A Probabilistic Graphical Model Based Data Compression Architecture for Gaussian Sources

by

Wai Lok Lai

S.B. in Mathematics, M.I.T. (2016)
S.B. in Computer Science and Engineering, M.I.T. (2016)

Submitted to the
Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2016

Author .....
Department of Electrical Engineering and Computer Science
September 6, 2016

Certified by .....
Gregory W. Wornell
Sumitomo Professor of Engineering
Thesis Supervisor

Accepted by .....
Christopher J. Terman
Chairman, Masters of Engineering Thesis Committee

A Probabilistic Graphical Model Based Data Compression Architecture for Gaussian Sources

by

Wai Lok Lai

Submitted to the
Department of Electrical Engineering and Computer Science
on September 6, 2016, in partial fulfillment of the requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

Data is compressible because of inherent redundancies in the data, which can be mathematically expressed as correlation structures. A data compression algorithm uses the knowledge of these structures to map the original data into a different encoding. The two aspects of data compression, *source modeling*, ie. using knowledge about the source, and *coding*, ie. assigning an output sequence of symbols to each input, are not inherently related, but most existing algorithms mix the two and treat the two as one.

This work builds on recent research on model-code separation compression architectures and extends this concept into the domain of lossy compression of continuous sources, in particular, Gaussian sources. To our knowledge, this is the first attempt at using *sparse linear coding* and *continuous-discrete hybrid belief propagation decoding* for compressing continuous sources.

With the flexibility afforded by the modularity of the architecture, we show that the proposed system is free from many inadequacies of existing algorithms, at the same time achieving competitive compression rates. Moreover, the modularity allows for many architectural extensions, with capabilities unimaginable for existing algorithms, including refining of source model after compression, robustness to data corruption, seamless interface with source model parameter learning, and joint homomorphic encryption-compression.

This work, meant to be an exploration in a new direction in data compression, is at the intersection of Electrical Engineering and Computer Science, tying together the disciplines of information theory, digital communication, data compression, machine learning, and cryptography.

Thesis Supervisor: Gregory W. Wornell
Title: Sumitomo Professor of Engineering

Acknowledgments

I would like to express my gratitude for the many people without whose help, direct or indirect, I would not have been able to complete this work.

First, I would like to thank Prof. Gregory Wornell for introducing me to the world of probabilistic inference through the class sequence 6.437-6.438, from which I learned to refine my understanding of probability and apply it to the problem of statistical inference. Prof. Wornell has been extremely patient when progress was slow, pointing me to useful resources, shedding light onto problems by helping me think in a different way. I feel fortunate to have the opportunity to be under his guidance, to have drawn from the breadth of his knowledge.

I would also like to thank Joshua Lee, who worked with me extensively on this project. The initial idea and the original MATLAB implementation of a low-density quantizer was due to him. Throughout this project, he has given me many crash courses in various topics in Electrical Engineering, which, coming from a Mathematical and Computer Science background, I lack, especially for a project fundamentally tied to concepts in Electrical Engineering.

This work has been inspired by Dr. Ying-Zong Huang, whose 2015 PhD dissertation details the idea and the implementation of a model-code separation architecture for compressing discrete sources. Ying-Zong communicated to Joshua and I valuable experiences he had with his system, forming the foundation of this work.

During the exploratory phase of this work, Joshua and I had the chance to speak with Dr. Ulugbek Kamilov, who researched in a related topic of message-passing de-quantization with respect to compressed sensing. This conversation impacted the design of the quantization structure presented in this work.

Next, I would like to express my appreciation for Prof. Katrina LaCurts and Shreya Saxena, who first introduced me to the concepts of channel coding, linear codes, and the Lempel-Ziv algorithm. This foundational knowledge was very useful for connecting the dots between channel coding and data compression.

I am thankful for the SIA community, especially for the support of Tricia and Giovanni, who helped me with various issues during my time at SIA.

With the mathematically heavy nature of this work, I would like to thank Martin Butzen, William Carroll, James Owens, Matthew O'Keefe, and Joseph Maggio for endowing me with a mathematical and a probabilistic understanding, and without whose help coming to, let alone graduating from, MIT would have been a mere dream.

Finally, I reserve my utmost gratitude for my parents, for being my first teachers, impressing on me the love of mathematical reasoning, the thirst for knowledge, and the determination to pursue my deepest dreams.

✠ $\mathcal{AMDG}$ ✠

Contents

List of Figures

List of Tables

List of Algorithms

1

Introduction

Data compression is a problem of fundamental importance at the intersection of electrical engineering and computer science. From the celebrated universal Lempel-Ziv algorithm which powers the ZIP compressor, to data specific compressors like JPEG, compression is essential to data transmission and storage.

Given the decades of research into data compression, it may be surprising to hear that existing algorithms have significant fundamental and theoretical inadequacies that inherently constrain its continued development. In this thesis, we will explore a new architecture for lossy compression that addresses these inadequacies, while at the same time giving competitive performance compared to existing algorithms.

1.1 Illustrative Example

To begin, let us consider an illustrative example. Suppose in a scientific experiment, 500 particles are fired one by one from a particle gun, and their terminal positions are thought to follow a Gaussian distribution with a known mean. The position of each particle is thought to be independent from one another, and standard deviation of the Gaussian distribution is thought to be 1. We would like to store the positions of the particles as data for future use.

Given that a Gaussian distribution is continuous that can take on any possible real value, it is impossible to consistently represent the positions of the particles to arbitrary precision with a finite number of bits. Therefore, some *distortion* is inevitably introduced between the actual position and the position value that we store.

One way to store the positions of the particles will be to store them as decimals, with precision up to a certain number of significant figures. However, with the number of bits we need to represent the truncated decimal numbers, we can do better by noting that in a Gaussian distribution, values closer to the mean are more likely to occur, and we can improve the *expected distortion* by recording values closer to the mean with a higher precision, rather than the uniform 10^{-x} precision implicitly imposed by a decimal representation or the 2^{-x} imposed by a floating point number.

The well known *Lloyd-Max algorithm* gives the optimum expected distortion for a given fixed number of bits used to represent a single value, provided that the distribution is known [26]. The `Lloyd-Max` algorithm returns a set of boundary values (on the real line $\mathbb{R}$) and representative values for each *bin* (defined by the boundaries), with the representative values of the bins being used as reconstructed values. Figure 1.1 illustrates the `Lloyd-Max` algorithm.

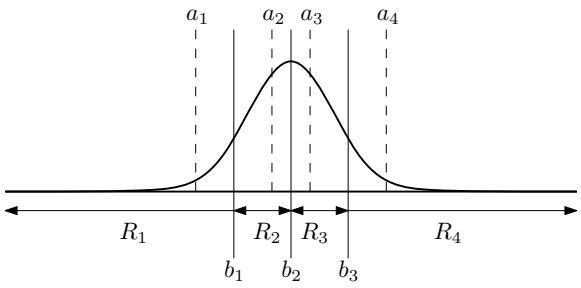


Figure 1.1: Given a probability distribution and a quantization level β ($\beta = 4$ in this example), the `Lloyd-Max` algorithm provides a set of boundary values b_i which separate the real line $\mathbb{R}$ into β slabs R_i, also called bins. If a data point falls in bin R_i, it is represented as bin i at compression time, and reconstructed as a_i at decompression time.

Given the optimum guarantee, it seems tempting to use the `Lloyd-Max` algorithm to store the observed positions of our particles. However, if we were to find out later that the underlying Gaussian distribution of the positions can be better modeled with a standard deviation of 2, instead of 1 as we previously thought, the optimum guarantee of the `Lloyd-Max` algorithm breaks down.

Moreover, if it turns out that the positions of the particles are not independent at all, that they can be modeled as a Markov chain with each position correlated to its immediate neighbors, we will be stuck with an extremely inefficient and imprecise way of representing the data. It is not clear how one would modify the existing structure to accommodate this new discovery, if possible at all. If luckily we still have access to the original position data, we will have to redesign a new way of representing and storing the values. If we do not, then not much can be done to improve the representation.

This example illustrates a very common design paradigm of using specific assumptions about the data (the *source model*) to optimize for the compression rate. It also exemplifies how dependent the compression representation is on the source model, that even a small change in the parameters will lead to significant incompatibilities. In this respect, most existing practical compression architectures suffer from the very same problem faced by our simple example.

1.2 Motivation

To better discuss existing compression architectures, let us first categorize them into four categories.

1.2.1 Data Compression Overview

Specificity \ Accuracy	Model specific	Model agnostic
Lossless	(I) Entropy coding *eg. Huffman coding*	(II) Universal entropy coding *eg. ZIP*
Lossy	(III) Rate-distortion coding *eg. JPEG, Lloyd-Max*	(IV) Universal Rate-distortion coding

Table 1.2: Classification of existing data compression algorithms into four categories, distinguished by their model specificity and reconstruction accuracy. In this thesis, we compare the performance of our architecture mainly to Domain III systems (shaded above).

In Table 1.2, the columns refer to whether the algorithm requires the form of the data to be specified at the time when the algorithm is designed.

- Model specific compression algorithms are designed around knowledge about the data type that they are built to compress. For example, JPEG is designed with knowledge that the data being compressed is a 2-dimensional array of color values, and MPEG is designed with knowledge that the data is a set of frames of images.

- Model agnostic compression algorithms, on the other hand, does not use prior knowledge about the source; instead, it learns and adapts to the source model from the data at compression time. For example, the Lempel-Ziv algorithm does not make any assumptions about the format of the data: at compress time it dynamically builds a dictionary of repeating symbols to extract 1-dimensional patterns from the data itself.

The rows refer to the accuracy of the reconstruction with respect to the original source.

- Lossless compression expects a perfect reconstruction of the original data. For example, an unzipped ZIP folder contains the exact same content as the original files. Lossless compression is usually applied to digital content, eg. documents or texts, which cannot afford any loss of original content.

- Lossy compression, on the other hand, allows for distortions in the reconstruction. For example, an image reconstructed from JPEG compression is a close approximate of the original image. Lossy compression is usually applied to analog content, eg. music, pictures, videos, that may be too costly and unnecessary, or even impossible, to digitally represent perfectly.

Domain I systems are know as entropy codes, which arose directly from the study of information theory and digital communication, which began with Claude Shannon's groundbreaking research in 1948 [35]. While not used practically in and of themselves, Domain I systems form the basis of most practical compression systems,

which lie in Domain II or Domain III with our categorization. In particular, most Domain II systems (universal entropy coding) are extensions of Domain I systems with adaptivity, and most Domain III systems (rate-distortion coding) are extensions of Domain I systems with preprocessing and quantization.

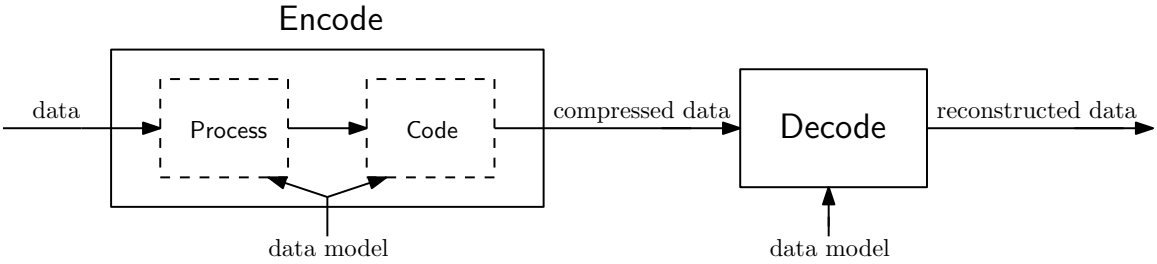


Figure 1.3: Most Domain II and Domain III systems process the data to extract compression gains, before using an entropy coder to remove any remaining memoryless structures.

As noted by Huang [21], the task of *coding* is thus very similar with most existing systems, for the paradigm of most systems is to preprocess the data to achieve compression with knowledge of the structure before using an entropy coder to clean up any remaining gains, illustrated by Figure 1.3. Inherent to this paradigm, however, is the fundamental problem of entanglement, as we shall now discuss.

1.2.2 Examples of Joint Design

A compression algorithm that follows the preprocess-code design pattern is the JPEG compressor, [37]. The JPEG compressor is a Domain III system (rate-distortion coding) that uses the Discrete Cosine Transform (DCT, the real portion of the Discrete Fourier Transform), quantization, and entropy coding. The model assumption is that still images decorrelate well in the DCT frequency domain. The quantization step discards the high frequency data, since the human eye is insensitive to high frequency components. Finally, the symbols are coded with a Huffman code. On the decoding side, the decoder reverses this procedure by first decoding the Huffman code, then piecing the data back together in the frequency domain, before applying the inverse DCT to reconstruct the image.

Unexpectedly, the popular ZIP compressor, a representative of a Domain II system (universal entropy coding), also follows this preprocess-code design pattern. The DEFLATE algorithm [9], which powers the ZIP compressor, involves using the Lempel-Ziv compressor followed by Huffman coding. Lempel-Ziv learns and adapts to the data source at compression time by building a dynamic dictionary to represent repeating symbols, and the residual entropy is again squeezed out by Huffman coding.

We note that the final step of both JPEG and ZIP is Huffman coding, a Domain I system being embedded within the larger architecture to clean up any remaining memoryless structures, with the majority of the compression gains already achieved by the preprocessing components.

1.2.3 Structural Challenges

As illustrated by the two examples above, the Encode procedure for most existing architectures can be broken down into two components: a data specific Process component, and an entropy coding Code component (Figure 1.3). Knowledge of the source model is required to design both Process and Code. However, there seem to be a lack of distinction of the roles of either components: in terms of fitting the source and extracting compression, it is not clear what one does that the other does not, and it is hard to draw a line between the two, for both are involved in *source modeling*, ie. using knowledge of the source, and *coding*, ie. assigning a compressed output for a given data input.

For JPEG, the source model explicitly assumes that images decorrelate well in the DCT frequency domain. On the other hand, while ZIP is a universal compressor that does not require an explicit source model at compression time, the Lempel-Ziv component implicitly assumes a 1-dimensional pattern in the data source, and practically will only compress well when this implicit model is satisfied. While it can still achieve some compression for non-1-dimensional data like photos, its performance is much better for sources like plain text, which satisfies its implicit model.

With this definition of source modeling and coding, we can see that most existing systems can be labeled as *joint model-code architectures*, which consider source modeling and coding together as a whole when designing the algorithm. Such algorithms, by their nature, entangle *model* and *code* in both the Process and Code components. This entanglement of model and code is usually justified by the opportunities for clever optimizations in the compression rate. However, this inherently constraints the power of the compression algorithm in terms of readiness for change and in terms of generalization to other (perhaps related) data sources.

1.2.4 Rigidity of Joint Design

We now illustrate the design rigidity of the joint model-code paradigm with JPEG as an example. While images decorrelate well in the DCT frequency domain, it was later discovered that they decorrelate better in the Wavelet frequency domain [2], a decision that has been incorporated into the newer JPEG2000 standard. This improvement of knowledge of the source model, however, cannot be incorporated into the existing JPEG algorithm as the DCT and the Wavelet Transform are not compatible. Like our illustrative example in Section 1.1, a refinement of the source model has led to the need for an entirely new algorithm. In this case, the new algorithm JPEG2000 sees a low adoption rate, since it is fundamentally incompatible with the original JPEG standard.

Furthermore, JPEG was originally intended for natural photographs, as suggested by its full name (Joint Photographic Experts Group). However, over time, as new types of images became more common, its function has been overloaded with users compressing artificial graphics with JPEG (Figure 1.4). Artificial graphics have a very different underlying model from natural photographs, but JPEG, even JPEG2000, has no way to handle this source model assumption violation, which tends to occur when

new types of data becomes available and there is not an algorithm outstanding to compress it. The result of this model mismatch is the infamous JPEG pixelation, where the wavy patterns of the Discrete Cosine basis become noticeable. To solve this problem, another entirely new algorithm mutually incompatible with JPEG will need to be designed for artificial graphics. In general, with the joint model-code paradigm, a new algorithm will need to be designed whenever a new class of data emerges.

Figure 1.4: Using JPEG on digitally created images is a model mismatch. The left image represent the original image, stored in the PNG format, and the right represent one compressed by JPEG. While natural photos decorrelate well in the DCT frequency domain, for digitally created images with sharp edges, this model assumption fails, leading to JPEG's infamous wavy pattern.

The rigidity of joint model-code design discussed above motivates us to consider a compression architecture with model-code separation, where there is a clear separation between using knowledge about the data source (ie. source modeling) and assigning output symbols (ie. coding).

1.3 Previous Work

To design a data compression algorithm, we draw from the work of the information theory and digital communications communities, the foundations of which was laid by Shannon's landmark 1948 paper [35]. The important concepts of entropy and linear coding are essential to our work. Particularly relevant to our work, however, would be the branch of information theory called *rate-distortion theory*, which investigates lossy compression and its fundamental limits. Rate-distortion theory is a well-developed theory that we shall describe in Section 2.2.4, which we shall reference frequently in this thesis.

Following Shannon's initial developments, Gallager developed Low-Denisity Parity-Check (LDPC) codes in his doctoral dissertation in 1960 [11]. LDPC codes, a type of linear code that will form the backbone of our compression algorithm, are discussed in more details in Section 2.2.3. LDPC codes were soon forgotten after its publication, as its computational complexity for exact decoding was too high for computer systems in the 1960s, and its development remained dormant until 1996.

During this time, Pearl introduced Belief Propagation (BP) in 1982 [30]. Also known as sum-product message passing, BP is an inferential algorithm for marginalization on probabilistic graphical models, as we shall introduce in Section 2.3.4.2. While originally developed for tree graphs, the machine learning community has refined the tool over the years as an approximation algorithm for general graphs. As we shall see, approximate BP is a tool key to decoding LDPC codes efficiently.

More recently, in 2003 Caire et al. researched in using LDPC codes for lossless compression of memoryless discrete sources [7], using belief propagation and *doping*

(discussed in Section 3.4) as the decoding mechanism. The researchers extended the algorithm to a universal lossless compressor by preprocessing the data using the Burrows-Wheeler transform [6], which helps the algorithm adapt to and decorrelate the data source at compression time. This preprocessing by itself does not achieve any compression, but allows LDPC codes to compress the processed data more efficiently.

This universal lossless compressor, by its nature, does not allow for incorporation of source model information. More relevant to our work would be Garcia-Frias and Zhong's 2003 paper [13], which experimented with compressing binary Markov chains with LDPC codes, the first instance of some conception of a model-code separation architecture. Then, Schonberg et al.'s 2006 paper [32] discussed the idea of a source model (in the sense of knowledge about the source, as we defined in Section 1.2.3) and experimented with a 2D Markov model on bi-level images encrypted with a one-time pad.

LDPC codes for lossy compression for binary sources was discussed in 2007 by Braunstein et al. [5], who used LDPC codes over $\mathbb{Z}_p$ (instead of the well established binary LDPC codes) to achieve results close to the rate-distortion bound for binary $\textbf{Bern}\left(\frac{1}{2}\right)$ sources.

The recent work of Huang and Wornell in 2014 [22, 21] brings together these concepts by proposing a general compression algorithm with a model-code separation architecture for discrete sources. The core component is a lossless binary compressor powered by LDPC encoding and BP decoding, with a flexible architecture to allow for lossless non-binary discrete sources (by incorporating a translator component) and for lossy binary compression (by incorporating a quantizer component).

1.4 Contributions

In this thesis, we expand on the work of Huang and Wornell [22, 21], who proposed the model-code separation architecture for data compression for discrete systems that addressed the concerns raised in Section 1.2.3. We shall apply the fundamental concept of model-code separation and belief-propagation decoding proposed by Huang, while exploring the results of extending it into the continuous domain. In particular, we shall focus our attention on compressing Gaussian sources, which are known to be the most difficult to compress due to the fact that the Gaussian distribution is the maximum entropy distribution for any given mean and variance [8]. Figure 1.5 illustrates our proposed architecture. To our knowledge, a lossy compressor for continuous sources based on sparse codes has not been attempted.

One important challenge faced by our architecture is the domain over which the inputs and the outputs are defined. The input source (model component) operates in the continuous domain, while the output bit stream (code component) operates in the discrete domain. To resolve the issue, we connect the two components by a quantizer, which bridges the continuous and the discrete domains into one structure while keeping the modularity of the model and code components. As we mentioned, Belief Propagation is the method of choice for decoding, and to our knowledge, this is the first attempt in using *continuous-discrete hybrid belief propagation* in the context

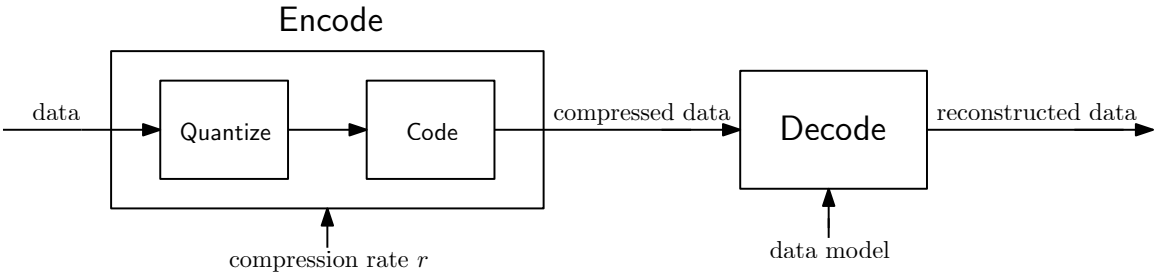


Figure 1.5: The block diagram showing information flow in our proposed architecture for lossy compression on continuous sources. Note that by design, the Encode procedure is model-free, ie. does not require information about the underlying source, while the Decode procedure applies its knowledge about the source model to reconstruct the data.

of lossy compression.

This architecture is extremely modular and flexible. Component-wise optimizations can be done independently of other components, such as refining the source model or searching for better codes. In addition to being free of the rigidity and inadequacies discussed in Section 1.2.3, our architecture also provides competitive performance against existing algorithms, as we shall show in Chapter 7.

Moreover, as we shall demonstrate in Chapter 9, the modularity of the architecture makes it easily extensible. Components with different self-contained add-on functionalities can be incorporated into the system, and different components can be modified to incorporated new features without changing the structure of the compression architecture, making it ready to solve future problems that have not arisen at the time of the design of the architecture.

1.5 Thesis Organization

Our discussion follows a natural development of the architecture. In particular, the thesis is organized as follows:

In Chapter 2 (Background), we first introduce important concepts and tools, including Gaussian distributions, linear coding, and probabilistic graphical models.

Next, in Chapter 3 (Compressor Architecture Overview), we give a self-contained overview of our proposed compression architecture, pointing out the different modular components of the architecture, including coding, quantization, and source modeling.

Then, in Chapters 4 (Code and Translator Structure), 5 (Quantizer Structure), and 6 (Source Modeling), we discuss the implementation details of the code, quantizer, and source model components of the architecture respectively.

After presenting the algorithm, in Chapter 7 (Compression Performance) we discuss our performance in light of the theoretical lower bounds as well as the bounds of other algorithms.

Chapter 8 (Discussion and Analysis) analyzes our design choices and how those decisions affect performance.

Chapter 9 (Realistic Applications and Extensions) presents real world applications of our architecture and proposes extensions to our system that have practical impacts.

Finally, in Chapter 10 (Conclusion), we summarize our work and point out potential ideas that call for further research.

1.6 Notation

While there are many commonly used sets of notations within the different communities of information theory, digital communication, and machine learning, for the sake of unity, clarity, and conciseness, we have chosen the following set of notation for this thesis:

We use lower case x to denote a scalar or vector variable. If necessary, we clarify the dimension of a vector with superscripts x^n or with $x \in \mathbb{R}^n$.

We use upper case A to denote a matrix. We clarify its dimension with $A \in \mathbb{R}^{m \times n}$.

We use upright x to denote a constant, either a scalar or a vector.

We use hat $\hat{x}$ to denote an estimated value.

We use asterisk x^* to denote an optimal value of a variable, judged with respect to some metric.

For vectors and matrices, we use subscripts x_i to denote the i^{th} entry of the vector x or A_{ij} to denote the $(i,j)^{\text{th}}$ entry of the matrix A.

For iterative algorithms, we use super script $m_i^{(\tau)}$ to denote the value of the variable m_i at iteration τ.

We use italicized sans serif x to denote a random variable, with dimensionality clarified as necessary.

We use boldface $\mathbf{N}(\cdot)$ to denote a distribution.

We use script $\mathscr{H}$ to denote a set.

We use $|\mathscr{H}|$ to denote the cardinality of a set.

We use $\|x\|_p$ to denote the L^p norm.

We use $\mathbb{R}$ to denote the real numbers, $\mathbb{R}_+$ to denote the positive real numbers, and $\mathbb{Z}_n$ to denote the integers mod n.

We use italics $f(\cdot)$ to denote a function. In particular, we use $\mathcal{N}(x; \mu, \sigma^2)$ to denote the probability density function (pdf) of a Gaussian with mean μ and variance σ^2, which is a function in x.

We use double strike $\mathbb{H}(\cdot)$ to denote a statistical function of random variables. With $\mathbb{E}(\cdot)$ denoting the expected value of random variable, subscripts $\mathbb{E}_{\mathsf{s}}(\cdot)$ clarify the variable with respect to which the expectation is taken.

We use calligraphic $\mathcal{G} = (\mathcal{V}; \mathcal{E})$ to denote a graph with vertices $\mathcal{V}$ and edges $\mathcal{E}$, and $\mathcal{G} = (\mathcal{V}_1, \ldots, \mathcal{V}_n; \mathcal{E})$ to denote an n-partite graph. We use $\mathcal{N}_i^{\mathcal{V}_k}$ to denote the set of vertices in $\mathcal{V}_k$ that are neighbors of vertex i. The super script will be dropped when unambiguous.

We use sans serif Decode to denote a procedure.

We use monospace `Lloyd-Max` to denote an established system or scheme.

For summations, to simplify notation, we only write one summation symbol even if the sum is taken over multiple variables. Moreover, the summation is taken all possible

values of the variables. Eg. if $\mathscr{A} = \{1, 2\}$ with $x_1 \in \{1, \ldots, m\}$ and $x_2 \in \{1, \ldots, n\}$, we write

$$\sum_{x_{\mathscr{A}}} f(x_{\mathscr{A}}) := \sum_{x_1=1}^{m} \sum_{x_2=1}^{n} f(x_1, x_2)$$

For integrals, we adopt the same simplification. In addition, we write the integrated variables as a subscript under the integral sign, with the integral understood to be taken over the whole $\mathbb{R}^n$ space unless otherwise specified. Eg. if $\mathscr{A} = \{1, 2\}$, we write

$$\int_{x_{\mathscr{A}}} f(x_{\mathscr{A}}) := \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} f(x_1, x_2) \, \mathrm{d}x_2 \, \mathrm{d}x_1$$

2

Background

An architecture that compresses a sequence of continuous variables into a finite number of bits unavoidably faces a three-fold challenge: *source modeling*, *quantization*, and *encoding*. In this chapter, we give an overview of the background, tools, and methodologies required for such a system before we present our compression architecture.

Section 2.1 reviews the definition and the properties of a joint Gaussian distribution, the class of sources we focus on in this thesis. Section 2.2 presents classical coding theory as a method to reframe the data compression problem, as well as rate-distortion theory as the baseline with which we compare our compression performance. Section 2.3 introduces probabilistic graphical models, the main tool that we use to construct the architecture.

2.1 Gaussian Distributions

2.1.1 Univariate Gaussian Random Variables

The following is the familiar definition of a univariate Gaussian (normal) random variable:

Definition 2.1. (Univariate Gaussian). A random variable $\mathsf{s} \in \mathbb{R}$ is called Gaussian if its probability density function (pdf) $p_{\mathsf{s}}(\cdot)$ can be written as

$$p_{\mathsf{s}}(s) = \frac{1}{\sqrt{2\pi\sigma^2}} \cdot \exp\left\{-\frac{1}{2\sigma^2}(s-\mu)^2\right\} \tag{2.2}$$

for some $\mu \in \mathbb{R}$, known as the *mean*, and $\sigma^2 \in \mathbb{R}_+$, known as the *variance*.

Following standard notation, we write $\mathsf{s} \sim \mathbf{N}(\mu, \sigma^2)$ to denote that s is a Gaussian random variable with mean μ and variance σ^2. In particular, we call $\mathbf{N}(0,1)$ the *standard normal distribution*. We use $p_{\mathsf{s}}(s) = \mathcal{N}(s; \mu, \sigma^2)$ to denote the probability density function in Equation 2.2 itself.

2.1.2 Jointly Gaussian Random Variables

Definition 2.3. (Multivariate Gaussian). A random vector $\boldsymbol{s}^n \in \mathbb{R}^n$ is *jointly Gaussian* if its joint distribution $p_{\boldsymbol{s}^n}(\cdot)$ can be written as

$$p_{\boldsymbol{s}^n}(s^n) = \left((2\pi)^{-\frac{n}{2}}|\Sigma|^{-\frac{1}{2}}\right) \cdot \exp\left\{-\frac{1}{2}(s-\mu)^T\Sigma^{-1}(s-\mu)\right\} \tag{2.4}$$

for some $\mu \in \mathbb{R}^n$, known as the *mean vector*, and some positive definite matrix $\Sigma \in \mathbb{R}^{n\times n}$, known as the *covariance matrix*. $|\cdot|$ represents the determinant of a matrix. This parametrization of the joint Gaussian distribution is know as the *covariance form*.

Through a change of variables, we can obtain an alternate parametrization, known as the *information form*, of the jointly Gaussian distribution. Letting $\Lambda = \Sigma^{-1}$ and $\eta = \Lambda\mu$, a simple substitution gives

$$p_{\boldsymbol{s}}(s) = \left((2\pi)^{-\frac{n}{2}}|\Lambda|^{\frac{1}{2}}\exp\left\{-\frac{1}{2}\eta^T\Lambda^{-1}\eta\right\}\right) \cdot \exp\left\{-\frac{1}{2}s^T\Lambda s + \eta^T s\right\} \tag{2.5}$$

where η is known as the *potential vector* and Λ is known as the *precision matrix*.

Following standard notation, we write $\boldsymbol{s}^n \sim \mathbf{N}(\mu, \Sigma)$ or $\boldsymbol{s}^n \sim \mathbf{N}^{-1}(\eta, \Lambda)$ to denote that $\boldsymbol{s}$ is jointly Gaussian with the corresponding parameters. We use $p_{\boldsymbol{s}^n}(x^n) = \mathcal{N}(s^n; \mu, \Sigma)$ or $p_{\boldsymbol{s}^n}(x^n) = \mathcal{N}^{-1}(s^n; \eta, \Lambda)$ to denote the joint probability density function of a joint Gaussian distribution.

In the remainder of the thesis, we will be using both the covariance form and the information form of the joint Gaussian distribution, wherever convenient.

2.1.2.1 Properties of Joint Gaussians

We now present some important facts that are central to our development of Gaussian random variables [36]:

Fact 2.6. A random vector $\boldsymbol{s}$ is jointly Gaussian if and only if it can be expressed as a linear combination of independent and identically distributed (iid) univariate standard normal variables, ie. there exists some matrix A and some offset vector b such that $\boldsymbol{s} = A\boldsymbol{w} + b$, where $\boldsymbol{w} \sim \mathbf{N}(0, I)$ is the iid standard normal distribution.

Fact 2.7. A random vector $\boldsymbol{s}$ is jointly Gaussian if and only if every linear combination of the elements of $\boldsymbol{s}$ is a univariate Gaussian variable, ie. for any constant vector a, there exists scalars $\mu \in \mathbb{R}$ and $\sigma^2 \in \mathbb{R}_+$ such that $a^T\boldsymbol{s} \sim \mathbf{N}(\mu, \sigma^2)$.

Fact 2.8. (Conditionals of Joint Gaussians). The conditional distribution of any subset of a set of jointly Gaussian random variable, conditioned on any other subset of the set, is also jointly Gaussian.

Fact 2.9. (Marginalization of Joint Gaussians). Any marginalization of a jointly Gaussian distribution is also a jointly Gaussian distribution, ie., if $\boldsymbol{s}$ is a set of jointly

Gaussian variables with mutually exclusive subsets s_1 and s_2 such that

$$\mathsf{s} = \begin{bmatrix} \mathsf{s}_1 \\ \mathsf{s}_2 \end{bmatrix} \sim \mathbf{N}^{-1}\left(\begin{bmatrix} \eta_1 \\ \eta_2 \end{bmatrix}, \begin{bmatrix} \Lambda_{11} & \Lambda_{12} \\ \Lambda_{21} & \Lambda_{22} \end{bmatrix} \right) \tag{2.10}$$

then

$$\mathsf{s}_1 \sim \mathbf{N}^{-1}(\eta', \Lambda') \tag{2.11}$$

where

$$\eta' = \eta_1 - \Lambda_{12}\Lambda_{22}^{-1}\eta_2 \tag{2.12}$$
$$\Lambda' = \Lambda_{11} - \Lambda_{12}\Lambda_{22}^{-1}\Lambda_{21} \tag{2.13}$$

Fact 2.14. (Pointwise Products of Gaussian pdf's). Consider a set of k joint Gaussian pdf's $\mathcal{N}^{-1}(s; \eta_i, \Lambda_i)$ for $i \in \{1, \ldots, k\}$, each of n dimensions. Their pointwise product is

$$\prod_{i=1}^{k} \mathcal{N}^{-1}(s; \eta_i, \Lambda_i) = \zeta \cdot \mathcal{N}^{-1}(s; \eta', \Lambda') \tag{2.15}$$

where

$$\eta' = \sum_{i=1}^{k} \eta_i \tag{2.16}$$
$$\Lambda' = \sum_{i=1}^{k} \Lambda_i \tag{2.17}$$

and the constant of proportionality ζ is

$$\zeta = (2\pi)^{-\frac{n}{2}(k-1)} \frac{\prod_{i=1}^{k} |\Lambda_i|^{\frac{1}{2}}}{|\Lambda'|^{\frac{1}{2}}} \exp\left\{ -\frac{1}{2}\left(\sum_{i=1}^{k} \eta_i^T \Lambda_i^{-1} \eta_i - (\eta')^T (\Lambda')^{-1} \eta' \right) \right\} \tag{2.18}$$

2.2 Coding Theory

The study of data processing and compression began with Claude Shannon, the pioneer of the fields of communication and information theory. In his original work [35], he investigated two types of compression systems: lossless compression (also known as source coding theory), described in Section 2.2.1, and lossy compression (also known as rate-distortion theory), described in Section 2.2.4.

Central to the development of both theories is the concept of a source model, a probabilistic description of the source s. The source model represents prior knowledge of the data being compressed, assuming that each instance s^n of the class of data s^n is drawn independently from the known joint probability density function $p_{\mathsf{s}^n}(\cdot)$.

2.2.1 Entropy and Coding

The entropy of a source $\boldsymbol{s}^n$ is the measure of the average information contained in a sample sequence $s^n \sim p_{\boldsymbol{s}^n}$. Entropy, thus, serves as a natural lower bound to the compression rate if sample sequences were to be consistently recovered without loss. We first present a fundamental theorem of coding theory, due to Shannon [35]:

Theorem 2.19. (Shannon's Source Coding Theorem). For an iid source $\boldsymbol{s}^n$, as $n \to \infty$,

(a) There exists a uniquely decodable code that can compress instances s^n into $n \cdot \mathbb{H}(\boldsymbol{s})$ bits without any loss of information, and
(b) There does not exist a uniquely decodable code that can compress instances s^n into less than $n \cdot \mathbb{H}(\boldsymbol{s})$ bits without any loss of information,

where the value

$$\mathbb{H}(\boldsymbol{s}) := \lim_{n\to\infty} \frac{1}{n}\mathbb{H}(\boldsymbol{s}^n) = \lim_{n\to\infty} \frac{1}{n}\mathbb{E}_{\boldsymbol{s}^n}(-\log_2(p_{\boldsymbol{s}^n}(s^n))) \tag{2.20}$$

is known as the *entropy rate* of $\boldsymbol{s}$.

2.2.2 Linear Codes

Shannon's development of information theory makes it clear that he treated communication and compression as closely related problems, even as duals of the same problem. After Shannon's original publication in 1948, many others have discovered that random sparse linear codes are amongst the best codes for digital communication. However, the compression side of the problem has not seen much development along these lines.

Definition 2.21. (Linear Code). A linear code is a code that encodes a length-$(n-k)$ source sequence into a length-n *codeword* by using k extra parity check bits to correct for potential corruption of codewords when sent over a channel. In particular, for a source alphabet $\mathscr{S}^{n-k} = \mathbb{Z}_q^{n-k}$, a linear code is defined by a linear transform

$$L : \mathscr{S}^{n-k} \to \mathscr{S}^n \tag{2.22}$$

Such a linear transform can be characterized by a *generator matrix* $G \in \mathbb{Z}_q^{(n-k)\times n}$ with

$$L(s) = G^T s \tag{2.23}$$

or equivalently, by the *parity check matrix* $H \in \mathbb{Z}_q^{k\times n}$ which satisfies

$$HG^T = 0 \tag{2.24}$$

Linear codes are widely used as *error correcting codes* because of the following well known fact on its optimality [8]:

Fact 2.25. Linear codes over finite field alphabets $\mathbb{Z}_q$ achieve capacity over symmetric discrete memoryless channels (DMC).

Linear encoding, as described above, amounts to multiplication by G^T, which has time complexity $O(n(n-k)) = O(n^2)$. Decoding, on the other hand, requires some more work. Letting $\mathscr{L} = \{v : v = G^T u\}$ with $\{u\}$ being the set of all source sequences, we note that the set of received codewords, with channel noise $w \in \mathscr{S}^n$, would be

$$\mathscr{L} + w = \{y : y = G^T u + w\} = \{y : Hy = Hw\} \tag{2.26}$$

Hence, decoding amounts to determining the maximum likelihood (ML) estimate

$$\hat{w}^* = \underset{\{\hat{w}: Hy = Hw\}}{\arg\max}\ p_w(w) \tag{2.27}$$

which has time complexity $O(n|\mathscr{S}|^k)$ [8], too large to be efficiently decoded.

Low-density parity-check (LDPC) codes are codes with parity check matrices H that have insignificant row and column weights, negligible compared to the growth of n. With iterative local decoding, for which good performance has been proven, decoding has time complexity $O(k|\mathscr{S}|^\rho \tau^*)$, where ρ is the maximum row weight and τ^* is the number of iterations till convergence, with $\tau^* = O(1)$ for fixed rates [24]. With ρ set to a constant, LDPC codes for channel coding have encoding complexity of $O(n^2)$ and decoding complexity $O(n)$.

2.2.3 LDPC Matrices for Compression

As previously noted, there is a strong duality between communication and compression. Identifying the channel noise in communication with the source distribution in compression, we get the dual of Fact 2.25:

Fact 2.28. Linear codes over finite field alphabets $\mathbb{Z}_q$ achieve entropy in lossless compression of discrete memoryless sources (DMS).

In this dual problem, compression amounts to a linear projection by the parity-check matrix $H \in \mathbb{Z}_q^{k \times n}$, with compressed data $x \in \mathscr{S}^k$ computed as

$$x = Hz \tag{2.29}$$

for source sequence $z \in \mathscr{S}^n$. Compression thus has time complexity $O(nk)$.

As in Section 2.2.2, computing the maximum likelihood (ML) $\hat{z}^*$ for a general parity check matrix H will have time complexity $O(n|\mathscr{S}|^k)$. However, if we resort to LDPC matrices with iterative local decoding, we can bring the decoding complexity down to $O(k|\mathscr{S}|^\rho \tau^*)$, while encoding will be $O(\rho k)$. It is important to note that the time complexity of both operations are thus both $O(n)$, linear in the length of the source sequence.

While Fact 2.28 only applies to discrete memoryless sources, [7] asserts the same result for general sources including sources with memory and non-stationary sources. Although it does not immediately follow that LDPC codes and ML decoding achieve the theoretical bound, we take this result with an optimistic note that recent research on much larger classes of sources with LDPC encoding yields mostly positive results [3].

2.2.4 Rate-Distortion Theory

When the size of the alphabet is too large, eg. infinite, or that too many bits are required to represent each possible value distinctly, we may be willing to sacrifice some accuracy in return for a much smaller compressed output. Rate-Distortion Theory, the study of this trade-off between rate and distortion, is central to our development of a lossy compression architecture.

Lossy compression, unlike lossless compression, is a many-to-one compression method, which means its reconstruction of the source will be imperfect. To judge the goodness of a reconstruction $\hat{s}$ of the original sequence s, therefore, a measure of distortion $\Delta(\hat{s}, s)$ is needed.

For discrete sources of a finite alphabet, a natural distortion measure is the Hamming Distortion:

$$\Delta_{\mathrm{H}}(\hat{s}^n, s^n) := \frac{1}{n}\sum_{i=1}^{n} \mathbb{1}\{\hat{s}_i \neq s_i\} \tag{2.30}$$

and for continuous sources, the common choice is the Mean Squared Error:

$$\Delta_{\mathrm{MSE}}(\hat{s}^n, s^n) := \frac{1}{n}\sum_{i=1}^{n} (\hat{s}_i - s_i)^2 \tag{2.31}$$

The problem of lossy data compression can thus be formulated as the following: For a source s with alphabet $\mathscr{S}$, and given a fixed distortion value δ, find the lowest achievable compression rate r such that the distortion between $s \in \mathscr{S}^n$ and $\hat{s} \in \mathscr{S}^n$ is at most δ, where $\hat{s}$ is reconstructed from a bit stream $x \in \mathbb{Z}_2^{nr}$.

The *rate-distortion function* $\mathbb{R}(\delta; \mathsf{s})$ provides a lower bound on the compression rate r as a function of δ.

Definition 2.32. (Rate-Distortion Function). The rate-distortion function, also known as the rate-distortion curve, is defined to be

$$\mathbb{R}(\delta; \mathsf{s}) := \lim_{n\to\infty} \frac{1}{n}\mathbb{R}(\delta; \mathsf{s}^n) = \lim_{n\to\infty} \frac{1}{n}\left(\inf_{\mathscr{P}} \mathbb{I}(\hat{\mathsf{s}}^n; \mathsf{s}^n)\right) \tag{2.33}$$

where $\mathscr{P} := \{p_{\hat{\mathsf{s}}^n|\mathsf{s}^n} : \mathbb{E}_{\mathsf{s}^n}(\Delta(\hat{s}^n, s^n)) < \delta\}$ represents the set of reconstruction methods that has expected distortion smaller than δ, and $\mathbb{I}(\cdot;\cdot)$ denotes the mutual information.

Therefore, the rate-distortion function serves as the theoretical lower bound with which we compare lossy compression algorithms. In particular, we will focus on using

the Mean Squared Error (MSE) as our distortion measure. Given the magnitude of the MSE error and its dependence on the variance of the source, it is customary to represent the error in decibels, ie. on the logarithmic scale, a value known as the *signal to quantization noise ratio* (SQNR). The conversion between δ_{MSE} distortion and $\text{SQNR}|_{dB}$ is defined to be

$$\text{SQNR}|_{dB} = -10 \cdot \log_{10}\left(\frac{\delta_{\text{MSE}}}{\sigma_s^2}\right) \tag{2.34}$$

where σ_s^2 is the variance of a single symbol s_i of the source s^n.

2.3 Probabilistic Graphical Models

Graphs provides a natural representation of the relationships between random variables. Recent developments within the machine learning community have led to the standardization of many tools for statistical inference, in particular, probabilistic graphical models for the task of inference. Such models provides a compact representation of large sets of random variables with sparse correlation. While graphical models are primarily used for parameter estimation within the context of machine learning, this tool, with an appropriate set-up, is also suited for the problem of data compression.

2.3.1 Undirected Graphs

Definition 2.35. (Undirected Graphical Models). An *undirected graphical model* $\mathcal{G} = (\mathcal{V}; \mathcal{E})$ of a set of random variables s^n (also known as a *Markov random field* on s) consists of a set of vertices $\mathcal{V}$, with each vertex $\mathcal{V}_i$ representing a random variable s_i, and a set of edges $\mathcal{E}$ connecting the vertices. For any mutually exclusive subsets $\mathcal{A}, \mathcal{B}, \mathcal{C} \subset \mathcal{V}$, the edges $\mathcal{E}$ satisfy the following conditional independence property:

$$\mathsf{s}_{\mathcal{A}} \perp\!\!\!\perp \mathsf{s}_{\mathcal{B}} \mid \mathsf{s}_{\mathcal{C}} \tag{2.36}$$

holds (ie. $\mathsf{s}_{\mathcal{A}}$ is conditionally independent of $\mathsf{s}_{\mathcal{B}}$ given $\mathsf{s}_{\mathcal{C}}$) whenever there does not exist a path in $\mathcal{E}$ from any vertex in $\mathcal{A}$ to any vertex in $\mathcal{B}$ that does not pass through a vertex in $\mathcal{C}$.

While it is not clear at first glance whether a graph satisfying this property can be constructed, the following asserts that such a graph exists for any given joint distribution:

Fact 2.38. (Universality of Undirected Graphs). Any strictly positive distribution $p_{\mathsf{s}^n}(s^n)$ over an alphabet $\mathscr{S}^n$ can be represented as an undirected graph $\mathcal{G} = (\mathcal{V}; \mathcal{E})$, with $\mathcal{V}$ representing the set of variables s_i and the set of edges $\mathcal{E}$ such that

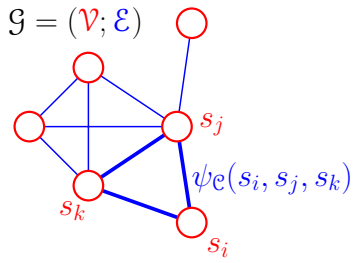


Figure 2.37: An undirected graph $\mathcal{G}$ is defined by its variable nodes $\mathcal{V}$ and the cliques formed by its edges $\mathcal{E}$, where each clique is associated with a clique potentials $\psi_{\mathcal{C}}(s_{\mathcal{C}})$.

$p_{\boldsymbol{s}^n}(s^n)$ can be factored over the maximal cliques $\mathcal{C}$ of $\mathcal{G}$ as

$$p_{\boldsymbol{s}^n}(s^n) \propto \prod_{\mathcal{C} \in \text{MaxClq}(\mathcal{G})} \psi_{\mathcal{C}}(s_{\mathcal{C}}) \tag{2.39}$$

where $\psi_{\mathcal{C}}(\cdot)$ are non-negative functions known as potential functions. In addition, by the Hammersley-Clifford theorem [17], this factorization satisfies the conditional independence property in 2.36 when applied on the graph $\mathcal{G}$.

2.3.1.1 Pairwise Undirected Graphs

A subset of undirected graphical models, known as *pairwise models*, is a less powerful class of models, representing distributions where cliques are of size at most 2, ie. it suffices to take the product in Equation 2.39 only over the edges. For this class of models, while the factorization in Equation 2.39 is complete, in practice it is more convenient to consider potentials on edges and assign unary potential functions to variable nodes, resulting in the alternate factorization of

$$p_{\boldsymbol{s}^n}(s^n) \propto \prod_{i \in \mathcal{V}} \phi_i(s_i) \prod_{(i,j) \in \mathcal{E}} \psi_{ij}(s_i, s_j) \tag{2.40}$$

for pairwise undirected graphs.

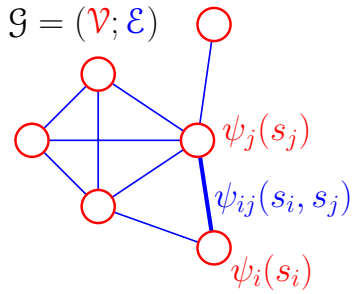


Figure 2.41: The same undirected graph $\mathcal{G}$ as in Figure 2.37, but as a pairwise model. This model requires that the joint distribution be factorable into pairwise relations, and does not allow factorization to be over cliques (of size > 2). $\mathcal{G}$ is defined by its variable nodes $\mathcal{V}$ and their associated node potentials $\phi_i(s_i)$, and its edges $\mathcal{E}$ and their associated edge potentials $\psi_{ij}(s_i, s_j)$.

Due to the factorization structure of joint Gaussians, however, it can be proven that all Gaussian graphical models can be expressed as a pairwise undirected graph, meaning that we do not lose representation power in using pairwise undirected graphs for any arbitrary Gaussian distribution. Therefore, for the rest of this thesis, we shall assume the use of pairwise models on Gaussian distributions to simplify equations.

2.3.2 Factor Graphs

General undirected graphs can be consolidated to provide a more effective graphical representation than factorization over maximal cliques, which are hard to handle algorithmically and hard to decouple conceptually. Factor graphs provide an alternative representation of the same probability distribution on the same set of random variables.

Definition 2.42. (Factor Graphs). A *factor graph* $\mathcal{G} = (\mathcal{V}, \mathcal{F}; \mathcal{E})$ of a set of random variables z^n is a bipartite graph with edges $\mathcal{E}$ joining *variable nodes* $\mathcal{V}$ and *factor nodes* $\mathcal{F}$. A factor graph represents a factorization of

$$p_{\mathsf{z}^n}(z^n) \propto \prod_{a \in \mathcal{F}} f_a(z_{\mathcal{N}_a^{\mathcal{V}}}) \tag{2.43}$$

where $f_a(\cdot)$ are non-negative functions associated with a factor node a known as factors, and $\mathcal{N}_a^{\mathcal{V}}$ represents the neighbors of factor node a in the set of variable nodes $\mathcal{V}$.

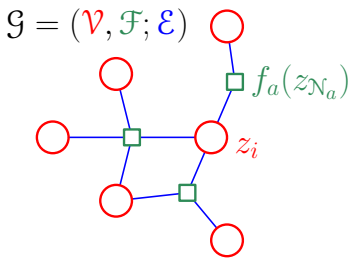


Figure 2.44: The factor graph representation of the undirected graph in Figure 2.37. A factor graph $\mathcal{G}$ is defined by its variable nodes $\mathcal{V}$, its factor nodes $\mathcal{F}$ and their associated factors $f_a(z_{\mathcal{N}_a})$, and its edges $\mathcal{E}$.

The universality of factor graphs in representing arbitrary joint distributions can be seen by applying Fact 2.38. Given an undirected graph, we can construct a factor representing the same distribution by replacing maximal cliques with factor nodes over the same variable nodes.

2.3.3 Gibbs Sampling

A graphical model represents the independence structure of a set of random variables. A particularly important task would be to sample from a distribution given its graphical model representation. The undirected nature of Markov random fields

lends itself to a well-known sampling algorithm known as Gibbs sampling [15], as we describe now.

We assume that we have access to a single-element sampler $\mathsf{Samp}(p_{\mathsf{s}})$ that can draw a single-element sample $s_i \in \mathcal{S}$ according to the distribution $p_{\mathsf{s}}(\cdot)$ in $O(|\mathcal{S}|)$ time. We first initialize $s^{(0)}$ to some (potentially deterministic) value, and initialize σ to be a permutation of $\{1 \dots n\}$ as the order we update corresponding index of the sample. We then perform the updates in Algorithm 2.45.

Algorithm 2.45 Gibbs Sampling

1: **procedure** $\mathsf{GibbsSampler}(p_{\mathsf{s}^n})$
2: $\quad\sigma \leftarrow \mathsf{Perm}(n)$ ▷ a permutation of $\{1 \dots n\}$
3: $\quad s^{(0)} \leftarrow 0$ ▷ initialization
4: $\quad$ **for** $\tau \leftarrow 1 \,..\, \tau^*$ **:**
5: $\quad\quad$ **for** $i \leftarrow \sigma(1) \,..\, \sigma(n)$ **:** ▷ sample every element of the vector
6: $\quad\quad\quad s_i^{(\tau)} \leftarrow \mathsf{Samp}\left(p_{\mathsf{s}^n}(\cdot, s_{\setminus i}^{(\tau-1)})\right)$
7: $\quad$ **return** $s^{(\tau^*)}$

For an undirected graphical model, we can express the expression on the right hand side of line 2.45.6 explicitly as

$$s_i^{(\tau)} \leftarrow \mathsf{Samp}\left(p_{\mathsf{s}^n}(\cdot, s_{\setminus i}^{(\tau-1)})\right) := \mathsf{Samp}\left(\prod_{\mathcal{C} \in \mathrm{MaxClq}(\mathcal{G})} \psi_{\mathcal{C}}(\cdot, s_{\mathcal{C} \setminus i}^{(\tau-1)})\right) \tag{2.46}$$

in terms of the parameters of the undirected graph $\mathcal{G}$.

After enough iterations, ie. for a large enough τ^*, we observe asymptotically independent samples, under moderate conditions on p_{s^n}. The Gibbs sampler has a time complexity of $O(n|\mathcal{S}|\tau^*)$ [15].

2.3.4 Marginalization and Belief Propagation

2.3.4.1 Maximum a Posteriori and Marginalization

When we are faced with the task of inference, a frequent approach would be to represent underlying variables z and observed variables x as a graphical model, given its ease of representing conditional relationships between random variables. Thus, the relations among the underlying variables z represent a *prior belief*, and by having access to the values of the observed variables x, we seek to compute a *posterior belief* on the underlying variables.

In mathematical terms, the task of inference is to find the *maximum a posteriori* (MAP) distribution, ie. compute

$$\hat{z} := \arg\max_z p_{\mathsf{z}|\mathsf{x}}(z \mid x) = \arg\max_z p_{\mathsf{z},\mathsf{x}}(z, x) \tag{2.47}$$

where the equality is due to $p_{\mathsf{z},\mathsf{x}}(z,x) = p_{\mathsf{z}|\mathsf{x}}(z \mid x) \cdot p_{\mathsf{x}}(x)$. Hence, Equation 2.47 reduces the posterior computation to a *marginalization* of z, ie. computing the unconditional distribution $p_{\mathsf{z},\mathsf{x}}$ given the observation x, which as we shall see is an operation central to our decompression system.

2.3.4.2 Belief Propagation

Given the importance of marginalization, we present the celebrated *sum-product algorithm*, an iterative algorithm that can efficiently compute the marginals on a graphical model representation of random variables. The sum-product algorithm, also known as *belief propagation* (BP) or *message passing*, works well for both undirected graphs and factor graphs. For our development, we present the message passing equations for only factor graphs and pairwise undirected graphs.

We first describe the algorithm for factor graphs, then treat pairwise undirected graphs as a special case of factor graphs. For factor graphs. we assume a factorization structure as introduced in 2.43 of

$$p_{\mathsf{z}^n}(z^n) \propto \prod_{a \in \mathcal{F}} f_a(z_{\mathcal{N}_a^{\mathcal{V}}}) \tag{2.48}$$

We let $m_{ia}^{(t)}(\cdot)$ denote the message from variable node $i \in \mathcal{V}$ to factor node $a \in \mathcal{F}$ and $m_{ai}^{(\tau)}(\cdot)$ denote the message from a to i at iteration τ, noting that each message is a single variable function. The sum product algorithm is defined in Algorithm 2.49.

Algorithm 2.49 Belief Propagation on Factor Graphs

1: **procedure** FactorBP($\mathcal{V}, \mathcal{F}, \mathcal{E}$)
2: **for** $(i,a) \in \mathcal{E}$ **:** ▷ initialization
3: $m_{ia}^{(0)}(z_i) \leftarrow 1$
4: $m_{ai}^{(0)}(z_i) \leftarrow 1$
5: **for** $\tau \leftarrow 1 \,..\, \tau^*$ **:**
6: **for** $(i,a) \in \mathcal{E}$ **:**
7: $m_{ia}^{(\tau)}(z_i) \leftarrow \prod_{b \in \mathcal{N}_i^{\mathcal{F}} \setminus \{a\}} m_{bi}^{(\tau-1)}(z_i)$ ▷ variable to factor messages
8: $m_{ai}^{(\tau)}(z_i) \leftarrow \sum_{x_{\mathcal{N}_a^{\mathcal{V}} \setminus \{i\}}} \left(f_a(z_{\mathcal{N}_a^{\mathcal{V}}}) \prod_{j \in \mathcal{N}_a^{\mathcal{V}} \setminus \{i\}} m_{ja}^{(\tau-1)}(z_j) \right)$ ▷ fac. to var. mes.
9: **for** $i \in \mathcal{V}$ **:**
10: $\hat{p}_{\mathsf{z}_i}(z_i) \leftarrow \prod_{a \in \mathcal{N}_i^{\mathcal{F}}} m_{ai}^{(\tau^*)}(z_i)$ ▷ compute total belief, ie. marginals
11: **return** $\hat{p}_{\mathsf{z}^n}$

The expression in 2.49.7 is called the *variable to factor update*, the expression in 2.49.8 is called the *factor to variable update*, and the expression in 2.49.10 is called the *total belief*.

For pairwise undirected graphs, as described in Section 2.3.1.1, we first note that the factor graph representation of a pairwise undirected graph will have factor nodes that are connected to at most 2 variables. Thus, the product in 2.49.8 will only involve one term. Thus, assuming the factorization in Equation 2.40 of

$$p_{\mathbf{z}^n}(z^n) \propto \prod_{i \in \mathcal{V}} \phi_i(z_i) \prod_{(i,j) \in \mathcal{E}} \psi_{ij}(z_i, z_j) \tag{2.50}$$

and denoting $m_{ij}^{(\tau)}(\cdot)$ as the message from node i to node j, we can derive the message passing equations for pairwise undirected graphs as in Algorithm 2.51.

Algorithm 2.51 Belief Propagation on Pairwise Undirected Graphs

procedure PairwiseUndirectedBP($\mathcal{V}, \mathcal{E}$)
  for $(i,j) \in \mathcal{E}$ **:** ▷ initialization
    $m_{ij}^{(0)}(z_j) \leftarrow 1$
  for $\tau \leftarrow 1 \,..\, \tau^*$ **:**
    for $(i,j) \in \mathcal{E}$ **:**
      $$m_{ij}^{(\tau)}(z_j) \leftarrow \sum_{z_i} \left(\phi_i(z_i)\psi_{ij}(z_i, z_j) \prod_{k \in \mathcal{N}_i \setminus \{j\}} m_{ki}^{(\tau-1)}(z_i) \right)$$ ▷ messages
  for $i \in \mathcal{V}$ **:**
    $\hat{p}_{\mathbf{z}_i}(z_i) \leftarrow \phi_i(z_i) \prod_{j \in \mathcal{N}_i} m_{ji}^{(\tau^*)}(z_i)$ ▷ marginalization
  return $\hat{p}_{\mathbf{z}^n}$

In both versions of the belief propagation, if any of the variables are continuous variables, we replace the sum with an integral of the corresponding variable.

As [30] has demonstrated, the iterative sum-product algorithm converges to the true belief (marginalization) within D iterations on a tree graph, where D is the diameter of the tree. The properties of running iterative sum-product on loopy graphs, known as *loopy belief propagation*, is an open problem and is under active research in the machine learning community, but loopy belief propagation nonetheless converges reliably in most cases.

For either versions, the time complexity of each iteration of the algorithm will be linear in the product of the number of edges and the number of computation done at each summation. Therefore, for discrete variables, the time complexity will be $O(\tau^*|\mathcal{E}||\mathscr{X}|^2)$. τ^*, the time to convergence, is $O(1)$ if the algorithm converges, although not necessarily the correct solution [7]. Therefore, for an alphabet of a fixed size, the algorithm will be linear in the size of the input.

2.4 Summary

In this chapter, we introduced important concepts and tools that are building blocks for the data compression scheme we will present in this thesis, including Gaussian distributions (Section 2.1), the class of distributions we seek to compress; coding theory for lossless and lossy schemes (Section 2.2), the framework with which we structure our compression scheme; and probabilistic graphical models (Section 2.3), the main tool we use for the decompressing algorithm.

With these concepts, in particular LDPC coding from digital communications and the sum-product algorithm from machine learning, we are now ready to present our general compression architecture.

3

Compressor Architecture Overview

In this chapter, we present the overview of the compressor architecture, which we will elaborate in details in Chapters 4, 5, and 6. We first introduce the components of the system in Section 3.1, then describe the encoding scheme using these components in Section 3.2, before elaborating on the decoding algorithm in Section 3.3. We then address some practical concerns about decoding in Section 3.4. Finally, we analyze the time and space complexity of the encoding and decoding algorithms, as well as the communication costs in Section 3.5.

3.1 Architecture Components

As we described in Chapter 1.4, the primary focus of this thesis is to construct a lossy compression architecture that compress a sequence of Gaussian variables $\mathbf{s}$ into a bit stream $\mathbf{x}$. To achieve such a task, we require the following inputs to the algorithm:

(i) a *source model* to represent the underlying correlation structure of the data (Section 3.1.1),
(ii) a *quantizer* that maps a real value to a discrete value (Section 3.1.2), and
(iii) a *code* that compresses the discrete values into a binary bit stream, which can be sub-divided into a *translator* that converts a set of discrete values into a binary representation (Section 3.1.3), and a *binary code* that compresses the translated binary representation into a shorter bit stream (Section 3.1.4).

3.1.1 Source Model

Let n denote the length of the source sequence. The source model is the joint probability distribution $p_{\mathbf{s}^n}$ from which s^n is drawn, represented in the form of a probabilistic graphical model, as introduced in Section 2.3. For ease of discussion, we represent it as a pairwise undirected graphical model (Section 2.3.1.1), noting that pairwise

undirected models also has universal representation power, for all Gaussian sources can be written with a factorization structure of

$$p_{\mathbf{s}^n}(s^n) \propto \prod_{i \in \mathcal{V}} \phi_i(s_i) \prod_{(i,j) \in \mathcal{E}} \psi_{ij}(s_i, s_j) \tag{3.1}$$

3.1.2 Quantizer

Essential to any lossy compression algorithm is quantization, the procedure of mapping a length-n source sequence s^n drawn from a large alphabet $\mathcal{S}^n$, in our case $\mathbb{R}^n$ which is uncountably large, into a length-m quantized sequence u^m from a smaller set $\mathcal{U}^m$. In our case, we let the cardinality of the set to be $|\mathcal{U}| =: \beta = 2^b$, a power of 2, such that each u can be represented with b bits.

For our system, we draw the quantizer $q(\cdot)$ from a collection $\mathcal{Q}(n, m)$, such that each q is a function satisfying

$$q : \mathcal{S}^n \to \mathcal{U}^m \tag{3.2}$$

which we write as

$$\mathrm{u}^m = q(\mathrm{s}^n) \tag{3.3}$$

3.1.3 Translator

To represent a set of discrete values with 0s and 1s, we need a one-to-one mapping t_b that maps

$$t_b : \mathcal{U} \to \mathbb{Z}_2^b \tag{3.4}$$

where $\mathbb{Z}_2$ represents the finite field of 2 and b is the size of the output of the translator. As a slight abuse of notation, we allow t_b to operate on vectors u^m with individual elements concatenated into one vector, ie.

$$\mathrm{z}^{mb} = t_b(\mathrm{u}^m) = t_b\left(\begin{bmatrix} \mathrm{u}_1 \\ \vdots \\ \mathrm{u}_m \end{bmatrix}\right) := \begin{bmatrix} t_b(\mathrm{u}_1) \\ \vdots \\ t_b(\mathrm{u}_m) \end{bmatrix} \tag{3.5}$$

with z^{mb} denoting the length-mb output vector of 0s and 1s. The subscript b of t_b will be dropped when unambiguous.

3.1.4 Binary Code

The last component of the algorithm would be the binary code, for which we drop the "binary" prefix for the remainder of the thesis when unambiguous. The *code* is a function $h(\cdot)$ that maps a length-mb bit stream into a shorter length-kb bit stream, an operation commonly known as hashing.

For our system, we draw the hashing functions $h(\cdot)$ from a collection $\mathscr{H}(mb, kb)$, such that each h is a function satisfying

$$h : \mathbb{Z}_2^{mb} \to \mathbb{Z}_2^{kb} \tag{3.6}$$

which we write as

$$\mathrm{x}^{kb} = h(\mathrm{z}^{mb}) \tag{3.7}$$

3.2 Encoder

The encoder, as we introduced in Section 1.4, is model-free, meaning that the encoding operation does not depend on the source model. By combining the components described in Section 3.1, we let the encoder output be

$$\mathrm{x}^{kb} = h(t(q(\mathrm{s}^n))) \tag{3.8}$$

and by choosing suitable parameters m and k, we achieve a *nominal compression rate* of

$$r_{\text{code}} := \frac{kb}{n} \tag{3.9}$$

bits per sample.

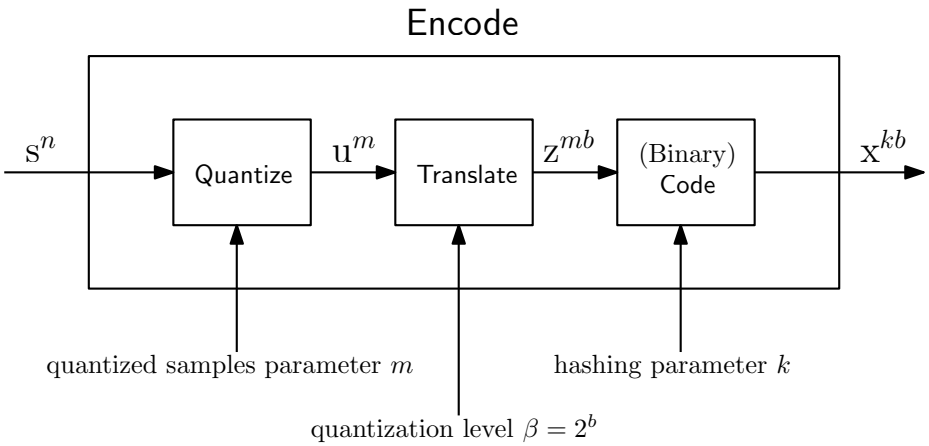


Figure 3.10: The Encode procedure takes in a source sequence s^n as an input, and outputs x^{kb} as the compressed bit stream. Encode contains three components: Quantize, Translate, and Code.

For our particular encoder, we let the set of quantizers to be sparse and linear, ie.

$$\mathscr{Q}(n, m) = \{q : q(\mathrm{s}^n) = \lfloor Q\mathrm{s}^n + Q_0 \rfloor\} \tag{3.11}$$

where the set of Q's and Q_0's we use are chosen from a subset of

$$\left\{Q : Q \in \mathbb{R}^{m\times n}, \max \deg(Q) \le \xi\right\} \tag{3.12}$$

$$\{Q_0 : Q_0 \in \mathbb{R}^m\} \tag{3.13}$$

for some constant ξ, where $Q\mathrm{s}^n$ denotes a matrix multiplication by a matrix Q, and $\lfloor\cdot\rfloor$ represents the floor function where we take the integer part each element in the vector. We note that the standard uniform quantizer

$$q_{\mathrm{us}}(x) := \lfloor x \rfloor \tag{3.14}$$

is an instance of a sparse linear quantizer with $Q = I$.

For the translator $t_b(\cdot)$, we shall focus on the b-bit gray code, which has the special property that the encoding of adjacent numbers being off by exactly one bit. We write

$$t_b(\mathrm{u}^m) = \begin{bmatrix} \mathsf{Gray}_b(\mathrm{u}_1) \\ \vdots \\ \mathsf{Gray}_b(\mathrm{u}_m) \end{bmatrix} \tag{3.15}$$

For the code, we focus on low-density parity-check (LDPC) codes, a class of linear codes introduced in Section 2.2.3, such that

$$\mathscr{H}(mb, kb) = \{h : h(\mathrm{z}^{mb}) = H\mathrm{z}^{mb}\} \tag{3.16}$$

where the set of H's we use are chosen from a subset of

$$\left\{H : H \in \mathbb{Z}_2^{kb\times mb}, \max \deg(H) \le \rho, \min \deg(H) \ge 1\right\} \tag{3.17}$$

We will discuss each of these choices in the following chapters. Note that this is one particular incarnation of the architecture: the architecture that we described in the previous sections is general and given its modular nature, different components can be switched out. Different choices of quantizer, translator, and code will result in difference in the implementation details of the Encode and Decode algorithms, but the general structure of the pseudocode remains the same.

With our choice of quantizer, translator, and code, the Encode function is thus a very light-weight model-free encoder, described in pseudocode in Algorithm 3.18.

Algorithm 3.18 Model-free Encoder

procedure $\mathsf{Encode}(\mathrm{s}^n; \mathscr{Q}, \mathscr{H})$
  $Q, Q_0 \leftarrow \mathsf{Rand}(\mathscr{Q}(n, m))$ ▷ randomly select quantization matrix
  $H \leftarrow \mathsf{Rand}(\mathscr{H}(mb, kb))$ ▷ randomly select LDPC matrix
  $\mathrm{u}^m \leftarrow \lfloor Q\mathrm{s}^n + Q_0 \rfloor$ ▷ quantization
  $\mathrm{z}^{mb} \leftarrow \mathrm{vec}(\{\mathsf{Gray}_b(\mathrm{u}_i)\}_{i=1}^m)$ ▷ translation
  $\mathrm{x}^{kb} \leftarrow H\mathrm{z}^{mb}$ ▷ hashing
  return x^{kb}

The output of the algorithm, the length-kb vector x, is the compressed bit stream, which the decoder (as we describe below in Section 3.3) decompresses to reconstruct a source sequence $\hat{\mathrm{s}}^n$ that minimizes the expected value of the distortion measure $\mathbb{E}_{\boldsymbol{s}}(\Delta(\hat{\mathrm{s}}^n, \mathrm{s}^n))$.

3.3 Decoder

Given the irreversible nature of our lossy compression architecture, multiple source sequences s^n will map to the same compressed bit stream x^{kb}. Using mean squared error (MSE, Equation 2.31) as our distortion measure, our goal in the decoding step is to find the $(\hat{\mathrm{s}}^n)^*$ that satisfies the constraints imposed by the the compressed vector x^{kb} that minimize the expected distortion, ie.

$$(\hat{\mathrm{s}}^n)^* = \underset{\hat{\mathrm{s}}:h(t(q(\hat{\mathrm{s}})))=\mathrm{x}}{\arg\min}\ \mathbb{E}_{\boldsymbol{s}}(\Delta_{\mathrm{MSE}}(\hat{\mathrm{s}}^n, \mathrm{s}^n)) = \underset{\hat{\mathrm{s}}:h(t(q(\hat{\mathrm{s}})))=\mathrm{x}}{\arg\min}\ \mathbb{E}_{\boldsymbol{s}}\left(\frac{1}{n}\sum_{i=1}^{n}(\hat{\mathrm{s}}_i - \mathrm{s}_i)^2\right) \tag{3.19}$$

This estimate $(\hat{\mathrm{s}}^n)^*$ is know as the minimum mean squared error (MMSE) estimate.

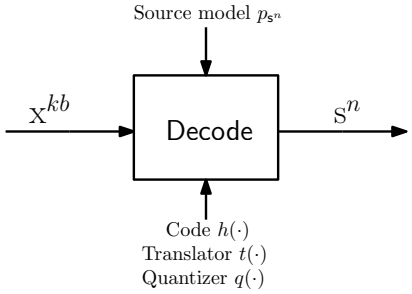


Figure 3.20: The Decode procedure takes in the compressed sequence x^{kb} as an input, and outputs $\hat{\mathrm{s}}^n$ as the reconstruction. Decode also needs the code, translator, and quantizer components, as well as the source model, to reconstruct the original source sequence.

To solve the minimization problem in Equation 3.19, the decoder needs the constrains between $\boldsymbol{x}^{kb}$ and $\boldsymbol{s}^n$, as well as the correlation structure among the elements of $\boldsymbol{s}^n$, as illustrated in Figure 3.20. This can be achieved by having an undirected graphical representation of $p_{\boldsymbol{s}^n}$ and the factor graph representations of $q(\cdot)$, $t(\cdot)$, and $h(\cdot)$. We then combine these subgraphs into a joint graph $\mathcal{G}$ (Figure 3.21), on which we run graphical inference to infer the MMSE estimate $(\hat{\mathrm{s}}^n)^*$.

We now describe the subgraphs of $\mathcal{G}$, each corresponding to a component describe in Section 3.1.

3.3.1 Source Subgraph

The *source subgraph* $\mathcal{C}$ contains prior knowledge of the source $p_{\boldsymbol{s}^n}(\cdot)$. If $p_{\boldsymbol{s}^n}(\cdot)$ is available in its algebraic form, then we can construct $\mathcal{C}$ through its factorization structure. If the prior knowledge is available through an undirected graphical model

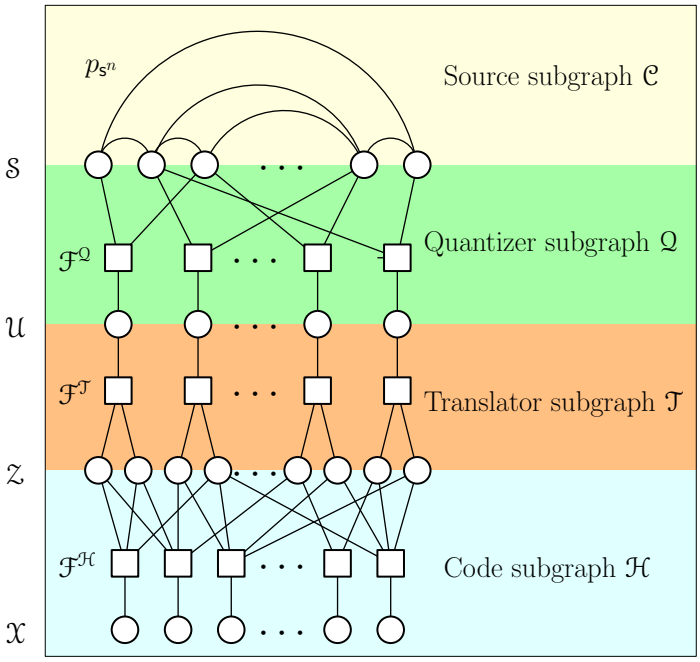


Figure 3.21: The joint graph $\mathcal{G}$ has four subgraphs: source $\mathcal{C}$, quantizer $\mathcal{Q}$, translator $\mathcal{T}$, and code $\mathcal{H}$, with factor nodes within each of the latter three components. There are four sets of variable nodes: source $\mathcal{S}$, quantized bins $\mathcal{U}$, translated bits $\mathcal{Z}$, and hashed bits $\mathcal{X}$. The relationships between the variables are embedded within the graph $\mathcal{G}$.

or a factor graph, we can represent it with a pairwise undirected graphical model (Section 2.3.1.1) with factorization

$$p_{\mathbb{S}^n}(s^n) \propto \prod_{\mathcal{S}_i \in \mathcal{S}} \phi_i^{\mathcal{C}}(s_i) \prod_{(\mathcal{S}_i, \mathcal{S}_j) \in \mathcal{E}} \psi_{ij}^{\mathcal{C}}(s_i, s_j) \tag{3.22}$$

which simplifies the decoding algorithm, noting that pairwise models have universal representation power for Gaussian sources.

The source subgraph $\mathcal{C}$ can thus be represented as a pairwise undirected graph with variable nodes $\mathcal{S}$, and we use subscripts $\mathcal{S}_i$ to denote the i^{th} node.

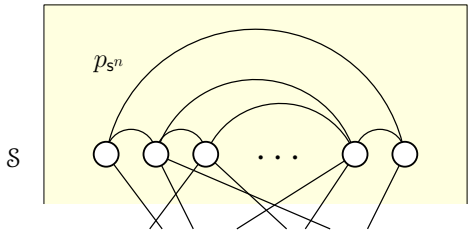


Figure 3.23: The source subgraph $\mathcal{C}$ is a pairwise undirected graphical model, constructed from the correlation structure among the source variables s_i.

3.3.2 Quantizer Subgraph

The quantization procedure contains two steps: linear transform by matrix $Q \in \mathcal{Q}(n, m)$ and standard uniform scalar quantization $q_{\text{us}}(\cdot) = \lfloor \cdot \rfloor$. The relationship between the source sequence s^n and the quantized sequence u^m, also called the *bin sequence*, can be described as a factor graph. Given the linear transform Q, we note that each row of Q corresponds to an output of the quantizer. Thus, we construct the factor graph $\mathfrak{Q}$ such that a quantizer output node a is connected to source node i if and only if $Q_{ai} \neq 0$. This $\mathfrak{Q}$ graph is called the *quantizer subgraph*.

In particular, the function $f_a^{\mathfrak{Q}}(\cdot)$ associated with each quantizer factor node is an indicator function that evaluates to 1 if and only if the quantizer constraints are met, ie. the quantized source sequence satisfies the constraints imposed by u^m. Thus, the function $f_a^{\mathfrak{Q}}(\cdot)$ has the form

$$f_a^{\mathfrak{Q}}(s, u) = \mathbb{1}\left\{ u_a = \left\lfloor \sum_{i=1}^{n} Q_{ai} s_i + Q_{0a} \right\rfloor \right\} \tag{3.24}$$

where $\mathbb{1}\{\cdot\}$ represents the indicator function. This gives the unnormalized distribution over the quantizer subgraph, which is also called the *quantizer constraint function*:

$$c^{\mathfrak{Q}}(s^n, u^m) := \prod_{a=1}^{m} f_a^{\mathfrak{Q}}(s_{\mathcal{N}_a^{\mathfrak{Q}}}, u_a) = \prod_{\mathcal{F}_a^{\mathfrak{Q}} \in \mathcal{F}^{\mathfrak{Q}}} \mathbb{1}\left\{ u_a = \left\lfloor \sum_{i \in \mathcal{N}_a^{\mathfrak{Q}}} Q_{ai} s_i + Q_{0a} \right\rfloor \right\} \tag{3.25}$$

This imposes an unnormalized conditional probability mass function (pmf) that assigns equal probability to configurations s^n that satisfy the constraints imposed by the bin sequence u^m, and 0 to those that do not. Note that the sums in Equations 3.24 and 3.25 are equivalent because $Q_{ai} = 0$ for $i \notin \mathcal{N}_a^{\mathfrak{Q}}$.

The quantizer subgraph $\mathfrak{Q}$ is thus a factor graph with variable nodes $\mathcal{S}$ representing the source variables s_i, variable nodes $\mathcal{U}$ representing the bin sequence u_a, and factor nodes $\mathcal{F}^{\mathfrak{Q}}$ representing the structure of the quantization matrix Q and offset vector Q_0.

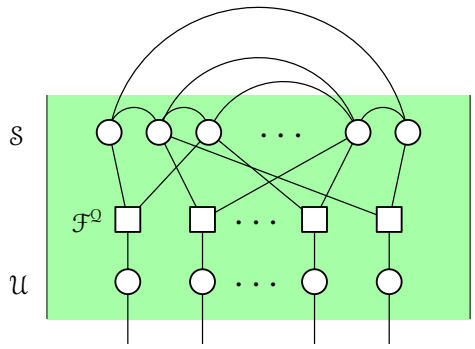


Figure 3.26: The quantizer subgraph $\mathfrak{Q}$ is a factor graph constructed from the quantizer function $q(\cdot)$, whose structure is defined by the quantization matrix Q and the offset vector Q_0.

3.3.3 Translator Subgraph

As described in Section 3.1.3, the translator $t(\cdot)$ is a one-to-one mapping from $\mathcal{U}^m$ to $\mathbb{Z}_2^{mb}$. We can describe the relationship between the bin sequence u^m and the translated sequence z^{mb} with a factor graph with the m variable nodes $\mathcal{U}$ representing the bin sequence, mb variables nodes $\mathcal{Z}$ representing the translated sequence, and m factor nodes $\mathcal{F}^{\mathcal{T}}$ representing the relationship between $\mathcal{U}$ and $\mathcal{Z}$. A translated variable node $\mathcal{Z}_i$ is connected to the bin factor node $\mathcal{F}_a^{\mathcal{T}}$ from which it is translated. We thus obtain the following factorization:

$$c^{\mathcal{T}}(u^m, z^{mb}) := \prod_{a=1}^{m} f_a^{\mathcal{T}}(u_a, z_{\mathcal{N}_a^{\mathcal{T}}}) = \prod_{\mathcal{F}_a^{\mathcal{T}} \in \mathcal{F}^{\mathcal{T}}} \mathbb{1}\left\{u_a = t^{-1}\left(z_{\mathcal{N}_a^{\mathcal{T}}}\right)\right\} \tag{3.27}$$

where $t^{-1}(\cdot)$ is the *inverse translator*, the inverse of $t(\cdot)$, which is uniquely defined because $t(\cdot)$ is one-to-one.

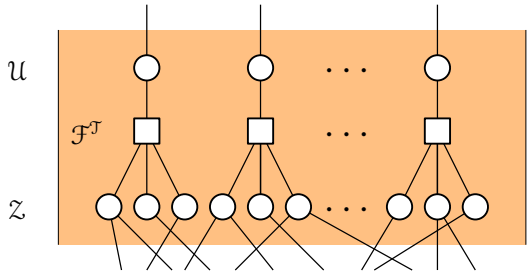


Figure 3.28: The translator subgraph $\mathcal{T}$ is a factor graph, with the variable nodes $\mathcal{Z}$ representing the translated sequence z being divided into groups of b, where the nodes of each group corresponds to an element in u.

3.3.4 Code Subgraph

The coding matrix H defines the parity check bit constraints of

$$x_a = \bigoplus_{i=1}^{mb} H_{ai} z_i \tag{3.29}$$

where a is the index of the parity check bit, i is the index of the translated bits, with $\oplus$ being the xor operation, ie. summation over $\mathbb{Z}_2$. We can represent this relationship with a bipartite graph $\mathcal{H}$, such that a parity check factor node a is connected to translated node i if and only if $H_{ai} = 1$. This $\mathcal{H}$ graph is called the *code subgraph*. With this notation, we can rewrite the sum as

$$x_a = \bigoplus_{i \in \mathcal{N}_a^{\mathcal{H}}} z_i \tag{3.30}$$

where $\mathcal{N}_a^{\mathcal{H}}$ represents the neighbors of a in the $\mathcal{H}$ graph. With this constraint, we can assign an unnormalized probability mass function (pmf) over all possible translated

sequences z^{mb}, with sequences that satisfy the parity check constraints imposed by x having equal probabilities and those that do not satisfy the constraints having a probability of 0. The unnormalized distribution, which is also known as the *hash constraint function*, is thus

$$c^{\mathcal{H}}(z^{mb}, x^{kb}) := \prod_{a=1}^{kb} f_a^{\mathcal{H}}(z_{\mathcal{N}_a^{\mathcal{H}}}, x_a) = \prod_{\mathcal{F}_a^{\mathcal{H}} \in \mathcal{F}^{\mathcal{H}}} \mathbb{1}\left\{ x_a = \bigoplus_{i \in \mathcal{N}_a^{\mathcal{H}}} z_i \right\} \tag{3.31}$$

which evaluates to 1 if z^{mb} satisfies all constraints imposed by x^{kb}, and 0 otherwise.

The code subgraph $\mathcal{H}$ thus has variable nodes $\mathcal{Z}$ representing the translated bits z_i, variable nodes $\mathcal{X}$ representing the hashed bit sequence x_a, and factor nodes $\mathcal{F}^{\mathcal{H}}$ representing the constraints imposed by H.

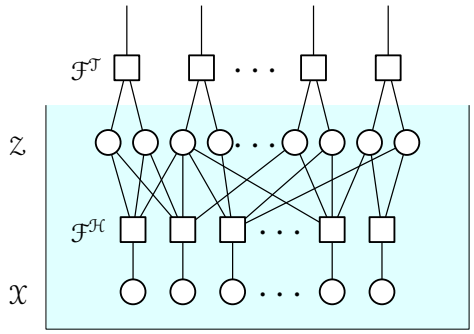


Figure 3.32: The code subgraph $\mathcal{H}$ is a factor graph, with the variable nodes $\mathcal{Z}$ representing the translated sequence z, the variable nodes $\mathcal{X}$ representing the hashed bit sequence x, and the factor nodes $\mathcal{F}^{\mathcal{H}}$ representing the relations imposed by the LDPC matrix H.

3.3.5 Decoding Algorithm

With the subgraphs defined above, we are now ready to present the decoding algorithm. With $\mathcal{C}$ as the source subgraph, $\mathcal{Q}$ as the quantizer subgraph, $\mathcal{T}$ as the translator subgraph, and $\mathcal{H}$ as the code subgraph, let $\mathcal{G} = \mathcal{C} \cup \mathcal{Q} \cup \mathcal{T} \cup \mathcal{H}$ be the joint graph. With this joint graph, the decoding algorithm is straight forward: we run loopy belief propagation, as introduced in Section 2.3.4.2, on the joint graph $\mathcal{G}$ until convergence. The overview of the belief propagation algorithm is listed in Algorithm 3.33, and the details of the message passing equations will be explained in the following chapters.

The four subgraphs (source, quantizer, translator, code) of the joint graph $\mathcal{G}$ can be divided into two portions, operating in different domains: (i) the continuous portion, which operates in $\mathbb{R}$ on the source subgraph $\mathcal{C}$, and (ii) the discrete portion, which operates in $\mathbb{Z}_p$ on the translator subgraph $\mathcal{T}$ with $p = 2^b$ and the code subgraph $\mathcal{H}$ with $p = 2$.

As noted in Section 1.4, to the best of our knowledge, this work is the first attempt in using continuous-discrete hybrid belief propagation in the context of data compression. With a hybrid joint distribution over the graphical model, the quantizer

Algorithm 3.33 Decoding Algorithm: Belief Propagation on Joint Graph $\mathcal{G}$

procedure Decode($\mathrm{x}, \mathcal{C}, \mathcal{Q}, \mathcal{T}, \mathcal{H}$)
  for $\tau \leftarrow 1 \,..\, \tau^*$ **:** ▷ Iterate until convergence
    for $(\mathcal{S}_i, \mathcal{S}_j) \in \mathcal{E}_{\mathcal{S}}$ **:** ▷ BP within $\mathcal{S}$

$$m^{(\tau)}_{\mathcal{S}_i\mathcal{S}_j}(s_j) \leftarrow \int_{s_i} \left(\phi^{\mathcal{C}}_i(s_i)\psi^{\mathcal{C}}_{ij}(s_i, s_j) \prod_{k \in \mathcal{N}^{\mathcal{C}}_i \setminus \{j\}} m^{(\tau-1)}_{\mathcal{S}_k\mathcal{S}_i}(s_i) \prod_{a \in \mathcal{N}^{\mathcal{Q}}_i} m^{(\tau-1)}_{\mathcal{F}^{\mathcal{Q}}_a\mathcal{S}_i}(s_i) \right)$$

    for $(\mathcal{S}_i, \mathcal{F}^{\mathcal{Q}}_a) \in \mathcal{E}_{\mathcal{S},\mathcal{F}^{\mathcal{Q}}}$ **:** ▷ BP between $\mathcal{S}$ and $\mathcal{F}^{\mathcal{Q}}$

$$m^{(\tau)}_{\mathcal{S}_i\mathcal{F}^{\mathcal{Q}}_a}(s_i) \leftarrow \prod_{b \in \mathcal{N}^{\mathcal{Q}}_i \setminus \{a\}} m^{(\tau-1)}_{\mathcal{F}^{\mathcal{Q}}_b\mathcal{S}_i}(s_i) \prod_{j \in \mathcal{N}^{\mathcal{C}}_i} m^{(\tau-1)}_{\mathcal{S}_j\mathcal{S}_i}(s_i)$$

$$m^{(\tau)}_{\mathcal{F}^{\mathcal{Q}}_a\mathcal{S}_i}(s_i) \leftarrow \sum_{u_a} \int_{s_{\mathcal{N}^{\mathcal{Q}}_a \setminus \{i\}}} \left(f^{\mathcal{Q}}_a(s_{\mathcal{N}^{\mathcal{Q}}_a}, u_a) \cdot m^{(\tau-1)}_{\mathcal{U}_a\mathcal{F}^{\mathcal{Q}}_a}(u_a) \prod_{j \in \mathcal{N}^{\mathcal{Q}}_a \setminus \{i\}} m^{(\tau-1)}_{\mathcal{S}_j\mathcal{F}^{\mathcal{Q}}_a}(s_j) \right)$$

    for $(\mathcal{F}^{\mathcal{Q}}_a, \mathcal{U}_a) \in \mathcal{E}_{\mathcal{F}^{\mathcal{Q}},\mathcal{U}}$ **:** ▷ BP between $\mathcal{F}^{\mathcal{Q}}$ and $\mathcal{U}$

$$m^{(\tau)}_{\mathcal{F}^{\mathcal{Q}}_a\mathcal{U}_a}(u_a) \leftarrow \int_{s_{\mathcal{N}^{\mathcal{Q}}_a}} \left(f^{\mathcal{Q}}_a(s_{\mathcal{N}^{\mathcal{Q}}_a}, u_a) \prod_{i \in \mathcal{N}^{\mathcal{Q}}_a} m^{(\tau-1)}_{\mathcal{S}_i\mathcal{F}^{\mathcal{Q}}_a}(s_i) \right)$$

$$m^{(\tau)}_{\mathcal{U}_a\mathcal{F}^{\mathcal{Q}}_a}(u_a) \leftarrow m^{(\tau-1)}_{\mathcal{F}^{\mathcal{T}}_a\mathcal{U}_a}(u_a)$$

    for $(\mathcal{U}_a, \mathcal{F}^{\mathcal{T}}_a) \in \mathcal{E}_{\mathcal{U},\mathcal{F}^{\mathcal{T}}}$ **:** ▷ BP between $\mathcal{U}$ and $\mathcal{F}^{\mathcal{T}}$

$$m^{(\tau)}_{\mathcal{U}_a\mathcal{F}^{\mathcal{T}}_a}(u_a) \leftarrow m^{(\tau-1)}_{\mathcal{F}^{\mathcal{Q}}_a\mathcal{U}_a}(u_a)$$

$$m^{(\tau)}_{\mathcal{F}^{\mathcal{T}}_a\mathcal{U}_a}(u_a) \leftarrow \sum_{z_{\mathcal{N}^{\mathcal{T}}_a}} \left(f^{\mathcal{T}}_a(u_a, z_{\mathcal{N}^{\mathcal{T}}_a}) \prod_{i \in \mathcal{N}^{\mathcal{H}}_a} m^{(\tau-1)}_{\mathcal{Z}_i\mathcal{F}^{\mathcal{T}}_a}(z_i) \right)$$

    for $(\mathcal{F}^{\mathcal{T}}_a, \mathcal{Z}_i) \in \mathcal{E}_{\mathcal{F}^{\mathcal{T}},\mathcal{Z}}$ **:** ▷ BP between $\mathcal{F}^{\mathcal{T}}$ and $\mathcal{Z}$

$$m^{(\tau)}_{\mathcal{F}^{\mathcal{T}}_a\mathcal{Z}_a}(z_i) \leftarrow \sum_{u_a, z_{\mathcal{N}^{\mathcal{T}}_a \setminus \{i\}}} \left(f^{\mathcal{T}}_a(u_a, z_{\mathcal{N}^{\mathcal{T}}_a}) \cdot m^{(\tau-1)}_{\mathcal{U}_a\mathcal{F}^{\mathcal{T}}_a}(u_a) \prod_{j \in \mathcal{N}^{\mathcal{T}}_a \setminus \{i\}} m^{(\tau-1)}_{\mathcal{Z}_j\mathcal{F}^{\mathcal{T}}_a}(z_j) \right)$$

$$m^{(\tau)}_{\mathcal{Z}_i\mathcal{F}^{\mathcal{T}}_c}(z_i) \leftarrow \prod_{a \in \mathcal{N}^{\mathcal{H}}_i} m^{(\tau-1)}_{\mathcal{F}^{\mathcal{H}}_a\mathcal{Z}_i}(z_i)$$

    for $(\mathcal{Z}_i, \mathcal{F}^{\mathcal{H}}_a) \in \mathcal{E}_{\mathcal{Z},\mathcal{F}^{\mathcal{H}}}$ **:** ▷ BP between $\mathcal{Z}$ and $\mathcal{F}^{\mathcal{H}}$

$$m^{(\tau)}_{\mathcal{Z}_i\mathcal{F}^{\mathcal{H}}_a}(z_i) \leftarrow m^{(\tau-1)}_{\mathcal{F}^{\mathcal{T}}_c\mathcal{Z}_i}(z_i) \prod_{b \in \mathcal{N}^{\mathcal{H}}_i \setminus \{a\}} m^{(\tau-1)}_{\mathcal{F}^{\mathcal{H}}_b\mathcal{Z}_i}(z_i)$$

$$m^{(\tau)}_{\mathcal{F}^{\mathcal{H}}_a\mathcal{Z}_i}(z_i) \leftarrow \sum_{x_a, z_{\mathcal{N}^{\mathcal{H}}_a \setminus \{i\}}} \left(f^{\mathcal{H}}_a(z_{\mathcal{N}^{\mathcal{H}}_a}, x_a) \cdot \mathbb{1}\{x_a = \mathrm{x}_a\} \prod_{j \in \mathcal{N}^{\mathcal{H}}_a \setminus \{i\}} m^{(\tau-1)}_{\mathcal{Z}_j\mathcal{F}^{\mathcal{H}}_a}(z_j) \right)$$

  for $\mathcal{S}_i \in \mathcal{S}$ **:** ▷ Compute total belief, ie. marginals

$$\hat{p}_{\boldsymbol{s}_i}(s_i) \leftarrow \prod_{a \in \mathcal{N}^{\mathcal{Q}}_i} m^{(\tau^*)}_{\mathcal{F}^{\mathcal{Q}}_a\mathcal{S}_i}(s_i) \prod_{j \in \mathcal{N}^{\mathcal{C}}_i} m^{(\tau^*)}_{\mathcal{S}_j\mathcal{S}_i}(s_i)$$

  return $\mathbb{E}_{\boldsymbol{s}}(s^n)$

subgraph $\mathcal{Q}$ discretizes the continuous source and acts as a bridge for information flow between the two portions. Hence, in terms of the *rate-distortion theory* introduced in Section 2.2.4, the choice of the quantizer subgraph $\mathcal{Q}$ is tied to the distortion δ of the reconstruction $\hat{\mathrm{s}}^n$, while the choice of the code subgraph $\mathcal{H}$ is tied to the rate r.

Given this observation, we note that rate and distortion can be optimized somewhat independently by optimizing the quantizer and the code, while still being bound by the rate distortion function (Definition 2.32). This means that for a fixed quantizer $q(\cdot)$ (hence a fixed distortion), we can optimize the rate by tuning the code $h(\cdot)$.

3.4 Doping Symbols

With the loopy belief propagation algorithm presented in Algorithm 3.33, depending on the source model, trivial initial messages $m^{(0)}_{\mathcal{G}_i\mathcal{G}_j}(\cdot)$ may itself be a stable point of convergence of the algorithm, which may cause decoding failure. Given this observation, the encoder may sometimes need to select a subset $\mathscr{D}$ of the translated z^{mb} bits to send directly to the decoder for initialization. This set of bits $\mathrm{z}_{\mathscr{D}}$, known as the *dope bits*, are interpreted by the decoder as deterministic messages from the $\mathcal{Z}_{\mathscr{D}}$ nodes to the $\mathcal{F}^{\mathcal{T}}$ and the $\mathcal{F}^{\mathcal{H}}$ factor nodes:

$$m^{(\tau)}_{\mathcal{Z}_i\mathcal{F}^{\mathcal{T}}_a}(z_i) = \mathbb{1}\{z_i = \mathrm{z}_i\} \quad \forall i \in \mathscr{D}, \ \ \forall a \in \mathcal{N}^{\mathcal{T}}_i \tag{3.34}$$

$$m^{(\tau)}_{\mathcal{Z}_i\mathcal{F}^{\mathcal{H}}_a}(z_i) = \mathbb{1}\{z_i = \mathrm{z}_i\} \quad \forall i \in \mathscr{D}, \ \ \forall a \in \mathcal{N}^{\mathcal{H}}_i \tag{3.35}$$

which overrides the messages in Lines 3.33.16 and 3.33.18 of the loopy belief propagation algorithm.

This process of sending translated $\mathrm{z}_{\mathscr{D}}$ bits to the decoder, interpreted as deterministic messages, is known as *doping*, which is introduced by [6]. The dope bits provide an anchor for the decoding process, and depending on the source model, only a fraction of the unhashed z bits need to be doped. The *dope rate*, which is defined to be

$$r_{\text{dope}} := \frac{|\mathscr{D}|}{n} \tag{3.36}$$

can be tuned to optimize for the total rate r, which is the sum of the nominal rate and the dope rate:

$$r := r_{\text{code}} + r_{\text{dope}} = \frac{kb + |\mathscr{D}|}{n} \tag{3.37}$$

As we shall see in Chapter 4, doping is intricately tied to the choice of translators and codes. We shall discuss doping in more detail along with the discussion of code structure.

3.5 Complexity

We now give the theoretical bounds on the time and space complexities of our algorithm, for both the Encode and Decode procedures.

3.5.1 Time Complexity of Encode

We first analyze the time complexity of the encoder. Quantization of s^n to u^m takes $O(\xi m)$ time, where ξ is the maximum row weight of the quantization matrix Q. Translation of u^m to z^{mb} takes $O(mb)$ time. LDPC coding of z^{mb} to x^{kb} takes $O(\rho kb)$ time, where ρ is the maximum row weight of the LDPC matrix H. The total complexity is thus

$$O(\xi m + mb + \rho kb) \tag{3.38}$$

If we choose the maximum row weight of the quantizer matrix and the code matrix to be constants, ie. $\xi = O(1)$ and $\rho = O(1)$. Together with $m = O(n)$ and $b = O(1)$, the time complexity for encoding will be linear in the input size n, ie. of $O(n)$.

3.5.2 Space Complexity of Encode

For space complexity, we need to store the quantization matrix Q which has $O(\xi m)$ elements, the translator which is a table with $O(2^b b)$ elements, and the LDPC matrix H which has $O(\rho kb)$ elements. Hence, the space complexity on the encoder side is

$$O(\xi m + 2^b b + \rho kb) \tag{3.39}$$

which is also linear in n, ie. of $O(n)$, with the choice of parameters as above.

3.5.3 Time Complexity of Decode

The time complexity of the decoder is dependent on the number of iterations till convergence τ^*.

For each iteration, messages passing among the source nodes $\mathcal{S}$ takes $O(I_{\mathcal{C}}|\mathcal{E}_{\mathcal{C}}|)$, where $I_{\mathcal{C}}$ is the time needed to calculate each integral. For univariate Gaussian variables, $I_{\mathcal{C}}$ is $O(1)$ since there are closed form expressions for the Gaussian mean and variance updates (Chapter 6).

Message passing between the $\mathcal{S}$ and the $\mathcal{U}$ nodes has time complexity $O(I_{\mathcal{Q}}\left|\mathcal{E}_{\mathcal{S},\mathcal{F}^{\mathcal{Q}}}\right|)$, where $I_{\mathcal{Q}}$ is the time to evaluate the integral in the quantization messages, which is $O(|\mathcal{U}|) \cdot O(1)$ given that there are $|\mathcal{U}|$ bins for each $\mathcal{U}$ node and that the messages from $\mathcal{S}$ to $\mathcal{U}$ are Gaussian (Chapter 5).

Message passing between the $\mathcal{U}$ and the $\mathcal{Z}$ nodes has time complexity of $O(mb)$, which is the cost of translation (Chapter 4). Finally, message passing between $\mathcal{Z}$ and $\mathcal{X}$ costs $O(\rho kb)$, which is the LDPC decoding cost we discussed in Section 2.3.4.2.

Hence, the total time complexity is

$$\tau^* \cdot \left(O(I_{\mathcal{C}}|\mathcal{E}_{\mathcal{C}}|) + O(I_{\mathcal{Q}}\left|\mathcal{E}_{\mathcal{S},\mathcal{F}^{\mathcal{Q}}}\right|) + O(mb) + O(\rho kb)\right) \tag{3.40}$$

$$= \tau^* \cdot (O(|\mathcal{E}_{\mathcal{C}}|) + O(|\mathcal{U}|\xi m) + O(mb) + O(\rho kb)) \tag{3.41}$$

$$= \tau^* \cdot O\left(|\mathcal{E}_{\mathcal{C}}| + 2^b \xi m + \rho kb\right) \tag{3.42}$$

If we choose the maximum row weight of the quantizer matrix and the code matrix to be $\xi = O(1)$ and $\rho = O(1)$ respectively, with $m = O(n)$ and $b = O(1)$ as we have above, the time complexity for decoding will be linear in the input size n and linear in the source model complexity $|\mathcal{E}_{\mathcal{C}}|$, ie. of $O(n + |\mathcal{E}_{\mathcal{C}}|)$, noting that $|\mathcal{E}_{\mathcal{C}}|$ is at most $O(n^2)$ for any source models.

3.5.4 Space Complexity of Decode

The space complexity of the decoder can be determined by analyzing the number of messages being passed in each iteration of message passing. We note that each message can be represented in $O(1)$, as we will see in the following chapters. Since a pair of messages is passed for each edge, we can simply count the number of edges. The source subgraph has $O(|\mathcal{E}_{\mathcal{C}}|)$ edges. The quantizer subgraph has $O(\xi m)$ edges. The translator subgraph has $O(mb)$ edges. The code subgraph has $O(\rho kb)$ edges.

Hence, the space complexity of the decoder is

$$O(|\mathcal{E}_{\mathcal{C}}| + \xi m + mb + \rho kb) \tag{3.43}$$

which is again linear in the input size n and linear in the source model complexity $|\mathcal{E}_{\mathcal{C}}|$, ie. of $O(n + |\mathcal{E}_{\mathcal{C}}|)$, at most $O(n^2)$ for any source models.

3.5.5 Communication Costs

We now analyze the communication costs between the Encode and Decode procedures. The Decode procedure needs the compressed bit stream x^{kb}, the doped bits $\mathrm{z}_{\mathscr{D}}$, the source model $p_{\boldsymbol{s}^n}(\cdot)$, the quantizer matrix Q, the translator $t(\cdot)$, and the code matrix H for correct decoding.

We first note that the source model $p_{\boldsymbol{s}^n}(\cdot)$ is information available on the decoder side, not information that is sequence dependent. Next, the translator $t(\cdot)$ can chosen to be a standardized code which does not need to be communicated from the encoder to the decoder. In addition, the randomly generated Q and H matrices can in practice be communicated to the decoder by sending the random seeds used to generate the (pseudo-)random matrices, which are of length $O(1)$, assuming that the decoder has the same matrix generating algorithm. This leaves the compressed stream x^{kb} and the doped bits $\mathrm{z}_{\mathscr{D}}$ to be sent. The total communication costs is thus

$$O(kb) + O(|\mathscr{D}|) = O(rn) \tag{3.44}$$

3.6 Summary

In this chapter, we gave an overview to the different components of the compression architecture, namely (i) a source model, (ii) a quantizer function, (iii) a translator function, and (iv) a code function. We then proceeded to give the specific choice of each of the components we described, with linear transform followed by uniform scalar quantization as the quantizer function, gray code as the translator function, and LDPC encoding as the code function. With this specific implementation, we provided a graphical model description, and we presented the loopy belief propagation algorithm with the message passing equations for Decode. We concluded by analyzing the time complexity of the Encode and the Decode algorithms.

In the following chapters, we shall explore each of the components of the algorithm in more detail and explain the different approximations, optimizations, and design choices of the implementation of the general architecture. Chapter 4 will be devoted to the code and translator components as well as doping, Chapter 5 will be devoted to the choice of quantizers, and Chapter 6 will be devoted to the discussion of source modeling. We will then present the results of this compression scheme in Chapter 7.

4

Code and Translator Structure

While flexibility and modularity is the main priority of the general architecture of the compression algorithm, practicality in terms of compression rate and computing resources are also important concerns for our algorithm. For the algorithm to be practical, tuning the set of codes $\mathscr{H}(mb, kb)$ is indispensable. In this chapter, we shall discuss the details of the choice of codes in Section 4.1, then introduce different translators in Section 4.2, before discussing different doping schemes in Section 4.3. Finally, we present the details of the message passing equations in Section 4.4.

4.1 Code Parameters

In Section 2.2.2, we introduced the concept of linear codes for channel coding, including low-density parity-check (LDPC) codes. LDPC codes are of particular interest to us, given that its sparse nature will significantly bring down the coding complexity. In Section 2.2.3, we briefly discussed using LDPC codes for compression, noting the inherent duality of the channel coding problem and the compression problem.

LDPC code optimization is an active field of research itself and is beyond the scope of this work; nevertheless, in this section, we introduce the parameters to tuning an LDPC code.

4.1.1 Row Weight and Column Weight

An LDPC code has two associated parameters $\varrho(\cdot)$ and $\varrho'(\cdot)$ known as degree distributions. The *row degree distribution* $\varrho(\cdot)$ is a probability distribution from which the row weights ρ of each row of the LDPC matrix H are drawn; similarly, the *column degree distribution* $\varrho'(x)$ is a probability distribution from which the column weights ρ' of each column of the LDPC matrix H are drawn.

4.1.2 Four Cycles in LDPC

Since the LDPC code is randomly generated, it is very likely that in its graphical representation, two hash nodes $\mathfrak{X}_a$, $\mathfrak{X}_b$ are each connected to the same two translated bit

nodes $\mathcal{Z}_i$, $\mathcal{Z}_j$, which is expressed graphically as a four cycle in the factor graph. Four cycles have the potential of adding redundant information to the hashed sequence x^{kb}, whose effects on the compression rate are unclear. In addition, successful decoding of a probabilistic graphical model by loopy belief propagation depends to a large extent the locality of the computation. With cycles of a small length, this assumption may break down and may affect the performance of our loopy BP decoder.

Given this, we experimented with removing four cycles in the randomly generated code, an operation supported by an off-the-shelf LDPC code generator [29]. We will discuss the effects of four cycles on rate performance in Section 8.1.2.

4.2 Translator Choices

In Sections 3.1.3 and 3.3.3, we described the concept of a translator and its associated graphical model. The translator, as its name suggests, translates symbols from the alphabet $\mathcal{U} = \{-2^{b-1}, \ldots, -1, 0, 1, \ldots, 2^{b-1} - 1\}$ to the alphabet $\mathbb{Z}_2 = \{0, 1\}$. In Section 3.2, we made a choice to use the b-bit Gray code. We now describe two choices of translators: (i) the b-bit standard binary code and (ii) the b-bit Gray code.

4.2.1 Standard Binary Code

The most intuitive way to encode a symbol from the alphabet

$$\mathcal{U} = \{-2^{b-1}, \ldots, -1, 0, 1, \ldots, 2^{b-1} - 1\} \tag{4.1}$$

is with the b-bit standard binary code. There are two choice that can be made to code the negative numbers:

(i) two's complement, where the first bit is used as a sign bit to represent -2^b, or
(ii) add 2^{b-1} to each symbol to make the sequence non-negative, on which we then use the unsigned binary representation.

4.2.2 Gray Code

Gray code is another way to encode $\mathcal{U} = \{-2^{b-1}, \ldots, -1, 0, 1, \ldots, 2^{b-1} - 1\}$ with b-bits. Gray code has the special property that adjacent symbols in $\mathcal{U}$ have a Hamming distance of exactly 1 in its coded $\mathbb{Z}_2$ representation.

Table 4.2 gives an illustration of a 3-bit Gray code, which codes the values $\{-4, \ldots, 3\}$. Given the 1-dimensional nature of the quantizer bins, we shall be using the PAM Gray code.

4.3 Doping

One important aspect of the translator is its ability to facilitate decoding. Since each translator will induce a different distribution on the translated bits z^{mb}, we need to

Value u	Encoding z
-4	000
-3	001
-2	011
-1	010
0	110
1	111
2	101
3	100

Table 4.2: A 3-bit Gray code that encodes $\{-4, \ldots, 3\}$ into $\mathbb{Z}_2^3$. All adjacent code words have a Hamming distance of exactly 1.

be careful in initializing message passing in these bits.

As alluded to in Section 3.4, the belief propagation decoding will likely be at a point of convergence if all messages are initialized to be trivial, meaning there will be no dynamics or updates in the messages over iterations. To introduce dynamics into the system, a potential solution could be random initializations to disrupt the initial equilibrium, but in practice such random initialization may cause messages to be contradictory or may start the algorithm in the wrong search space, making it difficult for an iterative algorithm like ours to escape.

To solve this problem, we use doping so that we can start the decoding process around the neighborhood of the real solution, at the cost of increasing the total compression rate. In this Section, we explore the details of the mechanics of doping.

From the set of $\mathcal{Z}$ nodes, we select a subset $\mathcal{D}$ of nodes to dope where $|\mathcal{D}| = n \cdot r_{\text{dope}}$. The doped bits are sent by the encoder along with the compressed bit stream to the decoder, which increases the total rate. The doped bits are interpreted as deterministic messages in the decoding process, which provides an anchor in both initialization and further iterations of belief propagation.

The choice of the bits being doped, however, will significantly affect the performance of the decoding algorithm in terms of the threshold compression rate for convergence. Here, we present three doping schemes, namely

(i) random doping, which chooses the bits to be doped randomly,
(ii) sample doping, which selects b-bit groups that each correspond to a quantized sample u_a, and
(iii) lattice doping, which selects the dope bits at regular intervals,

and intuitively discuss their expected performances.

4.3.1 Random Doping

Random doping selects the set $\mathcal{D}$ of dope bits randomly. Random doping is most in line with the model-code separation, for it requires knowledge of neither the source model, nor the translator, nor the code. Random doping is extremely flexible in that

our architecture allows for arbitrary dope rates for random doping.

Random doping, however, is suboptimal in terms of compression rate. Intuitively, the distribution of the $\mathcal{Z}$ bits is not uniform: its distribution is induced by the source model, the quantizer, and the translator. With knowledge of these components, we can choose our dope bits more wisely to decrease the total rate.

4.3.2 Sample Doping

Sample doping is on the other extreme of the spectrum of doping schemes. It chooses groups of bits, where each group corresponds to a quantized sample u_a. The choice of doped bits can be chosen with knowledge of the source, quantizer, translator, and code. However, this raises the concern of model-code separation by mixing source information into the decoder, something we actively try to avoid with our architecture.

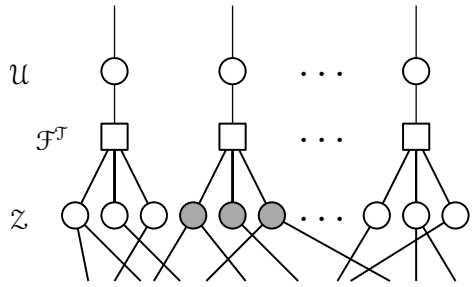


Figure 4.3: Sample doping, where all bits corresponding to whole samples are doped. The shaded nodes represent the bits doped.

4.3.3 Lattice Doping

A middle ground between random doping and sample doping is lattice doping, which does not use source information in deciding which bits to dope, but also achieves the purpose of giving each quantized sample u_a an initialization. In lattice doping, we exploit the structure of the b translator. Since every b^{th} bit in the translated sequence z^{mb} corresponds to the bit of the same significance in every sample (eg. every b^{th} starting from the first bit always corresponds to the most significant bit in each sample), we can exploit this structure by doping bits at a regular interval. Depending on the properties of the choice of translator, we may want to dope different bits, eg. dope the most significant bit vs. dope the least significant bit. Figure 4.4 illustrates this concept.

While taking advantage of the knowledge of the system, lattice doping does not require knowledge of the model, rather, only the translator. This maintains the model-code separation while achieving better performance than random doping, as we shall confirm empirically in Section 8.3.1.

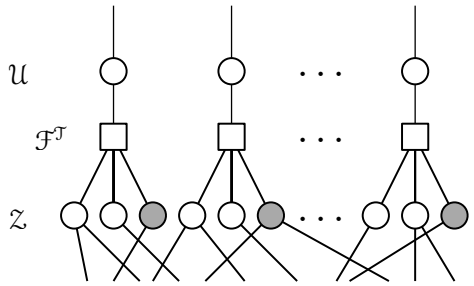


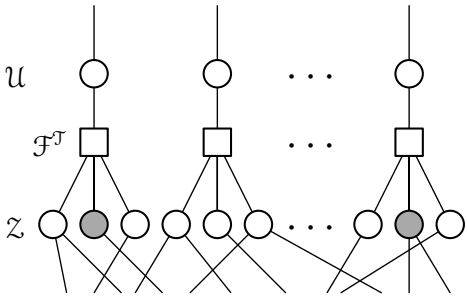
Figure 4.4: Lattice doping, where the same bit of each sample is doped. The shaded nodes represent the bits doped.

4.3.4 Lattice Doping Extensions

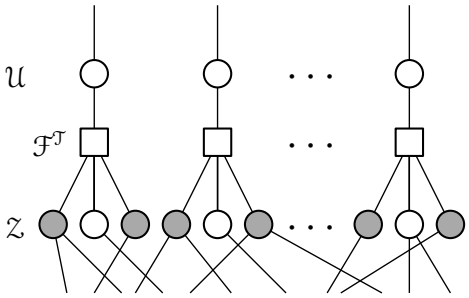
In light of the discussion on lattice doping and the structure of the translator, multiple related ideas can be explored, including (i) random sample lattice doping, which chooses sample groups u_a randomly, then dopes the same bit within each sample, which is illustrated in Figure 4.5a, and (ii) multiple lattice doping, which dope multiple bits for each b-bit group that corresponds to a sample u_a. Figure 4.5b illustrates this concept: for a 3-bit translator, we dope the first and third bits of every 3-bit group in this example.

However, it is not intuitively clear whether each of these extensions will contribute to better performance, for doping is very dependent on the choice of the translator. In particular, with multiple lattice doping, it is not clear whether an increase in dope rate r_{dope} can be compensated for by the decrease in the nominal rate r_{code} to decrease the total rate r.

We shall present and discuss the empirical performance of each of these doping choices with respect to different translators in Section 8.3.



(a) Random sample lattice doping.



(b) Multiple lattice doping.

Figure 4.5: Random sample lattice doping (left), where the same bit of random samples are doped; and multiple lattice doping (right), in this example the first and the third bis are doped. The shaded nodes represent the bits doped.

4.4 Decoding Mechanics

In this section, we expand on the general message passing equations listed in Algorithm 3.33, focusing on the implementation of message passing in the code subgraph (lines 18, 19, and 16) and the translator subgraph (lines 13 and 15).

4.4.1 Message Passing for the Code Subgraph $\mathcal{H}$

The message passing equations involving the translated sequence variable nodes $\mathcal{Z}$ are, from Algorithm 3.33,

$$m^{(\tau)}_{\mathcal{Z}_i\mathcal{F}^{\mathcal{H}}_a}(z_i) := m^{(\tau-1)}_{\mathcal{F}^{\mathcal{T}}_c\mathcal{Z}_i}(z_i) \prod_{b\in\mathcal{N}^{\mathcal{H}}_i\setminus\{a\}} m^{(\tau-1)}_{\mathcal{F}^{\mathcal{H}}_b\mathcal{Z}_i}(z_i) \tag{4.6}$$

$$m^{(\tau)}_{\mathcal{F}^{\mathcal{H}}_a\mathcal{Z}_i}(z_i) := \sum_{x_a, z_{\mathcal{N}^{\mathcal{H}}_a\setminus\{i\}}} \left(f^{\mathcal{H}}_a(z_{\mathcal{N}^{\mathcal{H}}_a}, x_a)\cdot \mathbb{1}\{x_a = \mathrm{x}_a\} \prod_{j\in\mathcal{N}^{\mathcal{H}}_a\setminus\{i\}} m^{(\tau-1)}_{\mathcal{Z}_j\mathcal{F}^{\mathcal{H}}_a}(z_j)\right) \tag{4.7}$$

$$m^{(\tau)}_{\mathcal{Z}_i\mathcal{F}^{\mathcal{T}}_c}(z_i) := \prod_{a\in\mathcal{N}^{\mathcal{H}}_i} m^{(\tau-1)}_{\mathcal{F}^{\mathcal{H}}_a\mathcal{Z}_i}(z_i) \tag{4.8}$$

We observe that the hashed bits x^{kb} comes into the equations through the term $\mathbb{1}\{x_a = \mathrm{x}_a\}$ in the $\mathcal{F}^{\mathcal{H}}$ to $\mathcal{Z}$ messages, which gives a sequence z^{mb} a probability of 0 if it does not satisfy the constraints imposed by $x^{kb} = h(z^{mb})$.

We first note that since the variables of the code graph is defined on the binary alphabet $\mathbb{Z}_2 = \{0, 1\}$, we can simplify the message passing equations by propagating the *log-likelihood ratio* (LLR). Operating in the log domain has the benefit of numerical stability, given that Equations 4.6 and 4.7 both involves taking products of small values. Hence, we define the LLR messages to be

$$\tilde{m}^{(\tau)}_{ia} := \log\left(m^{(\tau)}_{\mathcal{Z}_i\mathcal{F}^{\mathcal{H}}_a}(0)\right) - \log\left(m^{(\tau)}_{\mathcal{Z}_i\mathcal{F}^{\mathcal{H}}_a}(1)\right) \tag{4.9}$$

$$\tilde{m}^{(\tau)}_{ai} := \log\left(m^{(\tau)}_{\mathcal{F}^{\mathcal{H}}_a\mathcal{Z}_i}(0)\right) - \log\left(m^{(\tau)}_{\mathcal{F}^{\mathcal{H}}_a\mathcal{Z}_i}(1)\right) \tag{4.10}$$

and we re-write the incoming messages from the translator graph as

$$\tilde{\phi}^{(\tau)}_i := \log\left(m^{(\tau)}_{\mathcal{F}^{\mathcal{T}}_c\mathcal{Z}_i}(0)\right) - \log\left(m^{(\tau)}_{\mathcal{F}^{\mathcal{T}}_c\mathcal{Z}_i}(1)\right) \tag{4.11}$$

where we use tilde (eg. $\tilde{m}$) to denote the LLR messages.

With these notations, we now present the LLR messages along the edges illustrated in Figure 4.26. We first take the log of Equation 4.6 and get the $\mathcal{Z}$ to $\mathcal{F}^{\mathcal{H}}$ LLR messages to be

$$\tilde{m}^{(\tau)}_{ia} = \tilde{\phi}^{(\tau-1)}_i + \sum_{b\in\mathcal{N}^{\mathcal{H}}_i\setminus\{a\}} \tilde{m}^{(\tau-1)}_{bi} \tag{4.12}$$

For the $\mathcal{X}$ to $\mathcal{F}^{\mathcal{H}}$ LLR messages, we note by the normalization property that

$$m^{(\tau)}_{\mathcal{F}^{\mathcal{H}}_a \mathcal{Z}_i}(0) + m^{(\tau)}_{\mathcal{F}^{\mathcal{H}}_a \mathcal{Z}_i}(1) = 1 \tag{4.13}$$

$$m^{(\tau)}_{\mathcal{Z}_i \mathcal{F}^{\mathcal{H}}_a}(0) + m^{(\tau)}_{\mathcal{Z}_i \mathcal{F}^{\mathcal{H}}_a}(1) = 1 \tag{4.14}$$

and by algebraic manipulation of Equations 4.9 and 4.10 we get

$$m^{(\tau)}_{\mathcal{F}^{\mathcal{H}}_a \mathcal{Z}_i}(0) = \frac{\exp\{\tilde{m}_{ai}\}}{1+\exp\{\tilde{m}_{ai}\}} \qquad m^{(\tau)}_{\mathcal{F}^{\mathcal{H}}_a \mathcal{Z}_i}(1) = \frac{1}{1+\exp\{\tilde{m}_{ai}\}} \tag{4.15}$$

$$m^{(\tau)}_{\mathcal{Z}_i \mathcal{F}^{\mathcal{H}}_a}(0) = \frac{\exp\{\tilde{m}_{ia}\}}{1+\exp\{\tilde{m}_{ia}\}} \qquad m^{(\tau)}_{\mathcal{Z}_i \mathcal{F}^{\mathcal{H}}_a}(1) = \frac{1}{1+\exp\{\tilde{m}_{ia}\}} \tag{4.16}$$

which gives

$$m^{(\tau)}_{\mathcal{F}^{\mathcal{H}}_a \mathcal{Z}_i}(0) - m^{(\tau)}_{\mathcal{F}^{\mathcal{H}}_a \mathcal{Z}_i}(1) = \tanh \frac{\tilde{m}^{(\tau)}_{ai}}{2} \tag{4.17}$$

$$m^{(\tau)}_{\mathcal{Z}_i \mathcal{F}^{\mathcal{H}}_a}(0) - m^{(\tau)}_{\mathcal{Z}_i \mathcal{F}^{\mathcal{H}}_a}(1) = \tanh \frac{\tilde{m}^{(\tau)}_{ia}}{2} \tag{4.18}$$

where $\tanh(x) := \frac{e^x - e^{-x}}{e^x + e^{-x}}$ is the hyperbolic tangent function.

Next, we rewrite Equation 4.7 as

$$m^{(\tau)}_{\mathcal{F}^{\mathcal{H}}_a \mathcal{Z}_i}(z_i) \tag{4.19}$$

$$= \sum_{x_a, z_{\mathcal{N}^{\mathcal{H}}_a \setminus \{i\}}} \left(f^{\mathcal{H}}_a(z_{\mathcal{N}^{\mathcal{H}}_a}, x_a) \cdot \mathbb{1}\{x_a = \mathrm{x}_a\} \prod_{j \in \mathcal{N}^{\mathcal{H}}_a \setminus \{i\}} m^{(\tau-1)}_{\mathcal{Z}_j \mathcal{F}^{\mathcal{H}}_a}(z_j) \right) \tag{4.20}$$

$$= \sum_{z_{\mathcal{N}^{\mathcal{H}}_a \setminus \{i\}}} \left(\mathbb{1}\left\{ \left(\mathrm{x}_a \oplus z_i \oplus \bigoplus_{j \in \mathcal{N}^{\mathcal{H}}_a \setminus \{i\}} z_j \right) = 0 \right\} \prod_{j \in \mathcal{N}^{\mathcal{H}}_a \setminus \{i\}} m^{(\tau-1)}_{\mathcal{Z}_j \mathcal{F}^{\mathcal{H}}_a}(z_j) \right) \tag{4.21}$$

and taking the difference, we have

$$m^{(\tau)}_{\mathcal{F}^{\mathcal{H}}_a \mathcal{Z}_i}(0) - m^{(\tau)}_{\mathcal{F}^{\mathcal{H}}_a \mathcal{Z}_i}(1) \tag{4.22}$$

$$= (1 - 2 \cdot \mathbb{1}\{\mathrm{x}_a = 1\}) \left(\prod_{j \in \mathcal{N}^{\mathcal{H}}_a \setminus \{i\}} (1 - 2 \cdot \mathbb{1}\{z_j = 1\}) \cdot m^{(\tau)}_{\mathcal{Z}_j \mathcal{F}^{\mathcal{H}}_a}(z_j) \right) \tag{4.23}$$

$$= (1 - 2\mathrm{x}_a) \left(\prod_{j \in \mathcal{N}^{\mathcal{H}}_a \setminus \{i\}} \left(m^{(\tau)}_{\mathcal{Z}_j \mathcal{F}^{\mathcal{H}}_a}(0) - m^{(\tau)}_{\mathcal{Z}_j \mathcal{F}^{\mathcal{H}}_a}(1) \right) \right) \tag{4.24}$$

Combining Equations 4.17, 4.18, and 4.24, we thus get the $\mathcal{X}$ to $\mathcal{F}^{\mathcal{H}}$ LLR messages

to be

$$\tilde{m}_{ai}^{(\tau)} = 2 \cdot \operatorname{atanh}\left((1-2\mathrm{x}_a) \prod_{j \in \mathcal{N}_a^{\mathcal{H}} \setminus \{i\}} \tanh \frac{\tilde{m}_{ja}^{(\tau-1)}}{2}\right) \tag{4.25}$$

where $\operatorname{atanh}(x)$ is the inverse hyperbolic tangent function.

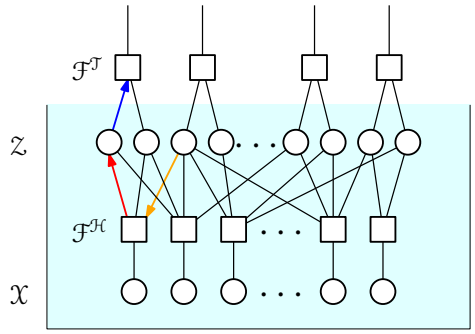


Figure 4.26: Message passing on the code subgraph: the orange arrow is a $\mathcal{Z}$ to $\mathcal{F}^{\mathcal{H}}$ message, the blue arrow is a $\mathcal{F}^{\mathcal{H}}$ to $\mathcal{Z}$ message, and the red arrow is a $\mathcal{Z}$ to $\mathcal{F}^{\mathcal{J}}$ message. Note that the information about the compressed bit stream comes into the system only through the red messages.

Finally, to compute the output $\mathcal{Z}$ to $\mathcal{F}^{\mathcal{J}}$ messages, we note that the output message in the LLR domain

$$\tilde{m}_{i\mathcal{J}}^{(\tau)} := \log\left(m_{\mathcal{Z}_i\mathcal{F}_a^{\mathcal{J}}}^{(\tau)}(0)\right) - \log\left(m_{\mathcal{Z}_i\mathcal{F}_a^{\mathcal{J}}}^{(\tau)}(1)\right) \tag{4.27}$$

can be expressed as

$$\tilde{m}_{i\mathcal{J}}^{(\tau)} := \sum_{a \in \mathcal{N}_i^{\mathcal{H}}} \tilde{m}_{ai}^{(\tau-1)} \tag{4.28}$$

by taking the log of Equation 4.8. Using the relations

$$m_{\mathcal{Z}_i\mathcal{F}_a^{\mathcal{J}}}^{(\tau)}(0) + m_{\mathcal{Z}_i\mathcal{F}_a^{\mathcal{J}}}^{(\tau)}(1) = 1 \tag{4.29}$$

$$m_{\mathcal{Z}_i\mathcal{F}_a^{\mathcal{J}}}^{(\tau)}(0) - m_{\mathcal{Z}_i\mathcal{F}_a^{\mathcal{J}}}^{(\tau)}(1) = \tanh \frac{\tilde{m}_{i\mathcal{J}}^{(\tau)}}{2} \tag{4.30}$$

we get

$$\tanh \frac{\tilde{m}_{i\mathcal{J}}^{(\tau)}}{2} = 1 - 2 \cdot m_{\mathcal{Z}_i\mathcal{F}_a^{\mathcal{J}}}^{(\tau)}(1) \tag{4.31}$$

Hence, the final output $\mathcal{Z}$ to $\mathcal{F}^{\mathcal{T}}$ messages in the linear (non-LLR) domain is

$$m^{(\tau)}_{\mathcal{Z}_i\mathcal{F}^{\mathcal{T}}_a}(1) = \frac{1}{2}\left(1 - \tanh\frac{\tilde{m}^{(\tau)}_{i\mathcal{T}}}{2}\right) \tag{4.32}$$

$$= \frac{1}{2}\left(1 - \tanh\left(\frac{1}{2}\sum_{a\in\mathcal{N}_i^{\mathcal{H}}}\tilde{m}^{(\tau-1)}_{ia}\right)\right) \tag{4.33}$$

and by the normalization property we have

$$m^{(\tau)}_{\mathcal{Z}_i\mathcal{F}^{\mathcal{T}}_a}(0) = 1 - m^{(\tau)}_{\mathcal{Z}_i\mathcal{F}^{\mathcal{T}}_a}(1) \tag{4.34}$$

4.4.2 Message Passing for the Translator Subgraph $\mathcal{T}$

The messages involving the translator factor nodes $\mathcal{F}^{\mathcal{T}}$ as illustrated in Figure 4.37 are, from Algorithm 3.33 lines 13 and 15,

$$m^{(\tau)}_{\mathcal{F}^{\mathcal{T}}_a\mathcal{U}_a}(u_a) := \sum_{z_{\mathcal{N}^{\mathcal{T}}_a}}\left(f^{\mathcal{T}}_a(u_a, z_{\mathcal{N}^{\mathcal{T}}_a})\prod_{i\in\mathcal{N}^{\mathcal{H}}_a} m^{(\tau-1)}_{\mathcal{Z}_i\mathcal{F}^{\mathcal{T}}_a}(z_i)\right) \tag{4.35}$$

$$m^{(\tau)}_{\mathcal{F}^{\mathcal{T}}_a\mathcal{Z}_a}(z_i) := \sum_{u_a, z_{\mathcal{N}^{\mathcal{T}}_a\setminus\{i\}}}\left(f^{\mathcal{T}}_a(u_a, z_{\mathcal{N}^{\mathcal{T}}_a})\cdot m^{(\tau-1)}_{\mathcal{U}_a\mathcal{F}^{\mathcal{T}}_a}(u_a)\prod_{j\in\mathcal{N}^{\mathcal{T}}_a\setminus\{i\}} m^{(\tau-1)}_{\mathcal{Z}_j\mathcal{F}^{\mathcal{T}}_a}(z_j)\right) \tag{4.36}$$

which are straight forward to implement.

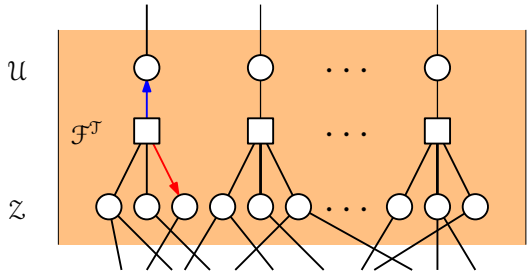


Figure 4.37: Message passing on the translator subgraph: the red arrow is a $\mathcal{F}^{\mathcal{T}}$ to $\mathcal{Z}$ message, and the blue arrow is a $\mathcal{F}^{\mathcal{T}}$ to $\mathcal{U}$ message.

An important note is that $z_i \in \mathbb{Z}_2$ and $u_a \in \mathcal{U} \cong \mathbb{Z}_{2^b}$. Hence, the $\mathcal{F}^{\mathcal{T}}$ to $\mathcal{U}$ messages will be an $m \times 2^b$ table, with the $(a,k)^{\text{th}}$ entry signifying the estimated probability that the quantized sample u_a is in the particular bin $k \in \mathcal{U}$.

On the other hand, the $\mathcal{F}^{\mathcal{T}}$ to $\mathcal{Z}$ messages can be expressed as an $mb \times 1$ vector, with the i^{th} entry signifying the estimated probability that translated bit z_i is equal to 1.

4.5 Summary

In this chapter, we have discussed various choices of code and translator selections. We introduced the different methods of doping, the results of which we will discuss in Chapter 8. Finally, we provided the implementation details of message passing in the code and translator subgraphs which solve the practical concern of numerical stability.

5

Quantizer Structure

A quantizer is a function that takes in a continuous value or vector and outputs a discrete value or vector. As described in Section 3.1.2, quantization is necessary for any lossy compression system, and the quantizer holds a central place in our architecture by implicitly controlling the rate-distortion trade-off, as we shall see in the following sections.

We first describe two ideas for the structure of our quantizer in Section 5.1. Then, we describe our particular choice of quantizer and elaborate on the message passing equations associated with our choice of quantizer in Section 5.2, detailing the approximations we use to keep the algorithm efficient and tractable. We note the flexibility of the architecture in allowing for component-wise optimizations whose structural changes do not affect that of the other components.

5.1 Quantizer Selection

Here, first we present two quantizers which can be adapted into the general architecture of graphical model decoding. In particular, the sparse structure of these quantizers allow for efficient computation and thus decoding.

5.1.1 Uniform Scalar Quantizer

The most simple quantizer would be the standard uniform scalar quantizer

$$q_{\text{us}}(x) = \lfloor x \rfloor \tag{5.1}$$

which simply rounds down the input value. The corresponding quantizer subgraph would be a set of n non-interacting nodes with edges connecting the source and the translator.

While the simplicity of the quantizer allows for efficient computation, there are two immediate issues that arise: (i) not taking advantage of the correlations the sample values, and (ii) the lack of flexibility to provide different degrees of refinement. While

the first issue of not using correlation does impact performance, it turns out this is not too much of a concern, with two reasons. First, the code $h(\cdot)$ already takes advantage of underlying correlation, so intuitively the actual performance loss would be mitigated. Second, using source correlation in the quantizer design, which is a part of the Encode algorithm, violates the code-model separation, as it uses knowledge of the model in the process of encoding. The more relevant concern is thus the second issue, the rigidity of the quantizer.

5.1.2 Low Density Hashing Quantizer

With the above concerns in mind, we shall use a *low density hashing quantizer*, which gives the quantizer more flexibility in terms of precision, but without using any knowledge about the correlation or variance of the underlying source. One thing should be noted, that all quantizers need to have an estimate for the mean of the sequence it is quantizing to provide reasonable performance, for without some knowledge of the mean, it would be impossible to even device a quantizer with a bounded length output.

As alluded to in Section 3.2, a low density hashing quantizer resembles an LDPC code in structure, as suggested by [25]. In particular, we shall do a sparse linear transform of the source sequence before applying the standard uniform scalar quantizer, mathematically defined by

$$\mathrm{u}^m = q_{\mathrm{us}}(Q\mathrm{s}^n + Q_0) \tag{5.2}$$

where $Q \in \mathbb{R}^{m\times n}$ is a sparse matrix with maximum row weight ξ, and $Q_0 \in \mathbb{R}^m$ is an offset vector to adjust for the mean of the sequence. We note that the uniform scalar quantizer is a special case of the low density hashing quantizer, with $Q = I$ and $Q_0 = 0$.

Note that linear transform is not the only operation possible. While maintaining the low density structure, the function $q_a(\cdot)$ for factor a can be changed. In the case of a linear transform, $q_a(\cdot)$ can be written as

$$q_a(s; Q, Q_0) = \left\lfloor \sum_{i\in\mathcal{N}_a^{\Omega}} Q_{ai}s_i + Q_{0a} \right\rfloor \tag{5.3}$$

Other functions can replace this particular choice of $q_a(\cdot)$ and the general structure of the algorithm will not change, although the message passing equations implementation will be dependent on the choice of $q_a(\cdot)$.

5.1.2.1 Tuning m and the Sub 1-Bit Regime

While m provides another parameter for fine-tuning the compression rate, either with $m < n$ for compression for $m > n$ for better precision, often times it is not necessary, for the code $h(\cdot)$ and the translator $t(\cdot)$ already serves similar functions. Therefore, generally we will keep $m = n$. The only scenario in which the value of m plays a

role is in the sub 1-bit regime, which has a compression rate of $r < 1$. In this case, the translator $t(\cdot)$ will not be able to provide any further lower precision, and the code $h(\cdot)$ will find it difficult to provide further lower compression rate below 1 with a translation bit length of 1.

One approach for the sub 1-bit regime would be to simply discard some translated bits z^{mb} and hash the rest, thereby achieving the linear time-sharing bound between entropy and 0-rate, a method proposed by Ancheta [1]. On the other hand, we can change the m parameter to compression the source with the quantizer before passing it onto the translator and code components. This operation, however, is still not too well understood and more research needs to be done with compressing in the sub 1-bit regime.

5.1.2.2 Practical Concerns and Choice of Q

Of the various configurations of the low density hashing quantizer that we experimented with, we find the most success in having Q as a diagonal matrix, meaning a quantizer graph with a set of non-interacting nodes. This, however, is different from the quantizer presented in Section 5.1.1 in that the quantizing width w of each region, instead of being uniformly 1, can be different for each element, as defined as the inverse of the diagonal entries of the matrix Q. By adjusting the width, we can fine tune the rate-distortion trade-off: a larger width means a smaller entropy rate in the quantized sequence u^m hence more compression, but at a cost of a less precise reconstruction; while a smaller width gives more entropy to u^m (hence less compression), but a more precise reconstruction. Note that regardless of the source variance, this scheme provides a level of performance parallel to the rate-distortion curve, hence this quantizer is source agnostic. To achieve a particular compression rate or a particular distortion though, the width w of the quantizer will need to be chosen with the knowledge of the source variance, something that no system can avoid.

Despite the simple structure we chose for Q as a diagonal matrix, the following development of the message passing equations is general for any quantizer that follows the linear transform then quantize pattern, ie. a general Q and Q_0.

5.1.2.3 Q_0 and the Sub 1-Bit Regime

Given that we choose Q to be a diagonal matrix, we notice that if we use only one bit, with Q_0 being the mean of the source, we transform the source sequence into a binary sequence whose marginal is $\mathbf{Bern}\left(\frac{1}{2}\right)^n$. If the source is iid, then the sequence will actually be $\mathbf{Bern}\left(\frac{1}{2}\right)^n$. We know that $\mathbf{Bern}\left(\frac{1}{2}\right)$ is the hardest binary source to compress: in fact, given that its entropy is 1, it will need 1 bit per sample to represent, meaning that the $\mathbf{Bern}\left(\frac{1}{2}\right)$ source is incompressible. For other values of p in a $\mathbf{Bern}(p)$ sequence, compression is possible as the entropy is less than 1.

With this observation, we see that another way to achieve performance in the sub 1-bit regime will be to set Q_0 to be different from the mean, hence inducing a non-$\mathbf{Bern}\left(\frac{1}{2}\right)$ sequence and giving less entropy to the translated sequence. On the

other extreme, setting Q_0 to ∞ (or $-\infty$) causes the translated sequence to be all 0's (or 1's), which gives us 0 rate. With this extreme, the MSE will be the mean of the sequence, which has expected distortion the same as the variance of the source, giving a SQNR of 0.

5.2 Decoding Mechanics

As described above, a non-intersecting quantizer graph is a special case of the low density hashing quantizer, with the quantizer matrix Q being diagonal and the maximum row weight of Q as $\xi = 1$.

As described in Algorithm 3.33 lines 6, 7, 9, and 10, the message passing equations on the quantizer nodes $\mathcal{F}^{\mathcal{Q}}$ (reproduced below) are

$$m^{(\tau)}_{\mathcal{S}_i \mathcal{F}^{\mathcal{Q}}_a}(s_i) := \prod_{b \in \mathcal{N}^{\mathcal{Q}}_i \setminus \{a\}} m^{(\tau-1)}_{\mathcal{F}^{\mathcal{Q}}_b \mathcal{S}_i}(s_i) \prod_{j \in \mathcal{N}^{\mathcal{C}}_i} m^{(\tau-1)}_{\mathcal{S}_j \mathcal{S}_i}(s_i) \tag{5.4}$$

$$m^{(\tau)}_{\mathcal{F}^{\mathcal{Q}}_a \mathcal{U}_a}(u_a) := \int_{s_{\mathcal{N}^{\mathcal{Q}}_a}} \left(f^{\mathcal{Q}}_a(s_{\mathcal{N}^{\mathcal{Q}}_a}, u_a) \prod_{i \in \mathcal{N}^{\mathcal{Q}}_a} m^{(\tau-1)}_{\mathcal{S}_i \mathcal{F}^{\mathcal{Q}}_a}(s_i) \right) \tag{5.5}$$

$$m^{(\tau)}_{\mathcal{U}_a \mathcal{F}^{\mathcal{Q}}_a}(u_a) := m^{(\tau-1)}_{\mathcal{F}^{\mathcal{T}}_a \mathcal{U}_a}(u_a) \tag{5.6}$$

$$m^{(\tau)}_{\mathcal{F}^{\mathcal{Q}}_a \mathcal{S}_i}(s_i) := \sum_{u_a} \int_{s_{\mathcal{N}^{\mathcal{Q}}_a \setminus \{i\}}} \left(f^{\mathcal{Q}}_a(s_{\mathcal{N}^{\mathcal{Q}}_a}, u_a) \cdot m^{(\tau-1)}_{\mathcal{U}_a \mathcal{F}^{\mathcal{Q}}_a}(u_a) \prod_{j \in \mathcal{N}^{\mathcal{Q}}_a \setminus \{i\}} m^{(\tau-1)}_{\mathcal{S}_j \mathcal{F}^{\mathcal{Q}}_a}(s_j) \right) \tag{5.7}$$

and the messages among the source nodes $\mathcal{S}$ are (Algorithm 3.33.4)

$$m^{(\tau)}_{\mathcal{S}_i \mathcal{S}_j}(s_j) := \int_{s_i} \left(\phi^{\mathcal{C}}_i(s_i) \psi^{\mathcal{C}}_{ij}(s_i, s_j) \prod_{k \in \mathcal{N}^{\mathcal{C}}_i \setminus \{j\}} m^{(\tau-1)}_{\mathcal{S}_k \mathcal{S}_i}(s_i) \prod_{a \in \mathcal{N}^{\mathcal{Q}}_i} m^{(\tau-1)}_{\mathcal{F}^{\mathcal{Q}}_a \mathcal{S}_i}(s_i) \right) \tag{5.8}$$

We first note that the messages among the source nodes $\mathcal{S}$ are Gaussian. To maintain the messages as Gaussian after the first iteration, we would need the messages from $\mathcal{F}^{\mathcal{Q}}$ to $\mathcal{S}$ to be Gaussian. However, the $\mathcal{F}^{\mathcal{Q}}$ to $\mathcal{S}$ messages described in Equation 5.7 are integrals over regions that satisfy the binning constraints. Therefore, we propose an approximation to maintain the Gaussian nature of the messages, as described below in Section 5.2.1.

5.2.1 The $\mathcal{F}^{\mathcal{Q}}$ to $\mathcal{S}$ Messages

The integral in Equation 5.7 is taken over all variables except i, the receiving node of the message. The first part of the integrand is

$$f^{\mathcal{Q}}_a(s_{\mathcal{N}^{\mathcal{Q}}_a}, u_a) \tag{5.10}$$

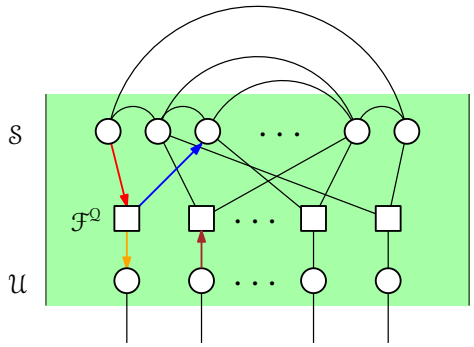


Figure 5.9: Message passing on the quantizer subgraph: the red arrow is a $\mathcal{S}$ to $\mathcal{F}^{\mathcal{Q}}$ message, the blue arrow is a $\mathcal{F}^{\mathcal{Q}}$ to $\mathcal{S}$ message, the brown arrow is a $\mathcal{U}$ to $\mathcal{F}^{\mathcal{Q}}$ message, the orange arrow is a $\mathcal{F}^{\mathcal{Q}}$ to $\mathcal{U}$ message.

which evaluates to 1 if and only if $q(s_{\mathcal{N}_a^{\mathcal{Q}}}) = u_a$, ie. the s sequences hashes to the correct bin. This implicitly defines the limits of integration. For the low density hashing quantizer for which $q(s) = \lfloor Q_0 + Qs \rfloor$, the set over which we integrate is thus

$$\left\{ s_{\mathcal{N}_a^{\mathcal{Q}} \setminus \{i\}} : u_a \leq Q_{0a} + \sum_{j \in \mathcal{N}_a^{\mathcal{Q}} \setminus \{i\}} Q_{aj} s_j < u_a + 1 \right\} \tag{5.11}$$

The second part of the integrand is the $\mathcal{U}$ to $\mathcal{F}^{\mathcal{Q}}$ message

$$m^{(\tau-1)}_{\mathcal{U}_a \mathcal{F}_a^{\mathcal{Q}}}(u_a) =: \hat{p}^{(\tau-1)}_{u_a}(u_a) \tag{5.12}$$

which represents the probability distribution over the 2^b bins, estimated at iteration $\tau - 1$. The last part of the integrand

$$\prod_{j \in \mathcal{N}_a^{\mathcal{Q}} \setminus \{i\}} m^{(\tau-1)}_{\mathcal{S}_j \mathcal{F}_a^{\mathcal{Q}}}(s_j) \tag{5.13}$$

is the product of the Gaussian $\mathcal{S}$ to $\mathcal{F}^{\mathcal{Q}}$ messages each in a different variable, hence this term is a joint independent Gaussian.

With these observations, we can re-write the message as

$$m^{(\tau)}_{\mathcal{F}_a^{\mathcal{Q}} \mathcal{S}_i}(s_i) = \sum_{u_a} \int_{s_{\mathcal{N}_a^{\mathcal{Q}} \setminus \{i\}}} \left(f_a^{\mathcal{Q}}(s_{\mathcal{N}_a^{\mathcal{Q}}}, u_a) \cdot m^{(\tau-1)}_{\mathcal{U}_a \mathcal{F}_a^{\mathcal{Q}}}(u_a) \prod_{j \in \mathcal{N}_a^{\mathcal{Q}} \setminus \{i\}} m^{(\tau-1)}_{\mathcal{S}_j \mathcal{F}_a^{\mathcal{Q}}}(s_j) \right) \tag{5.15}$$

$$= \sum_{u_a} m^{(\tau-1)}_{\mathcal{U}_a \mathcal{F}_a^{\mathcal{Q}}}(u_a) \int_{\{s : u_a \leq Qs + Q_0 < u_a + 1\}} \left(\prod_{j \in \mathcal{N}_a^{\mathcal{Q}} \setminus \{i\}} m^{(\tau-1)}_{\mathcal{S}_j \mathcal{F}_a^{\mathcal{Q}}}(s_j) \right) \tag{5.16}$$

which is a weighted sum of integrals of a joint independent Gaussian over slabs of the $\mathbb{R}^\xi$ space, which has boundaries being parallel $(\xi - 1)$-dimensional slices of the $\mathbb{R}^\xi$ space, as illustrated in Figure 5.14.

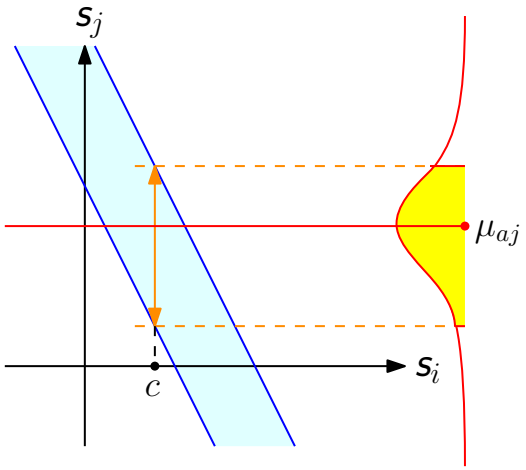


Figure 5.14: This figure illustrates the integral in Equation 5.16 for a single slab (ie. for a single u_a). The result of the integral is a function in s_i. The value of the function at c is the integral of the Gaussian over the slab, ie. the yellow area.

5.2.1.1 Gaussian Approximations for $\mathcal{F}^{\mathrm{Q}}$ to $\mathcal{S}$ Messages

We note that the integral of a joint independent Gaussian over a slab of the $\mathbb{R}^{\xi}$ space can be expressed as the difference between two Gaussian cumulative distribution functions (cdf's) whose probability distribution functions (pdf's) have the same variance but different means, as illustrated in Figure 5.17.

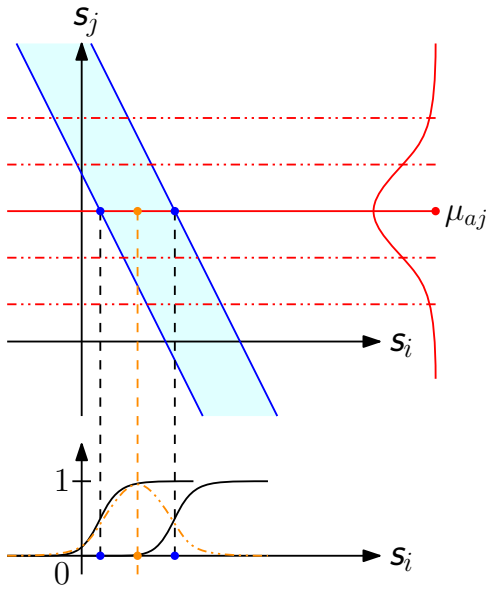


Figure 5.17: This figure illustrates integrating out s_2 over a slab over a Gaussian. The red lines represent the contour lines of the single variable Gaussian (in s_2), and the blue lines represent the boundary of the slabs we are integrating over. The boundaries are ($\xi-1$-dimensional slices of the $\mathbb{R}^{\xi}$ space. Thus, after integrating out s_2, the message is a function of s_1. We approximate the difference of the two Gaussian cdf's (black solid lines) as a single Gaussian pdf (orange dashed line).

The mean and variance of the pdf's associated with the two Gaussian cdf's are

$$\mu_{\text{cdf1},u_a} = -\frac{1}{Q_{ai}}\left(Q_{0a} + \sum_{j\in\mathcal{N}_a^{\mathcal{Q}}\setminus\{i\}} Q_{aj}\mu_{aj}\right) + \frac{1}{Q_{ai}}u_a \tag{5.18}$$

$$\mu_{\text{cdf2},u_a} = -\frac{1}{Q_{ai}}\left(Q_{0a} + \sum_{j\in\mathcal{N}_a^{\mathcal{Q}}\setminus\{i\}} Q_{aj}\mu_{aj}\right) + \frac{1}{Q_{ai}}(u_a+1) \tag{5.19}$$

and the variance is

$$\sigma^2_{\text{cdf}} = \frac{1}{Q_{ai}^2}\sum_{j\in\mathcal{N}_a^{\mathcal{Q}}\setminus\{i\}} Q_{aj}^2\sigma_{aj}^2 \tag{5.20}$$

where μ_{aj} and σ_{aj}^2 are the mean and the variance of the $\mathcal{S}$ to $\mathcal{F}^{\mathcal{Q}}$ Gaussian messages.

We note that when $\frac{1}{Q_{ai}}$ is small, the difference of two Gaussian cdf's can be well approximated with a Gaussian pdf. In particular, when $\frac{1}{Q_{ai}}$ approaches 0, the difference of the two cdf's approaches the Gaussian pdf, since by definition the pdf is the derivative of the cdf.

The center point of the difference will be the average of the two means. However, when $\frac{1}{Q_{ai}}$ is non-0, we need to adjust the variance by adding the variance of a uniform distribution with the same width $\frac{1}{Q_{ai}}$, which has the form $\frac{w^2}{12}$. Hence, if we approximate the difference with a Gaussian pdf, the parameters will be

$$\mu_{\text{diff},u_a} = -\frac{1}{Q_{ai}}\left(Q_{0a} + \sum_{j\in\mathcal{N}_a^{\mathcal{Q}}\setminus\{i\}} Q_{aj}\mu_{aj}\right) + \frac{1}{Q_{ai}}\left(u_a + \frac{1}{2}\right) \tag{5.21}$$

$$\sigma^2_{\text{diff}} = \frac{1}{Q_{ai}^2}\sum_{j\in\mathcal{N}_a^{\mathcal{Q}}\setminus\{i\}} Q_{aj}^2\sigma_{aj}^2 + \frac{1}{12Q_{ai}^2} \tag{5.22}$$

The final message is a weighted sum of these integrals over slabs, as illustrated in Figure 5.23. With the above approximation, this will thus be a Gaussian mixture. Since we require the message to be a Gaussian, we do one final approximation by approximating a Gaussian mixture as a single Gaussian, with its mean and variance being the mean and variance of the mixture. Thus, we pass the messages in information form as

$$m^{(\tau)}_{\mathcal{F}_a^{\mathcal{Q}}\mathcal{S}_i}(s_i) \approx \mathcal{N}^{-1}(s_i;\eta_{\mathcal{F}_a^{\mathcal{Q}}\mathcal{S}_i},\lambda_{\mathcal{F}_a^{\mathcal{Q}}\mathcal{S}_i}) = \mathcal{N}(s_i;\mu_{\mathcal{F}_a^{\mathcal{Q}}\mathcal{S}_i},\sigma^2_{\mathcal{F}_a^{\mathcal{Q}}\mathcal{S}_i}) \tag{5.24}$$

where

$$\lambda_{\mathcal{F}_a^{\mathcal{Q}}\mathcal{S}_i} = \left(\sigma^2_{\mathcal{F}_a^{\mathcal{Q}}\mathcal{S}_i}\right)^{-1} \tag{5.25}$$

$$\eta_{\mathcal{F}_a^{\mathcal{Q}}\mathcal{S}_i} = \lambda_{\mathcal{F}_a^{\mathcal{Q}}\mathcal{S}_i}\cdot\mu_{\mathcal{F}_a^{\mathcal{Q}}\mathcal{S}_i} \tag{5.26}$$

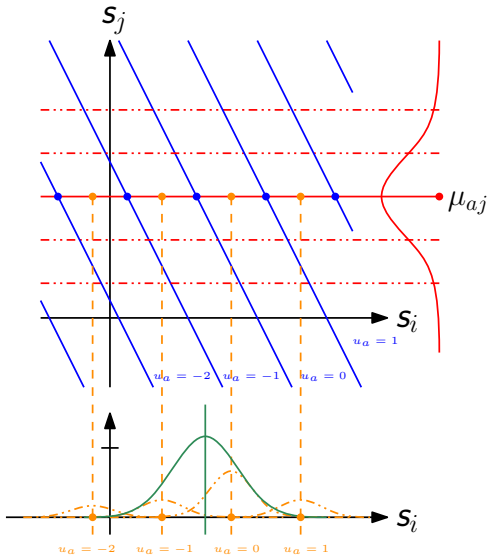


Figure 5.23: Each slab k (bin $u_a = k$) gives us a Gaussian pdf (orange), weighed by $\hat{p}_{u_a}(k)$. We approximate the Gaussian mixture as a single Gaussian (green) as $m^{(\tau)}_{\mathcal{F}^{\mathbb{Q}}_a \mathbb{S}_1}(s_1)$, the final message from quantizer node $\mathcal{F}^{\mathbb{Q}}_a$ to source node $\mathbb{S}_1$.

with

$$\mu_{\mathcal{F}^{\mathbb{Q}}_a \mathbb{S}_i} = \sum_{u_a=-2^{b-1}}^{2^{b-1}-1} \hat{p}_{u_a}(u_a) \cdot \mu_{\text{diff},u_a} \tag{5.27}$$

$$= -\frac{1}{Q_{ai}} \left(Q_{0a} + \sum_{j \in \mathcal{N}^{\mathbb{Q}}_a \setminus \{i\}} Q_{aj} \mu_{aj} \right) + \frac{1}{Q_{ai}} \left(\sum_{u_a} u_a \cdot \hat{p}^{(\tau-1)}_{u_a}(u_a) + \frac{1}{2} \right) \tag{5.28}$$

and

$$\sigma^2_{\mathcal{F}^{\mathbb{Q}}_a \mathbb{S}_i} = \sigma^2_{\text{diff}} + \sum_{u_a=-2^{b-1}}^{2^{b-1}-1} \hat{p}_{u_a}(u_a) \cdot \mu^2_{\text{diff},u_a} - \left(\sum_{u_a=-2^{b-1}}^{2^{b-1}-1} \hat{p}_{u_a}(u_a) \cdot \mu_{\text{diff},u_a} \right)^2 \tag{5.29}$$

$$= \frac{1}{Q^2_{ai}} \sum_{j \in \mathcal{N}^{\mathbb{Q}}_a \setminus \{i\}} Q^2_{aj} \sigma^2_{aj} + \frac{1}{12 Q^2_{ai}} + \frac{1}{Q^2_{ai}} \left(\sum_{u_a} u^2_a \cdot \hat{p}_{u_a}(u_a) - \left(\sum_{u_a} u_a \cdot \hat{p}_{u_a}(u_a) \right)^2 \right) \tag{5.30}$$

where

$$\hat{p}_{u_a}(u_a) := m^{(\tau-1)}_{\mathcal{U}_a \mathcal{F}^{\mathbb{Q}}_a}(u_a) \tag{5.31}$$

as defined in Equation 5.12.

5.2.2 The $\mathcal{S}$ to $\mathcal{F}^{\mathcal{Q}}$ Messages

With the $\mathcal{F}^{\mathcal{Q}}$ to $\mathcal{S}$ messages approximated as Gaussians, it follows that the messages among $\mathcal{S}$ (Equation 5.8) are also Gaussian. Hence, the messages from $\mathcal{S}$ to $\mathcal{F}^{\mathcal{Q}}$ (Equation 5.4) are also Gaussian, and thus Equation 5.4 can be expressed as

$$m^{(\tau)}_{\mathcal{S}_i\mathcal{F}^{\mathcal{Q}}_a}(s_i) := \prod_{b\in\mathcal{N}^{\mathcal{Q}}_i\setminus\{a\}} m^{(\tau-1)}_{\mathcal{F}^{\mathcal{Q}}_b\mathcal{S}_i}(s_i) \prod_{j\in\mathcal{N}^{\mathcal{C}}_i} m^{(\tau-1)}_{\mathcal{S}_j\mathcal{S}_i}(s_i) \tag{5.32}$$

$$= \prod_{b\in\mathcal{N}^{\mathcal{Q}}_i\setminus\{a\}} \mathcal{N}^{-1}\left(s_i;\eta^{(\tau-1)}_{\mathcal{F}^{\mathcal{Q}}_b\mathcal{S}_i},\lambda^{(\tau-1)}_{\mathcal{F}^{\mathcal{Q}}_b\mathcal{S}_i}\right) \prod_{j\in\mathcal{N}^{\mathcal{C}}_i} \mathcal{N}^{-1}\left(s_i;\eta^{(\tau-1)}_{\mathcal{S}_j\mathcal{S}_i},\lambda^{(\tau-1)}_{\mathcal{S}_j\mathcal{S}_i}\right) \tag{5.33}$$

$$= \mathcal{N}^{-1}(s_i;\eta^{(\tau)}_{\mathcal{S}_i\mathcal{F}^{\mathcal{Q}}_a},\lambda^{(\tau)}_{\mathcal{S}_i\mathcal{F}^{\mathcal{Q}}_a}) \tag{5.34}$$

where

$$\eta^{(\tau)}_{\mathcal{S}_i\mathcal{F}^{\mathcal{Q}}_a} = \sum_{b\in\mathcal{N}^{\mathcal{Q}}_i\setminus\{a\}} \eta^{(\tau-1)}_{\mathcal{F}^{\mathcal{Q}}_b\mathcal{S}_i} + \sum_{j\in\mathcal{N}^{\mathcal{C}}_i} \eta^{(\tau-1)}_{\mathcal{S}_j\mathcal{S}_i} \tag{5.35}$$

$$\lambda^{(\tau)}_{\mathcal{S}_i\mathcal{F}^{\mathcal{Q}}_a} = \sum_{b\in\mathcal{N}^{\mathcal{Q}}_i\setminus\{a\}} \lambda^{(\tau-1)}_{\mathcal{F}^{\mathcal{Q}}_b\mathcal{S}_i} + \sum_{j\in\mathcal{N}^{\mathcal{C}}_i} \lambda^{(\tau-1)}_{\mathcal{S}_j\mathcal{S}_i} \tag{5.36}$$

5.2.3 The $\mathcal{F}^{\mathcal{Q}}$ to $\mathcal{U}$ Messages

The messages from $\mathcal{F}^{\mathcal{Q}}$ to $\mathcal{U}$, reproduced from Equation 5.5,

$$m^{(\tau)}_{\mathcal{F}^{\mathcal{Q}}_a\mathcal{U}_a}(u_a) := \int_{s_{\mathcal{N}^{\mathcal{Q}}_a}} \left(f^{\mathcal{Q}}_a(s_{\mathcal{N}^{\mathcal{Q}}_a},u_a) \prod_{i\in\mathcal{N}^{\mathcal{Q}}_a} m^{(\tau-1)}_{\mathcal{S}_i\mathcal{F}^{\mathcal{Q}}_a}(s_i) \right) \tag{5.37}$$

are integrals of a Gaussian, with integration limits implicitly defined by the quantizer factors $f^{\mathcal{Q}}_a(s_{\mathcal{N}^{\mathcal{Q}}_a},u_a)$, as illustrated in Figure 5.40. For each u_a, the integral can thus be expressed in terms of the standard Gaussian cdf. To simplify our notation, let

$$\hat{p}_{u_a}(u_a) := m^{(\tau)}_{\mathcal{F}^{\mathcal{Q}}_a\mathcal{U}_a}(u_a) \tag{5.38}$$

$$\mathcal{N}(s,\mu_a,\Sigma_a) := \prod_{i\in\mathcal{N}^{\mathcal{Q}}_a} m^{(\tau-1)}_{\mathcal{S}_i\mathcal{F}^{\mathcal{Q}}_a}(s_i) \tag{5.39}$$

where μ_a is the mean vector and Σ_a is the diagonal covariance matrix of the product.

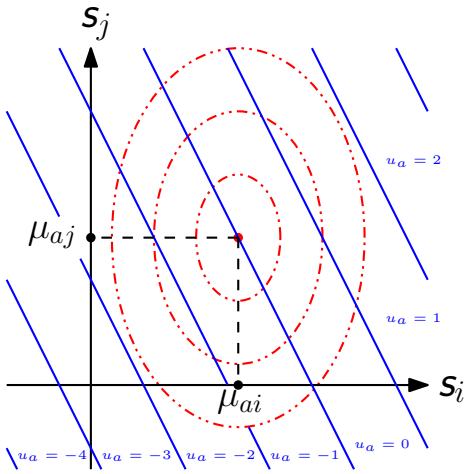


Figure 5.40: This figure illustrates integrating over parallel slabs over a joint independent Gaussian, with $\xi = 2$. The red lines represent the 3-sigma ellipsoid of the joint Gaussian, and the blue lines represent the boundary of the slabs.

Letting Q_a denote the a^{th} row of Q, we can rewrite Equation 5.37 as

$$\hat{p}_{u_a}(u_a) = \int_{\{s: u_a \le Q_a s + Q_{0a} < u_a + 1\}} \mathcal{N}(s, \mu_a, \Sigma_a) \tag{5.41}$$

$$= \int_{\{s: Q_a s + Q_{0a} - u_a \ge 0\}} \mathcal{N}(s, \mu_a, \Sigma_a) - \int_{\{s: Q_a s + Q_{0a} - (u_a+1) \ge 0\}} \mathcal{N}(s, \mu_a, \Sigma_a) \tag{5.42}$$

$$\tag{5.43}$$

Noting each the integral is taken over a half-space, we do a change in coordinates to convert it into the standard single variable Gaussian form, the integrals for which can be expressed in terms of the standard Gaussian cdf $\Phi(\cdot)$:

$$\int_{\{s: Q_a s + Q_{0a} - u_a \ge 0\}} \mathcal{N}(s, \mu_a, \Sigma_a) \tag{5.44}$$

$$= \int_{\left\{s': \Sigma_a^{1/2} Q_a s' + (Q_a \mu_a + Q_{0a} - u_a) \ge 0\right\}} \mathcal{N}(s', 0, I_n) \tag{5.45}$$

$$= \int_{\{s'': c_a \le s'' < \infty\}} \mathcal{N}(s'', 0, 1) \tag{5.46}$$

$$= 1 - \Phi(c_a) \tag{5.47}$$

where the scalar c_a is

$$c_a := -\frac{Q_a\mu_a + Q_{0a} - u_a}{\left\|\Sigma_a^{1/2}Q_a\right\|_2} \tag{5.48}$$

with $\|\cdot\|_2$ denoting the L^2 norm of a vector and $\Sigma_a^{1/2}$ denoting the diagonal matrix obtained by taking the square root of the entries of the diagonal matrix Σ_a. Hence, we have

$$m^{(\tau)}_{\mathcal{F}_a^{\mathcal{Q}}\mathcal{U}_a}(u_a) = \Phi\left(-\frac{Q_a\mu_a + Q_{0a} - (u_a+1)}{\left\|\Sigma_a^{1/2}Q_a\right\|_2}\right) - \Phi\left(-\frac{Q_a\mu_a + Q_{0a} - u_a}{\left\|\Sigma_a^{1/2}Q_a\right\|_2}\right) \tag{5.49}$$

as the $\mathcal{F}^{\mathcal{Q}}$ to $\mathcal{U}$ messages.

5.3 Summary

In this chapter, we have discussed two types of quantizers and their advantages and disadvantages, in addition to the practical concerns in each of them. We then derived the message passing equations for our choice of low density hashing quantizer, the equations of which also works for any general quantizer with the transform-then-quantize pattern. We note that with the structure of our general architecture, with the quantizer being one modular part of the system, the quantizer itself can be optimized and replaced individually without affecting any other parts of the system, a property that is rare among existing lossy compression algorithms. This modularity and flexibility of the architecture renders it ready for change.

6

Source Modeling

As described in Section 3.1.1, the *source model* refers to the underlying probability distribution $p_{\mathbf{s}^n}$ that generates the source sequence s^n. In Section 3.3.1 we briefly discussed the graphical representation of the source model based on its factorization structure, which is formalized as the *source subgraph* portion $\mathcal{C}$ of the full graph $\mathcal{G}$. In this chapter, we will explore the source modeling details.

In Section 6.1, we first describe in details two source models, namely (i) the Gaussian iid source, and (ii) the Gauss-Markov source, presenting their respective graphical model representation. Then, in Section 6.2, we describe their message passing equations and marginalization, an operation which gives us an estimate $\hat{\mathrm{s}}^n$. We note the universal nature of the source subgraph, an immediate advantage of our architecture that allows for compression of general sources.

6.1 Graphical Representation of Source Models

Probabilistic graphical models, introduced in Section 2.3, are universal in their ability to represent any probability distributions. In particular, pairwise undirected graphical models are universal for Gaussian sources (Section 2.3.1.1). In this section, we present two source models with relatively simple graphical representations: the Gaussian iid source and the Gauss-Markov source.

6.1.1 Gaussian iid Source

A Gaussian iid source $\mathbf{N}^{-1}(0, \lambda^s I)$, with λ^s being the inverse of the variance, is distributed according to

$$p_{\mathbf{s}^n}(s^n) = \mathcal{N}^{-1}(s; 0, \lambda^s I) \propto \exp\left\{-\frac{1}{2} s^T \lambda^s I s\right\} = \prod_{i=1}^{n} \exp\left\{-\frac{1}{2}\lambda^s s_i^2\right\} \tag{6.1}$$

which is trivially a pairwise model (Section 2.3.1.1) with only *node potentials*

$$\phi_i^{\mathcal{C}}(s_i) = \exp\left\{-\frac{1}{2}\lambda^s s_i^2\right\} \tag{6.2}$$

for all i, and no edge potentials. This gives rise to the graphical model in Figure 6.3.

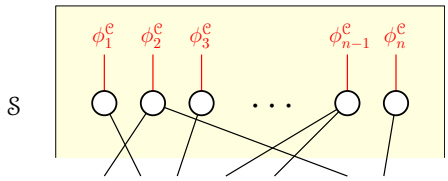


Figure 6.3: Graphical Model for the Guassian iid source, which only has node potentials $\phi_i^{\mathcal{C}}$.

6.1.2 Gauss-Markov Source

A *Gauss-Markov* source is a Markov chain of Gaussian variables generated by

$$\mathsf{s}_1 \sim \mathbf{N}^{-1}(0, \lambda_0^s) \tag{6.4}$$

$$\mathsf{s}_i = a\mathsf{s}_{i-1} + \mathsf{v}_i, \qquad \mathsf{v}_i \sim \mathbf{N}^{-1}(0, \lambda^s) \qquad \text{for } i \in \{2, \ldots, n\} \tag{6.5}$$

where a is the autocorrelation parameter, λ_0^s is the precision parameter for s_1, and λ^s is the precision parameter for v_i. The superscript s is to distinguish the parameters of the source model from the parameters of message passing, which shall be discussed in Section 6.2. The joint pdf of the Gauss-Markov source can thus be expressed as

$$p_{\mathsf{s}^n}(s^n) = p_{\mathsf{s}_1}(s_1) \cdot \prod_{i=2}^{n} p_{\mathsf{s}_i|\mathsf{s}_{i-1}}(s_i \mid s_{i-1}) \tag{6.6}$$

$$\propto \exp\left\{-\frac{1}{2}\lambda_0^s s_1^2\right\} \cdot \prod_{i=2}^{n} \exp\left\{-\frac{1}{2}\lambda^s \left(s_i - a s_{i-1}\right)^2\right\} \tag{6.7}$$

$$\begin{aligned} &= \exp\left\{-\frac{1}{2}\lambda_0^s s_1^2\right\} \cdot \prod_{i=2}^{n} \exp\left\{-\frac{1}{2}\lambda^s s_i^2\right\} \cdot \prod_{i=2}^{n} \exp\left\{-\frac{1}{2}\lambda^s a^2 s_{i-1}^2\right\} \\ &\qquad \cdot \prod_{i=2}^{n} \exp\left\{\lambda^s a s_i s_{i-1}\right\} \end{aligned} \tag{6.8}$$

$$\begin{aligned} &= \exp\left\{-\frac{1}{2}\left(\lambda_0^s + a^2\lambda^s\right) s_1^2\right\} \cdot \prod_{i=2}^{n-1} \exp\left\{-\frac{1}{2}\left(\lambda^s + a^2\lambda^s\right) s_i^2\right\} \\ &\qquad \cdot \exp\left\{-\frac{1}{2}\lambda^s s_n^2\right\} \cdot \prod_{i=2}^{n} \exp\left\{a\lambda^s s_i s_{i-1}\right\} \end{aligned} \tag{6.9}$$

$$= \prod_{i=1}^{n} \phi_i^{\mathcal{C}}(s_i) \cdot \prod_{i=2}^{n} \psi_{i-1,i}^{\mathcal{C}}(s_{i-1}, s_i) \tag{6.10}$$

which is pairwise model (Section 2.3.1.1) with node potentials $\phi_i^{\mathcal{C}}$ where

$$\phi_i^{\mathcal{C}}(s_i) \propto \begin{cases} \exp\left\{-\frac{1}{2}\left(\lambda_0^s + a^2\lambda^s\right)s_i^2\right\} & i = 1 \\ \exp\left\{-\frac{1}{2}\left(\lambda^s + a^2\lambda^s\right)s_i^2\right\} & i \in \{2,\ldots,n-1\} \\ \exp\left\{-\frac{1}{2}\lambda^s s_i^2\right\} & i = n \end{cases} \tag{6.11}$$

and edge potentials $\psi_{ij}^{\mathcal{C}}$ for $j = i-1$ where

$$\psi_{i-1,i}^{\mathcal{C}} \propto \exp\left\{a\lambda^s s_i s_{i-1}\right\} \qquad i \in \{2,\ldots,n\} \tag{6.12}$$

which gives rise to the following graphical model (Figure 6.13):

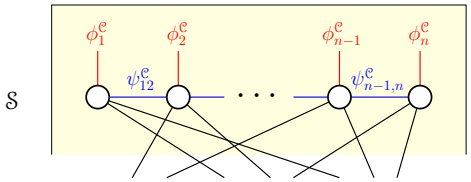


Figure 6.13: Graphical Model for the Guass-Markov source, which has node potentials $\phi_i^{\mathcal{C}}$ and edge potentials $\psi_{ij}^{\mathcal{C}}$ in the structure of a Markov chain.

6.2 Decoding Mechanics

As described in Algorithm 3.33 line 4, the message passing equations among the source nodes $\mathcal{S}$ are

$$m_{\mathcal{S}_i\mathcal{S}_j}^{(\tau)}(s_j) := \int_{s_i} \left(\phi_i^{\mathcal{C}}(s_i)\psi_{ij}^{\mathcal{C}}(s_i,s_j) \prod_{k\in\mathcal{N}_i^{\mathcal{C}}\setminus\{j\}} m_{\mathcal{S}_k\mathcal{S}_i}^{(\tau-1)}(s_i) \prod_{a\in\mathcal{N}_i^{\mathcal{Q}}} m_{\mathcal{F}_a^{\mathcal{Q}}\mathcal{S}_i}^{(\tau-1)}(s_i) \right) \tag{6.14}$$

for pairwise models. With the Gaussian approximation scheme presented in Section 5.2.1, the $\mathcal{F}^{\mathcal{Q}}$ to $\mathcal{S}$ messages $m_{\mathcal{F}_a^{\mathcal{Q}}\mathcal{S}_i}^{(\tau-1)}(s_i)$ are approximated as a Gaussian pdf. Therefore, we can use the following well known results in Gaussian message passing on the source graph.

6.2.1 Gaussian Potentials

We note that the node potentials $\phi_i^{\mathcal{C}}$ and edge potentials $\psi_{ij}^{\mathcal{C}}$ are in the quadratic exponential form for both the iid model and the Gauss-Markov model. In general, this is true for all Gaussian graphical models regardless of the underlying correlation of the variables: all node potentials and edge potentials assume the quadratic exponential form. All Gaussian graphical models can have their node and edge potentials be

written as

$$\phi_i^{\mathcal{C}}(s_i) \propto \exp\left\{-\frac{1}{2}\lambda_i^{\phi} s_i^2 + \eta_i^{\phi} s_i\right\} \tag{6.15}$$

$$\psi_{ij}^{\mathcal{C}}(s_i, s_j) \propto \exp\left\{-\lambda_{ij}^{\psi} s_i s_j\right\} \tag{6.16}$$

for some model dependent λ_i^{ϕ}, η_i^{ϕ}, and λ_{ij}^{ψ} for all nodes $\mathcal{S}_i$ and all edges $(\mathcal{S}_i, \mathcal{S}_j)$, where the superscripts are used to distinguish the parameters of the node and edge potentials from the messages parameters (Section 6.2.2).

In the case of the iid source, we have

$$\eta_i^{\phi} = 0 \tag{6.17}$$

$$\lambda_i^{\phi} = \lambda^s \tag{6.18}$$

and for the Gauss-Markov source, we have

$$\eta_i^{\phi} = 0 \tag{6.19}$$

$$\lambda_i^{\phi} = \begin{cases} \lambda_0^s + a^2\lambda^s & i = 1 \\ \lambda^s + a^2\lambda^s & i \in \{2, \ldots, n-1\} \\ \lambda^s & i = n \end{cases} \tag{6.20}$$

$$\lambda_{ij}^{\psi} = -a\lambda^s \tag{6.21}$$

6.2.2 Gaussian Message Passing

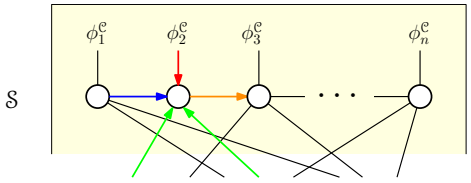


Figure 6.22: Belief propagation on the source subgraph $\mathcal{C}$. The source messages $m_{\mathcal{S}_i\mathcal{S}_j}^{(\tau)}$ (orange arrow) is computed as the integral over s_i of the product of the node potential (red arrow), neighboring edge potentials (blue arrows) except that from node $\mathcal{S}_j$, and incoming messages (green arrow).

With the potentials above, we consider the message passing equation in 6.14. We note that the integral in the expression is an marginalization operation of the joint distribution of $\boldsymbol{s}_i, \boldsymbol{s}_j$ by integrating out s_i. Parametrizing the Gaussian messages from $\mathcal{F}^{\mathcal{Q}}$ to $\mathcal{S}$ in the information form

$$m_{\mathcal{F}_a^{\mathcal{Q}}\mathcal{S}_i}^{(\tau)}(s_i) =: \mathcal{N}^{-1}(s_i; \eta_{\mathcal{F}_a^{\mathcal{Q}}\mathcal{S}_i}^{(\tau)}, \lambda_{\mathcal{F}_a^{\mathcal{Q}}\mathcal{S}_i}^{(\tau)}) \tag{6.23}$$

and applying Facts 2.9 (marginals of a joint Gaussian) and 2.14 (pointwise product

of Gaussian pdf's), we can thus derive the messages as

$$m^{(\tau)}_{\mathbb{S}_i\mathbb{S}_j}(s_j) \propto \mathcal{N}^{-1}(s_j; \eta^{(\tau)}_{\mathbb{S}_i\mathbb{S}_j}, \lambda^{(\tau)}_{\mathbb{S}_i\mathbb{S}_j}) \tag{6.24}$$

where

$$\begin{aligned}\eta^{(\tau)}_{\mathbb{S}_i\mathbb{S}_j} := -\lambda^{\psi}_{ji} &\left(\lambda^{\phi}_i + \sum_{k\in\mathcal{N}^{\mathbb{S}}_i\setminus\{j\}} \lambda^{(\tau-1)}_{\mathbb{S}_k\mathbb{S}_i} + \sum_{a\in\mathcal{N}^{\mathbb{Q}}_i} \lambda^{(\tau-1)}_{\mathcal{F}^{\mathbb{Q}}_a\mathbb{S}_i}\right)^{-1} \\ &\cdot \left(\eta^{\phi}_i + \sum_{k\in\mathcal{N}^{\mathbb{S}}_i\setminus\{j\}} \eta^{(\tau-1)}_{\mathbb{S}_k\mathbb{S}_i} + \sum_{a\in\mathcal{N}^{\mathbb{Q}}_i} \eta^{(\tau-1)}_{\mathcal{F}^{\mathbb{Q}}_a\mathbb{S}_i}\right)\end{aligned} \tag{6.25}$$

$$\lambda^{(\tau)}_{\mathbb{S}_i\mathbb{S}_j} := -\lambda^{\psi}_{ji}\left(\lambda^{\phi}_i + \sum_{k\in\mathcal{N}^{\mathbb{S}}_i\setminus\{j\}} \lambda^{(\tau-1)}_{\mathbb{S}_k\mathbb{S}_i} + \sum_{a\in\mathcal{N}^{\mathbb{Q}}_i} \lambda^{(\tau-1)}_{\mathcal{F}^{\mathbb{Q}}_a\mathbb{S}_i}\right)^{-1} \lambda^{\psi}_{ij} \tag{6.26}$$

For the iid model, the messages would be trivial, given the lack of edges in the source subgraph. For the Gauss-Markov model, the sums in Equations 9.26 and 9.27 will only consist of one term, given the Markov chain structure of the source subgraph. Message passing on the Gauss-Markov model is an instance of the famous *forward-backward algorithm* for Kalman smoothers [8].

6.2.2.1 Initialization

We initialize all messages to be trivial, ie.

$$m^{(0)}_{\mathbb{S}_i\mathbb{S}_j}(s_j) \propto \mathcal{N}^{-1}(s_j; \eta^{(0)}_{\mathbb{S}_i\mathbb{S}_j}, \lambda^{(0)}_{\mathbb{S}_i\mathbb{S}_j}) \quad \text{for } (i,j) \in \mathcal{E}_{\mathbb{S}} \tag{6.27}$$

where

$$\eta^{(0)}_{\mathbb{S}_i\mathbb{S}_j} = 0 \tag{6.28}$$

$$\lambda^{(0)}_{\mathbb{S}_i\mathbb{S}_j} = 1 \tag{6.29}$$

6.2.3 Marginalization and Convergence

At the end of each iteration τ of message passing, we estimate $\hat{\mathbb{s}}^{(\tau)}$ with the well known marginalization equation for graphical models, reproduced from Algorithm 3.33.21:

$$\hat{p}^{(\tau)}_{\boldsymbol{s}_i}(s_i) := \prod_{a\in\mathcal{N}^{\mathbb{Q}}_i} m^{(\tau)}_{\mathcal{F}^{\mathbb{Q}}_a\mathbb{S}_i}(s_i) \prod_{j\in\mathcal{N}^{\mathbb{C}}_i} m^{(\tau)}_{\mathbb{S}_j\mathbb{S}_i}(s_i) \tag{6.30}$$

which, in terms of the $\mathcal{S}$ message parameters $\eta^{(\tau)}_{\mathcal{S}_j\mathcal{S}_i}$ and $\lambda^{(\tau)}_{\mathcal{S}_j\mathcal{S}_i}$ and the $\mathcal{F}^{\mathcal{Q}}$ to $\mathcal{S}$ message parameters $\eta^{(\tau)}_{\mathcal{F}^{\mathcal{Q}}_a\mathcal{S}_i}$ and $\lambda^{(\tau)}_{\mathcal{F}^{\mathcal{Q}}_a\mathcal{S}_i}$, can be expressed as

$$\hat{p}^{(\tau)}_{\mathbf{s}_i}(s_i) \propto \mathcal{N}^{-1}(s_i, \hat{\eta}_i^{\mathrm{mrg}(\tau)}, \hat{\lambda}_i^{\mathrm{mrg}(\tau)}) \tag{6.31}$$

where

$$\hat{\eta}_i^{\mathrm{mrg}(\tau)} = \eta_i^{\phi} + \sum_{j \in \mathcal{N}_i^{\mathcal{S}}} \eta^{(\tau)}_{\mathcal{S}_j\mathcal{S}_i} + \sum_{a \in \mathcal{N}_i^{\mathcal{Q}}} \eta^{(\tau)}_{\mathcal{F}^{\mathcal{Q}}_a\mathcal{S}_i} \tag{6.32}$$

$$\hat{\lambda}_i^{\mathrm{mrg}(\tau)} = \lambda_i^{\phi} + \sum_{j \in \mathcal{N}_i^{\mathcal{S}}} \lambda^{(\tau)}_{\mathcal{S}_j\mathcal{S}_i} + \sum_{a \in \mathcal{N}_i^{\mathcal{Q}}} \lambda^{(\tau)}_{\mathcal{F}^{\mathcal{Q}}_a\mathcal{S}_i} \tag{6.33}$$

The *minimum mean squared error* (MMSE) estimate $\hat{\mathrm{s}}^{(\tau)}$ will thus be the mean of the marginal, ie.

$$\hat{\mathrm{s}}_i^{(\tau)} := \hat{\mu}_i^{\mathrm{mrg}(\tau)} = \left(\hat{\lambda}_i^{\mathrm{mrg}(\tau)}\right)^{-1} \cdot \hat{\eta}_i^{\mathrm{mrg}(\tau)} \tag{6.34}$$

We use convergence of the estimate $\hat{\mathrm{s}}$ to determine when to terminate the algorithm. In particular, we terminate at iteration τ^* and return the estimate $\hat{\mathrm{s}}^{(\tau^*)}$ when the MMSE estimates $\hat{\mathrm{s}}$ are ε-close under the L^∞ norm (the supremum norm), ie. satisfy

$$\varepsilon > \left\|\hat{\mathrm{s}}^{(\tau^*)} - \hat{\mathrm{s}}^{(\tau^*-1)}\right\|_\infty = \max_{i \in \{1,\ldots,n\}} \left|\hat{\mathrm{s}}_i^{(\tau^*)} - \hat{\mathrm{s}}_i^{(\tau^*-1)}\right| \tag{6.35}$$

for some convergence criterion ε, or for a maximum number of iterations $\tau^{\max}$. For our experiments in Chapter 7, we use $\varepsilon = 0.01$ and $\tau^{\max} = 100$.

6.3 Summary

The source model, representing the decoder's prior knowledge of the source, can be expressed as a pairwise undirected Gaussian graphical model. With our architecture, the source model forms a modular part of the decoding algorithm not entangled with any other parts of the system, and therefore gives the flexibility that, to our knowledge, no existing systems can provide.

Two of the advantages include

(i) having a universal compressor for Gaussian sources whose structure does not radically change depending on the underlying correlation, and
(ii) the ability to refine a model, hence improve the compression rate, *after* a source sequence has been compressed, when we have more information about the source to provide a better source model for the decoder. This will be discussed in more details in Section 9.2.

In addition, with the source model being represented as a graphical model, we can run inference tasks on top of the decoding algorithm, eg. to learn the parameters of the model, an architectural extension we will briefly explore in Section 9.3.

7

Compression Performance

In this chapter, we present the experimental results of applying our compression architecture to the two classes of Gaussian sources we described in Chapter 6, namely the Gaussian iid source and the Gauss-Markov source.

We first describe our experimental setup in Section 7.1. Then, in Sections 7.2 and 7.3, we present the performance of our algorithm on the Gaussian iid source and the Gauss-Markov source respectively. We compare our performance with the theoretical lower limit of compression (ie. the rate-distortion bound, Section 2.2.4), as well as with the theoretical bounds of classes of known algorithms.

7.1 Experimental Setup

For each source model, we select a set of representative parameter values for the model, which we then use to generate samples through Gibbs sampling (Section 2.3.3). For each set of model parameters, we test the architecture on different distortion levels, which are implicitly controlled by the quantizer $q(\cdot)$ and the precision parameter b, the number of bits used to represent the quantized values.

For each distortion level δ, we first generate 100 source sequences s^n. With each of these 100 sequences, we compress it with different compression rates: for each rate r, we generate 10 different random LDPC codes $h(\cdot)$ with an off-the-shelf LDPC generator [29], with uniform column weight $\rho' = 3$ and no four-cycles, and compress s into x with $h(\cdot)$. Then, for each x, we attempt to reconstruct the source sequence $\hat{\mathrm{s}}$ by running BP with its corresponding $h(\cdot)$. For this distortion level, we report the lowest rate

$$r^* := \min_{kb,\mathscr{D}} r_{\text{code}} + r_{\text{dope}} = \min_{kb,\mathscr{D}} \frac{kb + |\mathscr{D}|}{n} \tag{7.1}$$

and its associated distortion

$$\text{SQNR}|_{dB} = -10 \log \left(10 \cdot \Delta_{\text{MSE}}(\hat{\mathrm{s}}^n, \mathrm{s}^n)\right) \tag{7.2}$$

for which the system converges within $\tau^* = 100$ iterations with all constrains satisfied.

In particular, for each distortion level δ, we report two rates:

(1) `PGM-Proto`: The rate $r_p^*(\delta)$ for which *at least one* source sequence s^n converges for some code $h(\cdot)$ for our probabilistic graphical model (PGM) based data compressor, and
(2) `PGM-Convg`: The rate $r_h^*(\delta)$ for which *all* generated source sequences s^n converges for some code $h(\cdot)$ for our probabilistic graphical model based data compressor.

These two values are of special interest because of their implication on convergence: the `PGM-Proto` rate $r_p^*(\delta)$ signifies the rate that can be achieved with a perfect code $h(\cdot)$, while the `PGM-Convg` rate $r_h^*(\delta)$ signifies a good code $h(\cdot)$ that our system can use, one that causes belief propagation to almost always converge regardless of the source sequence.

7.2 Gaussian iid Sources

We presented the model of the Gaussian iid source in detail in Section 6.1.1. Now, in this section, we discuss its rate-distortion bound and the bounds for known classes algorithms for Gaussian iid sources. We then compare these bounds to the performance of our architecture, represented by the `PGM-Proto` rate $r_p^*(\delta)$ and the `PGM-Convg` rate $r_h^*(\delta)$.

7.2.1 Rate-Distortion Bound

We first describe the rate-distortion bound (Definition 2.32), the theoretical lower bound of the rate-distortion trade-off that no compression system can outperform.

Theorem 7.3. (Rate-Distortion Bound for Gaussian iid Sources). For high rates, the rate-distortion function of a Gaussian iid source $\boldsymbol{s}$ with variance $\sigma^2_{\mathbf{IID}}$ is, from [19],

$$\mathbb{R}^{\mathrm{RD}}(\delta; \boldsymbol{s}) = -\frac{1}{2}\log_2\left(\frac{\delta}{\sigma^2_{\mathbf{IID}}}\right) \tag{7.4}$$

For our test samples, we set $\sigma^2_{\mathbf{IID}} = 1$, so expressed in terms of $\mathrm{SQNR}|_{dB}$, the bound approximately

$$r^{\mathrm{RD}}_{\mathbf{IID}} \approx 0.166 \cdot \mathrm{SQNR}|_{dB} \tag{7.5}$$

7.2.2 Lower Bounds for Known Classes of Algorithms

Here, we present two classes of algorithms known for compressing Gaussian iid sources, and the theoretical rate-distortion lower bounds achieved by each of them.

7.2.2.1 Lloyd-Max Algorithm

The `Lloyd-Max` Algorithm [26] is an iterative algorithm that is similar to k-means clustering. With β quantization levels, it starts by randomly choosing β *representative values*, and set threshold values to be the equidistant point of each pair of representative values. Then within each quantization region, it calculates its centroid as the new representative value. The algorithm repeats the two steps until convergence.

For a Gaussian iid source, `Lloyd-Max` is run independently on every one of the n entries of the source sequence. For rate-distortion comparison, we give its lower bound below.

Theorem 7.6. (`Lloyd-Max` on Gaussian iid Sources). For a Gaussian iid source $\boldsymbol{s}$ with variance $\sigma^2_{\mathbf{IID}}$, the `Lloyd-Max` Algorithm [19] achieves a rate-distortion bound of

$$\mathbb{R}^{\mathrm{LM}}(\delta;\boldsymbol{s}) = -\frac{1}{2}\log_2\left(\frac{\delta}{\sigma^2_{\mathbf{IID}}}\right) + \frac{1}{2}\log_2\left(\frac{\sqrt{3}\pi}{2}\right) \tag{7.7}$$

Expressed in terms of $\mathrm{SQNR}|_{dB}$ for our samples, the `Lloyd-Max` bound is approximately

$$r^{\mathrm{LM}}_{\mathbf{IID}} \approx 0.166 \cdot \mathrm{SQNR}|_{dB} + 0.722 \tag{7.8}$$

We note that `Lloyd-Max` does not have a model free encoder, ie. the algorithm needs to know the model at encode time to be able to compute the quantized values.

7.2.2.2 Entropy-Coded Uniform Scalar Quantizer

The entropy-coded uniform scalar quantizer (`ECUS`) [16] is an algorithm that uses entropy coding, usually done with a trellis, on the output symbols of a uniform scalar quantizer, which we have described in Section 5.1.1. Here, we present its lower bound:

Theorem 7.9. (`ECUS` Quantizer). For a Gaussian iid source $\boldsymbol{s}$ with variance $\sigma^2_{\mathbf{IID}}$, the entropy-coded uniform scalar quantizer [19] achieves a rate-distortion bound of

$$\mathbb{R}^{\mathrm{ECUS}}(\delta;\boldsymbol{s}) = -\frac{1}{2}\log_2\left(\frac{\delta}{\sigma^2_{\mathbf{IID}}}\right) + \frac{1}{2}\log_2\left(\frac{\pi e}{6}\right) \tag{7.10}$$

Expressed in terms of $\mathrm{SQNR}|_{dB}$ for our samples, the entropy-coded uniform scalar quantizer bound is approximately

$$r^{\mathrm{ECUS}}_{\mathbf{IID}} \approx 0.166 \cdot \mathrm{SQNR}|_{dB} + 0.255 \tag{7.11}$$

We note that with the quantization matrix $Q = \frac{1}{w}I$, our algorithm can be treated as an instance of an `ECUS` quantizer scheme, since a quantizer function $q(\cdot)$ with $Q = \frac{1}{w}I$ is a uniform scalar quantizer and the code $h(\cdot)$ is an entropy coder in nature.

7.2.3 Results

In Figure 7.12, we plot the three bounds mentioned above (rate distortion, `Lloyd-Max`, and entropy-coded uniform scalar quantizer `ECUS`) with the performance of our architecture, plotted as the `PGM-Proto` rate and `PGM-Convg` rate.

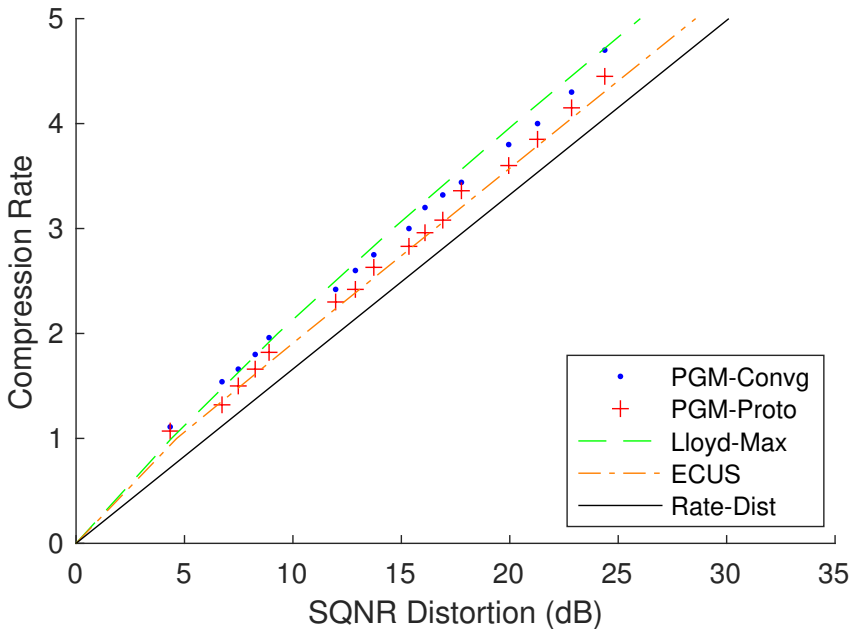


Figure 7.12: Performance of our algorithm on Gaussian iid sources against the rate distortion bound, `Lloyd-Max`, and `ECUS`. The y-axis is the rate and the x-axis is the SQNR.

Note that for the plot, we use $Q = \frac{1}{w}I$ for some constant w, hence the performance is lower-bounded by `ECUS`, which our architecture closely follows. We also consistently beat `Lloyd-Max` without using source information available to `Lloyd-Max`.

7.3 Gauss-Markov Sources

We presented the model of the Gauss-Markov source in detail in Section 6.1.2. We recall that a Gauss-Markov source has 3 parameters: (i) $\lambda_0^s = \frac{1}{\sigma_0^2}$ which is the inverse of the variance of the first element in the source sequence, (ii) a which is the auto-correlation factor, and (iii) $\lambda^s = \frac{1}{\sigma^2}$ which is the inverse of the variance of the additive noise, also known as the innovation. For our experiments, we choose $\lambda_0^s = 1$, and $a = 0.7$, and $\lambda^s = \frac{1}{0.51}$. This set of parameters has the property that the marginal distribution of each element of the source is $\sim \mathbf{N}(0, 1)$, a property we will discuss later.

In this section, we discuss the rate-distortion bound of the Gauss-Markov source and the bounds for known classes of algorithms that compress the Gauss-Markov source. We then compare these bounds with the performance of our architecture, represented by the `PGM-Proto` rate $r_p^*(\delta)$ and the `PGM-Convg` rate $r_h^*(\delta)$.

7.3.1 Rate-Distortion Bound

We first describe the rate-distortion bound (Definition 2.32) of the Gauss-Markov source [8].

Theorem 7.13. (Rate-Distortion Bound for Gauss-Markov Sources). For high rates, the rate-distortion function of a Gaussian-Markov source $\boldsymbol{s}$ with parameter λ^s is, from [18],

$$\mathbb{R}^{\mathrm{RD}}(\delta; \boldsymbol{s}) = -\frac{1}{2}\log_2\left(\delta\lambda^s\right) \tag{7.14}$$

For our test samples, we have $\lambda^s = \frac{1}{0.51}$, so expressed in terms of $\mathrm{SQNR}|_{dB}$, the bound is approximately

$$r^{\mathrm{RD}}_{\mathbf{GM}(\lambda^s)} \approx 0.166 \cdot \mathrm{SQNR}|_{dB} - \frac{1}{2}\log_2\left(\lambda^s\right) \tag{7.15}$$

$$\approx 0.166 \cdot \mathrm{SQNR}|_{dB} - 0.486 \tag{7.16}$$

7.3.2 Lower Bounds for Known Classes of Algorithms

Here, we present two classes of algorithms known for compressing Gauss-Markov sources, and the theoretical rate-distortion lower bounds achieved by each of them.

7.3.2.1 Entropy-Coded Uniform Scalar Quantizer

We described in Section 7.2.2.2 the `ECUS` quantizer. We now present its lower bound in terms of the appropriate set of parameters.

Theorem 7.17. (`ECUS` Quantizer). For a Gauss-Markov source $\boldsymbol{s}$ with parameters λ^s and a, the entropy-coded uniform scalar quantizer [19] achieves a rate-distortion bound of

$$\mathbb{R}^{\mathrm{ECUS}}(\delta; \boldsymbol{s}) = -\frac{1}{2}\log_2\left(\delta\lambda^s(1-a^2)\right) + \frac{1}{2}\log_2\left(\frac{\pi e}{6}\right) \tag{7.18}$$

Expressed in terms of $\mathrm{SQNR}|_{dB}$ for our samples, the entropy-coded uniform scalar quantizer bound is approximately

$$r^{\mathrm{ECUS}}_{\mathbf{GM}(\lambda^s)} \approx 0.166 \cdot \mathrm{SQNR}|_{dB} + 0.255 \tag{7.19}$$

since $\lambda^s = \frac{1}{0.51}$ and $a = 0.7$. This result is the same as that of Section 7.2.2.2.

We note that for our system, even when $Q = \frac{1}{w}I$, we are not necessarily lower bounded by the `ECUS` bound for Gauss-Markov sources, for we take advantage of the underlying correlation in the decoding process, which `ECUS` does not.

7.3.2.2 Differential Pulse Code Modulation

The Differential Pulse Code Modulation (DPCM) algorithm is specifically designed to quantize a Markov chain. In particular, it quantizes the *innovation*, ie. the additive Gaussian white noise v_i, described in Equation 6.5.

The DPCM reconstruction thus simulates the Markov generation process, adding the representative value of the quantized innovation $\hat{\mathrm{v}}_i$ to $a\hat{\mathrm{s}}_{i-1}$ as the reconstruction $\hat{\mathrm{s}}_i$. Here, we present its lower bound:

Theorem 7.20. (DPCM Quantizer on Gauss-Markov Sources). For a Gauss-Markov source $\boldsymbol{s}$ with parameters λ^s, the differential pulse code modulation quantizer [18] achieves a rate-distortion bound of

$$\mathbb{R}^{\mathrm{DPCM}}(\delta; \boldsymbol{s}) = -\frac{1}{2}\log_2\left(\delta\lambda^s\right) + \frac{1}{2}\log_2\left(\frac{\pi e}{6}\right) \tag{7.21}$$

Expressed in terms of $\mathrm{SQNR}|_{dB}$ for our samples, the entropy-coded uniform scalar quantizer bound is about

$$r^{\mathrm{DPCM}}_{\mathbf{GM}(\lambda^s)} \approx 0.166 \cdot \mathrm{SQNR}|_{dB} - \frac{1}{2}\log_2\left(\lambda^s\right) + 0.255 \tag{7.22}$$

$$\approx 0.166 \cdot \mathrm{SQNR}|_{dB} - 0.231 \tag{7.23}$$

since $\lambda^s = \frac{1}{0.51}$.

We note that DPCM's encoder is not model free. In fact, in addition to needing to know that the source has a Markov structure, it also needs to know the specific parameters a and λ^s to be able to produce the quantized values.

7.3.3 Results

In Figure 7.24, we plot the three bounds mentioned above (rate distortion, entropy-coded uniform scalar quantizer ECUS, and differential pulse code modulation DPCM) with the performance of our architecture, plotted as the PGM-Proto rate and PGM-Convg rate.

Note that for these plots, we use $Q = \frac{1}{w}I$ for some constant w, but as mentioned above, our performance is not lower bounded by ECUS. It seems, instead, our performance is lower bounded by DPCM, a bound which our algorithm closely follows.

We note that this is a very important result: even when the encoder completely ignores the underlying source correlation structure, the decoder can still decode at a rate that takes full advantage of the structure.

While the DPCM algorithm achieves a similar bound as our algorithm, our model-code separation architecture affords the flexibility that DPCM cannot, for at compression time the DPCM encoder needs to know everything about the source to be able to compress, whereas our architecture requires no such information, while still being able to provide comparable performance.

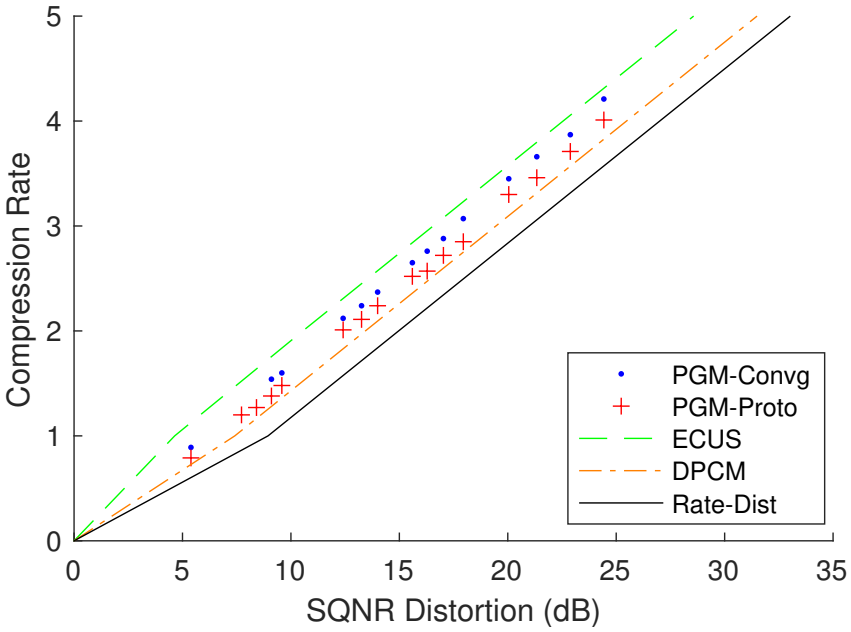


Figure 7.24: Performance of our algorithm on Gauss-Markov models $\mathbf{GM}(\lambda^s)$ (with $\lambda^s = \frac{1}{0.51}$) against the rate distortion bound, `ECUS`, and `DPCM`. The y-axis is the rate and the x-axis is the SQNR.

7.4 Summary

In this chapter, we presented the performance of our algorithm and compared it with the lower bounds of classes of algorithms, in addition to its theoretical lower bound, the rate-distortion bound. Specifically, we compressed Gaussian iid sources and Gauss-Markov sources, noting that in addition to providing performance comparable to existing algorithms, our algorithm is not burdened by the use of source model information at encoding time. By making the choice of not accessing source model information at the encoder, we avoid the entanglement of model and code, providing a flexible framework with modular components.

In the remaining chapters, we will discuss the optimization decisions made to the system, as well as potential extensions to the architecture.

8

Discussion and Analysis

In Chapters 3, we described the general architecture of our compression algorithm, defining the four main components of (i) source model $p_s(\cdot)$, (ii) quantizer $q(\cdot)$, (iii) translator $t(\cdot)$, and (iv) code $h(\cdot)$. In Chapters 4, 5, and 6, we presented various possible choices of each of the four components. Then, in Chapter 7, we discussed the performance of our algorithm on compressing the Gaussian iid source and the Gauss-Markov source. In this chapter, we will now compare how different choices of these architectural components affect compression performance.

8.1 LDPC Code Optimization

The code $h(\cdot)$ is the main component responsible for compression, and hence its parameters controls the compression rate. To optimize for the compression rate, therefore, we naturally focus on the code component first.

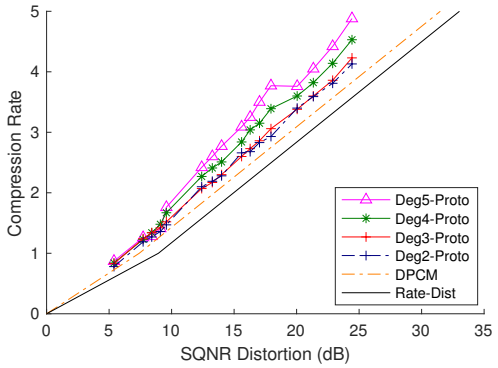
8.1.1 Column Weights and Degree Distribution

The degree distribution is an important parameter of an LDPC code. In this section, we experiment with different configurations of the degree distribution.

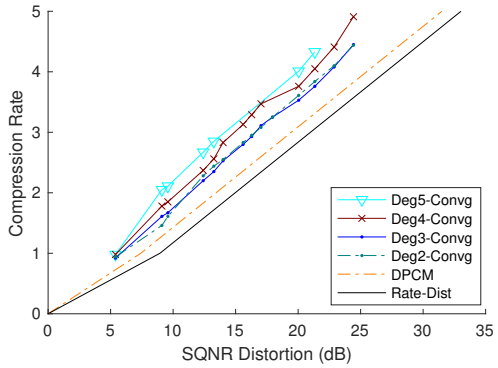
8.1.1.1 Regular LDPC Codes

We first experiment with different fixed column weights ρ', ie. regular LDPC codes. As we can see in Figure 8.3, increasing the column weight decreases performance in terms of compression rate.

We note that increasing ρ' increases the per iteration time complexity for decoding, as discussed in Section 3.5 where we argued that the time complexity of Decode is linear in the row weights ρ. In addition, the number of iterations needed for convergence also increases. Intuitively, with more dependencies between the translated bits z and the hashed bits x, more information is being passed around, making it harder for it to converge, thus a higher rate to ensure convergence.



(a) The `Proto` rate.



(b) The `Convg` rate.

Figure 8.1: Compressing Gaussian iid sources with $n = 500$, with regular LDPC codes of different degrees.

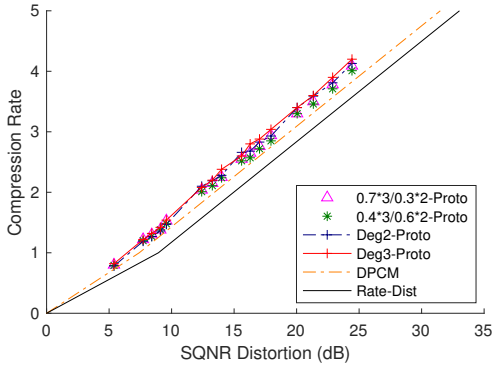
8.1.1.2 Irregular LDPC Codes

Next, we experiment with variable column weights, ie. irregular LDPC codes.

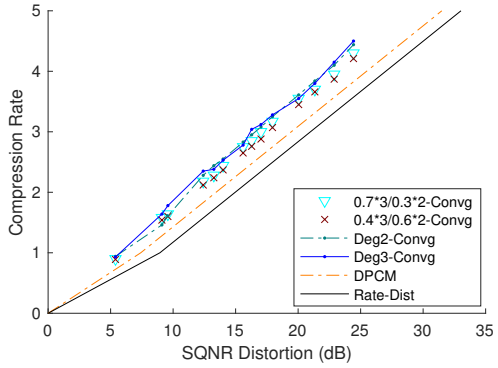
Let $P * A/Q * B$ be a shorthand for an irregular LDPC code with column degree distribution

$$\varrho'(x) = P \cdot \mathbf{Delta}(x - A) + Q \cdot \mathbf{Delta}(x - B) \tag{8.2}$$

We experimented with four configurations of LDPC codes, namely $(p) * 3/(1-p) * 2$ for $p = 0, 0.4, 0.7, 1$, and we plot our results in Figure 8.3. As we can see from the plots, irregular LDPC codes has a better rate performance than regular codes when compressing the Gauss-Markov source.



(a) The `Proto` rate.



(b) The `Convg` rate.

Figure 8.3: Compressing Gauss-Markov sources with $n = 500$, with different degree distributions. Irregular LDPC codes consistently beat the regular ones.

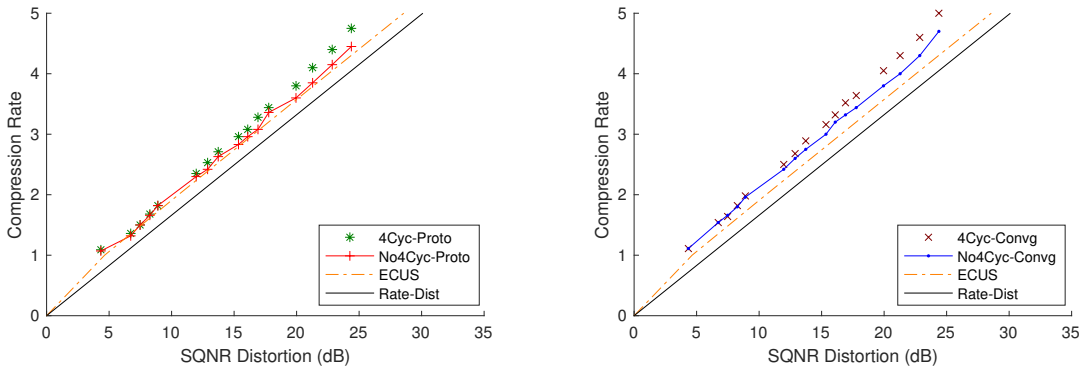
Irregular LDPC codes, however, does not provide further gains when compressing the Gaussian iid source. They give the same rate performance as the regular $\rho' = 3$ code, which is already on the `ECUS` lower bound (Section 7.2.2.2), as plotted in Figure 7.12.

The fact that the compression rate can be further decreased indicates that further work can be done on code selection. This ability to optimize only the code is made possible by the modular structure of our architecture.

8.1.2 Four Cycles

As introduced in Section 4.1.2, it is very likely that a randomly generated LDPC code contains four cycles. Here, we explore the effects of removing four cycles from the code.

We experiment with compressing Gaussian iid sources with sequence length $n = 500$. In Figure 8.4, we plot the performance of removing four-cycles against keeping four cycles at different compression rates. As can be seen in the plot, having the two sets of points closely trace each other, except at higher rates where having four cycles seem to increase the rate by a small amount. While four cycles in the LDPC code seem to have some negative effect on the compression performance, the effects are small.



(a) `Proto` rate. (b) `Convg` rate.

Figure 8.4: Compressing Gaussian iid sources with $n = 500$, allowing four cycles in the LDPC code. We note that there are no significant differences between the performances with or without four cycles, except at higher rates, where codes without four cycles outperform codes with four cycles by a slight margin.

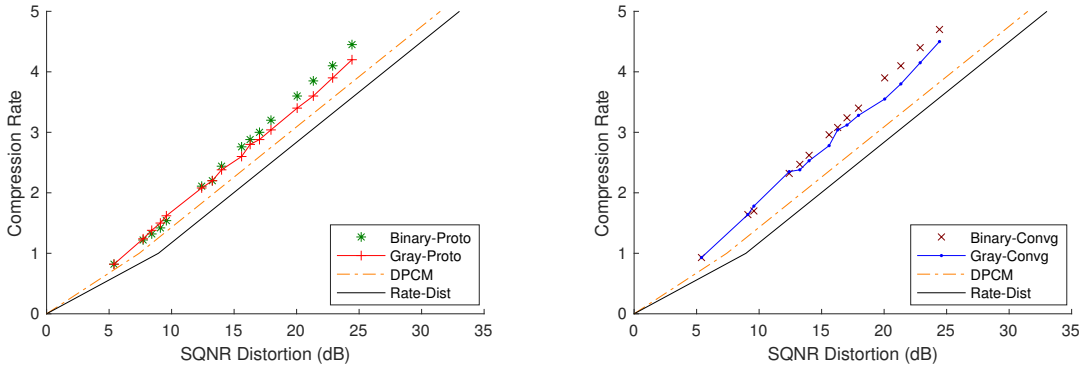
8.2 Translator Optimization

The next component to be tuned would be the translator. As mentioned, we have two candidates: standard binary code and Gray code. We now experiment with the two translators and compare their performance.

Figure 8.5 presents the total compression rate of Gray code vs. standard binary code on the Gauss-Markov source, with bits doped under lattice doping (Section 4.3). Gray code consistently beats standard binary code for $b \geq 3$, but for $b = 2$ the standard binary code beats Gray code. This is because for $b = 2$, the second bit of a Gray code denotes whether the sample is near the center, which is true most of the

time for a Gaussian variable. This means that the second bit of a Gray code does not contain useful information.

As we can see, the choice of translator affects the rate performance. This points to the inherent asymmetrical importance of the bits within each sample, a concept that should be further explored. In particular, while the translator itself has a small impact on performance, the design of the translator needs to be coordinated with the dope bits, especially with lattice doping, and that has a huge implication on rate performance.



(a) `Proto` rate. (b) `Convg` rate.

Figure 8.5: Compressing Gauss-Markov sources with $n = 500$, with standard binary code vs. PAM Gray code. We note that Gray code outperforms binary in higher bit quantizers, but for $b = 2$ binary code beats Gray code. We omit the results for $b = 1$ as the binary code and the Gray code coincide.

One intuitive explanation that Gray code provides a better performance than the standard binary code is the standard binary code's frequent change of bit values between adjacent coded values, a problem that Gray code avoids by enforcing only 1-bit Hamming distance between adjacent coded values. In particular, consider the case when standard binary code is used, and the decoder is almost certain that the least significant bit is 1. This information does not localize the search space, for the resulting distribution eliminates every other possible quantized sample bin value, but it does not make finding the actual bin too much easier. Gray code, on the other hand, provides a higher degree of localization, which allows it to consistently outperform standard binary code by a total rate of around $0.03b$ for $b \geq 3$.

8.3 Doping Optimization

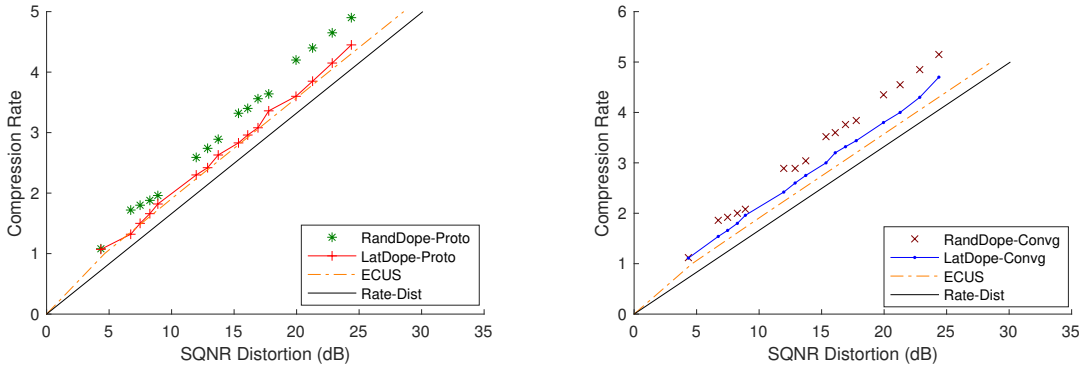
As mentioned above, doping is an operation that has to be designed in conjunction with the translator. For this section, we will focus on using the Gray code as the translator.

8.3.1 Random Doping vs. Lattice Doping

Recall that random doping (Section 4.3.1) chooses the set $\mathcal{D}$ of dope bits from $\mathcal{Z}$ randomly, and that lattice doping (Section 4.3.3) dopes every b^{th} bit of $\mathcal{Z}$. Therefore, random doping is *translator agnostic* and lattice doping is *translator aware*. As discussed in Section 4.3, lattice doping should, intuitively, outperform random doping since it uses knowledge of the probability distribution of $\mathcal{Z}$ induced by the translator. Here, we present quantitatively the gains from using lattice doping.

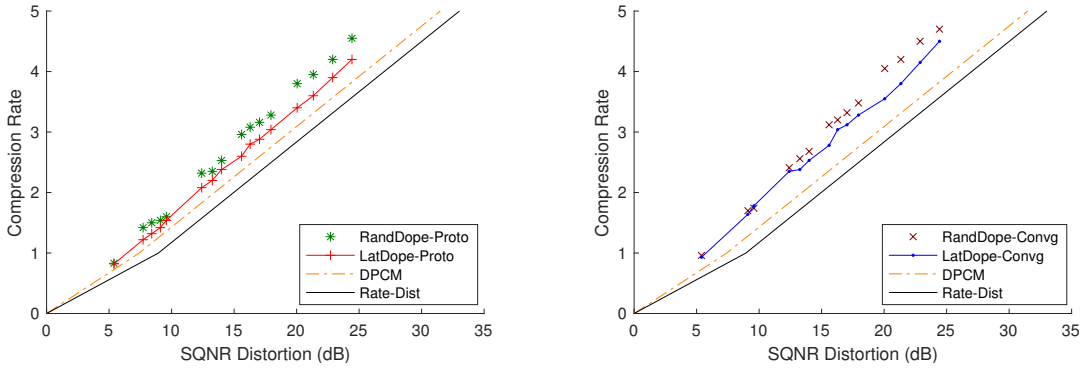
We compressed Gaussian iid sources with length $n = 500$. As seen in Figure 8.6, the performance of random doping (with the same dope rate as lattice doping when $\frac{1}{b}$ for $b \geq 2$) is consistently around 0.3 bits worse than lattice doping for a fixed distortion.

We note that for $b = 1$ random doping has the same performance as lattice doping on the iid source, for lattice doping takes advantage of the inherent asymmetry of bits of different significance, a property that a 1-bit quantizer lacks.



(a) `Proto` rate. (b) `Convg` rate.

Figure 8.6: Compressing Gaussian iid sources with $n = 500$, with random doping vs. lattice doping. We note that lattice doping outperforms random doping in all cases tried, except when $b = 1$.



(a) `Proto` rate. (b) `Convg` rate.

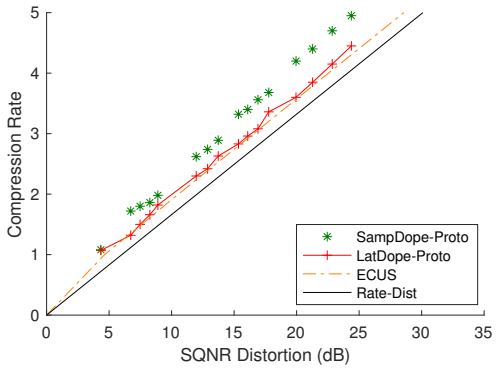
Figure 8.7: Compressing Gauss-Markov sources with $n = 500$, with random doping vs. lattice doping. Lattice doping outperforms random doping in all cases tried, except when $b = 1$.

We next compressed Gauss-Markov sources with length $n = 500$. As illustrated in Figure 8.7, for $b \geq 2$, the total rate for random doping is about 0.2 bits worse that that of the optimal lattice doping for a fixed distortion, a result consistent with that of the iid sequences above. We also note that for the 1-bit quantizer case, lattice doping outperforms random doping, unlike the Gaussian iid source. This advantage is due to the Markov chain structure of the Gauss-Markov source.

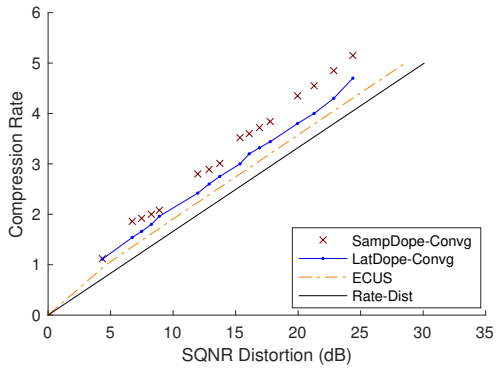
8.3.2 Sample Doping vs. Lattice Doping

Sample doping, as introduced in Section 4.3.2, chooses the dope bits in groups of b, the output size of each translator. We now present the results of using sample doping.

Noting that sample doping is meaningful only when $b \geq 2$, we found that for both Gaussian iid and Gauss-Markov sources, sample doping performs around 0.3 bits worse than optimized lattice doping for any fixed distortion. The results can be seen in Figures 8.8 and 8.9.

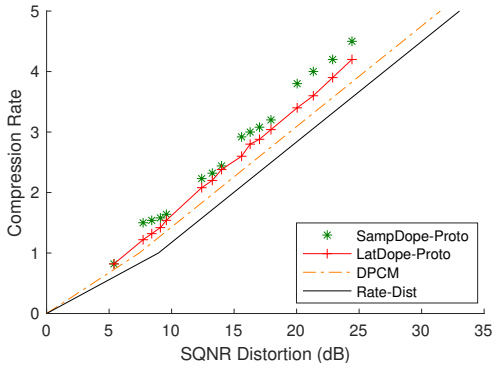


(a) `Proto` rate.

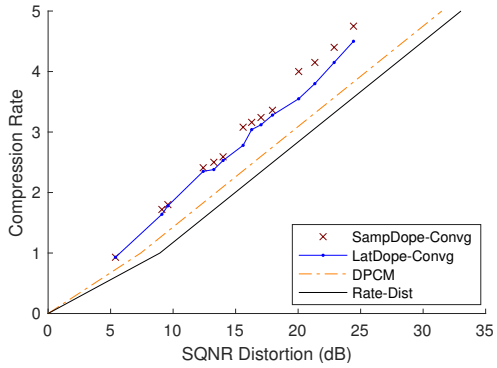


(b) `Convg` rate.

Figure 8.8: Compressing Gaussian iid sources with $n = 500$, with sample doping vs. lattice doping. We note that lattice doping outperforms sample doping consistently.



(a) `Proto` rate.



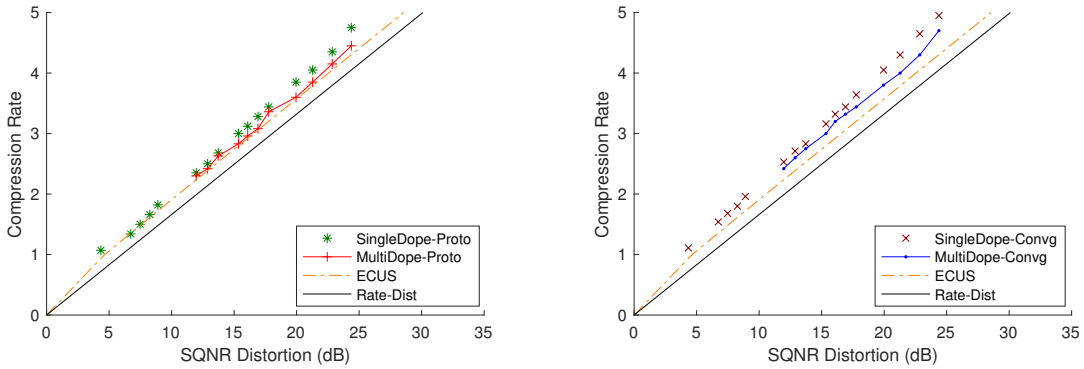
(b) `Convg` rate.

Figure 8.9: Compressing Gauss-Markov sources with $n = 500$, with sample doping vs. lattice doping. Lattice doping outperforms sample doping consistently.

This can be explained by the asymmetry of information within the bits induced by the translator. Under Gray coding, doping the second most significant bit wastes the dope bit because the second bit always indicates whether the sample is within certain bins of the middle, which for a Gaussian distribution is disproportionally true. Therefore, when we dope samples, some bits within each sample are not as useful as they can be, and thus sample doping performs worse than lattice doping.

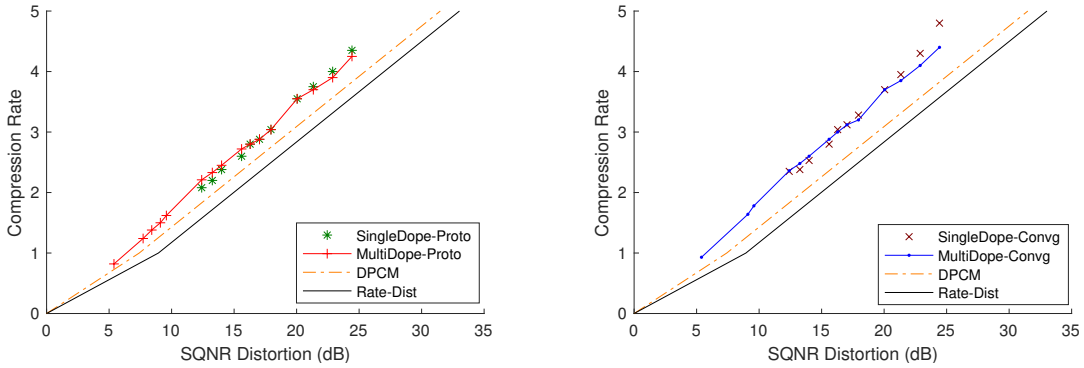
8.3.3 Multiple Lattice Doping

In Section 4.3.4, we described potential doping schemes that extend the advantages of lattice doping. Here, we present the results of multiple lattice doping. Multiple lattice doping refers to doping bits of the same significance for each b-bit group. For the Gaussian iid source, we see that multiple lattice doping gives the best results for $b \geq 3$, where multiple lattice doping applies. Figure 8.10 compares multiple lattice doping with normal lattice doping, showing that it outperforms normal lattice doping.



(a) `Proto` rate. (b) `Convg` rate.

Figure 8.10: Compressing Gaussian iid sources with $n = 500$, with multiple doping vs. single doping. Multiple doping outperforms single doping consistently for the iid source.



(a) `Proto` rate. (b) `Convg` rate.

Figure 8.11: Compressing Gauss-Markov sources with $n = 500$, with multiple doping vs. single doping. Multiple doping has similar performance as single doping for the Gauss-Markov source.

For the Gauss-Markov source, multiple lattice doping provides less gain than that of the Gaussian iid source. As seen in Figure 8.11, significant gains only happen in higher bit regimes (eg. $b = 5$). The difference in performance of multiple lattice doping between the iid source and the Markov source can be explained by the correlation structure: in the iid source each b-bit group is uncorrelated to one another, hence each group needs some initialization for better performance. For the Markov source, however, doping within each b-bit group provides redundant information, for the decoder can already infer from the Markov structure a sample's localization given its immediate neighbors.

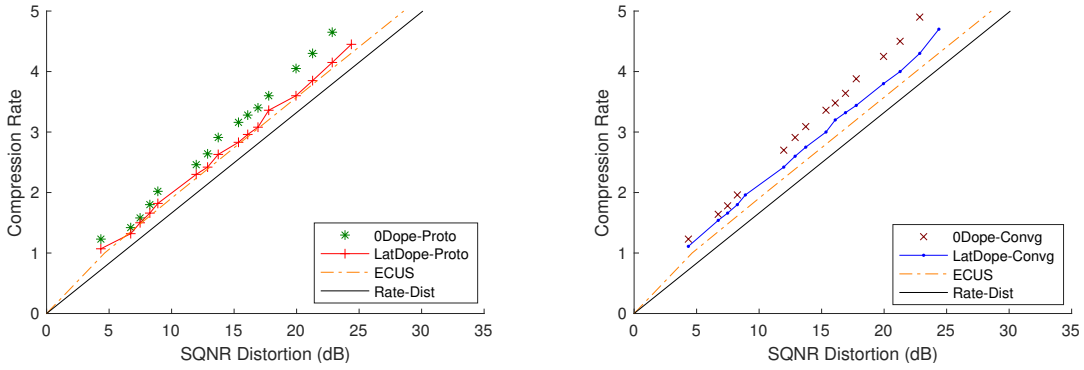
8.3.4 Hashing Doped Bits

Garcia-Frias and Zhong suggested in their work [13], which uses LDPC codes to compress binary Markov chains, that not hashing the doped bits will increase the rate performance for lossless compression. With our architecture, however, this idea does not seem to contribute any significant improvement over hashing the doped bits. In addition, in the case of random doping, it can even worsen the performance, since there is no anchor with which the algorithm can start message passing in the code graph. In the case of lattice doping, the negative effects are not as significant, for the non-doped initial messages are inherently biased, given our uniform scalar quantization of Gaussians.

8.3.5 Zero Doping

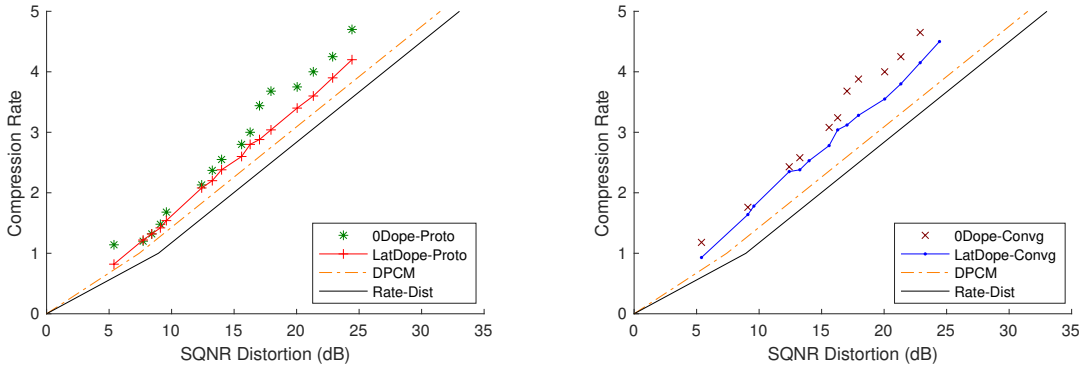
As mentioned in Section 3.4, doping provides an anchor for initialization for message passing. We now explore whether doping is necessary and observe its impact on performance.

We compressed Gaussian iid sources with length $n = 500$. From Figure 8.12, we see that the performance of zero doping for $b \geq 2$ is is consistently around 0.1 bits worse than lattice doping.



(a) `Proto` rate. (b) `Convg` rate.

Figure 8.12: Compressing Gaussian iid sources with $n = 500$, with zero doping vs. lattice doping. Zero doping converges, but only at a higher compression rate.



(a) `Proto` rate. (b) `Convg` rate.

Figure 8.13: Compressing Gauss-Markov sources with $n = 500$, with zero doping vs. lattice doping. Zero doping converges, but only at a higher compression rate.

Next, we compressed Gauss-Markov sources with length $n = 500$ with zero doping. For $b \geq 2$, the total rate for zero doping is consistently 0.2 worse than that of optimized lattice doping, as seen in figure 8.13.

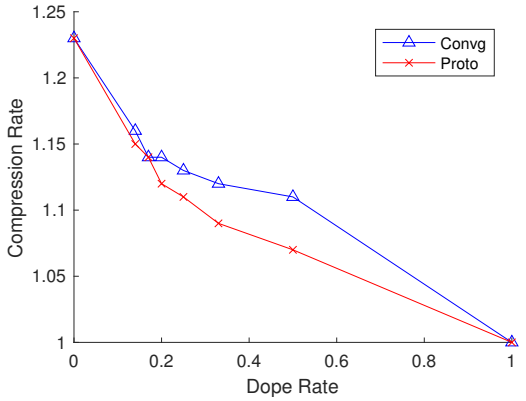
Intuitively, while we do not explicitly dope any of the bits, the inherent asymmetry of the distribution induced by the translator already implicitly provides a starting point for message passing, although the performance is not as good as explicit doping.

8.3.6 Doping in the Low Bit Regime

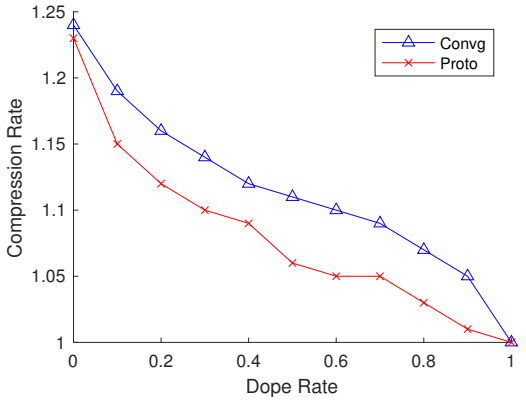
It should be noted that the low bit regime ($b = 1$ and $m \leq n$) is a special case for our system. Many of the translator properties does not apply for when the translator only has a 1-bit output. Hence, for $b = 1$, the dope bits need to be carefully chosen, since lattice doping, which works well for higher bit regimes $b \geq 2$ by exploiting the inherent asymmetry of the translated bits, does not have a logical parallel. Other methods, such as zero doping and random doping, does not seem to work well, as we shall discuss.

For the Gaussian iid source, our experiments shows that there is no difference in performance between random doping and lattice doping. This is because of the symmetry between all the bits in an iid source. As illustrated in Figure 8.14a, compression rate is decreasing with increasing dope rate, with the worse performance being $r_{\text{dope}} = 0$ (with $r_p^* = 1.23$) and the best performance being $r_{\text{dope}} = 1$, which trivially achieves a rate of $r = 1$. This result is source-dependent: with an iid source, there is no correlation that the entropy code $h(\cdot)$ can take advantage of, since the translated sequence is distributed according to $\mathbf{Bern}\left(\frac{1}{2}\right)$, which has an entropy of 1. Thus, no entropy coding is possible, and the best that can be done is to transmit the original sequence verbatim.

However, for Gauss-Markov sources for $b = 1$, the dope rate r_{dope} can be tuned to achieve non-trivial results. As seen in Figure 8.15a, zero doping achieves a total rate of $r_p^* = 1.14$, while the rate decreases to around $r_p^* = 0.82$ for $r_{\text{dope}} = 0.4$, before increasing to $r_p^* = 1$ for $r_{\text{dope}} = 1$. On the other hand, as seen in Figure 8.15b, the

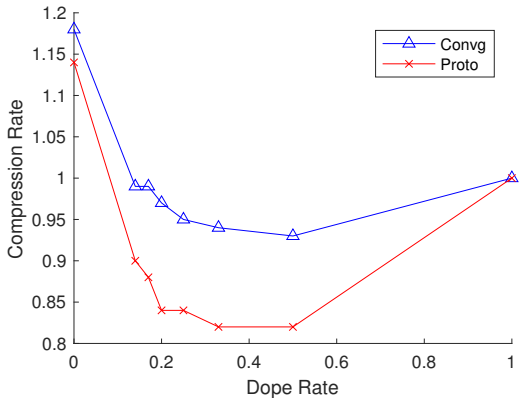


(a) Gaussian iid source with lattice doping.

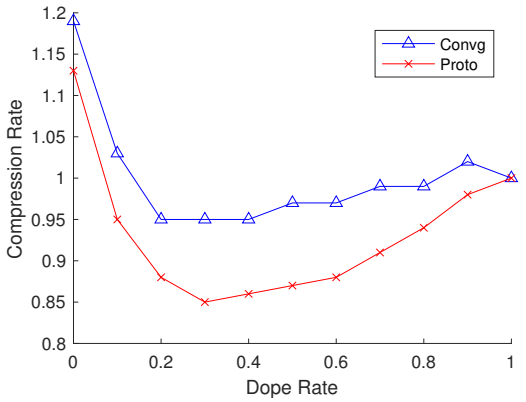


(b) Gaussian iid source with random doping.

Figure 8.14: Compressing Gaussian iid sources with $n = 500$ and $b = 1$, with lattice doping at different dope rates. Given the incompressible nature of the **Bern** $\left(\frac{1}{2}\right)$ source, we cannot get below a rate of 1. Note that the total compression is decreasing with an increasing dope rate.



(a) Gauss-Markov source with lattice doping.



(b) Gauss-Markov source with random doping.

Figure 8.15: Compressing Gauss-Markov sources with $n = 500$ and $b = 1$, with lattice doping and random doping at different dope rates. Note that the total compression can be optimized by choosing an appropriate dope rate.

optimized random doping achieves a worse rate of $r_p^* = 0.84$.

The advantage that lattice doping has on the Gauss-Markov source for $b = 1$ is of a different reason than that of lattice doping for $b \geq 2$, for this advantage is source dependent: with a Markov chain structure, doping every other bit minimizes the length of runs of undoped bits, starting off decoding in a better position.

Our results show that when there is correlation structure, entropy coding is possible. However, more research needs to be devoted to exploring the sub 1-bit regimes for our compression architecture.

8.4 Potential Quantizer Optimization

The quantizer component is perhaps the component with the most choices for optimization. We have discussed two choices for the quantization matrix Q, namely

(i) $Q = \frac{1}{w} I$ for some scalar w, and
(ii) $Q_{ai} \sim \mathbf{Bern}(p) \cdot \mathbf{N}(0, 1)$ for some small p, where Q is sparse and the non-0 entries are drawn from the standard normal distribution.

From our experiments, the identity matrix outperforms all trials of a sparse matrix with entries $\sim \mathbf{N}(0, 1)$ by an extremely wide margin. While this is a disappointing result, it suggests that much more work can be done in the optimization of the quantizer component $q(\cdot)$, especially in the exploration of the low bit (sub 1-bit) regime.

8.5 Summary

In this chapter, we expanded on the results in Chapter 7 and analyzed the components of the compression architecture. We justify our design choices by experimenting with different options of each component and explaining intuitively why certain choices will give better rate performance. In particular, we examined the choice of code, doping, translator, and quantizer. Our experiments with the quantization portion reveals that much more work can be done in choosing a good Q matrix.

9

Realistic Applications and Extensions

In this chapter, we present eight practical compression problems, broadly classified into the categories of source modeling, channel reliability, and network distributivity. We demonstrate that these challenges can be handled elegantly given the modularity and flexibility of our system.

We first describe problems related to source modeling. In Section 9.1, we address the problem of rate selection. In Section 9.2, we experiment with decoding with an incorrect data model. In Section 9.3, we present an augmented source model which can learn the parameters of the source model at decompression time.

We then present schemes to combat channel unreliability. In Section 9.4, we discuss the impact of loss or corruption of compressed data. In Section 9.5, we present existing schemes of encrypted compression and propose a new encrypted compression structure that allows for homomorphic encryption.

Finally, we discuss architectural extensions for network and distributivity. In Section 9.6, we describe a secret sharing scheme. In Section 9.7, we explore a block decoding scheme, which can be extended to a distributed memory parallel belief propagation. In Section 9.8, we present a progressive decoding scheme that allows refinements of decoded results.

9.1 Rate Selection

As an extension of the Huang's system [21], our compression algorithm is also a *fixed rate system* that does not inherently allow for feedback. This means that the encoder requires the compression rate r to be supplied, implied by the size of the doped indices $\mathscr{D}$ and input-output ratios of the functions $q(\cdot)$, $t(\cdot)$, and $h(\cdot)$. Therefore, we will need some estimate on the entropy $\mathbb{H}(\boldsymbol{z}^{mb})$ of the translated bits $\boldsymbol{z}^{mb}$, so that the rate r is chosen slightly above $\mathbb{H}(\boldsymbol{z}^{mb})$ to ensure correct decoding.

Such an estimate of entropy requires knowledge of the entropy of the source $\boldsymbol{s}$, as well as the structure of the quantizer. However, for each source and each quantizer, the encoder only needs to know of the value of the estimated entropy, while the actual

source and quantizer structures are not required. This still maintains the architectural separation between model and code.

9.1.1 Feedback System

If such prior knowledge of the entropy of the source is not available, another way to optimize for rate r is to allow for feedback, whereby the hashed sequence x^{kb} is sent bit by bit, until the decoder acknowledges sufficiency of information for reconstruction. This approach, mentioned by Huang [21] is justified by the conditional independence among the x^{kb} bits, conditioned on z^{mb}. This means that only the number of bits sent, but not the actual positional identity of the bits, matters for the purpose of decoding. This transforms our scheme into a rateless system, as illustrated in Figure 9.1. In fact, in a broadcast setting, even feedback in not necessary.

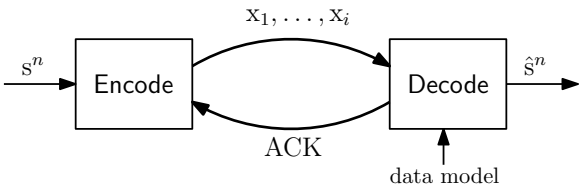


Figure 9.1: In a rateless system, the compressed bits x is streamed to the decoder, until the decoder acknowledges that there is enough information for successful decoding.

9.1.2 Extensions of the Feedback System

The feedback system above can be modified to serve in a data storage scenario. For a fixed source and quantizer, we can begin with a high rate, eg. an uncompressed code $h(\cdot)$ and dope $\mathscr{D}$ with $kb+|\mathscr{D}| = mb$. As we later discover that the decoder can decode at a lower rate, we can truncate x to a shorter length. This is again justified by the conditional independence among x, conditioned on z. More in depth discussions and experiments are presented in Section 9.2.

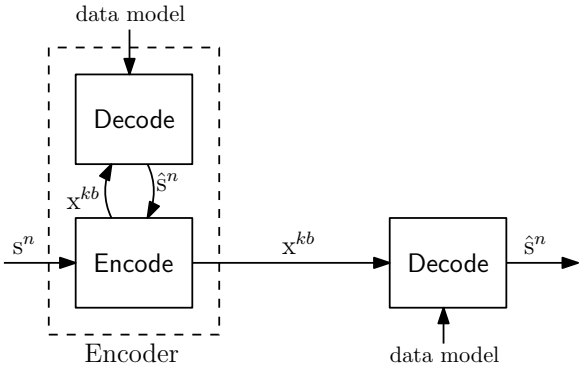


Figure 9.2: If feedback is not available, we can embed the Decode procedure within the encoder to simulate feedback, stopping when the Decode procedure acknowledges there being enough information for decoding.

If neither feedback nor source entropy is available, then we can simulate feedback by including the Decode algorithm in the encoder, as illustrated in Figure 9.2. Note that this does not break the model-code separation, for the Encode block is still model free, even though the encoder uses both Encode and Decode for compression. As Huang [21] explained, the fundamental advantage of model-code separation is not in designing the encoder and decoder separately, but in designing the coding structure to allow for separate optimization and flexibility.

9.2 Source Model Mismatch

As described in Section 9.1, the Encode algorithm does not need to know the source model, while the Decode algorithm uses its knowledge of the source to recover the MSE estimate $\hat{\mathrm{s}}^n$ of the source sequence s^n.

We now consider the effects of decoding with a model that does not correspond to the true source model. Such a scenario is of practical interest, for often times the true model is unknown or is not available, and we have to resort to estimating or learning the source structure and parameters, which may contain less or more structure than the true model.

9.2.1 Imprecise Model

In Section 9.1, in the discussion of rate selection, we mentioned as part of the data storage scenario that the encoder may sometimes want encode with a rate much higher than necessary for reconstruction, and if it is discovered later that the decoder can decode at a lower rate, the compressed sequence x^{kb} can be truncated without losing any information.

Consider the starting scenario, in which the decoder applies an imprecise source model for decoding, assuming that the rate r is high enough for decoding both the correct source model and the imprecise one. In particular, we consider the example of attempting to decode a Gauss-Markov source as an iid source. Noting that the marginal distribution of a Gauss-Markov source (with appropriate parameters λ_0^s, λ^s, and a such that $a^2 + \lambda^s = \lambda_0^s$) is an iid source, the iid model when used in this case is not incorrect, but imprecise, since it does not make use of the correlation of the source s^n.

	Proto rate r_p^*	Convg rate r_h^*	SQNR
Markov decoded as iid	2.83	3.00	15.35
iid	2.83	3.00	15.35
Markov	2.60	2.78	15.60

Table 9.3: Rate and SQNR of decoding a Markov source as an iid model, compared with the results of decoding with the correct models. In this example, we have $b = 4$ and $Q = \frac{1}{w}I$ with $w = 0.6$. We note that we achieve the same performance as that of a true iid source.

We present the results of our simulation in Table 9.3. We note that even with the imprecise model, we can still decode and have a reconstruction $\hat{\mathrm{s}}^n$ that has expected distortion fitting the bounds of that of an iid source. If later we realize that the source can be better modeled as a Gauss-Markov model, we do not need to change the compressed sequence nor the compression architecture itself, but simply replacing the source subgraph in the Decode algorithm will give us the expected performance. The immense flexibility of this architecture is extremely desirable, for it solves the backward compatibility issue that most existing data compression algorithms face. To our knowledge, there does not exist an algorithm that has this flexible decoding property that allows the source model to be refined *after* the source sequence has been compressed.

The rate-distortion gain of decoding with a more fitting model comes from both the quantization subgraph and the code subgraph. The distortion, controlled by the quantizer subgraph, is decreased with a better model, in addition to that the rate, controlled by code subgraph, can be lowered with more correlation in the source.

9.2.2 Wrong Model

Next, we explore the effects of decoding with a wrong model. In particular, we experimented with decoding a Gaussian iid source with a Gauss-Markov model. We note that unlike the previous scenario presented in Section 9.2.1, the model at the decoder side is not an imprecise model, rather, one that assumes too much structure.

	`Proto` rate r_p^*	SQNR
iid decoded as Markov	3.46	14.74
iid decoded as Markov (partial)	3.33	14.03
iid	2.83	15.35
Markov	2.60	15.60

Table 9.4: Rate and SQNR of decoding an iid source as a Markov model, compared with the results of decoding with the correct models. In this example, we have $b = 4$ and $Q = \frac{1}{w}I$ with $w = 0.6$. We note that we achieve both a worse rate performance and precision. The `Convg` rate is not presented because we cannot find a code that always decodes correctly given random bit flips.

Table 9.4 summarizes the results of our experiment. Our experiments show that for the configuration we tried, when the decoder assumes a model with structure non-existent in the actual source, the algorithm does not converge at moderate rates. We started observing a few instances of very low error at high rates, and eventually 0 hashing errors (ie. finding the correct bins u) at very high rates (about 0.6 bit higher than the rate of decoding with the correct iid model). Even though the algorithm manages to find the right bin, the messages, however, are not converging. Rather, they flicker around until the algorithm reached its maximum allowed iteration count $\tau^* = 100$, explaining why the $\mathrm{SQNR}|_{dB}$ is lower than both of the correct model and the assumed model. In addition to decoding at higher rates, the non-convergence also lead to increase in computation complexity.

This experiment shows that while decoding with too much incorrectly assumed structure has rate and computation consequences, it is not completely destructive. The hashed stream can still be decoded, albeit at higher rates, with a higher distortion. In fact, since we still have the access to the compressed bit stream, if we notice such behavior, we can attempt decoding with a model of less structure as a remedy.

9.2.3 Discussion

The above two experiments suggest that it is safe for the decoder to assume less structure at the cost of a worse compression rate, but assuming structure that are not present in the actual source is catastrophic. Hence, with enough compressed bits, the most conservative approach to decoding would be to use the iid model, which assumes the least structure.

This also confirms the intuition that with more structure hence correlation in the source, the more we can compress. Practically, this suggests that the compression rate of the iid model is the maximum that we would need, meaning that it is always safe to compress to the iid rate in the lack of any information about the source, as we can always decode at the iid rate, at the cost of some loss of rate performance. As we know more about the structure of source, we can refine the model, and we will start seeing gains in both rate and distortion. With more correctly identified structure, we can also start truncating from the compressed bit stream as we can decode at lower rates.

9.3 Source Model Learning

In Section 9.2, we observed that decoding is still possible when the model is inaccurate. In this section, we explore the potential of when the model has a known structure but unknown parameters.

9.3.1 Parameter Learning of Gaussian iid Source

As an illustrative example, consider the case when we are compressing a Gaussian iid source with unknown mean, and can be expressed in information form as

$$\mathsf{s}^n \sim \mathbf{N}^{-1}(\theta, \lambda^s I) \tag{9.5}$$

where λ^s is a known constant and the scalar parameter θ is uncertain.

While our encoder is model-free, our decoder needs the model parameters for successful decoding. Without an exact parameter, it may seem that decoding is impossible. With the decoder being a probabilistic graphical model, however, it is natural to use the inference capabilities inherent to graphical models to infer the model parameters, as Huang noted [21]. Therefore, we can incorporate this lack of knowledge into our model by augmenting the source subgraph with a parameter

learning subgraph $\mathcal{L}$. Assuming the prior distribution of θ to be a Gaussian

$$p_{\Theta}(\theta) = \mathcal{N}^{-1}(\theta; \eta^{\theta}, \lambda^{\theta}) \tag{9.6}$$

we can factor the joint distribution on $\boldsymbol{s}^n, \Theta$ as

$$\begin{aligned}
p_{\boldsymbol{s}^n,\Theta}(s^n, \theta) &= p_{\Theta}(\theta) \prod_{i=1}^{n} p_{\boldsymbol{s}_i|\Theta}(s_i|\theta) && (9.7)\\
&= \mathcal{N}^{-1}(\theta; \eta^{\theta}, \lambda^{\theta}) \prod_{i=1}^{n} \mathcal{N}^{-1}(s_i; \theta, \lambda^{s}) && (9.8)\\
&\propto \exp\left\{-\frac{1}{2}\lambda^{\theta}\theta^2 + \eta^{\theta}\theta\right\} \prod_{i=1}^{n} \exp\left\{-\frac{1}{2}\frac{\theta^2}{\lambda^{s}} - \frac{1}{2}\lambda^{s}s_i^2 + \theta s_i\right\} && (9.9)\\
&= \exp\left\{-\frac{1}{2}\left(\lambda^{\theta} + \frac{n}{\lambda^{s}}\right)\theta^2 + \eta^{\theta}\theta\right\} \prod_{i=1}^{n} \exp\left\{-\frac{1}{2}\lambda^{s}s_i^2\right\} \prod_{i=1}^{n} \exp\{\theta s_i\} && (9.10)\\
&= \phi_{\theta}^{\mathcal{L}}(\theta) \prod_{i=1}^{n} \phi_i^{\mathcal{S}}(s_i) \prod_{i=1}^{n} \psi_{\theta i}^{\mathcal{L}}(s_i, \theta) && (9.11)
\end{aligned}$$

following the factorization structure in Section 6.2.1, which corresponds to the graphical structure illustrated by Figure 9.12.

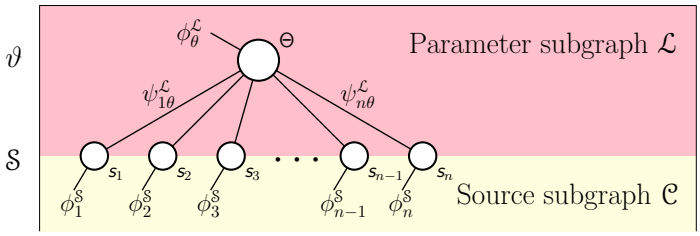


Figure 9.12: Augmented source model with parameter learning subgraph $\mathcal{L}$ to handle source models with known structures but unknown parameters.

The ϑ node potential is thus

$$\begin{aligned}
\phi_{\theta}^{\mathcal{L}}(\theta) &:= \exp\left\{-\frac{1}{2}\lambda_{\theta}^{\phi}\theta^2 + \eta_{\theta}^{\phi}\theta\right\} && (9.13)\\
\lambda_{\theta}^{\phi} &= \lambda^{\theta} + \frac{n}{\lambda^{s}} && (9.14)\\
\eta_{\theta}^{\phi} &= \eta^{\theta} && (9.15)
\end{aligned}$$

and the $\mathcal{S}_i$ node potentials are

$$\phi_i^{\mathcal{S}}(s_i) := \exp\left\{-\frac{1}{2}\lambda_i^{\phi} s_i^2 + \eta_i^{\phi} s_i\right\} \tag{9.16}$$

$$\lambda_i^{\phi} = \lambda^s \tag{9.17}$$

$$\eta_i^{\phi} = 0 \tag{9.18}$$

and the $\mathcal{S}_i, \vartheta$ edge potentials are

$$\psi_{\theta i}^{\mathcal{L}} := \exp\left\{-\lambda_{i\theta}^{\psi} \theta s_i\right\} \tag{9.19}$$

$$\lambda_{\theta i}^{\psi} = -1 \tag{9.20}$$

We now derive the message passing equations between nodes ϑ and $\mathcal{S}_i$. As in Section 6.2.2, we parametrize the Gaussian messages from $\mathcal{F}^{\mathcal{Q}}$ to $\mathcal{S}$ in the information form

$$\mathcal{N}^{-1}(s_i; \eta_{\mathcal{F}_a^{\mathcal{Q}}\mathcal{S}_i}^{(\tau-1)}, \lambda_{\mathcal{F}_a^{\mathcal{Q}}\mathcal{S}_i}^{(\tau-1)}) := m_{\mathcal{F}_a^{\mathcal{Q}}\mathcal{S}_i}^{(\tau-1)}(s_i) \tag{9.21}$$

where $a \in \mathcal{N}_i^{\mathcal{Q}}$. We apply then the results from Section 6.2.2 to get the $\mathcal{S}_i$ to ϑ messages to be

$$m_{\mathcal{S}_i\vartheta}^{(\tau)}(\theta) \propto \mathcal{N}^{-1}(\theta; \eta_{\mathcal{S}_i\vartheta}^{(\tau)}, \lambda_{\mathcal{S}_i\vartheta}^{(\tau)}) \tag{9.22}$$

where

$$\eta_{\mathcal{S}_i\vartheta}^{(\tau)} := -\lambda_{\theta i}^{\psi}\left(\lambda_i^{\phi} + \sum_{a\in\mathcal{N}_i^{\mathcal{Q}}} \lambda_{\mathcal{F}_a^{\mathcal{Q}}\mathcal{S}_i}^{(\tau-1)}\right)^{-1}\left(\eta_i^{\phi} + \sum_{a\in\mathcal{N}_i^{\mathcal{Q}}} \eta_{\mathcal{F}_a^{\mathcal{Q}}\mathcal{S}_i}^{(\tau-1)}\right) \tag{9.23}$$

$$\lambda_{\mathcal{S}_i\vartheta}^{(\tau)} := -\lambda_{\theta i}^{\psi}\left(\lambda_i^{\phi} + \sum_{a\in\mathcal{N}_i^{\mathcal{Q}}} \lambda_{\mathcal{F}_a^{\mathcal{Q}}\mathcal{S}_i}^{(\tau-1)}\right)^{-1}\lambda_{i\theta}^{\psi} \tag{9.24}$$

and the ϑ to $\mathcal{S}_i$ messages to be

$$m_{\vartheta\mathcal{S}_i}^{(\tau)}(s_i) \propto \mathcal{N}^{-1}(s_i; \eta_{\vartheta\mathcal{S}_i}^{(\tau)}, \lambda_{\vartheta\mathcal{S}_i}^{(\tau)}) \tag{9.25}$$

where

$$\eta_{\vartheta\mathcal{S}_i}^{(\tau)} := -\lambda_{\theta i}^{\psi}\left(\lambda_{\theta}^{\phi} + \sum_{j\in\mathcal{S}\setminus\{i\}} \lambda_{\mathcal{S}_j\vartheta}^{(\tau-1)}\right)^{-1}\left(\eta_{\theta}^{\phi} + \sum_{j\in\mathcal{S}\setminus\{i\}} \eta_{\mathcal{S}_i\vartheta}^{(\tau-1)}\right) \tag{9.26}$$

$$\lambda_{\vartheta\mathcal{S}_i}^{(\tau)} := -\lambda_{\theta i}^{\psi}\left(\lambda_{\theta}^{\phi} + \sum_{j\in\mathcal{S}\setminus\{i\}} \lambda_{\mathcal{S}_j\vartheta}^{(\tau-1)}\right)^{-1}\lambda_{\theta i}^{\psi} \tag{9.27}$$

For marginalizing s_i, we again follow Section 6.2.3 to get

$$\hat{p}^{(\tau)}_{\mathsf{s}_i}(s_i) := m^{(\tau)}_{\vartheta \mathcal{S}_i}(s_i) \prod_{a \in \mathcal{N}^{\Omega}_i} m^{(\tau)}_{\mathcal{F}^{\Omega}_a \mathcal{S}_i}(s_i) \tag{9.28}$$

which can be expressed as

$$\hat{p}^{(\tau)}_{\mathsf{s}_i}(s_i) \propto \mathcal{N}^{-1}(s_i, \hat{\eta}_i^{\mathrm{mrg}(\tau)}, \hat{\lambda}_i^{\mathrm{mrg}(\tau)}) \tag{9.29}$$

with

$$\hat{\eta}_i^{\mathrm{mrg}(\tau)} = \eta_i^{\phi} + \eta^{(\tau)}_{\vartheta \mathcal{S}_i} + \sum_{a \in \mathcal{N}^{\Omega}_i} \eta^{(\tau)}_{\mathcal{F}^{\Omega}_a \mathcal{S}_i} \tag{9.30}$$

$$\hat{\lambda}_i^{\mathrm{mrg}(\tau)} = \lambda_i^{\phi} + \lambda^{(\tau)}_{\vartheta \mathcal{S}_i} + \sum_{a \in \mathcal{N}^{\Omega}_i} \lambda^{(\tau)}_{\mathcal{F}^{\Omega}_a \mathcal{S}_i} \tag{9.31}$$

with the MMSE estimate of s^n being

$$\hat{\mathrm{s}}_i^{(\tau)} := \hat{\mu}_i^{\mathrm{mrg}(\tau)} = \left(\hat{\lambda}_i^{\mathrm{mrg}(\tau)}\right)^{-1} \cdot \hat{\eta}_i^{\mathrm{mrg}(\tau)} \tag{9.32}$$

We note that as a side benefit, we can also get an estimate $\hat{\theta}$ by marginalizing on Θ to if we want, since we already have access to node ϑ's incoming messages. The marginalization equation, again following the procedure in Section 6.2.3, will be

$$\hat{p}^{(\tau)}_{\Theta}(\theta) \propto \mathcal{N}^{-1}(\theta, \hat{\eta}_{\theta}^{\mathrm{mrg}(\tau)}, \hat{\lambda}_{\theta}^{\mathrm{mrg}(\tau)}) \tag{9.33}$$

with

$$\hat{\eta}_{\theta}^{\mathrm{mrg}(\tau)} = \eta_{\theta}^{\phi} + \sum_{i \in \mathcal{S}} \eta^{(\tau)}_{\mathcal{S}_i \vartheta} \tag{9.34}$$

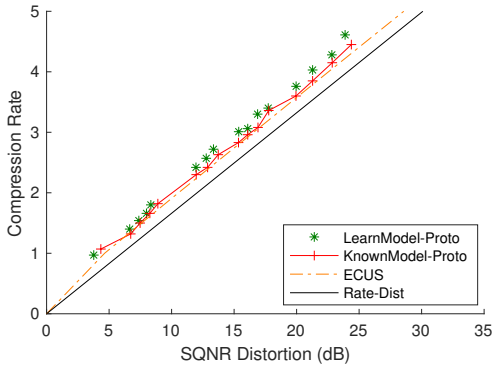
$$\hat{\lambda}_{\theta}^{\mathrm{mrg}(\tau)} = \lambda_{\theta}^{\phi} + \sum_{i \in \mathcal{S}} \lambda^{(\tau)}_{\mathcal{S}_i \vartheta} \tag{9.35}$$

with the MMSE estimate of θ being

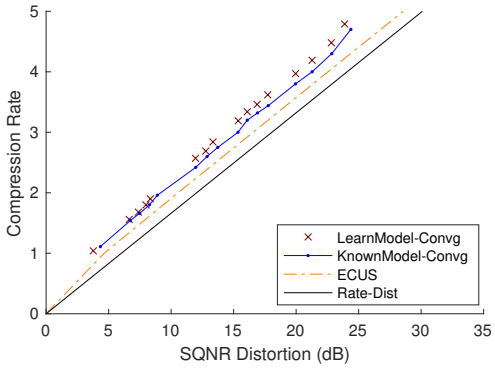
$$\hat{\theta}^{(\tau)} := \hat{\mu}_{\theta}^{\mathrm{mrg}(\tau)} = \left(\hat{\lambda}_{\theta}^{\mathrm{mrg}(\tau)}\right)^{-1} \cdot \hat{\eta}_{\theta}^{\mathrm{mrg}(\tau)} \tag{9.36}$$

9.3.2 Results

As illustrated in Figure 9.37, running our decoding algorithm with the source model augmented by the parameter subgraph $\mathcal{L}$ converges with a slightly higher rate than that of the known iid model. Moreover, with the augmented source subgraph, the SQNR values are also slightly lower. This slight increase in rate and slight degradation of accuracy can be interpreted the price we pay for not having the full model at the decoder. In particular, as θ deviates more and more from the mean of the prior $p_{\Theta}(\theta)$,



(a) The `Proto` rate r_p^*.



(b) The `Convg` rate r_h^*.

Figure 9.37: The rate performance of learning the source parameter θ vs. compression without learning. In our experiments we set $p_\Theta(\theta) \sim \mathbf{N}^{-1}(0,1)$, while we set $\theta = 0.5$, a value unbeknown to either the encoder or the decoder.

the gap widens. With a large deviation (eg. 2 standard deviations away from the mean), belief propagation is extremely hard to converge.

Figure 9.39 shows an instance of the evolution of $\hat{\theta}^{(\tau)}$. For the instances when rate is high enough for successful decoding, the estimates $\hat{\theta}$ are surprisingly accurate. An interesting note is that $\hat{\theta}$ converges to the empirical value

$$\theta_{\text{emp}} := \lambda^s \cdot \left(\frac{1}{n} \sum_{i=1}^{n} \mathrm{s}_i \right) \tag{9.38}$$

instead of the true θ, which agrees with the findings of [21]. This means that the true θ is irrelevant for the augmented source model, and if we had knowledge of the true θ, we would have done message passing differently.

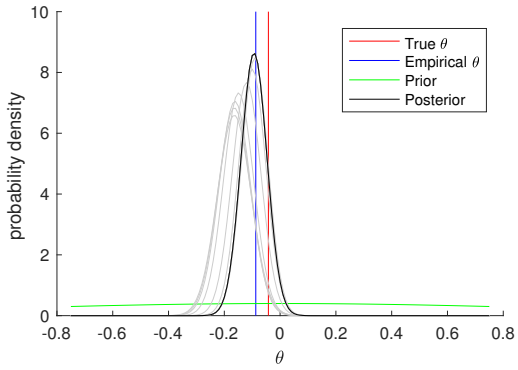


Figure 9.39: An instance of the evolution of the belief $\hat{p}_\Theta^{(\tau)}(\theta)$, with prior set to be $\Theta \sim \mathbf{N}^{-1}(0,1)$. The gray lines represent the estimated belief over the iterations. We note that the posterior converges to the empirical parameter θ_{emp} (blue), not the true θ (red).

9.3.3 Discussion

We note that with a graphical model, we treat unknown parameters the same way as hidden variables. The fact that no special treatment is needed to separately learn a parameter is yet another advantage that the architecture provides. The flexible structure allows manipulation of only the source subgraph to incorporate new functionalities, while leaving all other components of the general architecture untouched. While we experimented with a relatively simple source model with a small set of parameters, the same methodology can be generalized to more complicated sources with more parameters.

The architecture's ability to learn a parameter can further be extended to learning the true parameter θ over time and refining the decoder's knowledge of the source model as the same decoder processes more and more data of the same model. This can be achieved by storing the posterior belief $\hat{p}_{\Theta}(\theta)$ after each decoding and using it as the prior belief for the next time. This way, the additional cost we pay for not having a fully specified source model will decrease over time.

The ability to simultaneously learn from and decompress a data source sequence, combining the concept of machine learning into data compression, is immensely powerful. Our experiments serve as a glimpse into the potentials of a joint learn-decode architecture, and it has shown great promise.

We note that while our architecture can be easily extended to learn source model parameters, source model structure learning is a much more difficult task. In fact, graphical model structure learning is itself a field of active research. The potential for joint compression and structure learning seems promising and this potential application will benefit greatly from current research.

9.4 Robustness to Data Corruption

One important consequence of using an LDPC code to compute the hash x^{kb} is its robustness to data corruption. In this section, we follow the footsteps of Heydegger [20], who in 2009 analyzed the robustness of various compression algorithms. In particular, we experiment with bit erasures and bit flips of the compressed hash x^{kb}, and observe the system's performance in these scenarios.

9.4.1 Bit Erasure

Bit erasure is the scenario in which certain bits of the compressed bit stream x^{kb} are known by the decoder to be inaccurate, which the decoder will disregard. It is obvious that if enough bits are erased such that the number of remaining bits are below the decoding rate-distortion threshold, decoding is impossible. Therefore, if it is known that our communication channel will potentially erase bits, we should compress with redundancy, ie. with more bits than necessary for decoding.

To simulate bit erasure, we experimented with randomly choosing a subset $\mathscr{E}$ of the compressed bits to be erased, where $|\mathscr{E}| = p_{\text{erase}} \cdot kb$ with p_{erase} being the bit erasure

ratio. Table 9.41 summarizes our experiment results. We note that for all instances, the effective rates, defined to be

$$r_{\text{eff}} := \frac{|\mathscr{D}| + kb - |\mathscr{E}|}{n} \tag{9.40}$$

decreases as $\mathscr{E}$ increases. While this may seem surprising, we note that by erasing certain hashed bits x, we are also removing the edges associated with those hashed bit, inducing a non-uniform degree distribution on the effective LDPC code, ie. an irregular code. This agrees with the results of the experiments in Section 8.1.1.2, where we showed we can enhance performance by using an irregular LDPC code instead of a regular one.

p_{erase}	0	0.01	0.02	0.05	0.10	0.20	0.30	0.40
`Proto` rate r_p^*	2.60	2.61	2.63	2.66	2.75	N/A	N/A	N/A
Effective rate $r_{p,\text{eff}}^*$	2.60	2.59	2.60	2.58	2.58	N/A	N/A	N/A
Partial rate	2.60	2.61	2.63	2.66	2.75	2.96	3.12	3.50
Eff. partial rate	2.60	2.59	2.60	2.58	2.58	2.58	2.48	2.50
`Convg` rate r_h^*	2.78	2.78	2.79	2.88	N/A	N/A	N/A	N/A
Effective rate $r_{h,\text{eff}}^*$	2.78	2.76	2.75	2.79	N/A	N/A	N/A	N/A
Partial rate	2.78	2.78	2.79	2.82	2.86	3.02	3.40	3.74
Eff. partial rate	2.78	2.76	2.75	2.73	2.67	2.62	2.68	2.64

Table 9.41: `Proto` and `Convg` rate results for decoding with different bit erasure rates p_{erase} on a Gauss-Markov source with $n = 500$, $b = 4$, and $w = 0.60$. Note that the effective rates *decreases* as p_{erase} increases. This agrees with the results that using irregular LDPC codes enhances performance.

With a higher erasure ratio p_{erase}, however, we start noticing that some of the translated bits z were decoded incorrectly because all the hashed bits x connected to it were erased. The rest of the z bits, however, are still decoded correctly, thereby causing a distortion that is barely higher than complete correct decoding (ie. with a slightly lower SQNR_{dB}). We present the partial decoding threshold in Table 9.41, where all bits for which we have information are decoded correctly.

Note that we do not need to explicitly model our communication channel as a binary erasure channel (BEC) or do any additional error correcting code on top of our encoding, for our inferential decoder already has this inherent capability.

9.4.2 Bit Flip

Bit flip is the scenario in which certain bits of the compressed bit stream x^{kb} are flipped from 0 to 1 or vice versa, unbeknown to the decoder. In our experiments, we randomly choose a set $\mathscr{F}$ of hashed bits to be flipped, where $|\mathscr{F}| = p_{\text{flip}} \cdot kb$ with p_{flip} being the bit flip ratio.

9.4.2.1 Naive Decoding

We first experiment with encoding and decoding naively, ie. encoding without any additional structural changes other than increasing the compression rate, and on the decoder side, we pass

$$(1 - p_{\text{flip}}) \cdot \mathrm{x}_a + p_{\text{flip}} \cdot (1 - \mathrm{x}_a) \tag{9.42}$$

as $\mathcal{X}$ to $\mathcal{F}^{\mathcal{H}}$ messages instead of the 0-1 messages

$$\mathbb{1}\{x_a = \mathrm{x}_a\} \tag{9.43}$$

in Equations 3.33.19 and 4.25.

p_{flip}	1 bit	0.001	0.002	0.005	0.010	0.020	0.050
`Proto` rate r_p^*	2.61	2.63	2.63	2.69	2.72	N/A	N/A
Partial rate	2.61	2.63	2.63	2.67	2.69	2.67	3.07

Table 9.44: `Proto` rate results for decoding with different bit flip rates p_{flip} on a Gauss-Markov source with $n = 500$, $b = 4$, and $w = 0.60$. Note that convergence is very sensitive to p_{flip}, because as more and more bits becomes flipped, there are more and more conflicting information circulating due to the nature of message passing.

Table 9.44 summarizes our experimental results, recording the `Proto` rate. We note that the `Convg` rate cannot be reported, for we cannot find a code that always correctly decodes regardless of the positions of the bit flips. Moreover, as the bit flip rate p_{flip} is larger than 0.02, no complete reconstruction seem possible: the decoded values, even at extremely high rates, are always a couple bits off and are never perfect.

We observe that bit flips are much more destructive than bit erasures for our system. For low bit flip ratios, our algorithm can still largely recover from the corruption with more redundant bits, but for bit flip ratios higher than 0.02, the algorithm simply cannot converge, for there are many conflicting messages that causes the algorithm to loop between different nonsensical states.

9.4.2.2 Binary Symmetric Channel (BSC) Modeling

While theoretically the naive algorithm in Section 9.4.2.1 above has enough information to correct for the corruption, in practice it usually fails, as we have seen above. This is because of the lack of Hamming separation among the code words. Even with a non-binary message in Equation 9.42, BP still favors the incorrect values of the corrupted bits by a very wide margin.

To resolve this issue, one approach is to look to traditional channel coding methods and model the communication channel as a Binary Symmetric Channel (BSC). We can thus use channel coding on top of our architecture by encoding the compressed bit stream x^{kb} with a linear error correcting code G, and the resulting encoding $G^T x^{kb}$ will be sent over the channel. On the decoder side, we will have to first decode for the

compressed bit stream x^{kb}, before running BP to decompress for the original source sequence s. Such an approach, while it works, is very cumbersome.

We observe that our decompressor is itself an inferential algorithm that can elegantly handle the uncertainties of channel corruption. Instead of having two disjoint systems, we can augment our joint decode graph to perform joint channel denoising and decompression. We can use another LDPC code G to encode x and send the encoded bits over the channel. We note the important distinction that this usage of LDPC code is for *channel coding*, which is different from the core of our system of using LDPC codes for compression.

Since the capacity of a BSC with a flip probability p_{flip} is $1 - \mathbb{H}(\mathsf{e})$ [35], where $\mathsf{e} \sim \mathbf{Bern}(p_{\text{flip}})$, we need at least

$$\ell kb := \left(\frac{1}{1 - \mathbb{H}(\mathsf{e})} - 1 \right) kb \tag{9.45}$$

more bits to be sent over the channel for information recovery.

As mentioned in Section 2.2.2, however, general LDPC codes for *channel coding* have encoding time complexity $O(n^2)$. The sparseness of the parity check matrix H only has implications on the decoding complexity, and with the *generator matrix* G of the LDPC code usually not sparse, a full matrix multiplication is need to calculate the codeword. To keep the time complexity for coding low, therefore, we shall use a subset of LDPC codes called *low density generator matrix* (LDGM) codes, where the generator matrix G itself is also sparse [40].

Hence, at the Encode side, we shall use an LDGM code with generator matrix

$$G^T = \begin{bmatrix} I_{kb} \\ B \end{bmatrix} \tag{9.46}$$

with $B \in \mathbb{Z}_2^{\ell kb \times kb}$ being a sparse matrix and I_{kb} denoting the identity matrix, sending

$$G^T \mathrm{x} =: \begin{bmatrix} \mathrm{x}^{kb} \\ \mathrm{y}^{\ell kb} \end{bmatrix} \tag{9.47}$$

over the channel.

On the decoder side, the corrupted bits x' and y' are received. To decode, we run BP over the joint graph with the code subgraph replaced with a joint code-channel subgraph $\mathcal{H}$ and $\mathcal{B}$, as illustrated in Figure 9.48. In the figure, we see that the received corrupted values x' and y' enter the graph through the variable nodes $\mathcal{X}'$ and $\mathcal{Y}'$, which helps in determining the true values of x of the variable nodes $\mathcal{X}$.

We note that in channel coding, LDGM codes in the form of Equation 9.46 usually performs poorly in practice, because the $\mathcal{X}'$ nodes have degree 1 and are therefore not capable of updating their (potentially faulty) beliefs. For our purpose, however, it provides good performance, for we know the prior distribution of x through the top portion of the joint graph.

The message passing equations in the BSC subgraph $\mathcal{B}$ is the same as described in Section 4.4.1. We summarize our experimental results in Table 9.50, where the

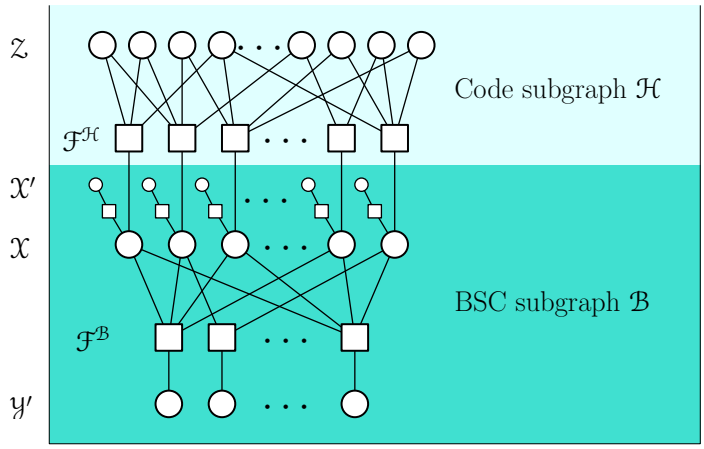


Figure 9.48: Handling flipped bits by modeling the received bits as communicated through a binary symmetric channel (BSC). We perform an additional coding with an LDGM code defined by matrix B. On the decoder side, we add a BSC subgraph $\mathcal{B}$ into the graph to represent the relationships imposed by B.

effective rate is defined to be

$$r_{\text{eff}} := \frac{|\mathcal{D}| + (kb + \ell kb)(1 - \mathbb{H}(e))}{n} \tag{9.49}$$

to correct for the capacity of the channel.

p_{flip}	1 bit	0.001	0.002	0.005	0.010	0.020	0.050	0.100	0.200	0.300
r_p^*	2.63	2.67	2.69	2.70	2.78	2.95	3.30	4.10	6.95	14.87
$r_{p,\text{eff}}^*$	2.62	2.65	2.65	2.62	2.64	2.67	2.64	2.65	2.65	2.65

Table 9.50: The `Proto` rate r_p^* and corresponding effective rate $r_{p,\text{eff}}^*$ for decoding with different bit flip rates p_{flip} on a Gauss-Markov source with $n = 500$, $b = 4$, and $w = 0.60$.

Unlike the naive method in Section 9.4.2.1 above which quickly breaks down with a small bit flip rate, BSC modeling and LDGM coding works for all values of p_{flip} attempted, even at moderately high corruption rates like $p_{\text{flip}} = 0.3$. In line with the well established results from channel coding, the rate needed for successful decoding increases drastically as p_{flip} approaches 0.5. However, we observe that the effective rate is essentially constant across different values of p_{flip} attempted.

9.4.3 Discussion

As we presented above, our system can elegantly handle data corruption without changes to the core architecture. By using an LDPC code to compute the hash, our algorithm is robust towards bit erasures, with the effective rate not affected even with considerable erasures. On the other hand, LDPC compression is less robust towards bit flips, as the corrupted bits contain erroneous information that tend to propagate throughout the system, causing confusion. However, with an additional

LDGM encoding on the compressed bits, data can be reconstructed without affecting the effective rate.

We note that we have only experimented with corrupting the hashed bits x, but not the dope bits $z_{\mathscr{D}}$. We observe that a dope erasure is equivalent to not doping the bit, which will have a small negative effect on the total compression rate. On the other hand, a dope flip will be trickier to handle, and we can potentially use another LDGM coding on the dope bits, but the actual implementation details will require further investigation.

The architecture's robustness to data corruption is remarkable. As Heydegger's 2009 paper [20] analyzed the robustness of compression algorithms, a single bit flip can have catastrophic implications for most image compressors, especially JPEG and PNG. With ZIP files, even a single bit erasure, let alone a bit flip, will render complete corruption of the data.

While one of the major challenges of digital communication has been combating channel noise, it is truly surprising that very little effort has been made on this subject in this dual problem of data compression. As Heydegger noted, a file's robustness to data corruption was not given the attention it deserves in the design of general file formats, or even in the design of compression algorithms, where data corruption has a much more catastrophic consequence. Heydegger's work was the first, and remains one of the very few, comprehensive survey of this topic. We believe that data robustness should be an important criterion in designing a compression system, and the lack of research in this fundamental subject is truly intriguing.

9.5 Encrypted Compression

Encrypted compression refers to compressing encrypted data without first decrypting the data. For traditional model-specific compression algorithms, such a task is impossible, for the source model is necessary at compression time. However, with a source-agnostic encoder, our architecture can handle such a task with some minor modifications.

9.5.1 One-Time Pad Encryption

As first introduced by Johnson et al. [23] in 2004, refined by Schonberg et al. [32] in 2006, and reaffirmed by Huang [21] in 2014, LDPC compression can be done on data encrypted with a one-time pad.

A one-time pad secret key k^n is a binary bit stream of length n that is randomly generated according to $\mathbf{Bern}\left(\frac{1}{2}\right)^n$. The pad length must be the same length as the plaintext s^n, and the encrypted ciphertext $\bar{\mathrm{s}}^n$ is taken to be

$$\bar{\mathrm{s}}^n = \mathrm{s}^n \oplus \mathrm{k}^n \tag{9.51}$$

where $\oplus$ denote bitwise addition over $\mathbb{Z}_2$, ie. the exclusive-or (xor) operation. As a notation, we use bar (eg. $\bar{s}$) to denote an encrypted version of a variable.

As Shannon has proved in his 1949 paper [34], a one-time pad achieves perfect secrecy in the sense that decryption is impossible without knowledge of the secret key k^n. However, as its name suggests, a one-time pad k^n can only be used once. If the same one-time pad is reused in part or in whole, the secrecy guarantees of the scheme breaks down, and information about the plaintext s^n can be recovered in part or in whole by an eavesdropper of the ciphertexts who has no a priori knowledge of k^n. This can be seen by observing that if an adversary takes the xor of two intercepted messages encrypted with the same one-time pad, the one-time pad is canceled out, leaving the adversary with the xor of the two plaintexts, which the adversary can decode by partial guessing or brute-forcing.

Therefore, to build the crypto-compression system around a one-time pad, there needs to be two communication channels: a public one that transmits the compressed bit stream, and a secure one that transmits the one-time pad which has the same length as the plaintext source. Given the non-reusable nature of the one-time pad, with the transmission cost of the pad we could have simply transmitted the unencrypted plaintext over the secure channel.

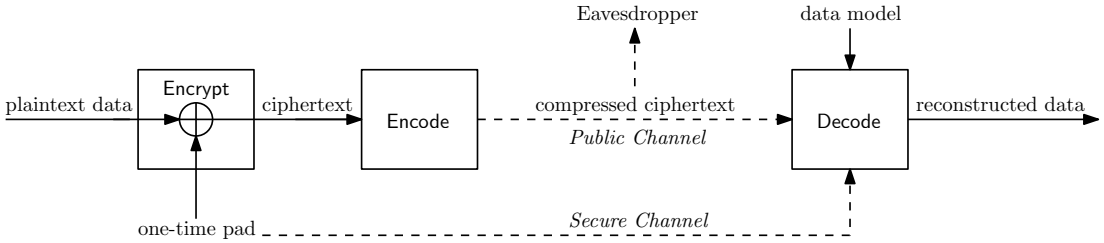
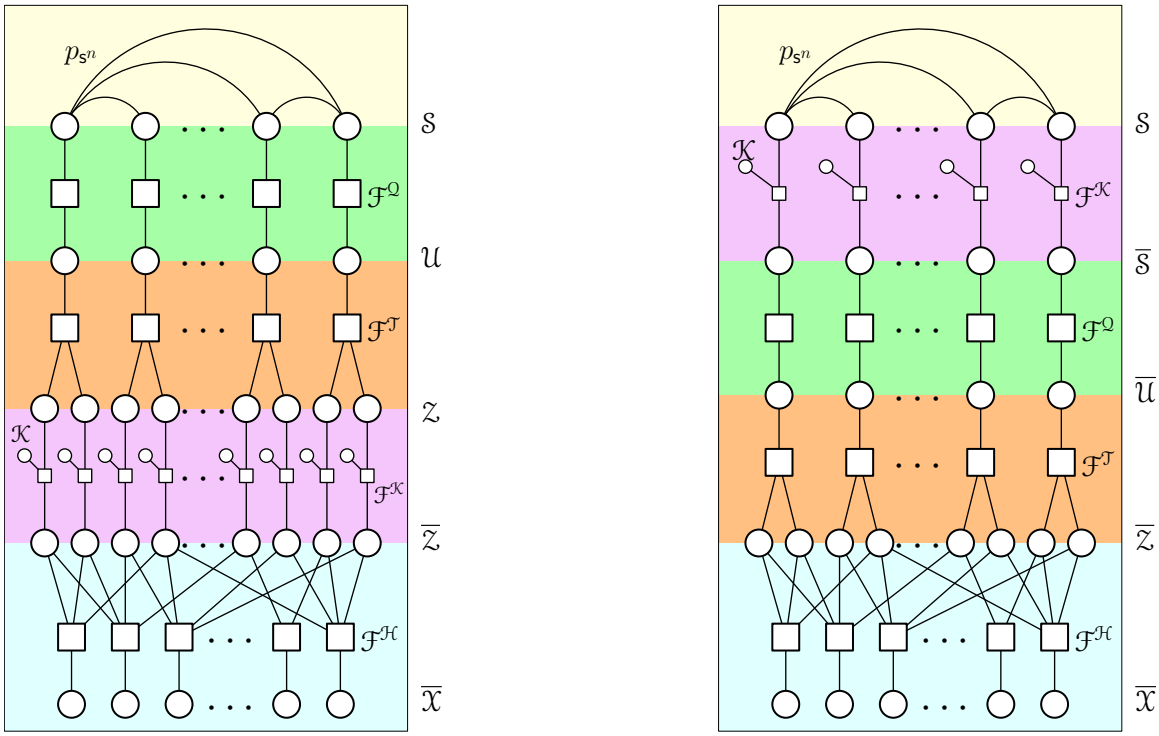


Figure 9.52: The block diagram showing information flow in the encrypted compression architecture proposed by Johnson et al. Note that there are two channels: the public one for the compressed ciphertext, and the private one for the one-time pad. An eavesdropper who only has access to the compressed ciphertext will not be able to decompress and decrypt the data.

We can thus replace the random one-time pad with a pseudo-random one, with the encryptor communicating only the random seed over the secure channel to the decoder. This transforms the one-time pad into a stream cipher, which has a much weaker security guarantee in that it is not informational theoretically secure. In particular, with the random seed being much smaller than the pad itself, stream cipher is potentially susceptible to brute force attacks, attacks that would have been ineffective on truly random one-time pads. Moreover, as with one-time pads, the secret key (ie. the random seed) of stream ciphers cannot be reused. Hence, the number of sequences that the system can compress is fundamentally limited by the size of the stream cipher secret key.

However, we can increase the size of the stream cipher key to defend against brute force attacks. With a key length in the hundreds, brute force attacks become practically infeasible and the number of possible keys will practically not limit the number of sequences that can be compressed, at the same time the secret key is small enough to justify sending it through a secure channel.

We note that our system can easily be modified to accommodate the compression of data encrypted with a one-time pad, given the source-agnostic nature of our en-



(a) A binary one-time pad.

(b) A Gaussian one-time pad.

Figure 9.53: The decoding graphs for our system modified for quantized and translated sources encrypted with a binary one-time pad (left) and for sources encrypted with an additive Gaussian one-time pad (right). Note that the secret key k enters the system through factor nodes (purple) corresponding to how it encrypts the data.

coder. For the sake of simplicity, we let our quantization matrix Q be the identity matrix I. At compress time, the compressor is given a quantized and translated source sequence $\bar{\mathrm{z}}^{mb}$ encrypted by a one-time pad k^{mb} drawn from $\mathbf{Bern}\left(\frac{1}{2}\right)^{mb}$ (or generated by a stream cipher). At decompress time, the decoder runs BP on the joint graph illustrated in Figure 9.53a, which takes both x^{kb} and k^{mb} as inputs. The one-time pad enters the graph through the additional factor nodes, highlighted in purple, which reverses the effects of xor-ing the one-time pad before passing information from the code subgraph to the rest of the graph.

Another technique suggested by Johnson et al. in [23] for continuous sources is to encrypt the original sequence with additive Gaussian noise, which are then quantized and translated into a binary sequence by a trellis code. We note that while Johnson et al. used a trellis code for compression, the idea of encryption with additive Gaussian noise can also be easily incorporated to our LDPC based system, as shown in Figure 9.53b. However, the issue of transmitting the secret key remains, and for a key that is itself Gaussian, quantization needs to be done on the key before transmission over the secure channel (Figure 9.52), another challenge that remains to be resolved.

9.5.2 Potentials for Homomorphic Encryption

Compressing encrypted data and joint decryption-decompression is of much theoretical interest. On the other hand, the ability to compute on compressed data addresses a very practical concern. Given that our compression architecture is based on linear codes, the linearity of matrix multiplication transfers over as an inherent property of our system as well.

Let s_1 and s_2 be two source sequences generated by two source models $\sim \mathbf{N}(\mu_1, \Sigma_1)$ and $\sim \mathbf{N}(\mu_2, \Sigma_2)$ respectively. For a moment, let us set aside the translator component, assuming access to a non binary $\mathbb{Z}_p$ LDPC code, which is discussed in Braunstein et al. [5]. We thus compress s_1 and s_2 by our architecture without a translator, ie. with only $q(\cdot)$ and $h(\cdot)$. If we only have access to compressed streams $x_1, x_2 \in \mathbb{Z}_p^m$, we have

$$\begin{aligned} x_1 + x_2 &= Hu_1 + Hu_2 \\ &= H(u_1 + u_2) \\ &= H \cdot (Qs_1 + Q_0 + Qs_2 + Q_0)) \\ &= H \cdot (Q(s_1 + s_2) + 2Q_0) \\ &= x_{12} + HQ_0 \end{aligned} \tag{9.54}$$

where x_{12} denotes the bit vector compressed from $(s_1 + s_2)$. Hence, our compression system is homomorphic for addition. For decompressing the sum of two vectors, however, we will need to use a source model that corresponds to the sum of the two distributions. With the two sequences being drawn independently from their respective models, the distribution of the sum will thus be

$$\mathbf{N}(\mu_1 + \mu_2, \Sigma_1 + \Sigma_2) \tag{9.55}$$

To represent an integer in $\mathbb{Z}_p$, however, we will eventually need a translator. To keep the whole system linear, therefore, our translator of choice $t(\cdot)$ needs to maintain this property as well. One such translator is the two's complement translator (Section 4.2.1), which preserves addition. With enough bits, the integer overflow issue of the two's complement representation can be resolved.

With our compression system being homomorphic, it is natural to extend our system with homomorphic encryption. In fact, any encryption homomorphic over addition can be applied to the compressed vector to give a homomorphic compressor-encryptor. However, encryption after compression is not a flexible architecture, as we lose the ability to manipulate the final result, the benefits of which include further compression, as described in Section 9.2. Encrypting before compression can harness the power of our compression architecture better.

In particular, we are interested in the integer vector homomorphic encryption scheme proposed in Zhou and Wornell's 2014 paper [41], which has been shown to be secure (via a reduction from the Learning with Errors problem) and demonstrated to be computationally viable by Yu et al. [39] in 2015.

Zhou's integer vector homomorphic encryption scheme is designed for client-cloud

interactions, where the client stores encrypted data in the cloud (an untrusted party), with the cloud doing blind computation on the encrypted data. The scheme supports three operations on integer vectors, namely addition, linear transformation, and weighted inner product. As a brief exploration of potential extensions to our system, incorporating this scheme into our architecture will allow homomorphic encryption and homomorphic compression for the addition operation.

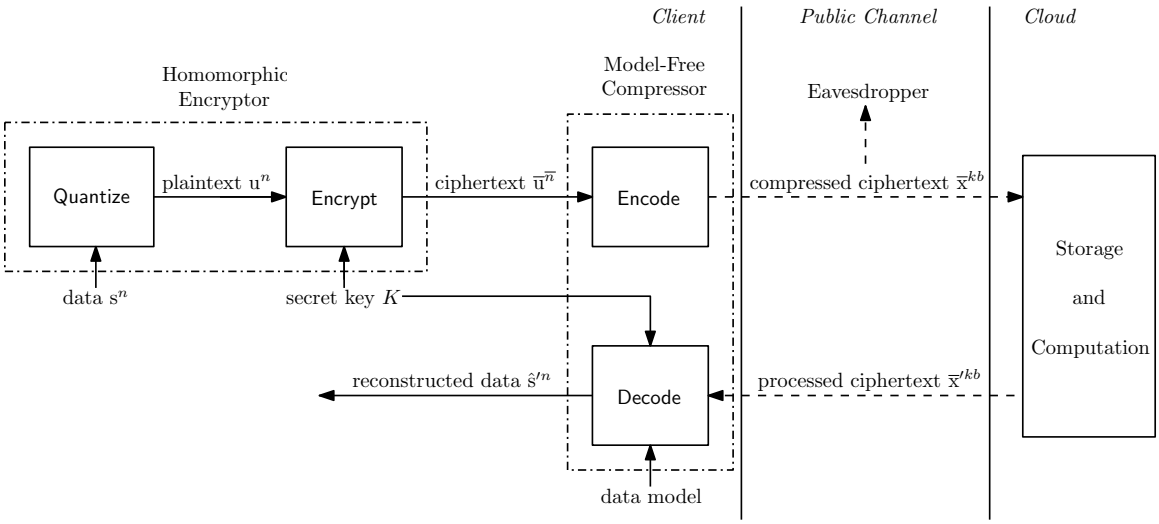


Figure 9.56: A block diagram for a homomorphic encryption-compression scheme, achieved by incorporating Zhou's integer vector homomorphic encryption scheme into our architecture.

Zhou's scheme defines the ciphertext $\bar{u} \in \mathbb{Z}^{\bar{n}}$ of an integer vector $u \in \mathbb{Z}^n$ encrypted by secret key $K \in \mathbb{Z}^{\bar{n} \times n}$ to be a vector that satisfies

$$K\bar{u} = \omega u + e \tag{9.57}$$

for some large scalar ω and for some error vector e. With the secret key, decryption becomes

$$u = \left\lceil \frac{K\bar{u}}{\omega} \right\rfloor \tag{9.58}$$

The method of encryption is mathematically heavy and is omitted here, for it distracts from our discussion in the sense that we are primarily concerned about compressing an already encrypted source and its joint decryption-decompression. Interested readers are encouraged to refer to [39], in which the encryption method is detailed.

Hence, for compressing the already encrypted source $\bar{u}^{\bar{n}}$, we simply run our Encode procedure without quantization, ie. only hashing then translating, as illustrated in Figure 9.56, with

$$\bar{z}^{kb} = t(\bar{x}^k) \tag{9.59}$$

$$= t(H\bar{u}^{\bar{n}}) \tag{9.60}$$

For joint decryption-decompression, we shall run loopy belief propagation on the

graph in Figure 9.61, noting that the secret key $K \in \mathbb{Z}^{\bar{n} \times n}$ is used in the factor nodes in implementing the decryption equation 9.58. It should be emphasized again that the source model in this case would be that in Equation 9.55, which corresponds to the sum of the two random variables. We also note that we cannot perform entropy coding after translating, for this causes the bit stream to lose its homomorphic property: while matrix multiplication by H is linear, having carry digits is not.

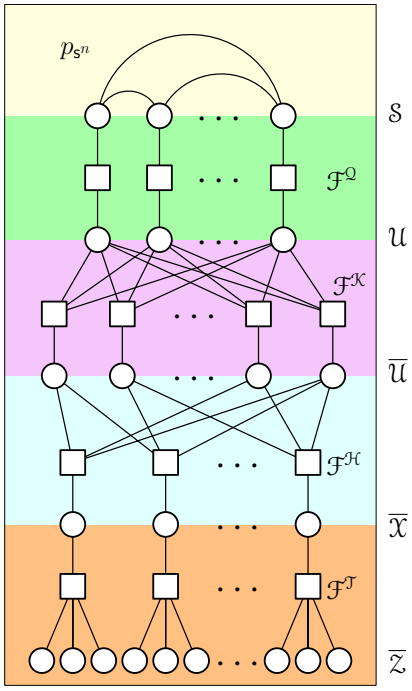


Figure 9.61: The decoding graphs for our system modified for Zhou's integer vector homomorphic encryption scheme. Note that the secret key K enters the system through factor nodes $\mathcal{F}^{\mathcal{K}}$ (purple), and that we hash with a $\mathbb{Z}_p$ LDPC code to get the hashed sequence $\bar{x}^k$ before translating the integer vector into a bit stream $\bar{z}^{kb}$.

We observe that the architectural advantages of our general system still holds for this joint encryption-compression system. In particular, should it be found out that less bits are required for correct reconstruction due to the discovery of a better model, extra bits in the compressed bit stream can be dropped. This operation is not possible if we have compression before encryption.

We note that this homomorphic encryption-compression scheme is by no means a mature system, but rather an exploration in extending our general architecture, meant to illustrate the flexibility of our system given its modularity. In particular, being only able to compute sums of Gaussian vectors seems limiting. It should be noted that many more ideas and research will need to be explored, for it is not obvious how one would extend LDPC compression to be homomorphic for linear transform and weighted inner product (the other two operations supported by Zhou's scheme), given that matrix multiplication by H is in general an non-commutative operation. Potential solutions will have to address this fundamental issue of commutativity.

Moreover, homomorphic encryption itself is an active field of research, the interests

for which was spurred by Gentry's dissertation in 2009 [14]. In particular, most existing homomorphic encryption systems are tied to Lattice-Based Encryption, which is known to be susceptible to Chosen Ciphertext Attacks, an attack model in which the adversary has access to a decryption oracle. Zhou's system is no exception [4]. Refining homomorphic encryption techniques, as well as the possibility of incorporating it to compression, is a problem that calls for further work.

9.6 Secret Sharing

As noted in Section 9.1, only the number of bits, but not the actual positional identity of the bits, matters for successful decoding. These quantifiable bits of information can be re-ordered, partially corrupted or lost, or accumulated from different sources, none of which affect decoding. One natural application of this important property would be secret sharing.

As first pioneered by Shamir in 1979 [33], secret sharing is the concept of distributing information about a secret among γ participants, with each participant given his share of the secret. The secret can be reconstructed when any κ shares of the secret are combined together, but cannot be reconstructed with less than κ shares.

Shamir's original scheme is information theoretically secure, meaning that an adversary will not gain any information about the secret until he has κ or more shares of the secret. Other schemes give a weaker guarantee, while still being secure under most practical scenarios.

9.6.1 Distribution Scheme

Our architecture can be trivially extended into a secret sharing scheme, with the original source sequence s^n being the secret, and the hashed bits x^{kb} being the shares of the secret. Let r be the entropy rate, ie. nr hashed bits are required for successful decoding, and let each share of the secret be of c bits. To ensure that decoding is successful with κ shares of the secret but not with $(\kappa - 1)$ shares, we thus have the inequality

$$(\kappa - 1)c < nr \leq \kappa c \tag{9.62}$$

and solving for c gives

$$\frac{nr}{\kappa} \leq c < \frac{nr}{\kappa - 1} \tag{9.63}$$

Thus, for a suitable c satisfying the above, we hash γc number of bits, and give each of the γ participant c bits.

We note that the total number of participants γ does not matter in the determination of c, but only the threshold κ, the length of the original sequence n, and the entropy rate r.

9.6.2 Security Analysis

We note that in our experiments, successful decoding is abrupt with respect to the rate, ie. even right below the threshold rate, partial decoding does not occur. This gives us more guarantee that adversaries with less than κ shares of the secret will not be able to partially decode the secret with their shares.

However, we note that unlike Shamir's scheme, our scheme is not information theoretically secure. This means that an adversary will gain more information about the secret as they gain more shares of the secret. As an illustrative case, an adversary with $\kappa - 1$ shares knows more about the source sequence than an adversary with 0 shares of the secret. Thus, the adversary with $\kappa - 1$ shares of the secret can potentially guess the last c bits by brute forcing all 2^c combinations, an attack that would be ineffective on information theoretically secure schemes. This, however, is not of practical concern, for nr is typically in the order of tens of thousands and κ is typically small, hence from Equation 9.63, c is typically in the order of the thousands. To launch a brute force attack, the adversary needs to run belief propagation to completion for 2^c times, thus the sheer number of combinations renders brute force attacks practically infeasible.

9.7 Block Decoding & Distributed Memory Parallelism

As noted above, successful decoding happens abruptly as we increase rate. Empirically, partial decoding does not occur when the rate is below the decoding threshold. However, there are circumstances in which partial decoding is desirable.

Decoding a long data sequence is computationally intensive given the size of the joint graph. As we look to compressing large data sequences, the graph structure and the messages associated with decoding a data sequence with length in the order of billions might not even fit in the memory of a single standard machine. In this section, we present a block decoding scheme that allows for partial decoding within a single machine, as well as describe a distributed version of the scheme that potentially allows for parallel decoding across different processors or machines.

9.7.1 Block Decoding Scheme

We now present a coding scheme that allows for partial decoding in terms of subsequences of the original data. We first observe that many sources have sparse correlation structures, hence the source graph $\mathcal{C}$ can be broken down into many subgraphs $\mathcal{C}_i$ with minimal edges between the subgraphs. As an example, the Gauss-Markov source can be broken down into α sub-chains, with only one edge connecting the sub-chains.

Given this observation, we propose the following quantizer and code structure. We break up the quantization matrix $Q \in \mathbb{R}^{m \times n}$ and LDPC matrix $H \in \mathbb{Z}_2^{kb \times mb}$ into

α different sub-matrices and place them in a block diagonal configuration

$$Q = \begin{bmatrix} Q_1 & & & \\ & Q_2 & & \\ & & \ddots & \\ & & & Q_\alpha \end{bmatrix}, \qquad H = \begin{bmatrix} H_1 & & & \\ & H_2 & & \\ & & \ddots & \\ & & & H_\alpha \end{bmatrix} \tag{9.64}$$

where each $Q_i \in \mathbb{R}_2^{(m/\alpha)\times(n/\alpha)}$ is a quantization matrix itself and each $H_i \in \mathbb{Z}_2^{(kb/\alpha)\times(mb/\alpha)}$ is an LDPC matrix itself. By choosing α to be an integer factor of m, we ensure we do not separate groups of b translated bits into two blocks. This corresponds to a graph structure as illustrated in Figure 9.65.

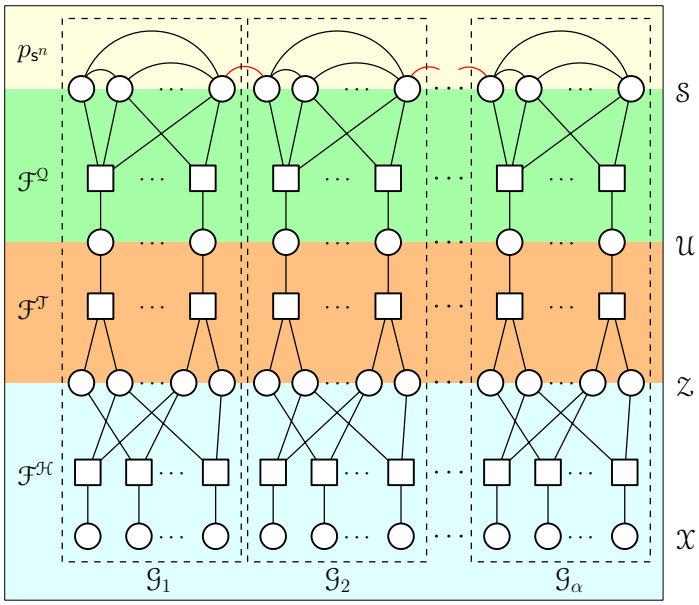


Figure 9.65: The block decoding graph, with the joint graph $\mathcal{G}$ is divided into subgraphs $\mathcal{G}_1, \ldots, \mathcal{G}_\alpha$. In block decoding, we can decode each block $\mathcal{G}_i$ somewhat independently one at a time. While we lose the correlation structure between blocks, we can replace the edge potentials of the affected edges (highlighted in red) with the values of $\hat{\mathbf{s}}_{\mathcal{G}_1}$ as deterministic node potentials for decoding $\hat{\mathbf{s}}_{\mathcal{G}_2}$, and so forth. For distributed memory parallel computing, we decode within each block for a number of iterations, then allow exchange of messages between different blocks, and iterate until convergence.

With this structure, we can thus decode each block $\mathcal{G}_i$ by itself, the size of which can be tuned to be small enough to fit in the memory of a single machine. We then decode the blocks one at a time on the same machine. Note that with this decoding procedure, we are treating the blocks as independent and thereby lose the correlation across different blocks. We remedy this by replacing the affected edge potentials (red edges in Figure 9.65) with node potentials on the undecoded nodes, using the already decoded values $\hat{\mathbf{s}}_{\mathcal{G}_i}$ to calculate the node potential of the affected nodes in $\mathcal{S}_{\mathcal{G}_{i+1}}$.

We implemented and tested block decoding on the Gauss-Markov model with $n = 5000$ and various values of α. We note that while the rate performance of all choices of α remain similar when decoding is successful, the time performances are not, as illustrated in Table 9.66. We note that with a smaller α, the decoding time

is longer, and the time decreases as we increase α. This difference, however, is not caused by the change in *time complexity* of each iteration, since our algorithm is linear, ie. of $O(n)$. Rather, the difference is caused by the *number of iterations* to convergence. With a larger code matrix with more loops, the number of iterations to convergence is larger.

α	1	2	5	10	20
Avg. total time to convergence (s)	6.79	4.33	2.48	2.37	2.56
Avg. iter. to convergence (per block)	51	35	24	21	20
Avg. time (μs) per iter. / mb	5.32	4.96	4.14	4.53	5.14

Table 9.66: Average iterations to convergence and time per iteration per element for different choices of α, with $n = m = 5000$, $b = 5$. We note that the decrease in overall time is due to the decrease of iterations to convergence, not the time complexity of each iteration.

On the other hand, the reduction in time for a higher α comes with a cost. With a larger choice of α, we start noticing more and more instances of non-convergence of at least one of the α blocks. This can be attributed to the fact that we are not using the full information available in the Gauss-Markov model. In particular, by severing the Markov chain into multiple pieces, we are making greedy local optimizations within each block, where each block only has information from previous blocks, but not from the succeeding blocks. In the extreme case where $\alpha = n$, the forward-backward algorithm on the Markov chain (a Kalman smoother) is transformed into a "forward only" algorithm (a Kalman filter), which has a worse accuracy than using the full correlation information. This suggests that to preserve accuracy, we want to have α small.

9.7.2 Potentials for Distributed Memory Parallel Decoding

The above structure, which is sequential in nature, can be modified to be implemented on multiple machines, each with a separate non-shared memory. This allows for parallelization over over a distributed network. To replace the deterministic messages from one block to another, we can configure the blocks to communicate the outgoing messages to one another periodically. This structure gives the advantage of parallel computation, while preserving message passing between the blocks and solves the issue with using only messages from one side.

We note that as [10] discussed, synchronizing message passing among different blocks is a difficult problem to solve in practice, for the computation of each block will vary in time, and the convergence of belief propagation is extremely sensitive to these implementation details. Parallelization of message passing is an area of active research, and inference algorithms including ours will benefit from it greatly.

9.8 Progressive Decoding

Distributed decoding allows for faster decompression, but users may not have access to multiple machines. On the other hand, block decoding on a single machine may not be useful in certain scenarios. Take, for example, an image being decompressed: block decoding decompresses contiguous regions of the image and shows it to the user one at a time. This partial decoding method is not the most useful because to the user, knowing part of the image to great detail is not as helpful as knowing the whole image to less details.

As a realistic scenario, consider a compressed image with a large file size being sent over a slow network: for a normally encoded image, the receiver will not be able to decode the image until the whole file finished downloading, which adversely affects user experience. It would be desirable for the receiver to be able to partially decode the image with the bits that it has received so far, generating a low quality preview of the image while waiting for the rest of the file to download, thereby greatly improving user experience.

This concept is called progressive decoding, and it has been incorporated into some versions of the new JPEG2000 standard [31], although support for it is far from universal. With the flexibility of our architecture, we note that we can also incorporate this concept into our algorithm. This is another way of partial decoding: instead of partial decoding in blocks of elements of the source, progressive decoding decompresses with increasing resolution.

9.8.1 Implementation Overview

Progressive coding requires a slight modification and coordination of the code component $h(\cdot)$ and the translator component $t(\cdot)$. In particular, in the compressed bit stream we want bits with usable self-containing information to be placed first, with the later bits serving as incremental refinements. Therefore, we adopt the following modifications.

On the encoder side, we use standard binary coding for the translator $t(\cdot)$, as its natural embeddable binning makes it ideal for refinements (Figure 9.68). Moreover, we choose code $h(\cdot)$ such that the corresponding H matrix hashes the translated bits z^{mb} in the order of bit significance. In particular, we define a permutation $\sigma(\cdot)$ of the z bits that groups the bits by significance, as illustrated in Figure 9.69. We then hash the permuted bits $\sigma(z)$ by a H matrix that has the same block diagonal configuration as in Section 9.7, ie.

$$H = \begin{bmatrix} H_1 & & & \\ & H_2 & & \\ & & \ddots & \\ & & & H_b \end{bmatrix} \tag{9.67}$$

where each H_i have the same width but can have different heights. Within each block H_i we randomly dope with a different rate.

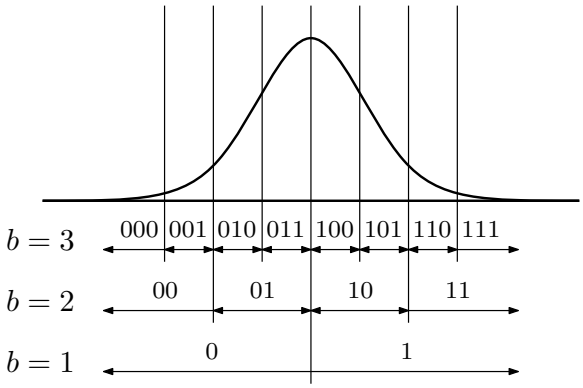


Figure 9.68: Progressive encoding with $b = 3$. Note that the embeddable nature of the standard binary code allows for progressive refinements over different quantization levels b.

On the decoder side, we run belief propagation with the hashed bits that corresponding to the first quantization level, as illustrated in Figure 9.69. To decode the next level, we repeat the procedure, with the previously decoded bits used as dope bits to send deterministic messages to the translator nodes. Note that in this structure, each level of significance has its own LDPC matrix. As we decode each level progressively, our estimate s^n gets more and more precise with each pass.

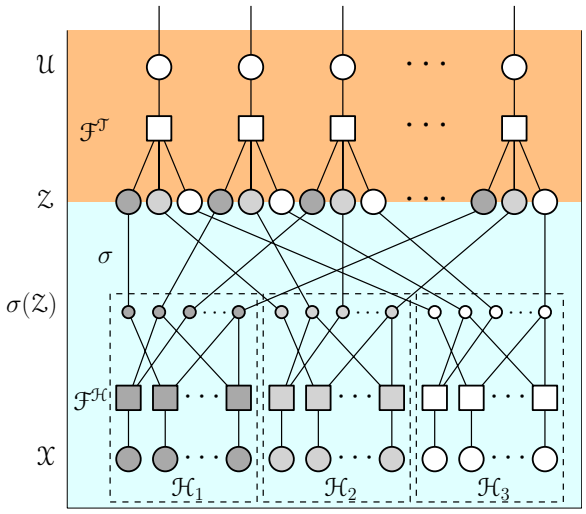


Figure 9.69: Progressive decoding with $b = 3$. The σ subgraph permutes the translated bits z so that they are grouped together by the significance of the bits (indicated by the shades of gray on the $\mathcal{Z}$ nodes). The mb translated bits are ordered into b groups, each with m bits. Each group is then hashed separately by an LDPC matrix $\mathcal{H}_i$, where each $\mathcal{H}_i$ can have a different compression ratio and dope rate. Decoding is done progressively: with the x bits received in order, we first only on $\mathcal{H}_1$ to decode the first bits of z, then as we receive more x bits, we use the decoded $z_{\mathcal{H}_1}$ bits as dope bits to decode on $\mathcal{H}_2$, and so on.

For doping in successive levels, we use the previously decoded bits as dope bits. This greedy approach, however, means that if a single level fails to converge to the correct value, all successive levels will fail. To remedy this, when we know that a level has not converged, for the next level we dope bits that we know for sure are

correct, with the hope that this next level can correct the error. This prevents the errors from snowballing through the levels. Note that it is fine if an intermediate level decodes incorrectly as long as the final level decodes correctly, for it only generates an imprecise preview, and will be overridden by the more precise version from the final level anyway.

9.8.2 Results and Discussion

With the progressive decoding setup above, we experiment with compressing a Gaussian iid source of length $n = 500$ with $b = 3$ levels of precision, the results of which are shown in Table 9.70. Note that with bits of successive levels of precision being used as dope bits for the next level, the computation needed for each level of decoding is reduced. We see that with progressive decoding, the overall rate achieves a similar performance as a normally compressed sequence.

Precision b	1	2	3
Slab width w	2.4	1.2	0.6
Progressive rate r_p^*	0.90	1.60	2.55
Non-progressive rate	0.88	1.54	2.54

Table 9.70: Compression rates of progressive decoding at different precision levels with $b = 3$. The non-progressive rate (from normally encoded sequences) are also presented for comparison. Note that progressive decoding has a similar rate performance as normally encoded sequences at the respective precision levels.

Progressive decoding is of much practical interest, for message passing decoding is a heavy weight process which can take considerable computational resources for large data sequences. Being able to generate a lower quality preview as the rest of the file is decompressing is thus especially valuable.

Another application of our progressive decoding scheme is that it provides different levels of precision. Depending on the need, the user can instruct the decoder to terminate at a lower level of precision without decoding the whole file, when a less precise reconstruction suffices. This flexibility gives users the freedom to make their own precision-computation trade off at decompression time.

We note that for a decoder that is not configured for progressive decoding, the normal procedure of running belief propagation on the full graph also works fine. Since the implementation of progressive decoding is not trivial, this compatibility gives the implementer the freedom to decide whether this additional feature is necessary. This flexibility is another advantage of this extension to our architecture, and our architecture in general.

9.9 Summary

In this chapter, we presented eight applications and extensions of our system, with the first four (rate selection, model imprecision, model learning and corruption robust-

ness) being realistic scenarios our system can elegantly handle with minor changes, and the other four (encrypted compression, secret sharing, progressive decoding, and parallel decoding) being of a more exploratory nature with glimpses of promise. As we can see in these brief surveys of different ideas, the extensibility of our system gives immense potential to the architecture, and it opens the door for further research to transform our system into mature products which can resolve many more problems not imaginable at the time of the writing of this thesis.

10

Conclusion

Let us summarize the contributions of our work and present insights for refining and extending our architecture for further research.

10.1 Summary

In this work, we have presented a data compression algorithm for Gaussian sources that separates the concerns of source modeling and encoding. With graphical model as our representation and message passing as our main tool for decoding, we can make the structure of the algorithm extremely flexible. Its modularity allows it to be a general compressor for Gaussian sources, working with different sources with different underlying source models, as well as a foundation on which different functional modules can be built. Its sparse nature gives the algorithm a low time complexity. Its method of decoding, message passing, implicitly handles the rate-distortion trade-off, affording it a reasonable performance despite the strong restriction that the algorithm does not have information about the source at compression time.

10.2 Future Work

Despite the advantages mentioned above, however, our system is far from perfect. In this section, we summarize insights that we gained in this work, offering potential ideas for refinement of our architecture and for further research. For reference, the MATLAB and C source code of this work will be made available at http://github.com/amagicube/compression.

10.2.1 Source Modeling

In our work, we experimented primarily with two source models: the Gaussian iid source and the Gauss Markov source. Real life data, however, have much more complicated correlation structure than either of them. More correlation structure

means less entropy hence more compression is possible, so in a sense the iid source with no correlation structure is the hardest to compress.

However, the practical question of determining the correlation structure of the source arises. As mentioned in Section 9.2, often times it is hard to know the correlation of the source a priori. In determining the correlation structure, however, we note that structure learning is a difficult task under active research. While more work needs to be done in this area, the results in Section 9.2 demonstrate that sources can be reconstructed at a higher rate with a trivial source graph (ie. independence), regardless of the actual underlying correlation structure, and this can serve as a baseline for us. On the other hand, the results from Section 9.3 show that modifying our architecture for parameter learning is trivial, when the graphical model structure is known.

Many fields of expertise, including financial, medical, biological, and natural image data sets, have accumulated domain specific knowledge over decades. While the process of quantifying these knowledge will not be trivial, we are excited by the prospect of consolidating these knowledge, and with the learning mechanism of our architecture, contributing to these fields with our data compression algorithm.

10.2.2 Quantizer Selection and the Sub 1-Bit Regime

In the discussion of a choice of quantizer in Chapter 5, we noted that choose a low density hashing quantizer with $Q \sim \mathbf{Bern}(\frac{\xi}{m})^{m \times n} \cdot \mathbf{N}(0,1)^{m \times n}$ does not seem to be able to provide good performance. For the rest of the thesis, we primarily used a uniform scalar quantizer with $Q = \frac{1}{w} I$ for various choices of quantizer width w.

While using a uniform scalar quantizer gives good rate performance, further research needs to be done on low density hashing quantizers. In particular, with a quantizer output that can be of a different size than the input, we will have a tool to tackle the sub 1-bit regime (Section 5.1.2.1), an interesting case is not well studied. With the modularity of our architecture though, we can explore different possibilities of quantizers, and potentially use a completely different quantizer structure all together, without impacting the rest of the algorithm.

Lossy compression in itself is a complex problem, for understanding human perception of distortion is an interesting field that calls for further research. In our work, we used the mathematically sensible distortion measure of minimum squared error (MSE). While the general structure of our algorithm will not change, the design of the quantizer will definitely depend on the choice of distortion measure.

10.2.3 Code Selection

In Chapter 4, we presented the different parameters of the LDPC code matrix H, and in Chapter 8, we briefly discussed some optimization choices those parameters. In Chapter 9, we experimented with different configurations of the code to allow for different functionalities like parallel decoding (Section 9.7) and progressive decoding (Section 9.8).

We can see that while the main purpose of the code portion is to perform the actual compression, its structure also needs to be carefully chosen should we want to incorporate additional features. As explored in Section 8.1, for the general case we have had the most success with H generated randomly with column degree distribution set to $\varrho'(x) = [0.4 * 3/0.6 * 2](x)$. While the `Proto` rate r_p^* is already on the theoretical lower bound of our algorithm, more tuning and a more refined choice of the degree distribution can be expected to close the gap between this bound and the `Convg` rate r_h^*.

10.2.4 Extensible Modules

In Chapter 9, we discussed multiple extensible modules that can augment our architecture for extra functionalities, including source model learning (Section 9.3), channel modeling (Section 9.4), and one-time pad encryption (Section 9.5).

These components are structurally independent from the general architecture, and can be used for additional features without affecting the core of our algorithm. As more and more data are generated, data compression are often coupled with other tasks, and with our architecture we can achieve simultaneous operation of decoding and other tasks that may not be known at the time of the design of the architecture, made possible by the modular nature of our architecture.

With different features desired, different modules can be devised to augment our architecture. This opens up a grand avenue for further work for our algorithm, and its potential is limitless.

10.2.5 BP Acceleration

Our MATLAB implementation of the decoding algorithm has been highly vectorized for time performance. For commercial usages, however, a C or C++ implementation might be warranted for further improvement. Yet, there is only so much that can be done on the software level. We note that the machine learning community has been researching in hardware accelerations for neural net computations, and with modern machine learning algorithms utilizing graphical models in increasing rates, a hardware acceleration of belief propagation is not too far fetched, and it will greatly benefit both our architecture and the machine learning community in general.

10.3 Concluding Remarks

With more data and more types of data available, more storage space is demanded. It was estimated in 2013 that the Internet giant Google has tens of exabytes (1024^6 bytes) in disk space [28] including caches of the whole Internet. With an exponential increase in the number of mobile devices and with more and more people in developing countries getting online for the first time, it is estimated that by 2020, 40 zettabytes (1024^7 bytes) of data will be generated, most of which will be new data [12].

This explosion of data generated has not been met with an equivalent growth in storage capacities. With the advancement of disk space technologies slowing down at a similar pace as Moore's Law at the beginning of 2015 [38, 27], the era of essentially unlimited storage space is ending.

This is the signal that now is the time to reconsider the problem of data compression. Instead of focusing on optimally designing different algorithms for different types of data individually, we should consider the problem as a whole, restructuring the landscape of data compression. This information age needs a data compression algorithm that is flexible and ready for change, and this architecture is a very promising candidate.

Bibliography

[1] Teofilo Ancheta. Bounds and Techniques for Linear Source Coding (Ph.D. Thesis Abstr.). *IEEE Transactions on Information Theory*, 24(2):276–276, 1978.

[2] Marc Antonini, Michel Barlaud, Pierre Mathieu, and Ingrid Daubechies. Image Coding Using Wavelet Transform. *IEEE Transactions on Image Processing*, 1(2):205–220, 1992.

[3] Jung Hyun Bae and Achilleas Anastasopoulos. Capacity-Achieving Codes for Finite-State Channels with Maximum-Likelihood Decoding. *IEEE Journal on Selected Areas in Communications*, 27(6):974–984, 2009.

[4] Sonia Bogos, John Gaspoz, and Serge Vaudenay. Cryptanalysis of a Homomorphic Encryption Scheme.

[5] Alfredo Braunstein, Farbod Kayhan, and Riccardo Zecchina. Efficient LDPC Codes Over GF(q) for Lossy Data Compression. In *2009 IEEE International Symposium on Information Theory*, pages 1978–1982. IEEE, 2009.

[6] Giuseppe Caire, Shlomo Shamai, and Sergio Verdú. Universal Data Compression with LDPC Codes. In *Third International Symposium On Turbo Codes and Related Topics*, pages 55–58, 2003.

[7] Giuseppe Caire, Shlomo Shamai, and Sergio Verdú. Noiseless Data Compression With Low-Density Parity-Check Codes. *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, 66:263–284, 2004.

[8] Thomas M. Cover and Joy A. Thomas. *Elements of Information Theory*. John Wiley & Sons, 2012.

[9] Peter Deutsch. DEFLATE Compressed Data Format Specification version 1.3. Technical report, 1996.

[10] Gal Elidan, Ian McGraw, and Daphne Koller. Residual Belief Propagation: Informed Scheduling for Asynchronous Message Passing. *arXiv preprint arXiv:1206.6837*, 2012.

[11] Robert Gallager. Low-Density Parity-Check Codes. *IRE Transactions on Information Theory*, 8(1):21–28, 1962.

[12] John Gantz and David Reinsel. The Digital Universe in 2020: Big Data, Bigger Digital Shadows, and Biggest Growth in the Far East. *IDC iView: IDC Analyze the future*, 2007:1–16, 2012.

[13] Javier Garcia-Frias and Wei Zhong. LDPC Codes for Compression of Multi-Terminal Sources With Hidden Markov Correlation. *IEEE Communications Letters*, 7(3):115–117, 2003.

[14] Craig Gentry. *A Fully Homomorphic Encryption Scheme*. PhD thesis, Stanford University, 2009.

[15] Walter R. Gilks and Pascal Wild. Adaptive Rejection Sampling for Gibbs Sampling. *Applied Statistics*, pages 337–348, 1992.

[16] H. Gish and J. N. Pierce. Asymptotically Efficient Quantizing. *IEEE Trans. on Information Theory*, IT-14(5):676, September 1968.

[17] Geoffrey R. Grimmett. A Theorem about Random Fields. *Bulletin of the London Mathematical Society*, 5(1):81–84, 1973.

[18] Onur G. Guleryuz and Michael T. Orchard. On the DPCM Compression of Gaussian Autoregressive Sequences. *IEEE Transactions on Information Theory*, 47(3):945–956, 2001.

[19] Jon Hamkins and Kenneth Zeger. Gaussian Source Coding with Spherical Codes. *IEEE Transactions on Information Theory*, 48(11):2980–2989, 2002.

[20] Volker Heydegger. Analysing the Impact of File Formats on Data Integrity. In *Archiving Conference*, volume 2008, pages 50–55. Society for Imaging Science and Technology, 2008.

[21] Ying-zong Huang. *Model-Code Separation Architectures for Compression Based on Message-Passing*. PhD thesis, Massachusetts Institute of Technology, 2015.

[22] Ying-zong Huang and Gregory W. Wornell. A Class of Compression Systems with Model-Free Encoding. In *Information Theory and Applications Workshop (ITA), 2014*, pages 1–7. IEEE, 2014.

[23] Mark Johnson, Prakash Ishwar, Vinod Prabhakaran, Daniel Schonberg, and Kannan Ramchandran. On Compressing Encrypted Data. *IEEE Transactions on Signal Processing*, 52(10):2992–3006, 2004.

[24] Santhosh Kumar, Andrew J. Young, Nicolas Maoris, and Henry D. Pfister. A Proof of Threshold Saturation for SPatially-Coupled LDPC Codes on BMS Channels. In *Communication, Control, and Computing (Allerton), 2012 50th Annual Allerton Conference on*, pages 176–184. IEEE, 2012.

[25] Joshua Lee. Personal Communication, 2015-2016.

[26] Stuart Lloyd. Least Squares Quantization in PCM. *IEEE Transactions on Information Theory*, 28(2):129–137, 1982.

[27] Chris Mellor. Kryder's Law Craps Out: Race to Uber-Cheap Storage Is Over, 2015.

[28] Randall Munroe. Google's Datacenters on Punch Cards. https://what-if.xkcd.com/63/. Accessed: 2016-07.

[29] Radford Neal. Software for Low Density Parity Check Codes. http://radfordneal.github.io/LDPC-codes/. Accessed: 2016-02.

[30] Judea Pearl. Reverend Bayes on Inference Engines: A Distributed Hierarchical Approach. In *AAAI*, pages 133–136, 1982.

[31] Diego Santa-Cruz, Touradj Ebrahimi, Joel Askelof, Mathias Larsson, and Charilaos A Christopoulos. JPEG 2000 Still Image Coding versus Other Standards. In *International Symposium on Optical Science and Technology*, pages 446–454. International Society for Optics and Photonics, 2000.

[32] Daniel Schonberg, Stark C. Draper, Chuohao Yeo, and Kannan Ramchandran. Toward Compression of Encrypted Images and Video Sequences. *IEEE Transactions on Information Forensics and Security*, 3(4):749–762, 2008.

[33] Adi Shamir. How to Share a Secret. *Communications of the ACM*, 22(11):612–613, 1979.

[34] Claude E. Shannon. Communication Theory of Secrecy Systems. *Bell System Technical Journal*, 28(4):656–715, 1949.

[35] Claude E. Shannon. A Mathematical Theory of Communication. *ACM SIGMOBILE Mobile Computing and Communications Review*, 5(1):3–55, 2001.

[36] Yung Liang Tong. *The Multivariate Normal Distribution*. Springer Science & Business Media, 2012.

[37] Gregory K. Wallace. The JPEG Still Picture Compression Standard. *IEEE Transactions on Consumer Electronics*, 38(1):xviii–xxxiv, 1992.

[38] Chip Walter. Kryder's Law. *Scientific American*, 293(2):32–33, 2005.

[39] Angel Yu, Wai Lok Lai, and James Payor. Efficient Integer Vector Homomorphic Encryption. 2015.

[40] Wei Zhong and Javier Garcia-Frias. LDGM Codes for Channel Coding and Joint Source-Channel Coding of Correlated Sources. *EURASIP Journal on Applied Signal Processing*, 2005:942–953, 2005.

[41] Hongchao Zhou and Gregory W. Wornell. Efficient Homomorphic Encryption on Integer Vectors and Its Applications. In *Information Theory and Applications Workshop (ITA), 2014*, pages 1–9. IEEE, 2014.