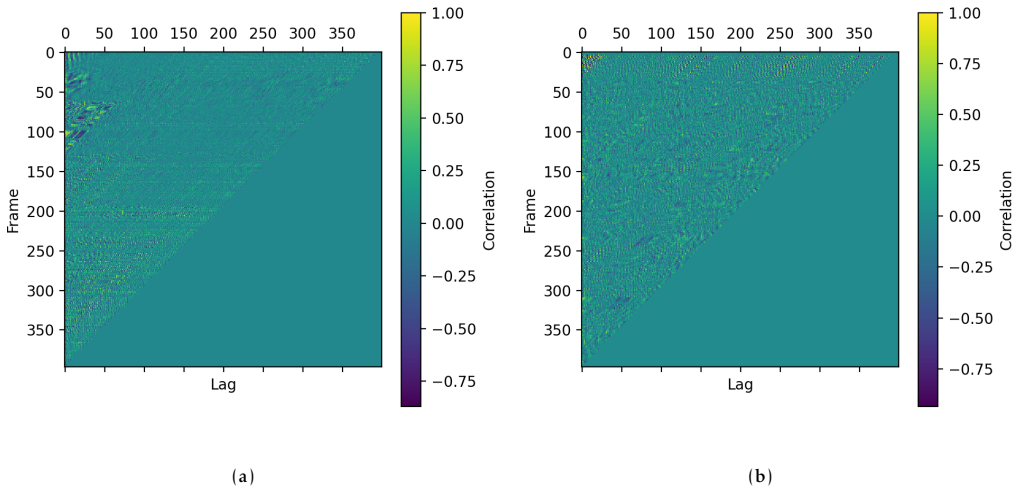

Figure 3.6: (**a**) Autocorrelation plot of unpacked MP3 frames of about 10s of classical music. Many short-term correlations and anticorrelations are visible (about 38 frames constitute a second of audio). (**b**) The autocorrelation plot of electronic music. Correlations are present on longer timescales compared to classical music and exhibit a wave-like pattern.

without revealing non-trivial structure (that is, structure beyond non-uniform distributions and run-length that arithmetic coders find natively), this is at the expense of entropic coding.

3.3.1 Dictionary

To exploit correlations that do not appear consecutively, a dictionary could be used to store frequent patterns. we choose a dictionary with a fixed length and evaluate it over two degrees of freedom: dictionary entries are $e \times f$ matrices with $1 \leq e \leq 10$ values along frames and $1 \leq f \leq 10$ frequency lines. The rationale to allow dictionary entries to span several frames is similar to the short-term probability model above, i.e. that sounds may be spread over several frames and thus be captured by the entries. Furthermore, allowing entries to be rectangular (rather than square), the asymmetry of the temporal and the frequency axis can better be accounted for. The compression ratio is determined as the bytes required to store the MP3 samples versus the bytes required for the dictionary plus the references to the dictionary entries.

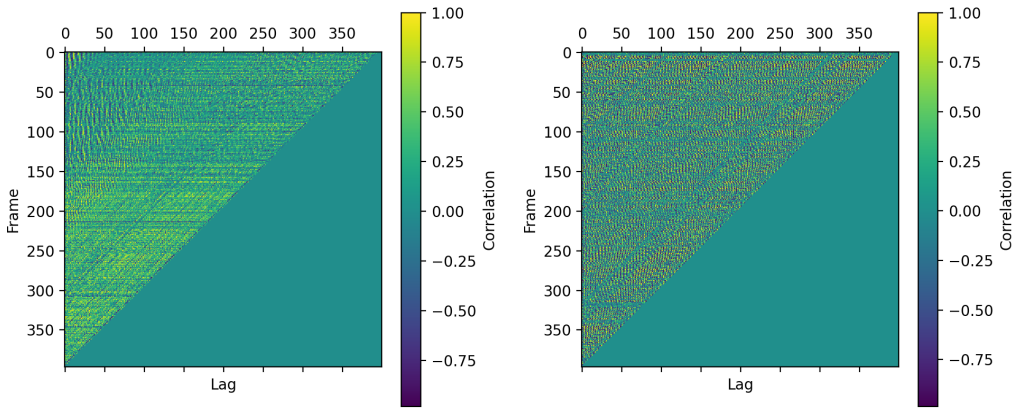

Figure 3.7: **(a)** Autocorrelation plot of only the lower 10 frequency lines of classical music. The (anti-)correlations are significantly stronger and more patterns appear to emerge. **(b)** The lower 10 frequency lines of electronic music. The patterns themselves seem to repeat, possibly indicating a repeating bass line with a notable interruption at about frame 310.

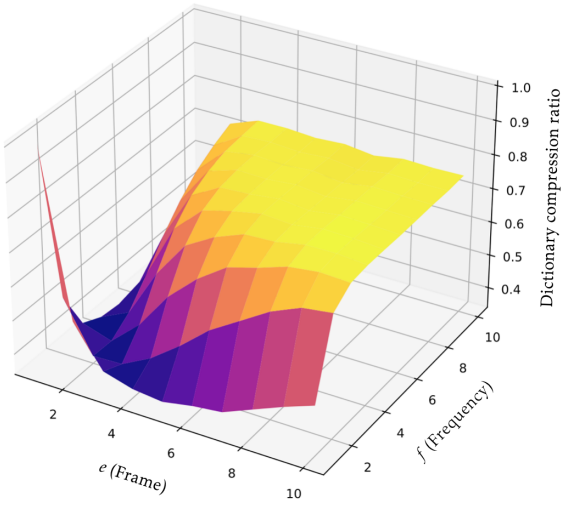

Figure 3.8: Dictionary compression ratio w.r.t. values along subsequent frames e and samples within frequency lines of that frame f. The minimum is reached at $e = 5$, $f = 1$

Figure 3.8 shows the compression ratio with respect to the entry size. An optimal compression ratio of 0.35 (or 1:2.86) is obtained with dictionary entries of size $e = 5$ and $f = 1$, and generally, matrices with 5-10 elements are preferred to larger or very small ones. When applying the dictionary compression with subsequent entropic coding, a compression ratio of 1:4.66 is achieved — an inferior performance compared to previous approaches. An explanation for the effectiveness of the 5×1 matrices is that they compress the low-variance *count1* and *rzero* regions, with amplitudes either being -1, 0 and 1 or all-zero. This diagnosis is supported by the quadruple- and run-length encoding of MP3 in these regions that serves a very similar purpose. Surprisingly, there is little to no advantage gained from similarities between subsequent frames as dictionary entries with fewer frames generally achieve higher compression. Simultaneously, the compression of low-variance regions achieved with this dictionary is unwanted, as arithmetic coding al-

ready performs well in this area. This is because the arithmetic coder encodes a flattened string of amplitudes along the frequency lines where it can perform run-length-encoding and adapt its probability model to the low variance. Instead, it would be more desirable to use a dictionary that operates across frames and finds structures that cannot be unveiled by the arithmetic coder.

A further conclusion is that this approach is subject to the problem of almost-repetitions and inexact matches that arises when applying dictionaries to data from a non-discrete source. As there is no immediate mitigation to this problem, the dictionary approach is discontinued at this point.

3.3.2 Persistent Homology

Persistent homology was introduced as a concept from mathematical topology, and can be used to compress data by removing intermediate points, which can be reconstructed approximately. It has been used for compressing the waveform of uncompressed audio as an example application of the TSC library that implements compression primitives based on persistent homology.

Algorithm 5 Persistent homology MP3 recompression

```
1: function Main(MP3 file M)
2:     R ← ∅
3:     for f ∈ M do
4:         h, s ← HuffmanDecode(f)          ▷ Obtain header h and amplitudes s from MP3 frame
5:         i, ŝ ← PHSimplification(s)       ▷ Persistent homology compression, i is index
6:         R̂ ← R ∪ ArithmeticCoder(s|i)     ▷ | denotes concatenation
7:     end for
8:     return R̂                             ▷ Recompressed file
9: end function
```

Within an MP3 frame, the amplitudes across the frequency bands resemble a signal, making them amenable to 1-d persistent homology simplification using the PHSimplification subroutine. For this, only the critical points (i.e. amplitudes) are preserved. In addition to the critical amplitudes, an index is kept that indicates which amplitudes have been discarded. Figure 3.9 shows the amplitudes of an uncompressed MP3 frame, its compressed representation, containing only critical amplitudes and the reconstructed version. The recompression procedure is shown in Algorithm 5. It recompresses the MP3 file frame-by-frame and therefore preserves independence between frames, meaning that an audio file can be (de-)compressed continuously (such as in a stream, or without loading the complete file into memory), or that the audio content can be split by splitting the byte sequence of the file.

The approach reduces the number of amplitudes depending on the range of frequency lines, or region. Minimal compression is achieved for approximately the first 100 frequency lines, but improves if higher regions are compressed as well (see Figure 3.10). Similarly as with the dictionary, this is due to the little variance found in the *count1* and *rzero* regions that contain few critical points. A complete frame can thus be compressed to approximately a quarter of the original size. Again, as with the dictionary, the persistent homology approach compresses at the expense of the arithmetic coder. Still, the arithmetic coder can reduce the size by about 1:2.86, jointly resulting in a compression of 1:13.2, which is almost 10% better than MP3, but incurs in an additional loss of information. Due to the additional loss of information and the overlap of arithmetic coding and persistent homology simplification, this approach is not continued as a MP3 recompressor, but with waveform audio in the next chapter.

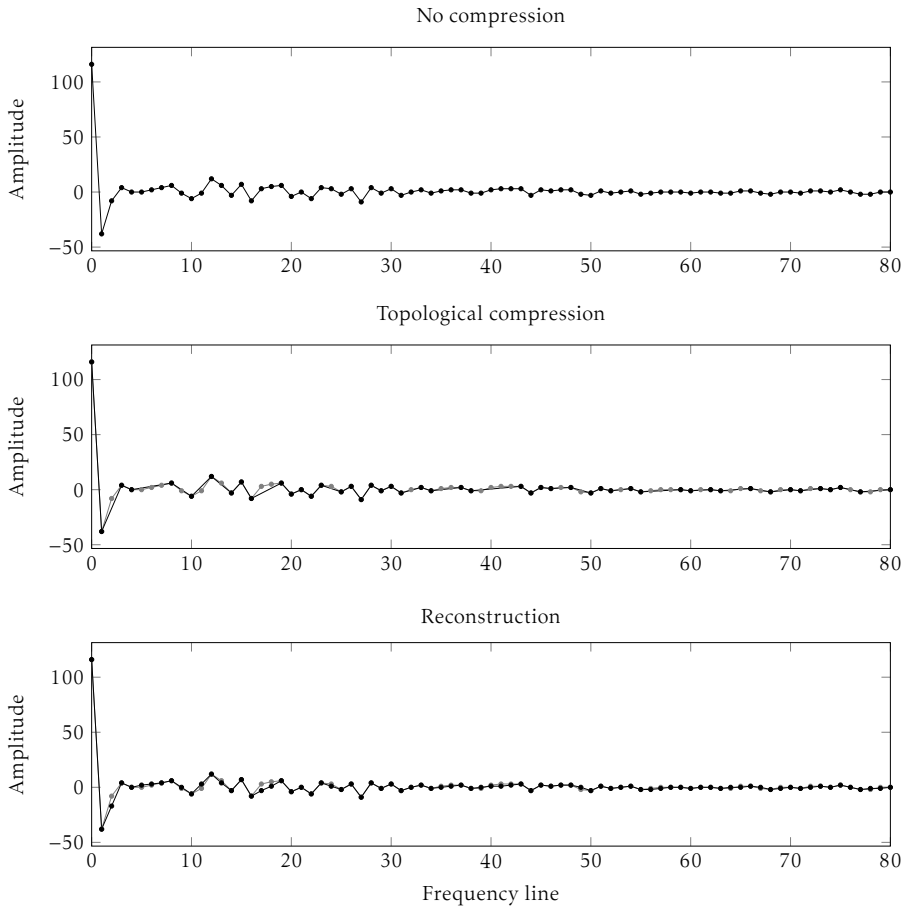

Figure 3.9: Comparison of the amplitudes of the lower 80 frequency lines of the original MP3 frame (top), the same amplitudes, compressed with the persistent homology approach (middle) and the reconstructed amplitudes (bottom). The original amplitudes from the MP3 are superimposed in grey. Note that the compressed frame in the middle panel leaves out all non-critical amplitudes, which can be approximated by the reconstruction. Also note that the remaining of 576 frequency lines have been omitted.

3.4 A Neural Network based MP3 Recompressor

After little success of finding structure within and among MP3 frames beyond order-0 entropic coding for lossless compression, it appears prudent to apply a universal coder as it has few assumptions about a file's content. For this, the PAQ compressors are promising candidates as they are most adaptive due to their neural network architecture, and while experimental, they are not impractically slow. A more formal description of the method is given in Algorithm 6.

The compression procedure is to revert the last stage of MP3 encoding, i.e. the MP3 Huffman coding and to then recompress the obtained data using a universal compressor. For decompression, the method is reversed, i.e. the compressed file is decompressed by the LPAQ1 universal compressor[3] and the samples reencoded again by any MP3 encoder. Note that reencoding has not been implemented.

Using this method, MP3 files can be recompressed by about 5%. However, the *PackMP3* recompressor compresses MP3 files by 16% on average using more straightforward, and computationally less expensive methods [102].

[3] `http://mattmahoney.net/dc/#lpaq`

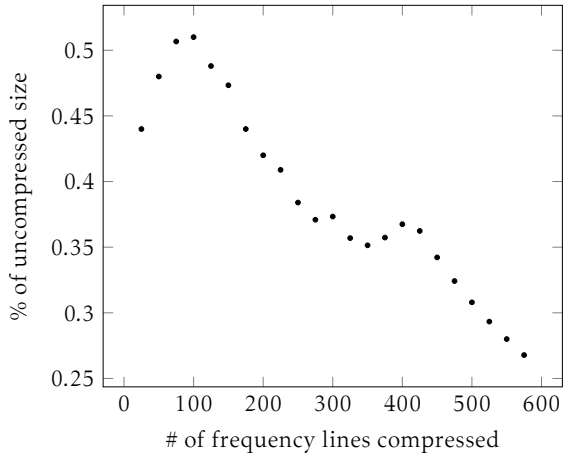

Figure 3.10: Ratios improve when more of the frame is compressed with the persistent homology approach. The number of frequency lines compressed are always counted from the lowest frequency line.

Algorithm 6 Neural network based MP3 recompression

```
1: function Main(MP3 file M)
2:     S ← ∅
3:     for f ∈ M do                                   ▷ Iterate over frames
4:         h, s ← HuffmanDecode(f)                    ▷ Obtain header h and amplitudes s from MP3 frame
5:         S ← S ∪ s
6:     end for
7:     C ← UniversalCompressor(S)
8:     return C                                       ▷ Compressed file
9: end function
```

3.5 Conclusion

The failure of using correlations and repetitions with the above mentions deserves a brief discussion. Two kinds of correlations were addressed thus far: those from immediate similarities in sound in subsequent frames and from similarities from repetitions that are not immediate. However, both cases rely upon the assumption that the MDCT time-frequency transformation preserves the structure in music, but this assumption is not necessarily true, as will be explained now. The time interval of approximately 26 ms encoded in an MP3 frame may both be too long, obscuring any patterns that are not a multiple of the interval that are further obfuscated by psychoacoustic lossy operations applied differently in every frame. Also, transforms from the psychoacoustic model may alter the time-frequency data, such as masking, even if it were structurally similar. From a data compression view, it is likely that the decorrelation procedure removes some correlations that are insignificant for a *single* frame, but not *beyond* the frame. For MP3, this is intended behaviour as it treats frames independently from each other. However, for recompression approaches targeting larger structure, these conditions are unfavourable. As PackMP3 is an effective MP3 recompressor, compressing audio data *ab initio* appears more promising and will be treated in the next chapter.

4 | Ab Initio Compression

> Truth is much too complicated to allow anything but approximations.
>
> John von Neumann

This chapter investigates the lossy and lossless compression of uncompressed audio data in waveform. This allows more degrees of freedom than the prior chapter, and will be used as a playground for many of the data compression methods introduced in Chapter 2.

4.1 Motivating Example

Adding dimensions to unravel structure inherent to data is a common theme in data compression, as for example with the discrete cosine transform, or more generally, time-frequency transforms. In the previous chapter, MP3's extensive reliance to this theme was showcased, yet it is not clear how much it contributes to compression in proportion with the lossy psychoacoustic model.

To examine the effect of the time-frequency transform for audio compression, we suggest a simplified compression method that only relies on three steps: transforming the audio signal into time-frequency space, minimal quantisation of the resulting time-frequency data and subsequent lossless coding using an universal compressor. Compared with MP3 or similar standards, this approach has fewer assumptions about the input signal: it only assumes that the input signal can be split and decomposed into frequency spectra. A high-level description of the method is given in Algorithm 7. Note that γ is a quantisation parameter that spreads or squeezes amplitudes to match the data type it is encoded with.

Algorithm 7 Simplified audio compression algorithm

```
1: function Main(WaveFile W)                          ▷ Waveform audio file
2:     S ← ShortTimeFourierTransform(W)               ▷ S is a N × T matrix
3:     S_abs ← |S|                                    ▷ Obtain magnitude of every amplitude
4:     S_q ← round(γ √S_abs)                          ▷ Quantisation
5:     C ← UniversalCompressor(S_q)
6:     return C                                       ▷ Compressed file
7: end function
```

An assumption of the DFT is that the frequency of the input signal is constant, i.e. that the frequency spectrum does not change. Clearly, music, or any audio signal that is not a constant sound invalidates this assumption and as a consequence, the DFT is applied individually on smaller time intervals of the signal. This technique is called *Short-time Fourier transform* (STFT)

and is very similar to the MDCT used in MP3. However, the DFT is not applied individually on groups of samples, but using overlapping windows. The window size determines a trade-off between how well closely spaced audio pulses can be resolved in either time with a small window size (temporal resolution) and in frequency with a large window size (frequency resolution)[1]. The window's shape is often chosen to weight samples in the centre of the window stronger than samples on its ends. Intuitively, this decision is taken to counteract noise in the frequency spectrum that emerges as the DFT assumes a constant signal (i.e. that sine waves continue in both directions at the same frequency), but receives a discontinued signal.

The result of the STFT is a $N \times T$ matrix S, where N is the number of frequency lines, or bins (here 513) and T the number of time intervals, or frames over the temporal axis. The matrix can be plotted as a spectrogram, as in Figure 4.1. The entries of the matrix are complex valued, describing the magnitude and angle of the frequency line $n \in N$ at time $t \in T$. For the magnitude, the absolute value $|S|$ is taken and the angle is discarded. For quantisation, the square root of every entry is computed to nonlinearly "squash" value range of amplitudes as entries become more sparse with higher frequency and therefore, have higher resolution as small values. Amplitude "squashing" evens out resolution along the frequency lines. MP3 uses a similar, yet more complex quantisation process, replacing the square root by the exponent 3/4 [20]. Sometimes, the logarithm is used for this purpose, but this produces negative values and the value range is not centred around zero, requiring signed data types and wasting space for representing very frequent entries with values far from zero. Multiplication by γ (here $\gamma = 10$) linearly stretches the value spectrum to give it more resolution when they are rounded to the next integer.

Another reason for quantisation is that Huffman or arithmetic coding, operating on symbols, can update the discrete probability distribution of input values (such as amplitudes) only if the byte representation of two "close" input values are identical. For example, the 32-bit floating point encodings of two amplitudes with values 0.4669 and 0.4670 as bytes are `840d` and `a01a`. Although the numerical values are very similar, an arithmetic coder will consider them as unrelated and maintain different frequencies of the symbols. After quantisation, the symbols may both be encoded as 467, yielding the same byte pattern. While some information is lost by quantisation, the arithmetic coder now encounters bytes that are more representative of the values they encode and can maintain a more accurate probability model. As a fix to the problem, the probability model of the arithmetic coder could be modified to support a continuous probability distribution understanding the floating-point encoding, allowing the coder to recognise "close" values and allowing to interpolate probabilities. The problem of compressing floating point numbers has been addressed by Lindstrom [103]. Still, as quantisation has proven to be a robust approach and integrates well with any byte-level compressor, this is the approach that is usually taken in this chapter.

Finally, the matrix is exported as 16-bit integers and compressed using the LPAQ1 compressor[2], a universal, context-sensitive and lossless encoder, comprising arithmetic entropy coding.

Decompression is a reversed application of the steps in Algorithm 7. The algorithm is not lossless, as both the STFT and the quantisation discard information. However, by choosing a larger c, the quantisation error can be minimised while increasing the file size. The audio quality is excellent and no compression artefacts are readily identified.

[1] This trade-off conceptually resembles the uncertainty principle in quantum mechanics, where the resolution of either momentum or position of a particle limit each other.

[2] `http://mattmahoney.net/dc/#lpaq`

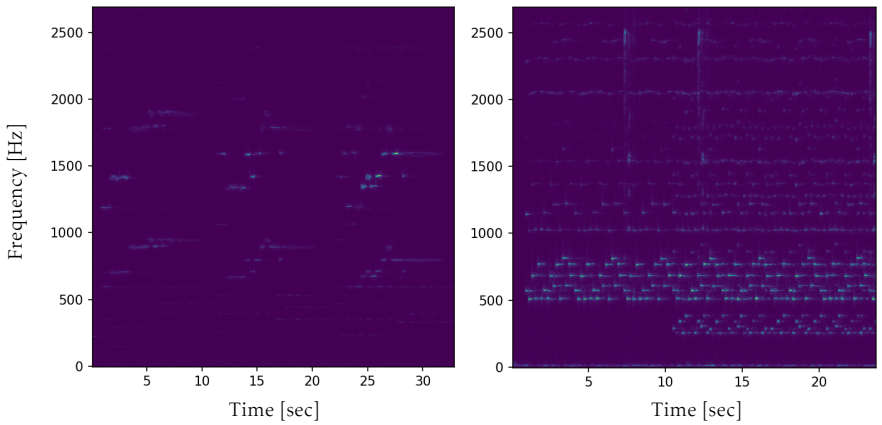

Figure 4.1: Spectrograms of ca 25 seconds of classical music (left panel, *Adagio* from Allessandro Marcello's oboe concerto) and electronic music (Commander Tom, Are Am Eye?). The classical sample has little repeating structure and is sparse. The electronic sample is denser and clearly bears structure in several frequency bands. Both spectrograms show only the lowest quarter of the frequency spectrum.

Results of the compression algorithm are listed in Table 4.1.

Sample	Classical	Electronic
Uncompressed (Wave)	5814 kb (1:1)	4198 kb (1:1)
Lossy compression (MP3)	528 kb (1:11.01)	382 (1:10.99)
Lossless compression (FLAC)	1940 kb (1:3.00)	1972 (1:2.13)
Algorithm 7 (motivational example)	561 kb (1:10.36)	703 kb (1:5.97)
LPAQ1 on Wave	4284 kb (1:1.35)	3579 kb (1:1.17)

Table 4.1: Comparison of various audio compression methods. Algorithm 7 compresses the "Classical" sample within 10% of MP3's size.

While MP3 still achieves the highest compression ratios, Algorithm 7 produces already significant compression, despite few optimisations and relatively few assumptions. In difference to most audio compression standards, Algorithm 7 compresses the complete matrix S as a whole, making it necessary to decompress the complete file before the compressed audio can be played back. It would however be simple to extend the method to allow for chunked compression, resolving this limitation.

The method presented in this example showcases the gain that can be achieved by transforming data into another representation that is more amenable to compression. It particularly exceeds in comparison to compressing the raw audio data with the same LPAQ compressor (last row of 4.1). Yet, the method fails to exploit repetition that is present in the audio files, particularly in the "Electronic" sample, as can visually be examined in the spectrogram in Figure 4.1. This is because the repetition resides at a higher abstraction level than the structure that is decorrelated by the STFT, a problem that was encountered before as the micro-macro problem. Applying transformations that attempt to decorrelate structure at these higher levels is the central goal in the next sections.

4.2 The Crux of Universal Compressors

It is a reoccurring theme that universal data compressors achieve relatively little compression with audio data, even if it is decorrelated. For example, ZIP, Lempel-Ziv and PAQ are competitive compressors in compression benchmarks [19], yet they fail to recognise the macrostructure of audio data, as seen in the motivational example, where structure-rich electronic music is compressed to a worse ratio as classical music with less repetitive structure.

These universal compressors are typically designed with kinds of data in mind that have short contexts, as they appear in natural language (where a word is guessed with initial characters, or subsequent words with the previous one). Their effect for compressing audio data is mainly due to entropic coding and results with more common compressors are within a 20% range of LPAQ1. Therefore, investigations on decorrelating data on a larger scale are in order.

4.3 Multi-level Dictionary

Following the intuition that structure in electronic music appears at several scales (see Figure 4.2), a naive approach would be to use dictionaries on different scales on the time-frequency representation to capture repetitions at each level. A dictionary is an associative list, each entry is a pair of a key (or index) and value (containing data from the source). Using the key, the entry can quickly be recovered. Recall that frequently occurring elements (such as words within a text) can be stored once as value in the dictionary, the key replaces the original occurrence by referring to the dictionary. Clearly, the key must be smaller than the value it refers to to achieve compression.

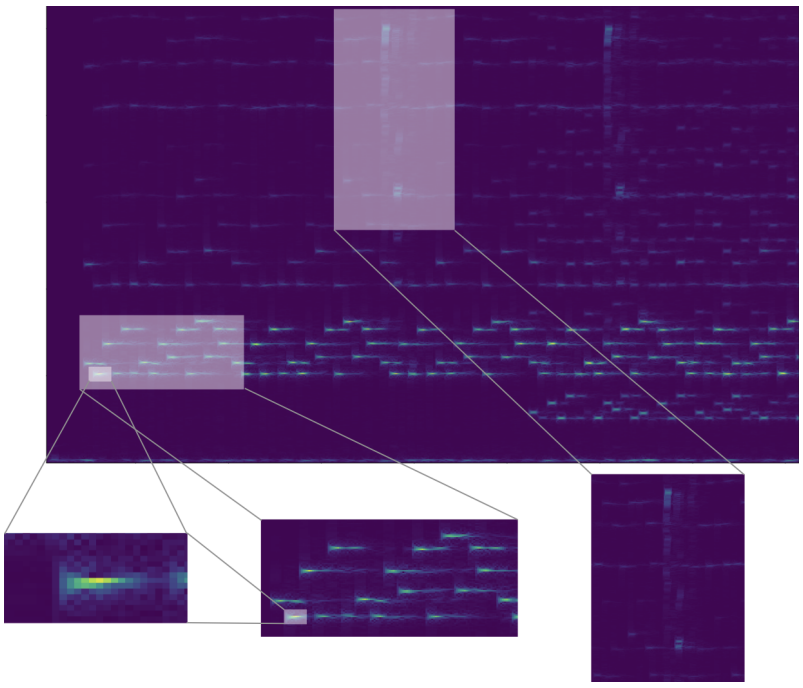

Figure 4.2: Structures appear at different scales in this spectrogram of approximately 16 seconds of electronic music. The structure on the left is the smallest and denoted as ζ_0. It is part part of a larger structure, shown in the middle and denoted as ζ_1.

In this approach, dictionaries are organised hierarchically (see Figure 4.3), similarly to a quadtree [104]. The lowest level dictionary d_0 has both keys and values as the $N \times T$ ampli-

tudes themselves. Each higher level dictionary d_l with $0 < l < L$ contains entries $e_{l,0}, e_{l,1}, \ldots$ (there is no fixed number) that each contains a block of four keys from the dictionary one level below d_{l-1}. A dictionary d_l can therefore be seen as a "translator" from level l (its key) to level $l-1$ (its value). The key is a single natural number, and the value is a 2×2 matrix of keys from the dictionary one level below (more generally, $w \times w$ matrices can be considered). At $l = 0$, the block matrices are sampled from the spectrogram, but at $l > 0$, the source consists of keys, each representing four values. This "input data" is represented using L collections c_l. Each collection consists only of entries of the dictionary of its level and can be imagined as a more "pixelated" spectrogram. Computationally, collections are the working memory that consists of references that describe the data recursively from the current level using dictionaries. As the dictionaries can resolve references from a higher to a lower level, it is sufficient to retain only the highest level collection[3].

The rationale is that a low-level repeating structure such as ζ_0 may be captured by a low-level dictionary, and that a higher-level dictionary must not be aware of the composition of ζ_0, but recognise that it is part of a larger-scale structure such as ζ_1. This can be done more efficiently when the lower-level values are represented by a single entry, preventing that larger-scale structures need to be matched by comparing their values on the lowest level. Matching is approximate because repetitions, especially at small scales, are not exact (see the discussion about dictionaries in Figure 2.10). Two entries $e_{0,a}, e_{0,b}$ will be recognised as *similar* if

$$\sum_{i \in w \times w} (e_{0,a_i} - e_{0,b_i})^2 < \theta$$

where θ is a threshold and i is an index iterating the 2×2 matrix. This formula requires that similar dictionary entries (i.e. the 2×2 matrices) are assigned keys that are similar natural numbers. This condition is trivially fulfilled if the matrices contain amplitudes, as at the lowest level, but if the keys are synthetically generated, this must be accounted for. More intuitively, if an image were to be processed, the requirement would assign blocks of four pixels a new pixel with the mean colour of the pixel block. The requirement can also be seen as a measure to maintain key aspects of the structure while moving from a lower to a higher level of abstraction.

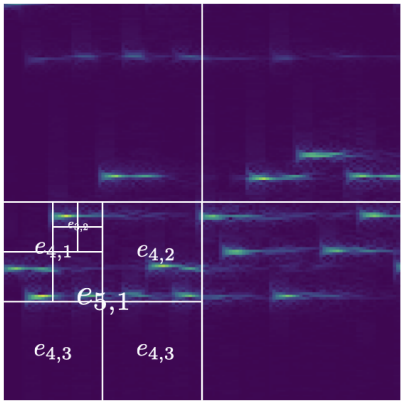

Figure 4.3: Incomplete collections from levels 5, 4 and 3 are shown. The entry on level $l = 5$ consists of four keys referring to level $l = 4$, although two entries are the same as their keys would recursively refer to very similar amplitudes.

[3] If the highest level L would cover the complete spectrogram (when $2^L \geq \max(N, T)$), its dictionary d_L would only consist of one entry, making even the highest-level collection obsolete. However, this approach would only be useful if repetitions would occur in the largest scale, such as if a piece of music would be placed twice in an audio file.

A procedure to create a multi-level dictionary is given in Algorithm 8, omitting any optimisations and proper handling of borders. Note that γ is a quantisation parameter. Also, note that dictionaries are accessed using a key k with brackets as $d(k)$ and matrices are accessed using square brackets using indexes starting at 0 with ranges denoted as colons. For example, $M[0:5, 0:2]$ accesses the upper left 5×2 submatrix of M.

Algorithm 8 Multi-level dictionary

function MAIN(WaveFile W) ▷ Waveform audio file
  $S \leftarrow$ SHORT-TIMEFOURIERTRANSFORM(W) ▷ S is a $N \times T$ matrix
  $S_{abs} \leftarrow |S|$ ▷ Obtain magnitude of every amplitude
  $S_q \leftarrow \text{ROUND}(\gamma \sqrt{S_{abs}})$ ▷ Quantisation
  $C \leftarrow \varnothing$ ▷ Set of L collections
  $d_0.keys \leftarrow S_{abs}$, $d_0.values \leftarrow S_q$ ▷ First dictionary is spectrogram itself
  $w = 2$
  $D \leftarrow \{d_0\}$
  for $l \in 1 \dots L$ **do**
    $m, n \leftarrow d_{l-1}.shape$
    $c_l \leftarrow \text{ZEROS}(\lfloor m/w \rfloor, \lfloor n/w \rfloor)$ ▷ Houses entries that are double (when $w = 2$) the width and height of lower level entries
    $d_l \leftarrow \varnothing$
    for $i \in \{0, w, 2w, \dots m\}$ **do** ▷ Iterate in strides of length w in first dimension
      for $j \in \{0, w, 2w, \dots n\}$ **do**
        $r \leftarrow c_{l-1}[i : i + w, j : j + w]$ ▷ $w \times w$ matrix from $l - 1$ that is being abstracted
        $k_{best}, v_{best} \leftarrow \text{FINDMOSTSIMILAR}(r, d_l, \theta)$ ▷ Seeks similar entry in d_l
        if $k_{best} = \varnothing$ **then** ▷ Create new entry
          $k_{new} \leftarrow \text{CREATEKEY}(r)$
          $d_l(k_{new}) \leftarrow r$
          $c_l[\lfloor i/w \rfloor, \lfloor j/w \rfloor] \leftarrow k_{new}$ ▷ Indexes are translated for level l
        else ▷ Refer to existing entry
          $c_l[\lfloor i/w \rfloor, \lfloor j/w \rfloor] \leftarrow k_{best}$
        end if
      end for
    end for
    $C \leftarrow C \cup \{c_l\}$
    $D \leftarrow D \cup \{d_l\}$
  end for
  return D, c_L
end function

Several limitations apply to this approach. Firstly, it cannot mitigate most of the weaknesses of dictionaries. Although the approximate matching can detect similar entries, it does not account for the distribution of entries. In particular, the parameter θ is fixed and may be too coarse or too sensitive. Vector quantisation addresses this problem better. Secondly, the $w \times w$ matrices are too rigid for structures, as can already be seen in Figure 4.3. This is most notably expressed by dictionary entries that cut structures into parts or capture them at different positions. Thus, a structure such as ζ_0 may be contained in three different entries, although the scale would be correct. More fatally, the structure may be contained differently at another repetition and this mismatch propagates through the scales, hindering capture of any structure within the dictionary.

The approach is slightly similar to fractal image compression, but instead of exploiting self-similarity where references point to similar structures within the same object at a different scale, the references here are to similar structures at the same scale or any structures at a smaller scale.

4.4 Topological Compression

Code reference. The lossless topological audio compressor is named `1toco.py` and the lossy compressor `toco.py`. The tested audio files are `monoadagio.wav` and `monotrance.wav`.

Simplification using persistent homology was already used on MP3 frames in the second chapter, now it is applied on spectrograms from an uncompressed waveform spectrogram similar to the one obtained in Algorithm 7. A significant difference to MP3 is that the amplitudes in the spectrogram represent the audio source more directly, while the content of the MP3 frames has been transformed both by the psychoacoustic model and transforms to improve entropic coding.

4.4.1 Approach

Although the persistent homology simplification can be applied to 2-d data, the first approach is to compress the spectrogram frame-by-frame (dashed arrow in Figure 4.4) using 1-d simplification. This has the risk of taking over the job of the arithmetic coder, as the simplification compresses most intensely in low-variance regions in the upper frequency spectrum, as seen in MP3. The advantage of this procedure is that frames are compressed individually and could be processed either during playing, or the audio file can be split by splitting the byte sequence. First attempts however show that the audio quality degrades significantly using this method. While the compression ratio is similar to MP3, the audio fidelity is clearly insufficient.

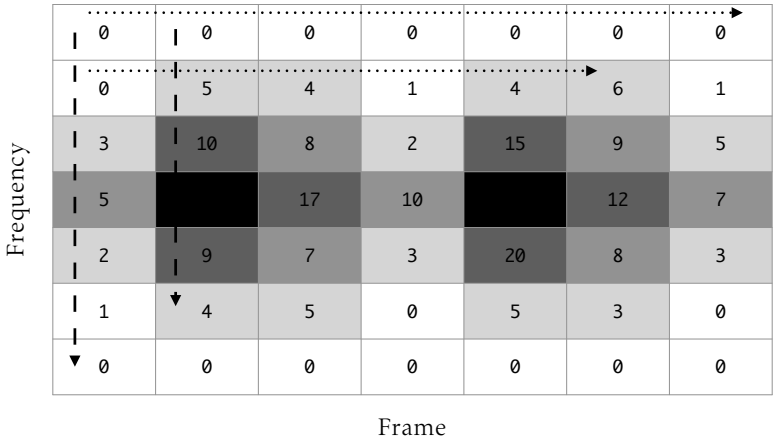

Figure 4.4: Conceptual overview of how the spectrogram is organised. Each column describes a point in time (frame) and is created from typically several thousand waveform samples. Each row corresponds to a narrow frequency band, often called frequency line. MP3 proceeds first within a frame (dashed arrow), the topological compressors in this thesis typically along the frequency lines (dotted arrow).

In the second approach, the frame-by-frame processing is abandoned and the spectrogram is processed row-by-row (dotted arrow in Figure 4.4). This resolves a problem of the first approach: entropic and run-length coding are less weakened as there is generally more variance along the frequency lines (see Figure 4.5). Furthermore, processing row-by-row gives access to the temporal axis, theoretically allowing for the exploitation of patterns within the signal, such as repetitions. In summary, the approach is to iterate row-by-row, feeding each row to persistent homology simplification (subsequently referred to as PHSimplification), serialise the result to a compact byte representation and apply entropic coding. The procedure is described more formally in Algorithm 9, and the corresponding decompression in Algorithm 10. Again, note that γ is a

quantisation parameter and the star denotes computations that are only made in the lossless compressor.

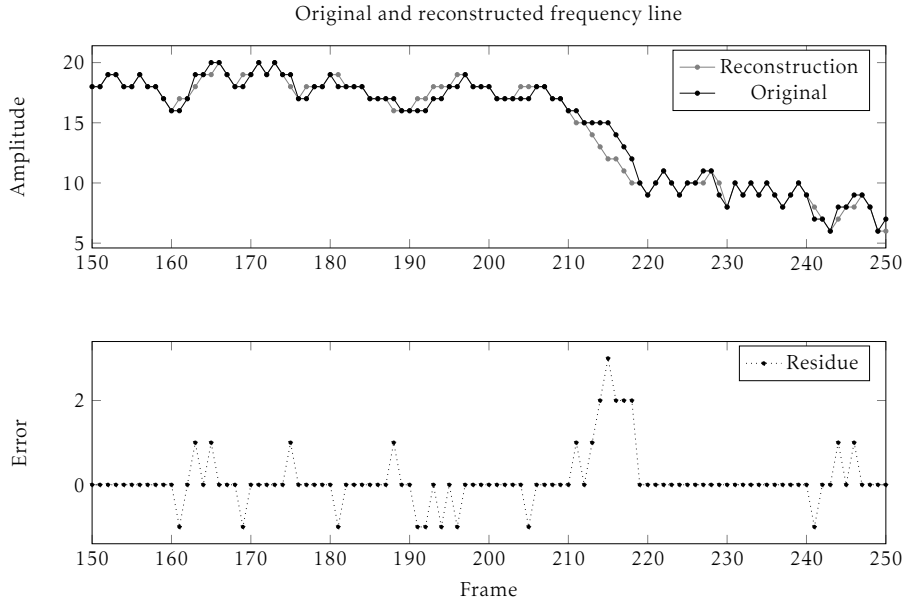

Figure 4.5: On the top panel, the original and the reconstructed amplitudes along the 20th frequency line are shown. Their difference is denoted as residue, whose distribution is strongly centred around 0. For easier inspection, a region with relatively little variance has been chosen. The increased residue between frames 210 and 220 derive from the removal of the monotonically falling amplitudes by the persistent homology simplification. Because several amplitudes are removed at once, fewer "clues" are available for reconstruction and allow greater divergence from the original. Conversely, where sole amplitudes are removed (e.g. around frame 230), the residue is much smaller.

There are two versions of the algorithm: one is a lossy compressor that preserves only the simplified amplitudes, but decreases audio quality. The other is an (almost-)lossless compressor that maintains the difference of the original and the simplified amplitudes, known as residue, and compresses it separately. This strategy is also employed by SHORTEN or FLAC [73, 72]. The losses incurred in the almost-lossless variant are due to the conversion of the waveform audio into the time-frequency spectrum and its inverse, as well as small quantisation errors. They could be eliminated by considering the residue as difference to the waveform rather as the difference to the spectrogram. The compressors only support mono audio files. An extension for multiple channels is straightforward and could be optimised by synchronising the PHSimplification between the channels such that the index would only be needed once.

4.4.2 Results

The results of the compressors are shown in Table 4.2. In general, topological audio compression performs well in the lossless category, being very similar with FLAC. However, the compressed files of the lossy coder are 20% to 50% larger than MP3 and have worse audio quality.

4.4.3 Conclusion

Although the lossy compressor is inferior to MP3 (and Algorithm 7), the (almost-) lossless compressor is comparable with FLAC. An advantage of FLAC is that its frames are independent, making FLAC files more manageable. However, the topological compression algorithms 9 and 10

Sample	Classical	Electronic
Uncompressed (Wave)	2907 kb (1:1)	2100 kb (1:1)
Lossy compression (MP3)	264 kb (1:11.01)	191 kb (1:10.99)
Lossless compression (FLAC)	989 kb (1:2.94)	1018 kb (1:2.06)
Topological compression (lossy)	316 kb (1:9.20)	372 kb (1:5.65)
Topological compression (lossless)	998 kb (1:2.91)	977 kb (1:2.15)

Table 4.2: Comparison of topological audio compression with FLAC and MP3. The lossy compressor is inferior in both compression ratio and audio quality compared to MP3. However, the lossless compressor is on par with FLAC.

Algorithm 9 Topological waveform compression

function MAIN(WaveFile W) ▷ Waveform audio file
  $I \leftarrow \varnothing$ ▷ Indexes
  $C \leftarrow \varnothing$ ▷ Compressed frequency lines
  $*R \leftarrow \varnothing$ ▷ Residue (lossless only)
  $S \leftarrow$ SHORT-TIMEFOURIERTRANSFORM(W) ▷ S is a $N \times T$ matrix
  $S_{abs} \leftarrow |S|$ ▷ Obtain magnitude of every amplitude
  $S_q \leftarrow \text{ROUND}(\gamma \sqrt{S_{abs}})$ ▷ Quantisation
  for $f \in S_q$ **do** ▷ Iterate over N frequency lines
    $i, \hat{f} \leftarrow$ PHSIMPLIFICATION(f) ▷ Persistent homology compression, i is index
    $I \leftarrow I \cup i$
    $C \leftarrow C \cup \hat{f}$
    $*R \leftarrow R \cup (f - \hat{f})$ ▷ Compute residue (lossless only)
  end for
  $\hat{I} \leftarrow$ ARITHMETICCODER(I)
  $\hat{C} \leftarrow$ ARITHMETICCODER(C)
  $*\hat{R} \leftarrow$ ARITHMETICCODER(R)
  return $(\hat{I}|\hat{C}|\hat{R})$ ▷ | denotes concatenation, $\hat{R}$ only in lossless
end function

could be modified to entropically code on a frame-by-frame manner, enabling similar independence when handling files and decoding.

The compressors are set to only discard non-critical points, for which the persistence diagram must not be computed. Thus, the simplified procedure for identifying critical points may be used, running in linear time. Practically, it is difficult to compare the performance of the compressors as MP3 and FLAC have highly optimised implementations [20], while the topological compressors are implemented in the slower Python language and feature almost no optimisations. However, from a theoretical view, there are no obstacles to faster implementations for the topological compressors.

Also, several continuations of the approach are imaginable. Firstly, the PHSIMPLIFICATION subroutine only removes non-critical points in Algorithm 9. By reducing the number of points preserved, linear size reductions could be expected. Tests would be required to observe the effect on audio quality, but gains in compression ratio may again be compensated by the residue. Furthermore, the 1-d persistent homology simplification could be applied directly on the audio waveform while maintaining the residue[4]. This would be similar to replacing the curve-fitting approach in FLAC.

[4] An approach discarding the residue is given on the website of the TSC library, but the audio quality is reduced significantly.

Algorithm 10 Topological waveform decompression

function MAIN(Compressed file $(\hat{I}|\hat{C}|\hat{R})$)
  $I \leftarrow$ ARITHMETICDECODER$(\hat{I})$
  $C \leftarrow$ ARITHMETICDECODER$(\hat{C})$
  $*R \leftarrow$ ARITHMETICDECODER$(\hat{R})$ ▷ (lossless only)
  $\tilde{S}_q \leftarrow \varnothing$
  for $i \in I, \tilde{f} \in C, r \in R$ **do** ▷ I and C have same size (R only in lossless)
    $\tilde{f} \leftarrow$ PHRECONSTRUCTION$(i, \hat{f})$ ▷ $\tilde{l}$ is reconstruction of l
    $*\tilde{f} \leftarrow \tilde{f} + r$ ▷ Residue correction (lossless only)
    $\tilde{S}_q \leftarrow \tilde{S}_q \cup \tilde{f}$
  end for
  $\tilde{S}_{abs} \leftarrow (\tilde{S}_q/\gamma)^2$ ▷ Dequantise
  $\tilde{W} \leftarrow$ INVERSESHORT-TIMEFOURIERTRANSFORM$(\tilde{S}_{abs})$ ▷ Reconstruct waveform
  return $\tilde{W}$
end function

4.5 Tensor Factorisation

Code reference. Code for this section is contained in the `notebooks/tensor.ipynb` Jupyter Notebook.

As noted in the second chapter, tensor factorisation has already been applied for the task of audio analysis [39] and audio compression [40, 41]. The focus of the audio compression work using tensor factorisation is on decorrelating between audio channels rather than *within* an audio sequence. Recall that two problems emerging from this is that the period Δt has to be found on which repetitions may be exploited and even if a good value for Δt is found for a few repetitions, it is not constant within a piece of music. However, as a proxy for a fixed period Δt, the autocorrelation of the audio file as spectrogram can be used to give clues about repetitions (see Figure 4.6).

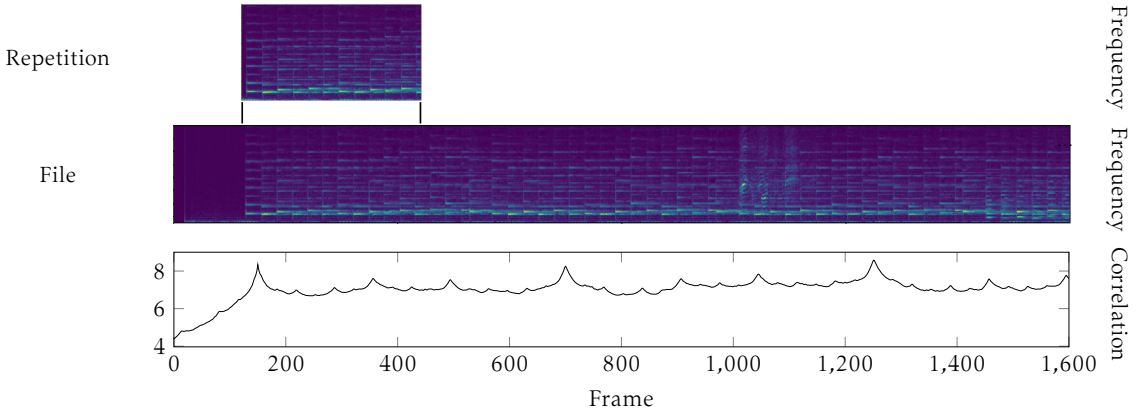

Figure 4.6: The autocorrelation of a repeating block within the audio file is an indicator for when the repetition occurs. Here, repetition occurs approximately every 550 frames, and only the first approximately 250 frames of the repetition have been used for the block. However, successful compression requires that the repetitions are known exactly.

4.5.1 3-d Tensor

The first approach is to use tensor factorisation targeting only the decorrelation of channels, meaning that the tensor to be factorised has three dimensions: channel, frequency and time. It is

given as a $C \times N \times T$ spectrogram with C channels, N frequency lines and T time slices (or frames) is obtained by the STFT. The variable names are changed here to be consistent with the names used in the introduction of tensor factorisation in Chapter 2. The spectrogram $\mathcal{X}$ is factorised into the core tensor $\mathcal{S}$ and the three factor matrices U_1, U_2 and U_3. The truncated core tensor is written as $\tilde{\mathcal{S}}$ and the corresponding truncated spectrogram is $\tilde{\mathcal{X}}$. A visualisation of the tensors and matrices is given in Figure 4.7

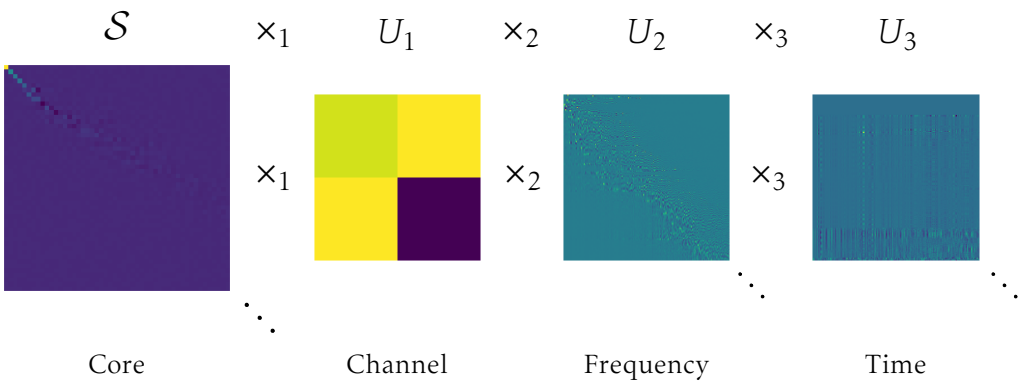

Figure 4.7: Example tensors and matrices of the factorisation. Only the first 50 entries of the left channel plane of the core tensor $\mathcal{S}$ are shown, and the first 200 entries of the factor matrices U_2 and U_3. U_1 is a 2×2 matrix as there are two channels.

This approach is subdivided into two methods: in one method, the spectrogram is quantised into integer values, as was done in previous approaches. In another method, the spectrogram is factorised in its original floating-point representation, as tensor factorisation is not reliant on integer values. For tensor factorisation, the TTHRESH [38] library is used, motivated by a fast C++ implementation and integrated arithmetic coding which is optimised for core tensor and factor matrices. The latter reason is particularly relevant for reporting results as the efficient entropic coding of core tensors and factor matrices is not trivial [38], and if done wrong can degrade good compression results obtained by tensor factorisation.

The measure of loss allowed when truncating the core tensor is defined as the relative error

$$\frac{\|\tilde{\mathcal{X}} - \mathcal{X}\|}{\|\mathcal{X}\|}$$

where $\mathcal{X}$ is the input tensor and the Euclidean norm on the flattened tensor (Frobenius norm) is used [38]. In context with the results, the allowed error is denoted with the `-e` flag. The results of this approach are reported in Table 4.3.

In comparison with topological audio compression, MP3 and FLAC encoding, compressing using tensor factorisation is computationally more intensive and encoding may take several minutes[5] compared to a minute for topological audio compression and seconds for FLAC and MP3. Decompression is significantly faster, typically only using few seconds. This asymmetry is characteristic for tensor factorisation. While finding an appropriate factorisation is computationally intensive [36], restoring the original tensor is a simple tensor-matrix multiplication.

[5] Using a 2015 Macbook Air with a 1.6 GHz Dual-Core Intel i5 processor and 8 GB of RAM.

Sample	Classical	Electronic
Uncompressed (Wave)	5814 kb (1:1)	4198 kb (1:1)
Lossy compression (MP3)	528 kb (1:11.01)	382 kb (1:10.99)
Lossy compression (MP3, highest compression)	312 kb (1:18.63)	249 kb (1:16.86)
Lossless compression (FLAC)	1940 kb (1:3.00)	1972 kb (1:2.13)
Tensor factorisation (quantised, `-e 0.02`	338 kb (1:17.20)	498 kb (1:8.43)
Tensor factorisation (unquantised, `-e 0.01`)	212 kb (1:27.42)	406 kb (1:10.34)
Tensor factorisation (split, `-e 0.1`)	-	199 kb (1:21.01)
Tensor factorisation (split, `-e 0.15`)	-	48 kb (1:84.46)
Tensor factorisation (split, `-e 0.2`)	-	15 kb (1:279.87)

Table 4.3: Results of audio compression with tensor factorisation. Depending on the tolerated error of tensor factorisation, better compression ratios than MP3 are obtained. As error accumulates quicker in the quantised variant due to scaling, the error limit is set higher.

4.5.2 4-d Tensor

In this approach, the time dimension from the 3-d tensor from above approach is split into time slices of length Δt to match the autocorrelation along the time axis induced by the repeating pattern (see Figure 4.8).

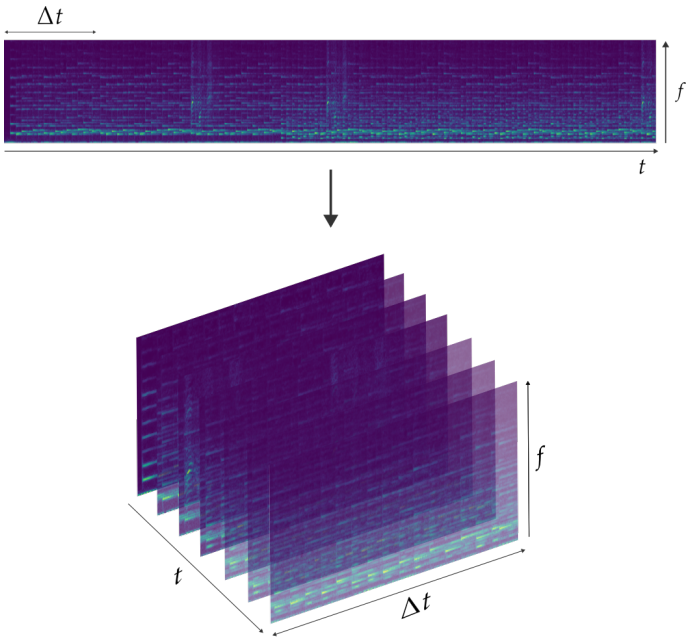

Figure 4.8: The 3-d tensor is split into time slices of length Δt, resulting in a 4-d tensor where the time slices are stacked. Note that the channel dimension was omitted to show the structure in a 3-d view.

The length of Δt was established separately by examining local maxima on the autocorrelation plot seen in Figure 4.6. Factorising the 4-d tensor with the same accepted error as above (i.e. 0.01 and 0.02) yields compressed audio with better audio quality, prompting to attempt compression runs allowing more error, and thus significantly higher compression ratios (see Figure 4.9). Remarkably, even with the relative error set to 0.2, the music is well-recognisable and the repeating pattern is only slightly degenerated by compression. On this setting, a compression ratio of 1:280 is obtained. When reducing the accepted error, the file size quickly increases with the quality. Still, a better audio quality example shows a compression ratio of 1:84 (see Figure 4.3 for all results). With an error of 0.1, the quality is well-comparable with normal MP3 while the

file size is almost half of that of MP3. However, this example showcases an extreme example of a sample where the repetition in the music sample is present without much noise, is extracted in a half-manual manner and over a fairly short time scale. In general, it cannot be expected that compression ratios of that order can be achieved without further modifications.

Furthermore, compression is significantly faster when the length of the axes is smaller (although the overall number of amplitudes remains), reducing the compression runtime to several seconds on the same hardware.

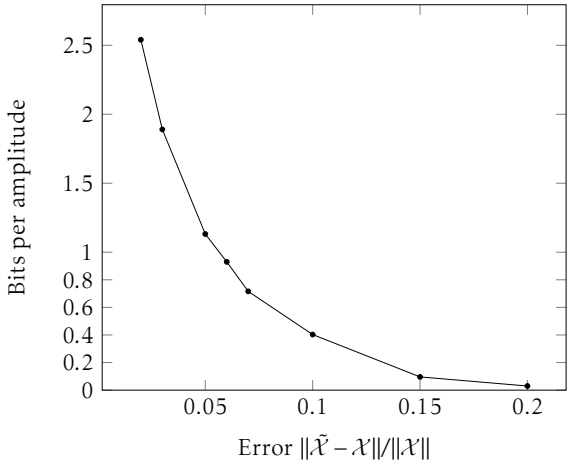

Figure 4.9: Allowing more error decreases the number of bits to describe an symbol (i.e. an amplitude). The rightmost point denotes 0.03 bits used per amplitude.

4.5.3 Conclusion

As tensor factorisation is used with truncation of the core tensor, it is a lossy method and therefore comparable with MP3 rather than with FLAC. The difference of audio quality between the quantised and unquantised variants varies on the audio sample taken, although both are comparable with a low-rate MP3 compression. Compression ratios from the 3-d tensor approach range from close to better as MP3, and with the "classical" audio sample, compression ratios are considerably higher, beating MP3 by more than the double compression with the unquantised spectrogram. In the 4-d tensor approach, compression ratios are significantly better than MP3 on its lowest settings, while audio quality remains acceptable. The extreme cases where compression ratios of 1:84 and 1:280 correspond to bit rates of 2.18 kbit/s and 0.68 kbit/s, outperforming the neural audio compression due to Défossez et al. [13], achieving the lowest bit rates on stereo music of 4.2 kbit/s. However, a specially prepared audio sample was used, and the approach is not likely to generalise to any kind of audio.

Although this approach compresses the audio file as a whole and does thus not provide independent frames along with its advantages, the extremely fast decompression and relatively small file sizes make it relatively practicable and competitive to lossy formats as MP3. Possible optimisations were to automate the detection of repetitions from the data's autocorrelation, a fast and integrated implementation and the examination of further correlations within the signal that could be separated into an additional tensor dimension and allow further compression.

4.6 Vector Quantisation

Code reference. The vector quantisation audio compressor is named `veco.py`. The tested audio files are `adagio.wav` and `trance.wav`, although the compressor can also compress mono audio. All files are in the `compressors` subdirectory.
Additionally, a Jupyter notebook is given in `vector_quantisation.ipynb`, illustrating the compression procedure with intermediate results.

The rationale of vector quantisation is mainly as to address the limitations of dictionaries. The main issue with dictionaries when data is approximate is that entries will either not be matched (see the discussion around Figure 2.10), or when an approximate matching technique is introduced (as in the multi-level dictionary in this chapter), the chosen dictionary entry may not be a good representative of the instances in the data it replaces. The latter issue is related to the issue of not representing the distribution of the underlying data, which is better solved by vector quantisation. Recall that in vector quantisation, instead of dictionary entries, centroids (or prototype vectors) are chosen that represent values from the input data that are close to them according to some metric. Thus, the centroids tessellate the space the values are in (see Figure 2.13), and the larger the number of centroids is, the less quantisation occurs. In one extreme case, if there are as many centroids as values, every centroids can represent exactly one value, causing no quantisation error, but also no compression.

4.6.1 Approach

Vector quantisation has been used for audio compression, as for example by Vorbis [69] and is used as a part of MPEG4[6]. These classic applications operate on a frame-by-frame basis and can therefore not choose the centroids that will represent future data ideally. In contrast, this approach applies vector quantisation on a complete audio sequence, enabling it to select centroids that represent the audio data after it has been seen. More concretely, the input data is given as a $C \times N \times T$ spectrogram with C channels, N frequency lines and T frames. A point is chosen to be one of the T frames, i.e. it is a vector with N elements. Instead of centroids, the term of prototype vectors will be used for the remaining section. In short, the algorithm selects $r = T/f_c$ prototype vectors from the spectrogram, stores them as a codebook P and replaces the T vectors by the closest prototype vector (see Figure 4.10 for a visualisation of the codebook). f_c is a compression factor. This assignment forms a set of labels L that is stored separately. L can be understood as a list of indexes that point to prototype vectors in the codebook.

The prototype vectors P and the labels L are entropy-coded and represent the compressed file. A more formal description of the encoding and decoding procedures is given in Algorithms 11 and 12. Finding the prototype vectors can be formulated as k-means problem and is solved using Lloyd's algorithm that was originally designed for use in data compression [105, 106]. The quantisation error of a vector t is given by $\|t - \textsc{closest}(t, P)\|$ where $\textsc{closest}(t, P) = \arg\min_{p \in P} \|p - t\|$ selects the closest prototype vector. The quantisation error is also part of the quantity that is minimised in the k-means problem

$$\text{minimise} \sum_{i=0}^{T} \|t_i - \textsc{closest}(t_i, P)\|$$

[6] `https://www.rarewares.org/rrw/nttvqf.php`

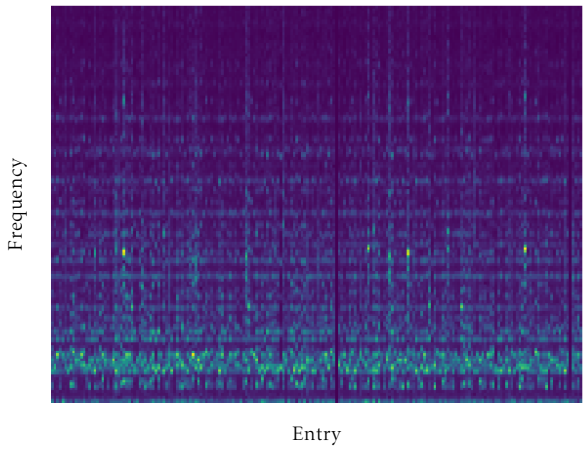

Figure 4.10: The first 200 entries with the lower 150 frequency lines of the codebook. Each entry replaces either an identical, or a close vector in the spectrogram, providing compression if there are fewer entries in the codebook than in the spectrogram.

For the distance metric $\|\cdot\|$, Euclidean distance is used.

Algorithm 11 Vector quantisation compression

function MAIN(WaveFile W, CompressionFactor f_c) ▷ Waveform audio file
  $P \leftarrow \varnothing$ ▷ Prototype vector codebook
  $L \leftarrow \varnothing$ ▷ Labels
  $S \leftarrow$ SHORT-TIMEFOURIERTRANSFORM(W) ▷ S is a $C \times N \times T$ matrix
  $S_{abs} \leftarrow |S|$ ▷ Obtain magnitude of every amplitude
  $S_q \leftarrow$ ROUND($\gamma\sqrt{S_{abs}}$) ▷ Quantisation
  $r \leftarrow T/f_c$
  $S_q \leftarrow$ CONCAT(S_q^l, S_q^r) ▷ Concatenate left and right channel along the time axis
  $P \leftarrow$ LLOYD(S_q, r) ▷ Choose r prototype vectors from spectrogram, over both channels
  for $t \in S_q$ **do** ▷ Iterate over T frames
    $L \leftarrow L \cup \arg\min_i \|t - P[i]\|$ ▷ Choose index i of prototype vector which is closest to l
  end for
  $\hat{P} \leftarrow$ ARITHMETICCODER(P)
  $\hat{L} \leftarrow$ ARITHMETICCODER(L)
  return $(\hat{P}|\hat{L})$ ▷ | denotes concatenation
end function

4.6.2 Results

The results of the compression algorithm are given in Table 4.4. As the compression factor f_c is linearly related to the number or prototype vectors, the compression ratio is influenced directly by f_c. Typical values for f_c are between 1 and 10. However, computation becomes more expensive with higher f_c.

As the compression ratio is predictable with setting the compression factor, the effectiveness of this approach is to be judged by the audio quality of the compressed samples. The audio quality is generally good, but compression artefacts are perceivable, especially in the less repetitive "classical" sample and with high f_c. However, the "electronic" remains relatively clear even with $f_c = 10$ and a compression ratio that is almost 4 times better than MP3. Especially repetitive

Algorithm 12 Vector quantisation decompression

function MAIN(Compressed file $(\hat{P}|\hat{L})$)
  $P \leftarrow$ ARITHMETICDECODER($\hat{P}$)
  $L \leftarrow$ ARITHMETICDECODER($\hat{L}$)
  $\tilde{S}_q \leftarrow P[L]$ ▷ $[\cdot]$ accesses codebook using index
  $\tilde{S}_{abs} \leftarrow (\tilde{S}_q/\gamma)^2$ ▷ Dequantise
  $\tilde{W} \leftarrow$ INVERSESHORT-TIMEFOURIERTRANSFORM($\tilde{S}_{abs}$) ▷ Reconstruct waveform
  return $\tilde{W}$
end function

Sample	Classical	Electronic
Uncompressed (Wave)	5814 kb (1:1)	4198 kb (1:1)
Lossy compression (MP3)	528 kb (1:11.01)	382 kb (1:10.99)
Lossless compression (FLAC)	1940 kb (1:3.00)	1972 kb (1:2.13)
Vector quantisation ($f_c = 1$)	840 kb (1:6.92)	-
Vector quantisation ($f_c = 2$)	433 kb (1:13.43)	447 kb (1:9.39)
Vector quantisation ($f_c = 4$)	228 kb (1:25.50)	229 kb (1:18.33)
Vector quantisation ($f_c = 10$)	-	98 kb (1:42.84)

Table 4.4: Comparison of compressing audio using tensor factorisation. Here, the "classical" sample compresses worse, and the audio quality of the "classical" sample is similar to the quality of the "electronic" sample compressed with the compression factor shifted by one.

elements are approximated well with the prototype vectors, while other musical features are neglected. For low f_c compression ratios of vector quantisation and audio quality are both worse than comparable lossy compression, but for higher f_c and music with repetitive features, audio quality degrades slowly while high compression ratios can be achieved.

4.6.3 Conclusion

Although the results of this naive vector quantisation method are promising, especially at high f_c and for music with repetitive patterns, several issues would need to be addressed to make it more competitive. Starting with the compression side, the codebook is passed to arithmetic coding without taking advantage of its structure. A first improvement would be to sort the prototype vectors by their similarity (e.g. by using the Euclidean distance metric) and then store the difference of one vector to another. These differences would be minimised by sorting by similarity and may be encoded efficiently. For this, the labels would need to be updated to the prototype vector's indexes after sorting.

Although this vector quantisation approach violates independence between frames (the codebook needs to be transmitted to decode one vector), infinite streams were possible by transmitting a fixed codebook beforehand to a client and then streaming labels to it (given that they refer to entries in the codebook). An imaginable application would be to define a codebook derived from typical audio files, transmit only the labels and if necessary, transmit new prototype vectors to update the client's codebook.

In terms of audio quality, vector quantisation of this approach has to deal with prototype vectors that are still fairly correlated (see codebook in Figure 4.10) *within* the frequency spectrum. For example, in a prototype vector that consists of three overlaid musical patterns (e.g. bass, percussion and melody), these elements become correlated by being packed into the same vector although they may appear independently (e.g. melody and bass without percussion). The

k-means approach may then assign this prototype vector to a frame where only melody and bass appear, falsely introducing the percussion to that frame. This is an instance of the problem of composite patterns (see Chapter 2), where elements are represented jointly although they could in principle be well-separated. Consequently, the storage capacity of the codebook is consumed to represent rare combinations of patterns instead of absorbing more, but less correlated individual patterns. Using this idea, it can be explained why the vector quantisation approach is more effective with the "electronic" sample in which melody and bass appear in almost in lockstep (see e.g. Figure 4.6) and that musical elements deviating from this structure receive strong compression artefacts. The crux of decomposing the combinations into independent patterns is that although the musical elements may appear to the human sense as clearly distinct, they are intertwined in the spectrogram representation. In the individual case, it may be possible to separate the patterns by separating the frequency bands, although in general, the patterns are superimposed in the spectrogram and attempts to obtain decorrelation by separating frequency bands was not successful.

4.7 Neural Network Approaches

Code reference. Code for this section can be found in the `notebooks/nn.ipynb` Jupyter Notebook.

The use of neural networks are well motivated for the task of data compression. First, they are well suited to extract high-level information even from low-level representation of data [10, 16], as for example in the well-known task of recognising hand-written digits in the form of 64×64 grayscale images [48]. Related to this is the capability that neural network approaches often have very few assumptions about the input data, allowing them to use structure that has not been hard-coded by a human programmer [48]. Secondly, neural networks have been used in data compression starting with text compression [47], succeeded with the more practical PAQ compressors [16] and most recently for audio compression [13]. Finally, neural networks have, at least in theory, the ability to decompose composite patterns [10], providing important decorrelation. Concretely, subdivided attractor networks — an alternative architecture to feedforward neural networks — are able to detect a pattern in each subdivision instead of learning the composite patterns as a whole, which are, using a combinatorial argument, much more in number.

4.7.1 Recurrent Neural Networks

Inspired by the success of gated recurrent neural networks for predicting samples of audio sequences, including music [107], the class of recurrent neural networks (RNN) is reviewed. RNNs are extensions of neural networks that are used for modelling sequence-like data [48] and are particularly successful for machine translation [107], or in generative language models [49, 50, 51].

While a conventional feedforward neural network maps input values through hidden layers to an output layer, a RNN maintains a hidden state h that maintains a memory over the sequence that has been fed to the network (see Figure 4.11). More formally, in every evaluation, the hidden layer is first updated as

$$h_t = \sigma(W_{xh}x + W_{hh}h_{t-1} + b_h)$$

where σ is an activation function, x, b_h and h are vectors and W_{hh} and W_{xh} are matrices. The matrices represent the synapses between neurons and quantify the strength of their interaction

and b_h is an additive bias to the interaction. The network's output is given by

$$y = \sigma(W_{hy} h_t) + b_y$$

where W_{hy} is a matrix for the interaction strength between the hidden state h_t and the output y. b_y is again an additive bias.

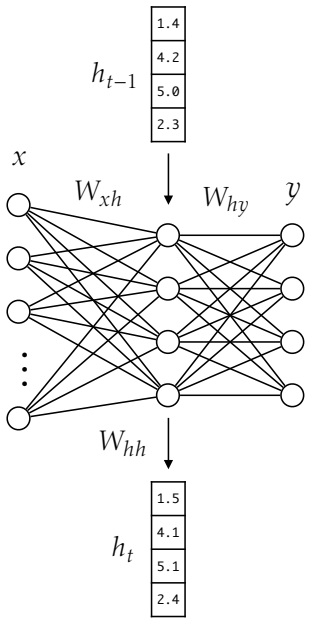

Figure 4.11: Structure of an RNN. The hidden layer h_t is updated from the input x and a previous hidden layer h_{l-1}. The output y only depends on the hidden layer h_l.

Here, there is only a single hidden layer but in practice, multiple hidden layers are used. Such RNNs are usually trained by the backpropagation through time algorithm [108]. Intuitively, it starts by measuring the error of the output y against training data and then adjusting weights first in the W_{hy} matrix with respect to the error, then proceeding towards the input, adjusting the matrices W_{hh} and W_{xh}. To bring the error in relation to the matrix weights, derivatives of the network in the form of gradients are used. The details of this algorithm are beyond the scope of this thesis, but is explained in more detail by Nielsen [48]. RNNs are seldom used in this form as they have difficulties retaining long-range dependencies. As mitigation, the mechanism for updating the hidden state is modified, resulting in long short-term memory (LSTM) or gated recurrent units (GRU) [107].

Still, we make a trivial attempt to predict the amplitudes from a spectrogram. Given a sliding window of width Δt frames, the window of same width, shifted by one frame should be predicted. Two prediction results are shown in Figure 4.12 with $\Delta t = 5$ (attempts with $1 \leq \Delta t \leq 5$ were conducted). It becomes apparent that the neural network generally predicts the amplitudes correctly if they remain mostly constant, but cannot predict time-varied patterns. Furthermore, the RNN used here is too limited in its functionality and the ability of non LSTM- or GRU networks difficulties to model long-range dependencies is long known [109]. As this thesis is not focused on deep learning methods, the route to deep RNNs for modelling and compressing audio sequences will not be pursued.

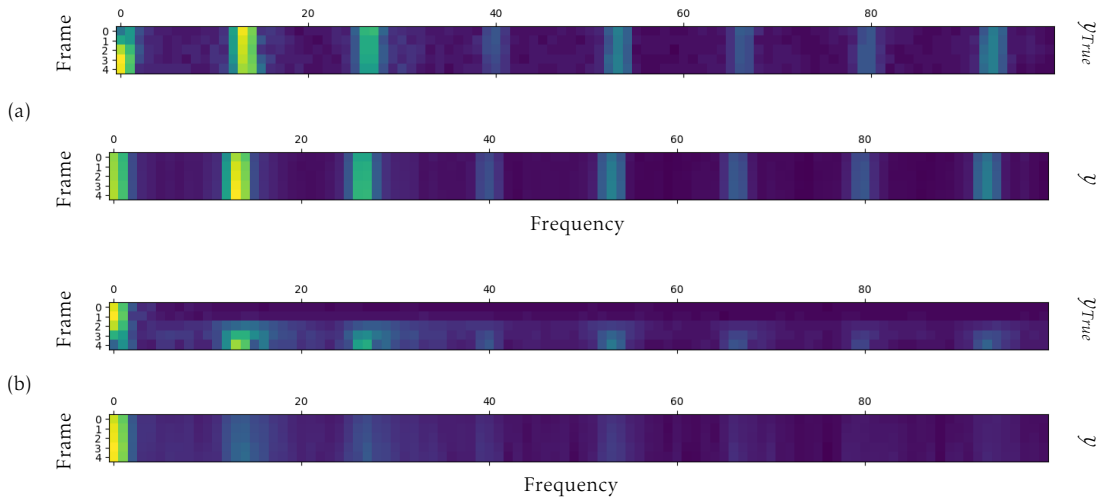

Figure 4.12: Two extracts of the results obtained with the RNN. (**a**) The predictions y are accurate, except that the RNN does not predict the noise within the spectrogram of the ground truth y_{True}. (**b**) The audio data is more time-varied and it becomes apparent that the neural network has not modelled changes that occur within the Δt window.

4.7.2 Reservoir Computing

In nonlinear and dynamical systems, *echo-state neural networks* (ESN), part of modern *reservoir computing*, have been used to model chaotic time series [110]. While the concentration of most current machine learning research lies on conventional neural networks, reservoir computing is gaining attention again [111, 112]. As ESN often deal with sequential data, they have been seen as a kind of RNN [110], although the modern term refers more to RNNs as described above. In contrast to conventional deep neural networks, methods from reservoir computing require smaller data sets and can be learned with efficient algorithms. Reservoir computing has also been used for audio analysis and prediction by Holzmann [113], but not for compression.

An ESN can have multiple in- and output neurons, although in this case, there is only one input neuron that represents a sample of the audio waveform and one output neuron for prediction. Informally, the reservoir can be understood as a collection of functions (linear and nonlinear) of the input values, produced by random connections between the neurons, given by the matrix W. To make the ESN's predictions reproducible (as would be needed in data compression when the same predictions are required by the encoder and decoder), the seed of the random number generator used for W can be fixed, making the generation of W deterministic. In training, the linkage of neurons in the reservoir with the output elements is adjusted by tuning weights $w \in W_{out}$ by minimising the mean squared error

$$\frac{1}{N}\sum_{i=0}^{N}(s_i - \sum_{j=0}^{r} w_j a_j(s_i))^2$$

with N samples s, and where w_j is the weight from the j-th neuron to the output (from W_{out} using a one-dimensional index), $a_j(s_i)$ is the activation of the j-th neuron when given sample s_i. The minimisation of this quantity can be formulated as linear regression [110].

Here, the ESN is run in two configurations. In the first one (data-stimulated), the ESN is fed 2000 audio waveform samples, with the first 700 as input and the last 700 as output, training the network to predict samples 300 samples into the future. Testing is then done by feeding the

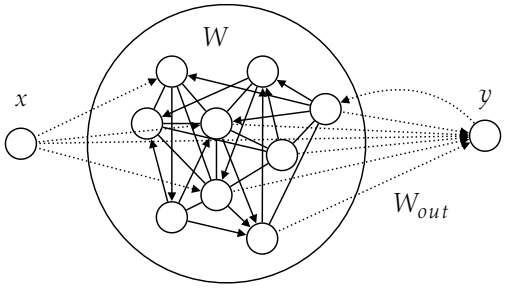

Figure 4.13: The ESN has a pool of interconnected neurons (the reservoir). Only the weights W_{out} from the reservoir to the output neuron are trained. The interaction strength of the reservoir neurons is defined by a random matrix W.

last 300 samples to the network as input, predicting for 300 samples into the future. The second configuration (constant-stimulated) uses 2000 audio waveform samples as output, but no data as input (a string of ones is fed to the network). This ESN is also tasked to predict 300 samples into the future but is again only stimulated by ones. The idea of the second configuration is that the ESN learns the dynamics of the audio waveform independently of previous data and only by adjusting its output weights. Parameters of the models are listed in Table 4.5. For dynamical systems, this is a frequent approach, as it is desired that the ESN simulates the behaviour of the observed dynamical system [111]. As a baseline for comparison, an autoregressive model AR(10) is used. The results are compared in Figure 4.14. The data-stimulated ESN produces the lowest mean squared error (MSE) of 0.099, while the constant-stimulated ESN has an MSE of 0.11 and the autoregressive model 0.13.

However, the predicted dynamics are generally better by ESNs rather than the autoregressive model. Still, compared to the results from Holzmann [113], the predictions of the ESNs shown here are significantly worse, although very similar parameters were attempted. The difference may be explained by the choice of standard neurons rather than filter neurons that are sensible to frequency bands and thus able to decompose the signal into a frequency spectrum [113]. This observation was made by Holzmann himself, noting that standard neurons did not predict audio data as well.

Configuration	Data-stimulated	Constant-simulated
Reservoir size	200	300
Spectral radius	0.4	1
Sparsity	0.1	0
Noise	0.0002	0.02

Table 4.5: Parameter configuration of the ESN. The base parameters were taken from [113], but adjusted by trial-and-error.

4.7.3 Conclusion

Deep recurrent neural networks are especially powerful in natural language processing and in audio modelling. Meanwhile, audio data can be compressed efficiently using neural networks in the form of autoencoders. Still, there remains the trade-off that small, interpretable networks as the RNN in this thesis are too weak for serious audio modelling and deeper RNNs too obscure for the goals of this thesis. For lossless compression in the style of the PAQ compressors, the predictions obtained from ESNs are insufficient to serve as probability models, having too much error. However, with an improved model, e.g. using filter neurons, the ESN may be considered for

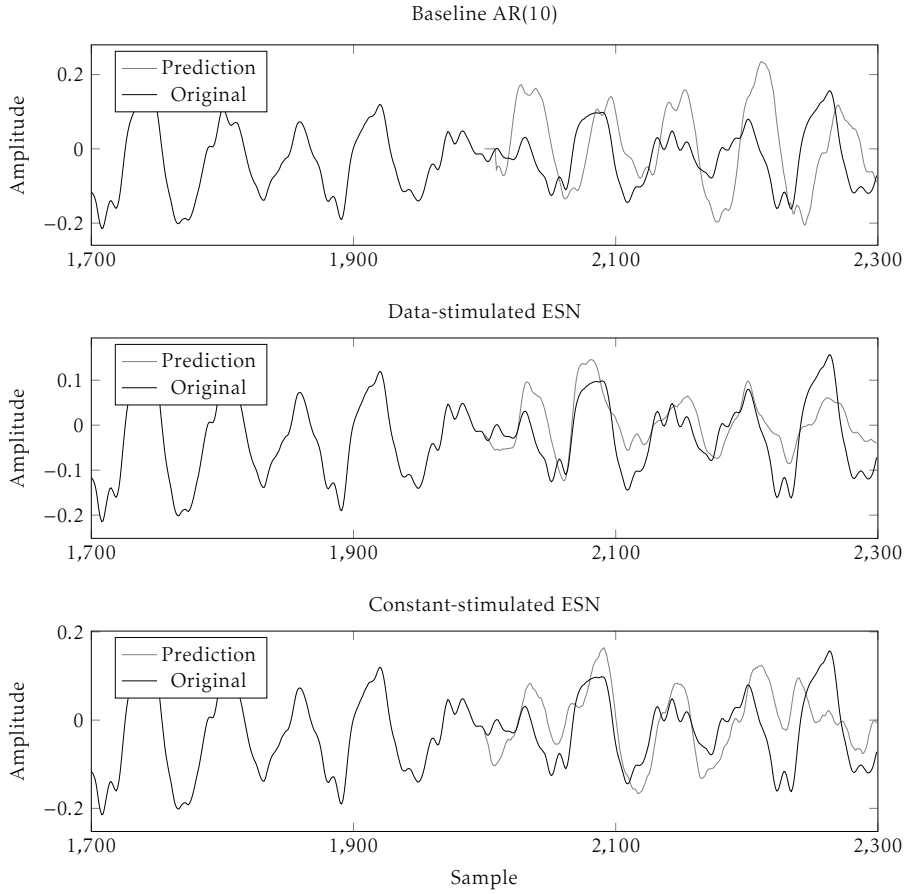

Figure 4.14: The original waveform audio is predicted by an autoregressive model using 10 lags AR(10), the data-stimulated ESN and the constant-stimulated ESN.

prediction. An alternative application is the replacement of the polynomial fit or linear predictive coding (LPC) of FLAC.

A sketch of the operation of such an encoder is as follows. Recall that in FLAC, typically 4096 waveform samples are approximated using either a polynomial or with LPC and the error (residue) stored separately to prevent loss of information. A ESN could be trained, given the samples of a frame, to predict some or all of the samples of a subsequent frame. For this, the ESN of the encoder could be fed with the uncompressed samples (or the freshly decompressed samples in case of the decoder) to predict the next frame. As the algorithms for fitting a polynomial and for learning an ESN are similar[7], the computational cost may be similar as well. Using filter neurons, ESNs may achieve sufficient accuracy to outperform the polynomial or LPC approaches.

[7] Linear regression is the task of fitting a first-order polynomial. Yet, learning the ESN is additionally dependent on the reservoir neurons, while curve fitting in FLAC is only dependent on the samples.

4.8 Audio Compression using Image Compression Methods

Code reference. Code for this subsection can be found in the `notebooks/fractal.ipynb` Jupyter Notebook.

As spectrograms as 2-d matrices with scalar-valued elements have a similar data structure to image matrices, there is a justified question whether they are amenable to image compression techniques. For the task of audio identification, the task of identifying a piece of music given a noisy sample of it, methods from computer vision have already been used, following a similar intuition [114]. Shazam, an online service providing audio identification uses a similar method based on image processing [115]. This section examines fractal image compression and gives some limitations of image compression methods.

4.8.1 Fractal Image Compression

In short, the compressed representation obtained from fractal image compression consists of a collection of transformations that relate a smaller block of an image to a larger block of it. If the transformations W are applied repeatedly to any image matrix f_0 (e.g. a random matrix), it converges towards an approximation f^* of the original matrix f_{orig}. Typically, f_{orig} cannot be reconstructed as it cannot be captured perfectly by the transformations. Also, due to the nature of transformations relating a smaller and a larger block of an image matrix, the method works better if the image matrix is self-similar, i.e. similar elements at different scales.

The rationale to use it for audio compression is that repeating patterns that appear in some spectrograms may be captured by fractal image compression. Even if these patterns are not self-similar, similar instances may be described using the same, larger block and thus be encoded more efficiently, remotely similar to the rationale of dictionaries and vector quantisation. In Figure 4.15, the original and the reconstruction of its compression are shown.

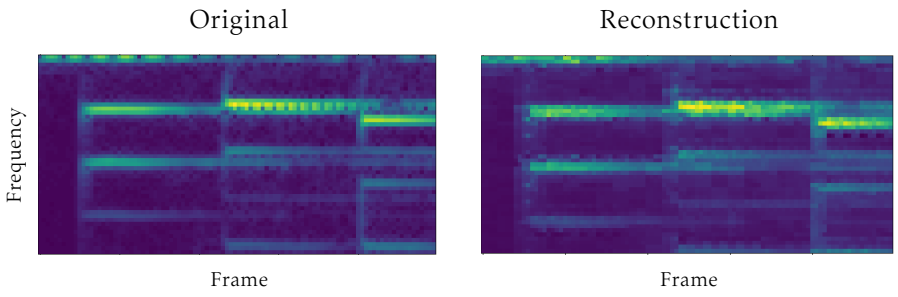

Figure 4.15: Original block of a spectrogram and its fractal reconstruction. Fractal image compression introduces some noise into the spectrogram.

The state after each iteration $f = W(f)$ for reconstruction is shown in Figure 4.16. Note that the initial matrix f_0 is a random matrix, and that due to the contractive property, any image matrix converges to the approximated original matrix.

An issue with fractal image compression is that it is computationally expensive — for every range block R_i (the smaller blocks) an appropriate variant ω_{jk} of a domain block D_j must be found. As the number of range and domain blocks, are both proportional to the size of the image matrix, the number of evaluations of similar blocks is quadratically dependent on the size of the image matrix. The evaluation is constant as the difference of R_i and D_j are computed as

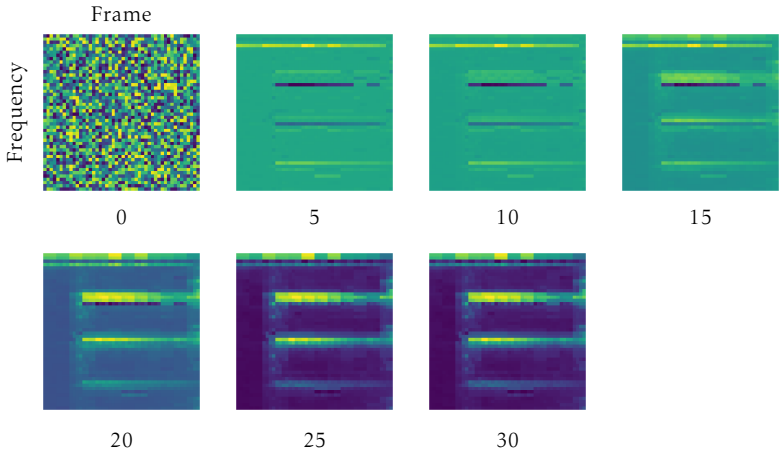

Figure 4.16: Iterations of the reconstruction of fractal image compression from a 50×50 block of a spectrogram. The first panel is a random matrix, which converges towards an approximation of the original matrix by repeatedly applying transformations.

error $e(\omega, R_i)$. However, as this necessitates rescaling and comparing all of the block elements, many operations are required. Note that in JPEG, for example, the 8×8 blocks are processed independently, making its encoding procedure linear. With the current implementation, it takes several seconds to encode a 50×50 block and larger blocks require as expected, quadratically more time. It would be conflicting with the rationale of reusing self-similar patterns within an image by applying fractional compression on smaller sections of the spectrogram, hindering the algorithm to find similar patterns further away. While the computational effort may be reduced, image quality decreases as fewer self-similar blocks are available.

Additionally, because the number of affine transformations is dependent on the number of range and domain blocks, an image (or spectrogram) with high self-similarity could not be encoded more efficiently. Rather, the redundancy of the image would be encoded into the transformations (pointing to few domain blocks), but may remain too obscure to be detected by a second lossless compression round.

Due to the computational intensity, no audio samples are compressed with fractional image compression and instead, its compression ratio is estimated analytically. The image matrix of Figure 4.16 is compressed with 12×12 affine transformations, each containing 6 numbers (two indexes for referring to the larger block, direction, angle, contrast and brightness). As the number of transformations proportional to the image size, it makes no difference to the compression ratio whether the spectrogram is compressed in smaller blocks or as a whole. The two indexes and the direction and angle allow for 288 choices that can be encoded using 9 bits. The contrast and brightness levels can be encoded as two 16-bit floating point numbers, yielding 41 bit per transformation. Thus, all 144 transformations require 738 bytes. The spectrogram of the "electronic" sample has $C \times N \times T = 2 \times 513 \times 4101$ elements, which would require at least $2 \times 10 \times 82$ blocks, for a total approximately 1210 kilobytes. Omitting concerns over quality and gains of entropic coding, the compression ratio of 1:4.80 is good, but would not be competitive with lossy coders.

4.8.2 Limitations

Although the application of image compression methods has is effective to a limited degree to audio data, it reveals several limitations. Firstly, there is an inherent asymmetry between the axes

of audio data, namely that audio is time-dependent while static images are not. For compression, this asymmetry is relevant in at least two ways.

First, in audio data with a repetitive structure, the repetitions appear in the time axis, but not in the frequency axis, which is best visible in a spectrogram. Meanwhile, image compression algorithms assume that structure along two axes is similar, as can be seen for example in the symmetrical block splitting of JPEG or fractal image compression.

Secondly, due to the temporal axis, it may be desirable to process audio data in slices such that an audio file can be played back while it is decoded, or that splitting an audio file's byte sequence results in a split in the temporal axis of the audio content. This property is particularly relevant if audio files are long or are infinite, such as in a live stream. Meanwhile, it is normal to encode an image as a whole, already because there is no "natural" partitioning.

A second limitation is that most image compression algorithms imply strong assumptions about the image's structure, for example that repetition does rarely occurs, or that patterns are generic and self-similar (whereas in audio, they are specific and different at micro- and macro-scales), a feature exploited directly by fractal image compression. Preliminary tests with a JPEG-inspired approach indicate that small amplitudes in high frequencies are treated as noise and therefore discarded. This assumption is typically true for the visual case, but not for the auditory — discarding the noise removes high-frequency components from the audio, essentially acting as a low-pass filter, making audio samples sound muffled. These limitations led to the decision to discontinue the approach of applying image compression methods to audio.

4.9 Further Approaches

This section describes sketches of approaches responding to the issues encountered with the methods above. However, they are neither theoretically rigorous nor practically validated but may serve as inspiration for future investigations of structure discovery or data compression.

4.9.1 Topological Delay Embeddings

Skraba et al.'s [97] approach to embed 1-d data of dynamical systems in higher-dimensional space to analyse their periodicity and dynamics appears very promising to decompose the repetitive structure of some music, especially as the repetitions appear on different time scales (see 2.4.2 micro-macro problem). However, the systems analysed by the authors are classical examples with low embedding dimensions and well-recognisable cycles.

In contrast, the 1-d waveform signals of audio data clearly have periodicity as well (see Figure 4.17), as they can be transformed into spectrograms using tools such as the STFT. However, parting from the spectrogram form, periodic structure is contained in the order of seconds (or tens to hundreds of STFT frames) and resides is on specific frequency bands. Figure 4.17 additionally shows the waveform signal on a phase plot with one axis showing the delayed signal using $\Delta t = 100$, illustrating periods as circles.

Given the degrees of freedom and the superposition of the patterns, combined with preliminary tests using Takens's embeddings on audio waveform data, more theoretical groundwork is necessary for a topological delay embedding approach to audio data and music. In particular, it is unclear how to proceed with 2-d representation that already reveals structure in form of the distribution of amplitudes and small-scale patterns, but the macrostructure in the form of loops and repetitions remains to be extracted. In other words, there need to be found ways that elevate

the topological delay embeddings approach to serve a purpose beyond simply replacing Fourier- or related transformation approaches.

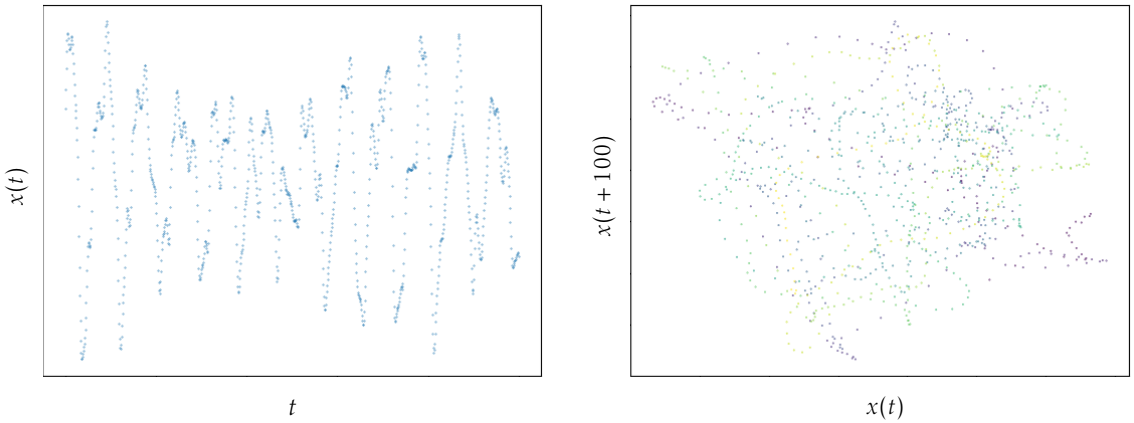

Figure 4.17: The left panel shows the time-varied waveform signal of 1000 amplitudes $x(t)$. On the right, a phase plot of the same signal reveals periods within the signal as loops, as targeted by Takens's theorem, or Skraba et al.'s [97] approach.

4.9.2 Fractal Wave Compression

Inspired by fractal image compression that operates in two dimensions, the same approach could be used for one-dimensional waveform audio. For this, range and domain regions can be imagined as one-dimensional intervals on the waveform curve (see left panel of Figure 4.17. Also note the self-similarity of the curve.). While fewer degrees are possible in this situation (e.g. contrast, brightness or the angle can be discarded), a condition should be introduced that the curve is matched approximately between successive blocks in order to reduce discontinuities in the signal.

5 | Conclusion

Data compression is a wide research area that maintains connections to almost the whole field of computer science and beyond. At its core, data compression is about discovering and using structure, and the sophistication of the structure found often correlates with the gains in compression, on the extreme end theoretically illustrated by Kolmogorov complexity seeking an algorithmic description of data. Audio compression is no exception to this observation. The attempt to find structures beyond the level of entropic coding has led through a survey through data compression methods and relationships to data analysis, topology, neural networks and others.

While it is simple for humans to separate different sounds and patterns within a piece of music, this task appears to be difficult even for modern-day neural networks that otherwise excel in pattern recognition. Reviewing the literature, the recognition and exploitation of audio patterns in the order of seconds for compression has remained elusive in both practice and research, though this thesis makes several approaches to it and gathers insights in more and less promising approaches.

The subgoal of replacing the existing entropic coder of MP3 has not brought gains in compression. However, a neural-network based universal compressor has brought 5% additional reduction in file size of MP3-encoded audio, and 10% could be gained by employing a compression method relying on a method from topology.

Drawing from topology, tensor factorisation and vector quantisation, we propose three prototypical audio compressors operating on uncompressed audio waveforms. The compressors are tested under different configurations and with different audio samples. The lossless topology-based compressor operates on-par with FLAC and for a specially prepared sample, the tensor factorisation approach achieves a compression ratio of 1:280 while retaining much of audio quality. Finally, we evaluate how suitable image compression and recurrent and echo state neural networks are for audio compression.

This thesis is more explorative in nature than what it targets an implementation or the evaluation of a specific method for audio compression. As a consequence of this exploration, we conclude this thesis with a brief outlook into future compression methods.

5.1 Outlook

As a subfield of mathematics, topology finds increasing application as topological data analysis. There, it helps unravel structure in data that is obscured by noise, high dimensionality or where there is no inherent metric. Simultaneously, self-organising maps are learning frameworks that pursue the topologically intact capture of data points. Using topological delay embeddings, data from complex and dynamical systems can be described in a concise way.

While topology has remained relatively weakly connected to computer science, the above examples illustrate several success stories of their combination. Because topology is strongly intertwined with the concepts of dimensionality, shape and structure that play an important role in data compression, deepening the connections between these topics may be a part of taming todays data deluge and reaching the 4.5 compression factor of today's data mentioned in the introduction.

As current developments of conceptual compression and the employment of neural architectures in computer science gear towards higher-level representations of data, these developments seem compatible with the non-geometric spirit of topology. Finally, this higher-level data representation performed by deep neural networks deserves studies in its own right and may have significance well beyond data compression. Simultaneously, ideas from information theory and data compression are already used to better understand neural networks. There seems to be plenty of space for fruitful cross-fertilisation between these topics.

Bibliography

[1] M. Hilbert and P. López, "The world's technological capacity to store, communicate, and compute information," *Science*, vol. 332, no. 6025, pp. 60–65, 2011.

[2] G. Hudson, A. Léger, B. Niss, and I. Sebestyén, "Jpeg at 25: Still going strong," *IEEE MultiMedia*, vol. 24, no. 2, pp. 96–103, 2017.

[3] D. Le Gall, "Mpeg: A video compression standard for multimedia applications," *Communications of the ACM*, vol. 34, no. 4, pp. 46–58, 1991.

[4] K. Brandenburg, "Mp3 and aac explained," in *Audio Engineering Society Conference: 17th International Conference: High-Quality Audio Coding*, Audio Engineering Society, 1999.

[5] J. Sterne, "The mp3 as cultural artifact," *New media & society*, vol. 8, no. 5, pp. 825–842, 2006.

[6] D. Rohr, "Gpu-based reconstruction and data compression at alice during lhc run 3," in *EPJ Web of Conferences*, vol. 245, p. 10005, EDP Sciences, 2020.

[7] M. Soler, M. Plainchault, B. Conche, and J. Tierny, "Topologically controlled lossy compression," in *2018 IEEE Pacific Visualization Symposium (PacificVis)*, pp. 46–55, IEEE, 2018.

[8] L. Amarú, P.-E. Gaillardon, A. Burg, and G. De Micheli, "Data compression via logic synthesis," in *2014 19th Asia and South Pacific Design Automation Conference (ASP-DAC)*, pp. 628–633, IEEE, 2014.

[9] M. Mahoney, "Data compression explained," 2013. `http://mattmahoney.net/dc/dce.html` [Accessed on 21.03.2023].

[10] Y. Bar-Yam, *Dynamics of complex systems*. CRC Press, 1998.

[11] D. J. MacKay, *Information theory, inference and learning algorithms*. Cambridge University Press, 2003.

[12] C. E. Shannon, "A mathematical theory of communication," *The Bell system technical journal*, vol. 27, no. 3, pp. 379–423, 1948.

[13] A. Défossez, J. Copet, G. Synnaeve, and Y. Adi, "High fidelity neural audio compression," *arXiv preprint arXiv:2210.13438*, 2022.

[14] M. A. Kramer, "Nonlinear principal component analysis using autoassociative neural networks," *AIChE journal*, vol. 37, no. 2, pp. 233–243, 1991.

[15] F. Bellard, "Lossless data compression with neural networks," 2019. https://bellard.org/nncp/nncp.pdf [Accessed 28.07.2023].

[16] M. V. Mahoney, "Fast text compression with neural networks.," in *FLAIRS conference*, pp. 230–234, 2000.

[17] G. E. Blelloch, "Introduction to data compression," 2013. https://www.cs.cmu.edu/~guyb/realworld.html [Accessed on 15.03.2023].

[18] A. N. Kolmogorov, "On tables of random numbers," *Theoretical Computer Science*, vol. 207, no. 2, pp. 387–395, 1998.

[19] M. Mahoney, "Rationale for a large text compression benchmark," 2009. http://mattmahoney.net/dc/rationale.html [Accessed on 15.03.2023].

[20] K. Lagerstrom, "Design and implementation of an mpeg-1 layer iii audio decoder," *Chalmers University of Technology, Department of Computer Engineering Gothenburg, Sweden*, 2001.

[21] M. Hutter, *Universal artificial intelligence: Sequential decisions based on algorithmic probability*. Springer Science & Business Media, 2004.

[22] I. H. Witten, R. M. Neal, and J. G. Cleary, "Arithmetic coding for data compression," *Communications of the ACM*, vol. 30, no. 6, pp. 520–540, 1987.

[23] Undisclosed authors, "Dirac video codec 1.0 released," 2008. https://lwn.net/Articles/272520/ [Accessed on 17.03.2023].

[24] J. Bankoski, J. Koleszar, L. Quillio, J. Salonen, P. Wilkins, and Y. Xu, "Vp8 data format and decoding guide," 2011. https://datatracker.ietf.org/doc/html/rfc6386 [Accessed on 17.03.2023].

[25] D. R. Horn, K. Elkabany, C. Lesniewski-Laas, and K. Winstein, "The design, implementation, and deployment of a system to transparently compress hundreds of petabytes of image files for a file-storage service.," in *NSDI*, pp. 1–15, 2017.

[26] J. Duda, "Asymmetric numeral systems," 2009.

[27] J. Duda, "Asymmetric numeral systems: entropy coding combining speed of huffman coding with compression rate of arithmetic coding," 2014.

[28] S. Golomb, "Run-length encodings (corresp.)," *IEEE transactions on information theory*, vol. 12, no. 3, pp. 399–401, 1966.

[29] J. M. Stettbacher, "Lecture notes on the "Information und Codierung" course at the Zurich School of Applied Sciences," 2018.

[30] G. V. Cormack and R. N. S. Horspool, "Data compression using dynamic markov modelling," *The Computer Journal*, vol. 30, no. 6, pp. 541–550, 1987.

[31] J. Cleary and I. Witten, "Data compression using adaptive coding and partial string matching," *IEEE transactions on Communications*, vol. 32, no. 4, pp. 396–402, 1984.

[32] N. Ahmed, T. Natarajan, and K. R. Rao, "Discrete cosine transform," *IEEE transactions on Computers*, vol. 100, no. 1, pp. 90–93, 1974.

[33] "Computer Image Processing and Analysis (E161) lecture notes, Harvey Mudd College," 2018. https://web.archive.org/web/20180214073623/http://fourier.eng.hmc.edu/e161/lectures/klt/node3.html (Archived) [Accessed 22.03.2023].

[34] T. Strohmer, "Singular value decomposition and principal component analysis," 2017. http://math.ucdavis.edu/~strohmer/courses/180BigData/180lecture_svd_pca.pdf [Accessed 28.05.2023].

[35] V. Loan, "Structured matrix computations from structured tensors lectures," 2015. https://www.dm.unibo.it/~simoncin/CIME/vanloan3.pdf.

[36] S. Rabanser, O. Shchur, and S. Günnemann, "Introduction to tensor decompositions and their applications in machine learning," *arXiv preprint arXiv:1711.10781*, 2017.

[37] L. De Lathauwer, B. De Moor, and J. Vandewalle, "On the best rank-1 and rank-(r 1, r 2,..., rn) approximation of higher-order tensors," *SIAM journal on Matrix Analysis and Applications*, vol. 21, no. 4, pp. 1324–1342, 2000.

[38] R. Ballester-Ripoll, P. Lindstrom, and R. Pajarola, "Tthresh: Tensor compression for multidimensional visual data," *IEEE transactions on visualization and computer graphics*, vol. 26, no. 9, pp. 2891–2903, 2019.

[39] J. B. Smith and M. Goto, "Nonnegative tensor factorization for source separation of loops in audio," in *2018 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pp. 171–175, IEEE, 2018.

[40] W. Jing, X. Xiang, and K. Jingming, "A novel multichannel audio signal compression method based on tensor representation and decomposition," *China Communications*, vol. 11, no. 3, pp. 80–90, 2014.

[41] C. Rohlfing, J. E. Cohen, and A. Liutkus, "Very low bitrate spatial audio coding with dimensionality reduction," in *2017 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pp. 741–745, IEEE, 2017.

[42] S. Quinlan, S. Dorward, *et al.*, "Venti: A new approach to archival storage.," in *FAST*, vol. 2, pp. 89–101, 2002.

[43] A. Tridgell, P. Mackerras, *et al.*, "The rsync algorithm," 1996.

[44] D. Vatolin, I. Seleznev, and M. Smirnov, "Lossless video codecs comparison 2007," *Inf. téc., Graphics & Media Lab (Video Group) of Moscow State University*, 2007.

[45] M. Z. Alom, T. M. Taha, C. Yakopcic, S. Westberg, P. Sidike, M. S. Nasrin, B. C. Van Esesn, A. A. S. Awwal, and V. K. Asari, "The history began from alexnet: A comprehensive survey on deep learning approaches," *arXiv preprint arXiv:1803.01164*, 2018.

[46] F. Rosenblatt, "The perceptron: a probabilistic model for information storage and organization in the brain.," *Psychological review*, vol. 65, no. 6, p. 386, 1958.

[47] J. Schmidhuber and S. Heil, "Sequential neural text compression," *IEEE Transactions on Neural Networks*, vol. 7, no. 1, pp. 142–146, 1996.

[48] M. A. Nielsen, *Neural networks and deep learning*, vol. 25. Determination press San Francisco, CA, USA, 2015.

[49] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, *et al.*, "Language models are few-shot learners," *Advances in neural information processing systems*, vol. 33, pp. 1877–1901, 2020.

[50] H. Touvron, L. Martin, K. Stone, P. Albert, A. Almahairi, Y. Babaei, N. Bashlykov, S. Batra, P. Bhargava, S. Bhosale, *et al.*, "Llama 2: Open foundation and fine-tuned chat models," *arXiv preprint arXiv:2307.09288*, 2023.

[51] A. Karpathy, "State of gpt," 2023. `https://karpathy.ai/stateofgpt.pdf` [Accessed 28.07.2023].

[52] K. Gregor, F. Besse, D. Jimenez Rezende, I. Danihelka, and D. Wierstra, "Towards conceptual compression," *Advances In Neural Information Processing Systems*, vol. 29, 2016.

[53] L. Ziyin, T. Hartwig, and M. Ueda, "Neural networks fail to learn periodic functions and how to fix it," *Advances in Neural Information Processing Systems*, vol. 33, pp. 1583–1594, 2020.

[54] G. P. Hom, "Lecture notes on the mit introductory digital systems course (6.111), lecture 2," 2016. `https://web.mit.edu/6.111/www/f2016/handouts/L02_4.pdf` [Accessed 26.07.2023].

[55] S. Shirani, "Lecture notes on the Multimedia Communications at the McMaster University," 2012. `https://www.ece.mcmaster.ca/~shirani/multi12/vectorq.pdf`, [Accessed 30.07.2023].

[56] D. H. Ballard, *An introduction to natural computation*. MIT press, 1999.

[57] T. Kohonen, "Self-organized formation of topologically correct feature maps," *Biological cybernetics*, vol. 43, no. 1, pp. 59–69, 1982.

[58] R. Wang, "Self-organized patterns in the som network," in *2015 11th International Conference on Natural Computation (ICNC)*, pp. 123–128, IEEE, 2015.

[59] B. B. Mandelbrot and B. B. Mandelbrot, *The fractal geometry of nature*, vol. 1. WH freeman New York, 1982.

[60] Y. Fisher, "Fractal image compression," *Fractals*, vol. 2, no. 03, pp. 347–361, 1994.

[61] J. E. Hutchinson, "Fractals and self similarity," *Indiana University Mathematics Journal*, vol. 30, no. 5, pp. 713–747, 1981.

[62] R. F. Haines, *The effects of video compression on acceptability of images for monitoring life sciences experiments*, vol. 3239. National Aeronautics and Space Administration, Office of Management . . . , 1992.

[63] G. Anderson, "Compression - gif vs png," 2000. http://ist.uwaterloo.ca/~anderson/images/GIFvsJPEG/compression_rates.html [Accessed 22.03.2023].

[64] D. Bryant, "WavPack About Page," 2023. https://www.wavpack.com [Accessed 22.03.2023].

[65] R. Raissi, "The theory behind mp3," *MP3'Tech*, 2002.

[66] "Sustainability of digital formats: Planning for library of congress collections," 2022. https://www.loc.gov/preservation/digital/formats/fdd/fdd000114.shtml [Accessed 23.03.2023].

[67] International Organization for Standardization/International Electrotechnical Commission, "Coding of moving pictures and associated audio for digital storage media at up to about 1.5 mbit/s," *ISO/IEC 11172*, 1993.

[68] E. P. Ruzanski, "Effects of mp3 encoding on the sounds of music," *IEEE Potentials*, vol. 25, no. 2, pp. 43–45, 2006.

[69] J. Moffitt, "Ogg vorbis—open, free audio—set your media free," *Linux journal*, vol. 2001, no. 81es, pp. 9–es, 2001.

[70] C. Montgomery, "Vorbis i specification," 2004. https://www.xiph.org/vorbis/doc/Vorbis_I_spec.html [Accessed 24.03.2023].

[71] E. J. Candès and M. B. Wakin, "An introduction to compressive sampling," *IEEE signal processing magazine*, vol. 25, no. 2, pp. 21–30, 2008.

[72] J. Coalson, "FLAC Format overview," 2022. https://xiph.org/flac/documentation_format_overview.html, [Accessed 31.07.2023].

[73] T. Robinson, "Shorten: Simple lossless and near-lossless waveform compression," 1994.

[74] T. Jehan, "Perceptual segment clustering for music description and time-axis redundancy cancellation.," in *ISMIR*, 2004.

[75] J. R. Weeks, *The shape of space*. CRC press, 2001.

[76] C. C. Adams, *The knot book*. American Mathematical Soc., 1994.

[77] M. Francl, "Stretching topology," *Nature chemistry*, vol. 1, no. 5, pp. 334–335, 2009.

[78] O. Anosova and V. Kurlin, "Introduction to periodic geometry and topology," *arXiv preprint arXiv:2103.02749*, 2021.

[79] T. Sousbie, "The persistent cosmic web and its filamentary structure–i. theory and implementation," *Monthly Notices of the Royal Astronomical Society*, vol. 414, no. 1, pp. 350–383, 2011.

[80] G. Carlsson, "Topology and data," *Bulletin of the American Mathematical Society*, vol. 46, no. 2, pp. 255–308, 2009.

[81] H. Edelsbrunner, D. Morozov, and V. Pascucci, "Persistence-sensitive simplification functions on 2-manifolds," in *Proceedings of the twenty-second annual symposium on Computational geometry*, pp. 127–134, 2006.

[82] J. A. Perea and J. Harer, "Sliding windows and persistence: An application of topological methods to signal analysis," *Foundations of Computational Mathematics*, vol. 15, pp. 799–838, 2015.

[83] S. Barbarossa and S. Sardellitti, "Topological signal processing: Making sense of data building on multiway relations," *IEEE Signal Processing Magazine*, vol. 37, no. 6, pp. 174–183, 2020.

[84] R. B. Gabrielsson, B. J. Nelson, A. Dwaraknath, and P. Skraba, "A topology layer for machine learning," in *International Conference on Artificial Intelligence and Statistics*, pp. 1553–1563, PMLR, 2020.

[85] S. Zeng, F. Graf, C. Hofer, and R. Kwitt, "Topological attention for time series forecasting," *Advances in Neural Information Processing Systems*, vol. 34, pp. 24871–24882, 2021.

[86] H. Edelsbrunner and J. L. Harer, *Computational topology: an introduction*. American Mathematical Society, 2022.

[87] Edelsbrunner, Harer, and Zomorodian, "Hierarchical morse—smale complexes for piecewise linear 2-manifolds," *Discrete & Computational Geometry*, vol. 30, pp. 87–107, 2003.

[88] Edelsbrunner, Letscher, and Zomorodian, "Topological persistence and simplification," *Discrete & Computational Geometry*, vol. 28, pp. 511–533, 2002.

[89] E. Munch, "A user's guide to topological data analysis," *Journal of Learning Analytics*, vol. 4, no. 2, pp. 47–61, 2017.

[90] P. Kanerva, "Hyperdimensional computing: An introduction to computing in distributed representation with high-dimensional random vectors," *Cognitive computation*, vol. 1, pp. 139–159, 2009.

[91] M. Hersche, M. Zeqiri, L. Benini, A. Sebastian, and A. Rahimi, "A neuro-vector-symbolic architecture for solving raven's progressive matrices," *Nature Machine Intelligence*, vol. 5, no. 4, pp. 363–375, 2023.

[92] G. James, D. Witten, T. Hastie, and R. Tibshirani, *An introduction to statistical learning*, vol. 112. Springer, 2013.

[93] D. L. Donoho *et al.*, "High-dimensional data analysis: The curses and blessings of dimensionality," *AMS math challenges lecture*, vol. 1, no. 2000, p. 32, 2000.

[94] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," *Advances in neural information processing systems*, vol. 26, 2013.

[95] F. Takens, "Detecting strange attractors in turbulence," in *Dynamical Systems and Turbulence, Warwick 1980: proceedings of a symposium held at the University of Warwick 1979/80*, pp. 366–381, Springer, 2006.

[96] C. R. Shalizi, "Methods and techniques of complex systems science: An overview," 2006.

[97] P. Skraba, V. De Silva, and M. Vejdemo-Johansson, "Topological analysis of recurrent systems," in *NIPS 2012 Workshop on Algebraic Topology and Machine Learning, December 8th, Lake Tahoe, Nevada*, pp. 1–5, 2012.

[98] R. M. Aarts, "Zak transform," 2023. https://mathworld.wolfram.com/ZakTransform.html [Accessed 22.05.2023].

[99] G. Kreitz and F. Niemela, "Spotify–large scale, low latency, p2p music-on-demand streaming," in *2010 IEEE Tenth International Conference on Peer-to-Peer Computing (P2P)*, pp. 1–10, IEEE, 2010.

[100] A. Singha, ""Future Internet" course at the ETH Zurich," 2021.

[101] H. G. Musmann, "Genesis of the mp3 audio coding standard," *IEEE Transactions on Consumer Electronics*, vol. 52, no. 3, pp. 1043–1049, 2006.

[102] M. Stirner, "Weitere verlustfreie kompression von mp3-dateien." 2012.

[103] P. Lindstrom and M. Isenburg, "Fast and efficient compression of floating-point data," *IEEE transactions on visualization and computer graphics*, vol. 12, no. 5, pp. 1245–1250, 2006.

[104] R. A. Finkel and J. L. Bentley, "Quad trees a data structure for retrieval on composite keys," *Acta informatica*, vol. 4, pp. 1–9, 1974.

[105] S. Lloyd, "Least squares quantization in pcm," *IEEE transactions on information theory*, vol. 28, no. 2, pp. 129–137, 1982.

[106] J. M. Phillips, ""Data Mining" course, University of Utah," 2013. https://users.cs.utah.edu/~jeffp/teaching/cs5955/L10-kmeans.pdf [Accessed 18.08.2023].

[107] J. Chung, C. Gulcehre, K. Cho, and Y. Bengio, "Empirical evaluation of gated recurrent neural networks on sequence modeling," *arXiv preprint arXiv:1412.3555*, 2014.

[108] A. Karpathy, "The unreasonable effectiveness of recurrent neural networks," 2015. http://karpathy.github.io/2015/05/21/rnn-effectiveness/ [Accessed 18.08.2023].

[109] S. Hochreiter, Y. Bengio, P. Frasconi, J. Schmidhuber, *et al.*, "Gradient flow in recurrent nets: the difficulty of learning long-term dependencies," 2001.

[110] H. Jaeger and H. Haas, "Harnessing nonlinearity: Predicting chaotic systems and saving energy in wireless communication," *science*, vol. 304, no. 5667, pp. 78–80, 2004.

[111] D. J. Gauthier, E. Bollt, A. Griffith, and W. A. Barbosa, "Next generation reservoir computing," *Nature communications*, vol. 12, no. 1, p. 5564, 2021.

[112] D. Patel and E. Ott, "Using machine learning to anticipate tipping points and extrapolate to post-tipping dynamics of non-stationary dynamical systems," *Chaos: An Interdisciplinary Journal of Nonlinear Science*, vol. 33, no. 2, 2023.

[113] G. Holzmann, "Reservoir computing: a powerful black-box framework for nonlinear audio processing," in *International Conference on Digital Audio Effects (DAFx)*, 2009.

[114] Y. Ke, D. Hoiem, and R. Sukthankar, "Computer vision for music identification," in *2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'05)*, vol. 1, pp. 597–604, IEEE, 2005.

[115] A. Wang *et al.*, "An industrial strength audio search algorithm.," in *Ismir*, vol. 2003, pp. 7–13, Washington, DC, 2003.

Declaration of Authorship

"I hereby declare,

- that I have written this thesis independently,
- that I have written the thesis using only the aids specified in the index;
- that all parts of the thesis produced with the help of aids have been precisely declared;
- that I have mentioned all sources used and cited them correctly according to established academic citation rules;
- that I have acquired all immaterial rights to any materials I may have used, such as images or graphics, or that these materials were created by me;
- that the topic, the thesis or parts of it have not already been the object of any work or examination of another course, unless this has been expressly agreed with the faculty member in advance and is stated as such in the thesis;
- that I am aware of the legal provisions regarding the publication and dissemination of parts or the entire thesis and that I comply with them accordingly;
- that I am aware that my thesis can be electronically checked for plagiarism and for third-party authorship of human or technical origin and that I hereby grant the University of St.Gallen the copyright according to the Examination Regulations as far as it is necessary for the administrative actions;
- that I am aware that the University will prosecute a violation of this Declaration of Authorship and that disciplinary as well as criminal consequences may result, which may lead to expulsion from the University or to the withdrawal of my title."

By submitting this thesis, I confirm through my conclusive action that I am submitting the Declaration of Authorship, that I have read and understood it, and that it is true.