Image Compression using Sum-Product Networks

by

Tejas K. Jayashankar

B.S., University of Illinois at Urbana-Champaign (2019)

Submitted to the Department of Electrical Engineering and Computer Science in Partial Fulfillment of the Requirements for the Degree of

Master of Science in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 2022

Signature of Author: ....
Department of Electrical Engineering and Computer Science
January 26, 2022

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

Accepted by: ....
Leslie A. Kolodziejski
Professor of Electrical Engineering and Computer Science
Chair, Department Committee on Graduate Students

Image Compression using Sum-Product Networks
by
Tejas K. Jayashankar

Submitted to the Department of Electrical Engineering and Computer Science
on January 26, 2022, in Partial Fulfillment of the
Requirements for the degree of
Master of Science in Electrical Engineering and Computer Science

Abstract

An estimated 79 zettabytes (10^{21} bytes) of data was generated worldwide in 2021 with even more data expected to be produced in the future. The effective storage and communication of such large amounts of data is an important problem. Data compression lies at the heart of the solution to this issue.

The two aspect of data compression — data modeling and coding — are typically jointly designed. As a result, it is difficult to evolve compression standards without a complete modification of the entire architecture. Recently, a model-code separation architecture for compression was proposed with a model-free encoder and model-adaptive decoder. The architecture uses a data independent encoder, and it employs a probabilistic graphical model (PGM) to model the source structure in the decoder. Decoding is performed by running belief propagation over the graphical models representing the modeling and coding aspects of compression.

In practical settings where we deal with naturally occurring data, e.g., CIFAR-10 images, the PGM underlying the source data is unknown. Existing structure learning algorithms for PGMs are inefficient for learning from large datasets and place additional constraints on the graphical model structure that diminishes a PGM's representational power. Due to the difficulty of inference and learning in complex PGMs, the current model-code separation architecture is limited in its use for many real world applications.

In this thesis, we develop a new separation architecture based on recently proposed sum-product networks (SPNs), a class of tractable probabilistic generative models, to model the source distribution. Our architecture strikes a balance between efficient learning of source structure and fast lossless decoding. We show that SPNs admit efficient parameter learning via gradient descent to learn statistical structure in synthetic and naturally occurring images. Furthermore, through modifications to the SPN architecture, we describe a procedure to assimilate external beliefs about the source and compute the marginal probabilities of all the source nodes in a single forward and backward pass of the SPN architecture. By using an SPN source model in place of a PGM, we obtain a new model-code separation architecture for compression.

Throughout this thesis, we focus on the efficient implementation of our compression architecture. We take advantage of modern deep learning frameworks and GPUs to implement our entire architecture using parallelized tensor operations. As a result, we are able to bridge the gap between traditional statistical inference algorithms and modern deep learning models by carefully developing the SPN source-code belief propagation algorithm for source decoding. The resulting algorithm can decode grayscale sources in under 0.04 seconds.

This work applies the proposed architecture for the lossless compression of binary and grayscale images. We compare our architecture against some of the most commonly used compression systems of today and theoretical limits.

We show that our architecture achieves a 1.7× gain in compression rate over the state-of-the-art JBIG2 compressor on the binarized MNIST dataset. Furthermore, our architecture does not incur a performance penalty on grayscale sources and is still able to achieve a 1.4× gain in compression rate on the grayscale CIFAR-10 and the Fashion MNIST datasets, as compared against some of the best universal compressors. Extensive analysis on synthetic binary sources show that our architecture can achieve near theoretical limits of compression and match the performance of baseline separation architectures with known PGM structure.

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

Acknowledgments

I would like to my extend my thanks and utmost appreciation to the many people that have made this thesis possible.

First and foremost, I am extremely grateful to my advisor, Professor Gregory Wornell, for his guidance, support and teaching throughout my time here at MIT. From our very first interaction before my acceptance into MIT, I knew that he was invested in my growth as a researcher and a student. Without his support of my research and professional goals, his constant motivation, and his broad technical knowledge, I would not be where I am right now. I am extremely grateful for his guidance and advice that has led to the discovery of new fields of research that I would not have immersed myself in otherwise. Through his classes and weekly meetings, I have added more skills to my toolkit and ventured beyond deep learning into the domains of information theory and statistical inference.

This thesis builds upon the ideas of Dr. Ying-zong Huang, whose 2015 PhD thesis forms the foundation of my work. Dr. Huang has been so kind in sharing his immense knowledge within the field of compression with me and he has taken so much time out of his schedule to guide me in the right direction. This thesis would have been impossible without his insights and suggestions, and for that I am extremely grateful.

My passion for research was fostered by my advisor at UIUC, Professor Pierre Moulin. I am lucky to have known Professor Moulin since 2015. His guidance shaped me into a more inquisitive researcher and I owe my admission into MIT's graduate program to his support and dedication towards my excellence.

Throughout my time at MIT I have always been lucky to have the support of my current and former labmates in the Signals, Information and Algorithms (SIA) laboratory. I would like to thank each of them for fostering a wonderful research environment for me at MIT — Safa Medin, Abhin Shah, Gary Lee, Toros Arikan, Joshua Lee, Mumin Jin, Maohao Shen, Dr. Yuheng Bu, Dr. Amir Weiss, Dr. Ganesh Ajjanagadde and Dr. Adam Yedidia.

Safa is one of my best friends at MIT and I have learned so much about deep generative models for computational imaging through his exemplary work in the area. Through Abhin I have gained a broader understanding about graphical models and inference. I am grateful to Gary who gave me the opportunity to help him mentor four UROPs in Fall 2020. Gary taught me so much about reinforcement learning and its applications to wireless communications. My discussions with Mumin about her research have also been beneficial to me for my growth as a researcher.

I would like to give a special shoutout to Tricia O'Donnell who has been there to help me since my first day at MIT. Even after the pandemic started and work went remote, Tricia has continued to offer a helping hand to me at times of need.

Having had the opportunity to be a teaching assistant for 6.438: Algorithms for Inference in Fall 2021, has truly enriched my graduate experience at MIT. Through my collaboration with other members of course staff — Professor Wornell, Dr. Atulya Yellepeddi, Jiejun Jin, Michael Truell and Romain Cosson — I gained a deeper insight into probabilistic graphical models and statistical inference theory. I am also immensely thankful for the hard work and curiosity of all the students in the course whose questions enabled me to solidify my understanding of the course material and deliver a stronger thesis.

Given the applied nature of this research, I greatly benefited from my prior research-based internships conducted in industry with these additional collaborators: Dr. Jonathan Le Roux and Dr. Chiori Hori at MERL where I developed deep learning models for multimodal audio-visual systems; Dr. Qing He, a former SIA group member, at Facebook AI where I worked on cutting-edge research for the development of variable rate neural audio codecs.

I had the opportunity to hold several leadership positions during my time at MIT. I have been involved with the EECS Graduate Student Association (GSA) as the VP of Publications and more recently as one of the co-Presidents. Throughout my time on the GSA, I have made so many friends in the department and I have had a great time collaborating with the EECS Graduate Office. I would like to convey a special thanks to everyone in the EECS Graduate Office for supporting my initiatives and ideas for the department and moreover, for allowing me to serve as a representative for the many graduate students in the EECS department. I would also like to thank my fellow executive members on the Ashdown House Executive Committee (AHEC) for helping build a safe and comfortable living environment on campus.

When the pandemic started I was welcomed in by the members of the MIT Sport Taekwondo, where I have been a member since. I am grateful to have represented MIT at tournaments, and I am thankful to all my teammates for bringing an extra amount of energy into my day during our workouts.

Graduate life at MIT would not be possible without all my close friends and family. I would like to thank Vivienne Zhang, Mark Saad, Johan Pereira, Pavani Majety, Karthiga Mahalingam, Siddharth Muralidaran, Rushik Desai and Aniket Patel for being there for me every day. They are my family away from home and their constant presence in my life brings a smile to my face. I would also like to thank all my friends at MIT of which there are so many to name.

I would also like to thank my parents and my sister. My parents have worked very hard to support my sister and me throughout our lives. I am very grateful for all the love and encouragement they have given me. This work is dedicated to my grandparents who have loved me from the day I was born. I am very grateful to have so many memories with them and their teachings live in me forever.

to my grandparents
for all their love and wisdom

Image Compression using Sum-Product Networks

Contents

Abstract **ii**

Acknowledgments **iv**

1 Introduction **13**
- 1.1 Compression Landscape . . . 13
- 1.2 Motivation . . . 15
  - 1.2.1 Joint Model-Code Architectures . . . 15
    - 1.2.1.1 Lossless Compression: Huffman Coding . . . 15
    - 1.2.1.2 Lossy Compression: JPEG and JPEG-2000 . . . 15
  - 1.2.2 Universal Data Compression . . . 16
  - 1.2.3 Neural Codecs . . . 16
- 1.3 Thesis Guide . . . 17
  - 1.3.1 Highlight . . . 17
  - 1.3.2 Organization . . . 18
  - 1.3.3 Notation . . . 18

2 Background and Prior Work **20**
- 2.1 Source Coding Theory . . . 20
- 2.2 Probabilistic Graphical Models . . . 22
  - 2.2.1 Undirected Graphical Models . . . 22
  - 2.2.2 Factor Graphs . . . 24
  - 2.2.3 Inference Routines . . . 24
    - 2.2.3.1 Belief Propagation . . . 24
    - 2.2.3.2 Sampling . . . 26
- 2.3 Model-Free Coding . . . 28
  - 2.3.1 Low Density parity-check (LDPC) Codes . . . 28
  - 2.3.2 Decoding LDPC Codes . . . 28
- 2.4 Prior Work . . . 30
- 2.5 Recent Work in Lossless Image Compression . . . 31

3 Sum-Product Networks **33**
- 3.1 Tractable Inference . . . 33
- 3.2 Architecture . . . 34
- 3.3 Comparison with Other Architectures . . . 37

3.3.1 SPNs vs. PGMs 37
3.3.2 SPNs vs. Arithmetic Circuits 39
3.3.3 SPNs vs. Deep Generative Models 39
3.4 Deep Generalized Convolutional Sum-Product Networks 40
3.5 Parameter Learning 41
3.6 Parallel Marginal Computation 42
3.7 Inference with External Beliefs 43

4 Model-Code Separation Architecture **45**
4.1 PGM-based Model-Code Separation Architecture 45
4.1.1 Encoder 45
4.1.2 Decoder 46
4.1.2.1 Code Graph 46
4.1.2.2 Source Graph 46
4.1.3 Source-Code Belief Propagation 46
4.1.4 Dealing with Large Alphabet Sources 48
4.1.5 Doping 49
4.1.6 Overall Architecture 50
4.1.7 Drawbacks of PGM-based Decoder 50
4.2 SPN-based Model-Code Separation Architecture 50
4.2.1 Architecture Details 50
4.2.2 SPN Source-Code Belief Propagation 52
4.2.3 Benefits of Separation Architecture with SPNs 52

5 Lossless Image Compression **55**
5.1 Belief Propagation Decoding Threshold 55
5.2 Experimental Setup 56
5.3 Compressing Binary Sources 57
5.3.1 Binary Ising Model 57
5.3.1.1 Experiments 59
5.3.1.2 Results 60
5.3.2 Binary MNIST 61
5.4 Compressing Grayscale Sources 63
5.4.1 Grayscale MNIST 63
5.4.2 Fashion MNIST 65
5.4.3 CIFAR-10 67
5.5 Summary 69

6 Conclusion **70**
6.1 Review 70
6.2 Applications and Future Work 70
6.3 Final Remarks 72

A Supplementary Images **73**

List of Figures

1-1 A lossless compression system with joint model-code architecture. The data model is used to design the data processor and the coding mechanism. . . . 15
1-2 We take an image, dither each pixel using uniform noise in the interval $[-0.5, 0.5]$ and then threshold the image to two levels using the modulo operator. The system achieves a compression rate of less than 0.125. The reconstructions from a superresolution GAN look perceptually similar to the input. . . . 17

2-1 To convert an undirected graph to a factor graph, create a factor node for each maximal clique. Nodes belonging to the same clique are connecting by edges of the same color. . . . 24
2-2 The Tanner graph representing the LDPC parity-check matrix from Example 2.1 . . . 29
2-3 Model-code separation for compression architectures. The data is coded using a coding mechanism that is independent of the data model. . . . 30

3-1 A normalized sum-product network over two random variables. The leaf nodes have singular scope and can contain any tractable distribution. If the leaf distribution is an indicator, the weights along the edges from the leaf to sum node are parameters for a categorical distribution. If the leaf distributions are, for example, Gaussian then the SPN can be interpreted as mixture model. . . . 35
3-2 An SPN can represent a distribution over all sequences with an even number of 1s with $\mathcal{O}(n)$ edges whereas a junction tree requires a fully connected graph with $\mathcal{O}(n^2)$ edges. . . . 38
3-3 Example DGCSPN architecture in 1D taken from [61]. Layer 0 contains the leaf distributions. Every product layer doubles the dilation rate, starting with a rate of 1. The scopes are indicated by the numbers within each node. All children of the same sum node have the same scope. . . . 40
3-4 The SPN from Example 3.1 with external beliefs (colored in pink) provided by multiplying them with the leaf distribution. . . . 44

4-1 The combined source-code decoder. The source and code graph share the common nodes denoting the source symbols which is represented by the virtual controller. 47
4-2 The combined source-code decoder with a translator introduced to compress large alphabet sources. 49
4-3 Combined SPN source-code decoder with arrows between the controller and SPN denoting the direction of message flow. The messages from the code graph are fed as external beliefs to the SPN. The messages from the source graph are accumulated from the gradients at the leaf nodes of the SPN and are then sent to the controller. 51

5-1 Binary Ising model **Ising**(p, q). 58
5-2 28×28 Gibbs samples images from **Ising**$(1/2, q)$ for different values of q. 59
5-3 Compression performance for the **Ising**(p, q) source family for $p = 1/2$ and different values of q. The dimensions of the image were fixed to $h = w = 28$ and $r_{\text{code}} = 0.04$. 60
5-4 Decoding a sample from **Ising**$(1/2,\ 3/4)$ using `SPN-SEP`. Intensities α_i range from black=0 to white=1. Intensities denote the value of the estimated marginal $\hat{p}_{\mathsf{s}_i}(1) \triangleq \alpha_i$. We set $r_{\text{code}} = 0.5$ and $r_{\text{dope}} = 0.04$. . 61
5-5 Sample images from the grayscale MNIST dataset. Images are binarized by using a mid-intensity threshold. 62
5-6 Decoding the digit 7 from the binary MNIST dataset using `SPN-SEP`. Intensities α_i range from black=0 to white=1. Intensities denote the value of the estimated marginal $\hat{p}_{\mathsf{s}_i}(1) \triangleq \alpha_i$. We set $r_{\text{code}} = 0.27$ and $r_{\text{dope}} = 0.04$ 63
5-7 Decoding the digit 9 from the MNIST dataset using `SPN-SEP`. Intensities α_i range from black=0 to white=1. Intensities denote the value of the max marginal $\alpha_i \triangleq \max_{s_i} \hat{p}_{\mathsf{s}_i}(s_i)$. We set $r_{\text{code}} = 0.23$ and $r_{\text{dope}} = 0.04$ 64
5-8 Samples from the Fashion MNIST dataset. Class labels are displayed below each image. 65
5-9 Decoding an image of a shoe from the Fashion MNIST dataset using `SPN-SEP`. Intensities α_i range from black=0 to white=1. Intensities denote the value of the estimated marginal $\hat{p}_{\mathsf{s}_i}(1) \triangleq \alpha_i$. We set $r_{\text{code}} = 0.45$ and $r_{\text{dope}} = 0.04$ 66
5-10 Samples from the CIFAR-10 dataset. Class labels are displayed below each image. 67
5-11 Decoding an image of a car from the CIFAR-10 dataset using `SPN-SEP`. Intensities α_i range from black=0 to white=1. Intensities denote the value of the estimated marginal $\hat{p}_{\mathsf{s}_i}(1) \triangleq \alpha_i$. We set $r_{\text{code}} = 0.85$ and $r_{\text{dope}} = 0.07$ 68

A-1 Decoding a sample from **Ising**$(1/2,\ 7/10)$ using `SPN-SEP`. Intensities α_i range from black=0 to white=1. Intensities denote the value of the estimated marginal $\hat{p}_{\mathsf{s}_i}(1) \triangleq \alpha_i$. We set $r_{\text{code}} = 0.70$ and $r_{\text{dope}} = 0.04$. 73

A-2 Decoding the same sample from Figure A-1 using PGM-SEP. Intensities α_i range from black=0 to white=1. Intensities denote the value of the estimated marginal $\hat{p}_{\mathsf{s}_i}(1) \triangleq \alpha_i$. We set $r_{\text{code}} = 0.70$ and $r_{\text{dope}} = 0.04$. 74
A-3 Decoding the same sample in Figure 5-4 using PGM-SEP. Intensities α_i range from black=0 to white=1. Intensities denote the value of the estimated marginal $\hat{p}_{\mathsf{s}_i}(1) \triangleq \alpha_i$. We set $r_{\text{code}} = 0.5$ and $r_{\text{dope}} = 0.04$. Compared to 5-4, we see that PGM-SEP arrived at the solution differently. PGM-SEP started decoding by growing outwards from the doped sites whereas SPN-SEP started to look for global structure from the first iteration. This is because BP on a PGM uses local markov structure to decode the source pixels whereas SPN use the complete structure of the source. 75
A-4 Decoding the digit 8 from the binary MNIST dataset using SPN-SEP. Intensities α_i range from black=0 to white=1. Intensities denote the value of the estimated marginal $\hat{p}_{\mathsf{s}_i}(1) \triangleq \alpha_i$. We set $r_{\text{code}} = 0.3$ and $r_{\text{dope}} = 0.04$ 76
A-5 Decoding the digit 5 from the binary MNIST dataset using SPN-SEP. Intensities α_i range from black=0 to white=1. Intensities denote the value of the estimated marginal $\hat{p}_{\mathsf{s}_i}(1) \triangleq \alpha_i$. We set $r_{\text{code}} = 0.27$ and $r_{\text{dope}} = 0.04$ 77
A-6 Decoding the digit 2 from the MNIST dataset using SPN-SEP. Intensities α_i range from black=0 to white=1. Intensities denote the value of the max marginal $\alpha_i \triangleq \max_{s_i} \hat{p}_{\mathsf{s}_i}(s_i)$. We set $r_{\text{code}} = 0.23$ and $r_{\text{dope}} = 0.04$ 78
A-7 Decoding an image of a pant from the Fashion MNIST dataset using SPN-SEP. Intensities α_i range from black=0 to white=1. Intensities denote the value of the estimated marginal $\hat{p}_{\mathsf{s}_i}(1) \triangleq \alpha_i$. We set $r_{\text{code}} = 0.45$ and $r_{\text{dope}} = 0.04$ 79
A-8 Decoding an image of a dress from the Fashion MNIST dataset using SPN-SEP. Intensities α_i range from black=0 to white=1. Intensities denote the value of the estimated marginal $\hat{p}_{\mathsf{s}_i}(1) \triangleq \alpha_i$. We set $r_{\text{code}} = 0.48$ and $r_{\text{dope}} = 0.04$. Notice how the SPN initially drives the decoder towards decoding the image as a pant. 80
A-9 Decoding an image of a horse from the CIFAR-10 dataset using SPN-SEP. Intensities α_i range from black=0 to white=1. Intensities denote the value of the estimated marginal $\hat{p}_{\mathsf{s}_i}(1) \triangleq \alpha_i$. We set $r_{\text{code}} = 0.80$ and $r_{\text{dope}} = 0.04$. 81
A-10 Decoding an image of a boat from the CIFAR-10 dataset using SPN-SEP. Intensities α_i range from black=0 to white=1. Intensities denote the value of the estimated marginal $\hat{p}_{\mathsf{s}_i}(1) \triangleq \alpha_i$. We set $r_{\text{code}} = 0.80$ and $r_{\text{dope}} = 0.04$. 82

List of Tables

1.1 The compression landscape can be divided into four domain based on model specificity and the reconstruction assessment metric. Most traditional systems fall under Domain II and III, but with the advent of deep generative models, Domain I systems have been the focus of current research. 14

5.1 Average compression rate of the binary MNIST images using different compression strategies. 62
5.2 Average compression rate of grayscale MNIST images using different compression strategies. 63
5.3 Average compression rate of Fashion MNIST images using different compression strategies. 65
5.4 Average compression rate of grayscale CIFAR-10 images using different compression strategies. 67

List of Algorithms

1 Belief propagation (BP) on a factor graph. 27

2 SPN Source-Code Belief Propagation. 54

Chapter 1

Introduction

The total amount of digital data that is created and consumed globally is increasing every day. According to Statista, in 2021, 79 zettabytes (10^{21} bytes) of data was generated worldwide [52]. Storing and sharing such large amounts of information is facilitated by efficient data compression algorithms. Data compression has a long history dating as far back as the 1800s, an era which witnessed the introduction of the Morse code and the telegraph, some of the earliest forms of text compression. The field has since made tremendous advances with the introduction of hundreds of compression systems that achieve excellent performance for specific data sources [49, 51]. In recent years, with the advent of machine learning and high-compute devices, the field of data compression is once again at the forefront of engineering. A recent breakthrough is the development of compression systems with model-code separation [27, 26, 34, 36]. In this thesis, we study model-code separation in compression architectures and demonstrate that such a constraint on our system leads to novel deep learning based architectures for modeling source structure.

1.1 Compression Landscape

The compression landscape can be divided into four domains depending on our prior knowledge about the data and the metric for assessing the data reconstruction quality, as shown in the Table 1.1. Along the vertical axis we can divide the systems based on the fidelity of reconstruction.

- If the reconstruction is perfect, i.e., the decoded output is the same as the input, the system performs *lossless* compression. An example of such a compression system is the Portable Networks Graphics (`PNG`) standard for images [3].

- On the other hand, *lossy* compression systems make a trade-off between compression rate and distortion to produce reconstructions that are perceptually "close" to the input. These systems are used for compressing perceptual content such as images or video. For example, the widely used Joint Photographics Experts Group (`JPEG`) image standard [56] leverages the Discrete Cosine Transform (DCT) [1] to perform energy compaction by filtering out high frequencies

Fidelity \Model	**Specified**	**Unspecified**
Lossless	I. Entropy Coding (processing + RLE, Huffman)	II. Universal Entropy Coding (LZW, CTW)
Lossy	III. Rate-Distortion Coding (processing + quantization + coding)	IV. Universal Rate-Distortion Coding

Table 1.1: The compression landscape can be divided into four domain based on model specificity and the reconstruction assessment metric. Most traditional systems fall under Domain II and III, but with the advent of deep generative models, Domain I systems have been the focus of current research.

in the image. The resulting reconstruction looks perceptually similar to the input with additional bit savings achieved by discarding bits from perceptually irrelevant regions.

Along the horizontal axis we can divide the systems based on data model specificity. The *data model* captures all prior knowledge about the data source. This knowledge could be in the form of perceptual information or statistical information about the source.

- If the data model is *specified*, then the system is typically used for compressing specialized data. For example, the Joint Bi-level Image Experts Group (`JBIG`/`JBIG2`) [23, 39] image standard was developed specifically for the compression of bi-level images, particularly for faxes. The system takes advantage of blockwise correlations in textual and halftone data by segmenting the image into blocks. Blocks with similar structures are compressed using fewer bits. Non-image bi-level data will not undergo significant compression using this system as a result of the previous assumptions. In this sense, the `JBIG2` data model is specific to bi-level image data.

- If the data model is *not specified*, then the system is typically used for compressing generic data, i.e., universal compression (see Section 2.1). Since a data model is required for compression, these systems learn statistical structure on-the-fly from the input data. For example, the `GZIP` standard that we use every day to compress our files (usually binary encoded) is a universal compression standard based on the Lempel-Ziv dictionary learning based source coder [59] and Huffman encoding [28]. These models often have to learn significant structure from the data and require long input sequences for optimal performance.

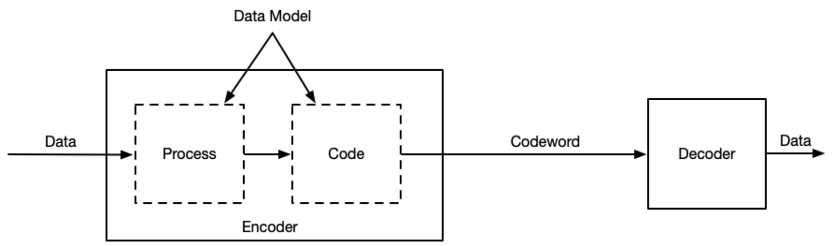


Figure 1-1: A lossless compression system with joint model-code architecture. The data model is used to design the data processor and the coding mechanism.

1.2 Motivation

We motivate our search for model-code separation compression architectures by looking at a few examples.

1.2.1 Joint Model-Code Architectures

Almost every familiar compressor that we use nowadays employs a joint model-code architecture in its system. We will study the drawbacks of such architectures by considering two examples.

1.2.1.1 Lossless Compression: Huffman Coding

Consider the simple yet tremendously powerful Huffman code that falls under Domain I. The Huffman coder learns the data model by building a binary tree that stores the frequency of symbols that appear in the input data. The construction of the binary tree can be interpreted as *processing* the data. As shown in Figure 1-1, the *coding mechanism*, sometimes called the *codec*, also uses the data model, i.e., the frequencies of the symbols, to efficiently compress the data by assigning smaller length binary codes to more frequently occurring symbols and assigning larger length codes to less frequently occurring symbols. In fact, as shown by Huffman in [28], this code achieves *entropy* (see Section 2.1) and is the most optimal code choice for the given data model. Hence, if the data model changes, the previously learned code will no longer be optimal. Moreover, the data model **and** the coding mechanism must be learned for every new instance of input data.

1.2.1.2 Lossy Compression: JPEG and JPEG-2000

An example that illustrates the drawbacks of joint model-code design is the story of `JPEG-2000` [55]. In 2000, the same group that developed `JPEG` introduced the `JEPG-2000` standard that uses the wavelet-transform [2] instead of the DCT. The benefits of using the wavelet-transform include a more scalable compressed bitstream, higher dynamic range reconstruction and progressive image transmission. However, despite these benefits, the older `JPEG` standard from 1987 is still more widely used for daily multimedia transmission and storage. The reason for `JPEG-2000` not replacing its predecessor is two-fold:

1. `JPEG-2000` depended on a completely new codec, and, thus, `JPEG` coded images could not be decompressed using the new standard.
2. `JPEG-2000` requires more memory and most software developers had already standardized their products for use with the `JPEG` standard.

While the second point might no longer be relevant in modern times, as we move into a new age with wearable devices and lightweight sensors, it is important to evolve compression standards with low-complexity codes that are highly optimized for embedded deployment.

Having seen some drawbacks of joint model code design, we seek to answer the following question.

Can we compress an input using a lightweight encoder independent of the data model and leverage the data model only within the decoder?

In [27, 26], Huang and Wornell showed that this is indeed possible. We revisit this question from a slightly different angle in this thesis.

1.2.2 Universal Data Compression

Universal compression is difficult because the system must code the data without any knowledge of the data model. The study of model-code separation is an important step in realizing practical universal compressors. Attempts have been made to ubiquitously adopt universal compression standards, but they usually end up becoming a standard for certain types of data, either due to sub-optimal performance on certain data types or due to compatibility issues with older standards. In this thesis, we propose a data model learning mechanism that can be used to statistically model the structure inherent to any data source type. Moreover, the encoding mechanism is a simple projection operator and only relies on the input data being represented as a bitstream.

1.2.3 Neural Codecs

Neural codecs, compression systems that use neural networks in the compression pipeline, boast state-of-the-art results for lossless compression tasks. Such codecs employ joint model-code design in their architecture and are often trained end-to-end on small datasets for days. Imposing model-code separation in compression systems that use neural data models forces the network to not only employ a deep generative convolutional architecture for rich feature extraction but to also admit tractable inference for probabilistic queries such as marginalization, sampling and most probable explanations (MPE). In this thesis, we show that imposing model-code separation in our system highlights the benefits of a new class of deep probabilistic models called Sum-Product Networks (SPNs) (see Sum-Product Networks).

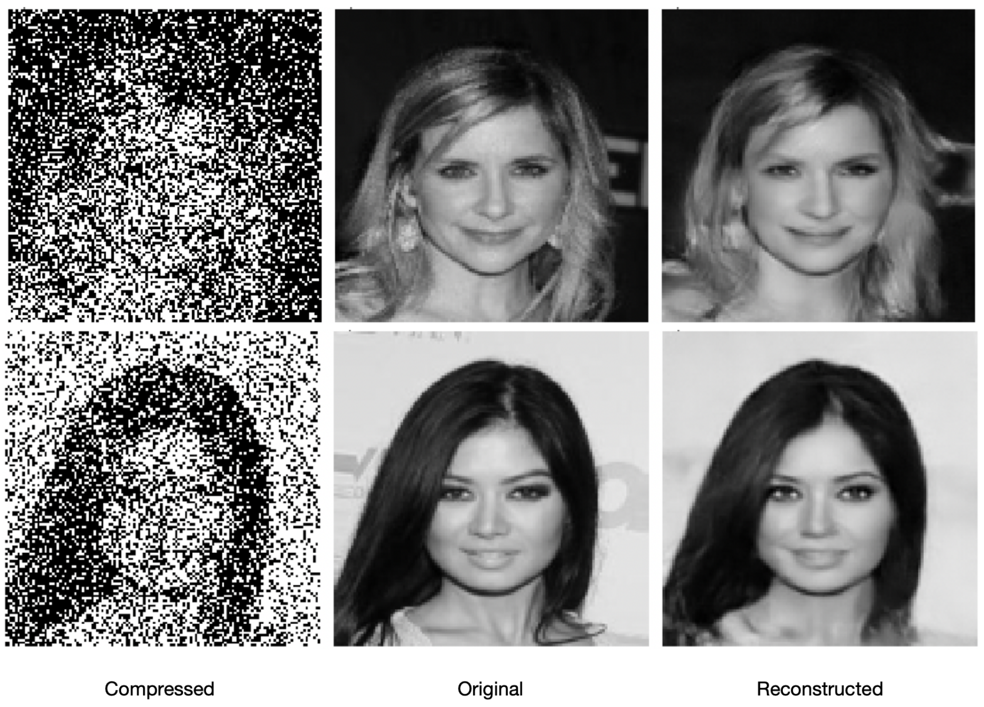


Figure 1-2: We take an image, dither each pixel using uniform noise in the interval $[-0.5, 0.5]$ and then threshold the image to two levels using the modulo operator. The system achieves a compression rate of less than 0.125. The reconstructions from a superresolution GAN look perceptually similar to the input.

1.3 Thesis Guide

1.3.1 Highlight

The inspiration for studying model-code separation stems from experiments on image reconstruction from *dither-quantized* images using generative adversarial networks (GANs) [22] and vector-quantized variational autoencoders (VQVAEs) [41]. We code an image from the CelebA dataset using the modulo operator and then use a deep generative network called a superresolution GAN (SRGAN) [58] to reconstruct the image. Let $\mathbf{s}^n$ be an image with $n = h \times w$ pixels over an alphabet of size M, and let $\mathbf{u}^n \sim \text{Uniform}([-0.5, 0.5]^n)$ be noise sampled from the uniform distribution. We compress $\mathbf{s}^n$ into a binary image $\mathbf{c}^n$ (alphabet size of 2) of the same spatial size,

$$\mathbf{c}^n = (\mathbf{s}^n + \mathbf{u}^n) \,\mathsf{modulo}\, 2, \tag{1.1}$$

where the process of adding uniform noise before quantization is called *dithering*. There were two key takeaways from this experiment — 1) the SRGAN was able to produce a perceptually similar reconstruction of the input, as shown in Figure 1-2, and 2) the coding mechanism was independent of the data model. This is an instance of a lossy compression system with the SRGAN discriminator playing the role of the distortion function. Ideally, to use this as a true compression system, we would like to trade-off between distortion and rate to compress an image. If we are equipped with a probabilistic interpretation for each layer of the discriminator, we can answer any number of probabilistic queries that can help quantify the level of distortion and rate of compression.

It is for these reasons that we revisit the model-code separation architecture for data compression proposed by Huang and Wornell in [27, 26]. We approach it with the goal of amalgamating statistical inference techniques such as *belief propagation (BP)* (see Section 2.2.3.1) with modern deep learning architectures. The end result is an architecture that is fast, flexible and highly parallelizable for deployment on powerful inference engines such as GPUs. Furthermore, the architecture uses a data model based on recent techniques in deep probabilistic modeling to learn powerful statistical structure in source sequences.

The experiments in this thesis focus on the lossless compression of images. The architecture is, however, general and can applied for the compression of any source modality. Furthermore, while our efforts are focused on Domain I in Table 1.1, the architecture can be extended for lossy compression of images similar to [34, 36].

1.3.2 Organization

This thesis is organized into six chapters. The first three chapters introduce the necessary background and tools that are needed to understand our compression architecture.

Important results from source coding theory, probabilistic graphical models and a brief survey of related work are presented in Chapter 2 (Background and Prior Work). We will build upon these concepts by introducing a recently proposed class of deep probabilistic generative models and describe the modifications to use them for lossless compression in Chapter 3 (Sum-Product Networks).

Chapter 4 (Model-Code Separation Architecture) describes the model-code separation compression architecture by using the concepts introduced in the previous chapters. Chapter 5 (Lossless Image Compression) presents experiments and results for lossless compression of images from various image datasets.

Finally, Chapter 6 (Conclusion) summarizes the work and elaborates on further ideas for research.

Appendix A (Supplementary Images) contains supplementary images that complement the results in Chapter 5 (Lossless Image Compression).

1.3.3 Notation

We write s to denote a random variable and s to denote a scalar variable. To denote a length n random vector or a sequence of n random variables we use the notation s^n. We denote the i'th element of s^n as s_i. Depending on the situation, we will write s^n as a column vector $[\mathsf{s}_1\ \mathsf{s}_2\ \ldots\ \mathsf{s}_n]^T$ or as a sequence $(\mathsf{s}_1, \mathsf{s}_2, \ldots, \mathsf{s}_n)$. We denote by s^n and s_i the non-random version of the same.

We reserve e.g., H for a matrix and e.g., $\mathcal{S}$ for a set. We use H^j to denote the vector representing the j'th row of H. We use $\mathbf{1}$ to denote the vector of all ones.

We sometimes use the notation $\mathbf{x}_j^r$ to denote j'th length r vector in the block vector $\mathbf{x}^{kr} = [\mathbf{x}_1^r \; \mathbf{x}_2^r \; \ldots \; \mathbf{x}_k^r]^T$.

We will use graphical models to represent probability distributions when necessary. We denote an undirected graph by $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ where $\mathcal{V}$ is the vertex/node set and $\mathcal{E}$ is the edge set. If a graph $\mathcal{G}$ represents the distribution p_{s^n}, we set $\mathcal{V} = \{1, \ldots, n\}$ such that node $v \in \mathcal{V}$ represents s_v. We use the shorthand $\mathsf{s}_\mathcal{V}$ to denote s^n and $\mathsf{s}_\mathcal{V}$ to denote $\mathbf{s}^n$.

We denote indexing of a random vector s^n with the index set $\mathcal{A}$ by $\mathsf{s}^n_\mathcal{A}$. In the graphical model setting we use the shorthand $\mathsf{s}_\mathcal{A}$. We use similar notation in the non-random setting.

Chapter 2

Background and Prior Work

The model-code separation architecture for compression can be decomposed into two sub-components — 1) the model-free code, and 2) the model-adaptive decoder. The fundamental concepts behind the *model-free code* come from the field of information theory, coding theory and communications. The development of the *model-adaptive decoder* draws inspiration from statistical inference and deep learning. In this chapter we introduce the key concepts required to understand the model-code separation architecture for compression. We end the chapter with details of prior work in this field and a brief survey of related works in the area of lossless image compression.

2.1 Source Coding Theory

In his seminal paper from 1948 [53], Claude Shannon showed that communications and data compression can be viewed as statistical information theoretic problems. He introduced two source-coding theories for lossless and lossy compression of data. We focus on lossless compression in this thesis and start by stating the source coding theorem.

Definition 2.1 (Entropy)**.** Given a random variable $\mathsf{s} \sim p_{\mathsf{s}}$ belonging to a finite alphabet $\mathcal{S}$, its *entropy* is defined as

$$H_p(\mathsf{s}) \triangleq -\sum_{s\in\mathcal{S}} p_{\mathsf{s}}(s)\log p_{\mathsf{s}}(s). \tag{2.1}$$

Theorem 2.1 (Source Coding Theorem)**.** *Given a random variable $\mathsf{s} \sim p_{\mathsf{s}}$ belonging to a finite alphabet $\mathcal{S}$, for all uniquely decodable coding functions $c : \mathcal{S} \to \mathcal{C}$, where $\mathcal{C}$ is the code alphabet, it holds that $\mathbb{E}[c(\mathsf{s})] \geq H_p(\mathsf{s})$.*

In other words, Shannon's source coding theorem for lossless compression states that a random variable s cannot be compressed into fewer than $H(\mathsf{s})$ bits without information loss.

Definition 2.2 (Entropy Rate)**.** Given a sequence of random variables represented as a random vector $\mathsf{s}^n = [\mathsf{s}_1, \mathsf{s}_2, \ldots, \mathsf{s}_n]^T$ with each s_i belonging to a finite alphabet $\mathcal{S}$, the *entropy rate* of the sequence is defined as

$$\mathbb{H}_p(\mathsf{s}) \triangleq \lim_{n\to\infty} \frac{1}{n} H_p(\mathsf{s}^n). \tag{2.2}$$

As described in Section 1.2.1.1, the Huffman code is an example of a code which achieves entropy since it is designed specifically for the true source distribution $p_{\mathsf{s}^n}(\mathsf{s}^n)$. Now assume that the coder was designed for some incorrect distribution $q_{\mathsf{s}^n}(\mathsf{s}^n)$. Clearly the Huffman coder is no longer optimal for source sequences $\mathsf{s}^n \sim p_{\mathsf{s}^n}$. Given no knowledge about $p_{\mathsf{s}^n}(\mathsf{s}^n)$, is it possible to still design a *universal code* with rate r_u such that every i.i.d. source with entropy $H_p(\mathsf{s}^n) < r_u$ can be described? The answer is yes, as we start to develop below. We will confine ourselves to a particular class of universal codes for ease of exposition.

Definition 2.3 (Weakly Typical Sequences)**.** Let $\mathsf{s}^n = (\mathsf{s}_1, \ldots, \mathsf{s}_n)$ be a sequence drawn from p_{s^n} over a finite alphabet $\mathcal{S}^n$. The typical set $\mathcal{A}_\epsilon \subset \mathcal{S}^n$ contains those sequences that satisfy,

$$2^{-n(H_p(\mathsf{s}^n)+\epsilon)} \leq p_{\mathsf{s}^n}(\mathsf{s}^n) \leq 2^{-n(H_p(\mathsf{s}^n)-\epsilon)}. \tag{2.3}$$

The above definition states that a typical sequence can be encoded in approximately $H(\mathsf{s}^n)$ number of bits. Thus, the typical set comprises of those sequences that are most representative of the source distribution. The size of the typical set is approximately $|\mathcal{A}_\epsilon| \approx 2^{nH_p(\mathsf{s}^n)}$.

If the code has a rate of $H_q(\mathsf{s}^n) = r_u$, then the typical set of sequences represented by this code has a size of approximately 2^{nr_u}. We now define the probability of error for *fixed rate block codes* — codes that use the same rate for all length n blocks in the input data stream. The rest of the definitions in this section are adapted from [9].

Definition 2.4 (Probability of Error for Fixed Rate Block Codes)**.** A *fixed rate block code* $B(2^{nr_u}, n)$ of rate r_u for a source s^n which has an unknown distribution p_{s^n} consists of two mappings, the encoder

$$f_n : \mathcal{S}^n \to \{1, 2, \ldots, 2^{nr_u}\}, \tag{2.4}$$

and the decoder,

$$\phi_n : \{1, 2, \ldots, 2^{nr_u}\} \to \mathcal{S}^n. \tag{2.5}$$

The probability of error of the code with respect to p_{s^n} is

$$P_\epsilon^{(n)} = \mathbb{P}\{\phi_n(f_n(\mathsf{s}^n)) \neq \mathsf{s}^n\}. \tag{2.6}$$

We are now ready to formally define a *universal* fixed rate block code.

Definition 2.5. A rate r_u block code for a source is called *universal* if the functions f_n and ϕ_n do not depend on the distribution p_{s^n} and if $P_\epsilon^{(n)} \to 0$ as $n \to \infty$ if $H_p(\mathsf{s}^n) < r_u$.

We now describe a universal code that exploits the fact that the size of the typical set is exponential in n.

Theorem 2.2 (Csiszár and Körner [8, 9]). *There exists a sequence of universal codes $B(2^{nr_u}, n)$ such that $P_\epsilon^{(n)} \to 0$ for every source $p_{\mathbf{s}^n}$ such that $H_p(\mathbf{s}^n) < r_u$.*

The encoder is very simple and it is defined by

$$f_n(\mathbf{s}^n) = \begin{cases} \text{index of } \mathbf{s}^n \text{ in } \mathcal{D} & \text{if } \mathbf{s}^n \in \mathcal{D} \\ 0 & \text{else} \end{cases} \tag{2.7}$$

where

$$\mathcal{D} = \{\mathbf{s}^n : \mathbf{s}^n \sim p_{\mathbf{s}^n}(\mathbf{s}^n) \text{ with } H_p(\mathbf{s}^n) \leq r_u\}.$$

Note that every element in $\mathcal{D}$ is decoded perfectly while elements with entropy larger than r_u are not. Thus, we have defined a universal coding scheme with fixed rate codes. In practice variable rate universal codes are used for compressing generic data. The Lempel-Ziv algorithm which is used in `GZIP` compressors is an example of such a scheme (see Section 1.1).

2.2 Probabilistic Graphical Models

Probabilistic Graphical Models (PGMs) are graphical models that can be used to compactly represent statistical structure and conditional (in)dependencies in data. In addition to modeling data, PGMs are capable of performing various inference tasks such as computing marginal distributions, sampling and computing moments. PGMs have been used to solve problems across various domains of engineering and science. For example, many traditional and modern speech recognition systems use *Hidden Markov Models (HMMs)* to model parts of speech [17, 57]. The HMM can also be used to model state evolution in linear dynamical systems and have powered space exploration through the *Kalman filter* [60]. In the early 2000s, factor graphs demonstrated powerful statistical modeling capabilities for error correcting codes and are still used in communication systems nowadays. In more recent years, PGMs have played an important role in modeling treatment effect in clinical machine learning models, where some form of intervention from domain experts is vital.

2.2.1 Undirected Graphical Models

An *undirected graphical model* is an undirected graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ where each $v \in \mathcal{V}$ represents a random variable and the edges represent conditional dependencies. If a *family of probability distributions* $p_{\mathbf{s}^n}$ can be represented as an undirected graph, where $\mathbf{s}^n$ is a random vector, then a node v in the graph represents one component scalar random variable s_v. Furthermore, for any two nodes $u, v \in \mathcal{V}$

$$(u, v) \notin \mathcal{E} \Rightarrow \mathsf{s}_u \perp\!\!\!\perp \mathsf{s}_v \,|\, \mathsf{s}_{\mathcal{V}\setminus\{u,v\}}, \tag{2.8}$$

where $\perp\!\!\!\perp$ denotes independence and $|$ denotes conditioning.

Remark: Note that we use the term *family of distributions* when describing an undirected graphical model. There could exist many distributions that have the same factorization structure and hence the same undirected graphical representation, but with different mass/density functions. There exists a stronger graph separation property in undirected graphical models which we state below.

Property 2.1 (Graph Separation Property). *Consider mutually disjoint subsets $\mathcal{A}, \mathcal{B}$ and $\mathcal{C}$ of $\mathcal{V}$. Then the conditional independence relation $\mathsf{s}_{\mathcal{A}} \perp\!\!\!\perp \mathsf{s}_{\mathcal{B}} \,|\, \mathsf{s}_{\mathcal{C}}$ holds for all distributions in the family of distributions represented by the undirected graph whenever there is no path from a node in $\mathcal{A}$ to a node in $\mathcal{B}$ that does not pass through a node in $\mathcal{C}$.*

We have thus far defined undirected graphs in terms of their conditional independencies via the graph separation property. We will now look at an alternate definition of an undirected graphical model by defining functions over the maximal cliques (fully connected subgraphs) of the graph. The definition is provided by the *Hammersley-Clifford theorem* [7].

Theorem 2.3 (Hammersley-Clifford Theorem). *Any strictly positive distribution p_{s^n} over $\mathcal{S}^n$ can be represented by an undirected $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ that satisfies the factorization over maximal cliques of the graph,* $\mathrm{cl}^*(\mathcal{G})$,

$$p_{\mathsf{s}^n}(s^n) = \frac{1}{Z} \prod_{\mathcal{C} \in \mathrm{cl}^*(\mathcal{G})} \phi_{\mathcal{C}}(s^n_{\mathcal{C}}), \tag{2.9}$$

if and only if it satisfies the conditional independencies implied by the graph separation property (Property 2.1). The functions defined over the maximal cliques are called the clique potentials and are positive everywhere.

Notation: When a distribution is represented by an undirected graph, we will use the equivalent notation $\mathsf{s}_{\mathcal{V}}$ to denote the random variable s^n. We denote the super-variable over a subset of nodes as $\mathsf{s}_{\mathcal{A}}$ for some $\mathcal{A} \subset \mathcal{V}$. Equation 4.4 can be equivalently expressed as

$$p_{\mathsf{s}_{\mathcal{V}}}(\mathrm{s}_{\mathcal{V}}) = \frac{1}{Z} \prod_{\mathcal{C} \in \mathrm{cl}^*(\mathcal{G})} \phi_{\mathcal{C}}(\mathrm{s}_{\mathcal{C}}). \tag{2.10}$$

Z is called the partition function. For large graphs the partition function is hard to compute as it requires a sum (or integral if variables are continuous) over all possible combinations of the inputs. Like undirected graphs, PGMs can be constructed over directed graphs in which conditional dependencies are conveyed by the edge direction. In directed graphs, the edge potentials represent conditional probability distributions. Note that the clique potentials in undirected graphs are not necessarily distributions.

2.2.2 Factor Graphs

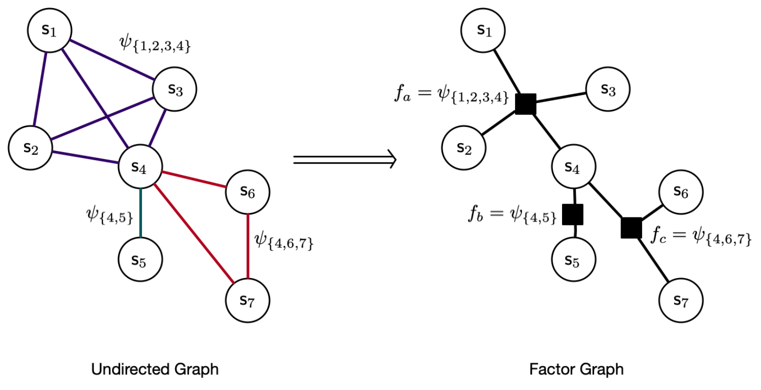


Figure 2-1: To convert an undirected graph to a factor graph, create a factor node for each maximal clique. Nodes belonging to the same clique are connecting by edges of the same color.

Another class of PGMs that have found extensive use in communications and coding theory are *factor graphs* [33]. A factor graph is a *bipartite graph* consisting of two types of nodes — variable nodes ($\mathcal{V}$) and factor nodes ($\mathcal{F}$). Variable nodes represent random variables and factor nodes represent functions over the random variables. The probability distribution represented by a factor graph $\mathcal{G} = (\mathcal{V}, \mathcal{F}, \mathcal{E})$ is given by

$$p_{\mathbf{s}_{\mathcal{V}}}(\mathbf{s}_{\mathcal{V}}) = \frac{1}{Z} \prod_{a \in \mathcal{F}} f_a(\mathbf{s}_{\mathcal{N}(a)}). \tag{2.11}$$

Note that any factor graph can be converted to an undirected graph and vice versa by creating $|\mathcal{F}|$ cliques such that $\phi_a \triangleq f_a$, with the clique size given by $|\psi_a| = \deg(a)$ and the clique itself defined by the set $\mathcal{N}(a)$. An example of this conversion is shown in Figure 2-1.

Factor graphs are often used in communication systems to model a distribution over the set of valid codewords identifiable by the system. The factors typically represent constraints between the symbols of the codeword. The Low Density parity-check (LDPC) code (see Section 2.3.1) is an example of such a code.

2.2.3 Inference Routines

Marginal distribution computation and sampling are two of the most important inference tasks on PGMs. If a model can perform one of these tasks efficiently, it can perform the other efficiently too, since the tasks of marginalization, sampling, computing the partition function Z and computing moments are all equivalent.

2.2.3.1 Belief Propagation

Approximate marginals of a distribution $p_{\mathbf{s}^n}$ can be computed using the sum-product or belief propagation algorithm (BP) [64]. The BP algorithm is iterative and converges to the (estimated) marginals with complexity exponential in the *treewidth* of the graph. We first describe the algorithm for undirected graphical models.

Let $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ be an undirected graphical model for $p_{\mathbf{s}^n}$ that factorizes as (2.10). Using naïve summation, with $\mathcal{O}(|\mathcal{S}|^n)$ operations, the marginal of a node can be computed as

$$p_{\mathsf{s}_i}(s_i) = \sum_{\mathbf{s}_{\mathcal{V}\setminus\{i\}}} \frac{1}{Z} \prod_{\mathcal{C}\in \mathrm{cl}^*(\mathcal{G})} \phi_{\mathcal{C}}(s_{\mathcal{C}}).$$

BP utilizes the factorization structure of the graph to compute marginals for all nodes by passing *messages* along the edges of the graph. They key idea is to reuse messages in the graph to compute marginals for all nodes in fewer operations than the naïve method. The messages can be interpreted as local beliefs about the marginal of a node. The two node computations of BP on an undirected graph are:

- The *message* from a node i to j. For every edge $(i, j) \in \mathcal{E}$ compute the messages

$$m_{i\to j}(s_j) \propto \sum_{\mathbf{s}_{\mathcal{D}\setminus\{j\}}} \phi_{\mathcal{D}}(\mathbf{s}_{\mathcal{D}}) \prod_{k\in\mathcal{N}(i)\setminus\{j\}} m_{k\to i}(s_i), \tag{2.12}$$

  where

$$\mathcal{D} = \bigcup_{\mathcal{C}\in \mathrm{cl}^*(\mathcal{G})} \mathcal{C} \;\text{ s.t. }\; (i, j) \in \mathcal{C}.$$

- The *total belief computation*. For each node $i \in \mathcal{V}$ compute the (approximate) unnormalized marginals/beliefs by accumulating messages from all neighboring nodes:

$$b_i(s_i) \propto \prod_{k\in\mathcal{N}(i)} m_{k\to i}(s_i). \tag{2.13}$$

BP can also be done on factor graphs and is often more easily described since the local factorization structure is clearly represented as factors. Since factor graphs consist of two types of nodes, we must now compute two types of messages. Since we can convert any undirected graph to a factor graph, the message computation described below is equivalent to what was already presented in (2.12) and (2.13). The three important node computations of BP on a factor graph are:

- The *variable to factor message* from a variable node i to factor node a. For every variable-factor edge $(i, a) \in \mathcal{E}$ compute the messages

$$m_{i\to a}(s_i) \propto \prod_{b\in\mathcal{N}(i)} m_{b\to i}(s_i), \tag{2.14}$$

- The *factor to variable message* from a factor node a to variable node i. For every factor-variable edge $(a, i) \in \mathcal{E}$ compute the messages

$$m_{a\to i}(s_i) \propto \sum_{\mathbf{s}_{\mathcal{N}\setminus\{i\}}} f_a(\mathbf{s}_{\mathcal{N}(a)}) \prod_{k\in\mathcal{N}(a)\setminus\{i\}} m_{k\to a}(s_k), \tag{2.15}$$

- The *total belief computation.* For each node $i \in \mathcal{V}$ compute the (approximate) unnormalized marginals/beliefs by accumulating messages from all neighboring factor nodes:
$$b_i(s_i) \propto \prod_{a \in \mathcal{N}(i)} m_{a \to i}(s_i). \tag{2.16}$$

We begin BP by initializing all messages proportional to a uniform distribution. The messages can also be initialized randomly. If the algorithm is carried out iteratively, the messages are computed for each edge according to a pre-selected order. In each iteration of the algorithm all messages are newly computed from the previous iteration's messages until convergence. To ensure stability of the algorithm the messages are normalized after each iteration.

The marginal estimates can be obtained from the beliefs as

$$\hat{p}_{\mathsf{s}_i}(s_i) = \frac{b_i(s_i)}{\|b_i\|_1}. \tag{2.17}$$

The complete BP algorithm for factor graphs is shown in Algorithm 1. The complexity of BP is $\mathcal{O}(|\mathcal{E}|T|S|^{\max_{\mathcal{C} \in \mathrm{cl}^* \mathcal{G}} |\mathcal{C}|})$ for an undirected graph or equivalently $\mathcal{O}(|\mathcal{E}|T|S|^{\max_{a \in \mathcal{F}} \deg(a)})$ for a factor graph, where T is the number of iterations.

Fact 2.1. If the graph has no loops, BP converges to the true marginals of the distribution. If the graph has loops, BP is not guaranteed to converge, but theory shows that the approximate beliefs are often close to the true marginals. In the presence of loops the BP algorithm is an *approximate inference* procedure and it is called *loopy BP.*

Fact 2.2. The *treewidth* of a graph is defined as one less than the size of the maximal clique,
$$\mathrm{tw}(\mathcal{G}) \triangleq \max_{\mathcal{C} \in \mathrm{cl}^* \mathcal{G}} |\mathcal{C}| - 1. \tag{2.18}$$

Hence the complexity of BP for an undirected graph can be equivalently stated as $\mathcal{O}(|\mathcal{E}|T|S|^{\mathrm{tw}(\mathcal{G})+1})$.

Notation: We denote by $\mathbf{m}_{i \to j}$ the vectorized version of the messages from node i to node j,

$$\mathbf{m}_{i \to j} = \begin{bmatrix} m_{i \to j}(0) & \dots & m_{i \to j}(|\mathcal{S}| - 1) \end{bmatrix}^T. \tag{2.19}$$

This is an abuse of notation since we drop the length of the vector in the subscript to keep the notation a bit more clean.

2.2.3.2 Sampling

Sampling is one of the most important routines in approximate inference. Given a large number of samples from a distribution, the empirical distribution of the samples

Algorithm 1: Belief propagation (BP) on a factor graph.

```
Data: Graph $\mathcal{G} = (\mathcal{V}, \mathcal{F}, \mathcal{E})$ and factor potentials $f_a$ for all $a \in \mathcal{F}$.
Result: Estimated marginals probabilities $\hat{p}_{\mathsf{s}_i}$ for all nodes $i \in \mathcal{V}$

/* Initialize messages to the uniform distribution over $|\mathcal{S}|$.       */
$\mathbf{m}_{i\to j} \leftarrow \frac{1}{|\mathcal{S}|}\mathbf{1}$;
/* Started iterative BP.                                                        */
$T \leftarrow 0$;
repeat
    $T \leftarrow T + 1$;
    for all variable-factor edges $(i, a) \in \mathcal{E}$ do
        $m^{(T)}_{i\to a}(s_i) \leftarrow \prod_{b\in\mathcal{N}(i)} m^{(T-1)}_{b\to i}(s_i)$;
    end
    for all factor-variable edges $(a, i) \in \mathcal{E}$ do
        $m^{(T)}_{a\to i}(s_i) \leftarrow \sum_{\mathbf{s}_{\mathcal{N}\setminus\{i\}}} f_a(\mathbf{s}_{\mathcal{N}(a)}) \prod_{k\in\mathcal{N}(a)\setminus\{i\}} m^{(T-1)}_{k\to a}(s_k)$;
    end
until convergence
/* Accumulate messages at each node to estimate marginals.                     */
for all variables $i \in \mathcal{V}$ do
    $b_i(s_i) \leftarrow \prod_{a\in\mathcal{N}(i)} m^{(T)}_{a\to i}(s_i)$;
    $\hat{p}_{\mathsf{s}_i}(s_i) \leftarrow \frac{b_i(s_i)}{\|b_i\|_1}$;
end
return $\hat{p}_{\mathsf{s}_i}$ for all nodes $i \in \mathcal{V}$.
```

converges to the true distribution, which we can use to approximate marginals. The *law of large numbers (LLN)* states that the expected value of random variable over a large number of trials approaches to the true mean of the distribution. Hence, we can compute moments of distributions if we can sample efficiently.

In this thesis we will use the celebrated *Metropolis-Hastings algorithm* [24] to sample from distributions with undirected graphical model representations. We will use a specialized version of this algorithm called the *Gibbs sampler* [19] to generate samples from PGMs. The Gibbs sampler works by first sampling a node i using its marginal distribution and then sampling a node j from the conditional distribution $p_{\mathsf{s}_j|\mathsf{s}_i}$ using the previously sampled value of s_i. With i and j sampled, a new node k is sampled from $p_{\mathsf{s}_j|\mathsf{s}_{\{i,j\}}}$. This process is carried on sequentially until all nodes have been sampled. The conditional distributions can be computed efficiently using the belief propagation algorithm.

2.3 Model-Free Coding

We would like to choose a code that is model-free and also simple to implement. We would also like a code that can not only be decoded via simple transformations but also via statistical inference algorithms. A suitable choice is a particular family of *linear codes* which we describe next.

2.3.1 Low Density parity-check (LDPC) Codes

Low Density parity-check (LDPC) codes are a family of linear codes developed by Gallager [18] in 1963 for use in channel coding. A code is linear if it is a subspace of some vector space $\mathbb{F}^n$, where $\mathbb{F}$ is a finite-field [47]. Let the source sequence belong to a $(n-k)$ dimensional subspace of the vector space $\mathcal{S}^n$ of dimension n, where $\mathcal{S}$ is a finite-field of size $|\mathcal{S}|$. The linear code $\mathcal{L} \subset \mathcal{S}^n$ maps the source sequence to a codeword of length n via multiplication with a generator matrix $G \in \mathcal{S}^{(n-k)\times n} = [I_{n-k} \,|\, P]$,

$$(\mathbf{s}^{(n-k)})^T G^T = (\mathbf{c}^n)^T = [(\mathbf{s}^{(n-k)})^T \quad (\mathbf{s}^{(n-k)})^T P]. \tag{2.20}$$

The k additional codeword symbols are called the parity-check symbols and are used to verify that the source sequence is correctly decoded. The linear code can be equivalently described using a parity-check matrix $H \in \mathcal{S}^{k\times n}$ that describes the computation of the parity information, where $HG^T = 0$:

$$\mathcal{L} = \{\mathbf{c}^n \,|\, (\mathbf{c}^n)^T = (\mathbf{s}^{(n-k)})^T G^T\} = \{\mathbf{c}^n \,|\, H\mathbf{c}^n = 0\}. \tag{2.21}$$

LDPC codes are sparse linear codes with minimal column and row weights in the parity-check matrix. In this thesis, we define all codes over the binary alphabet and use the parity-check matrix for source coding. H projects the source sequence $\mathbf{s}^n$ to a codeword $\mathbf{c}^k$ of length k via the projection

$$\mathbf{c}^k = H\mathbf{s}^n. \tag{2.22}$$

Remark: Note that multiplication and addition in the matrix operations are also defined over the finite-field $\mathcal{S}$. Hence, in the case of binary symbols, addition is equivalent to integer addition modulo 2.

2.3.2 Decoding LDPC Codes

Assume that each symbol in the source is binary. Given an LDPC code represented by the parity-check matrix $\mathcal{L} = C(H), H \in \{0,1\}^{k\times n}$, we can use *maximum a posteriori (MAP)* decoding to solve (2.22) for the source sequence $\mathbf{s}^n$. Given external beliefs

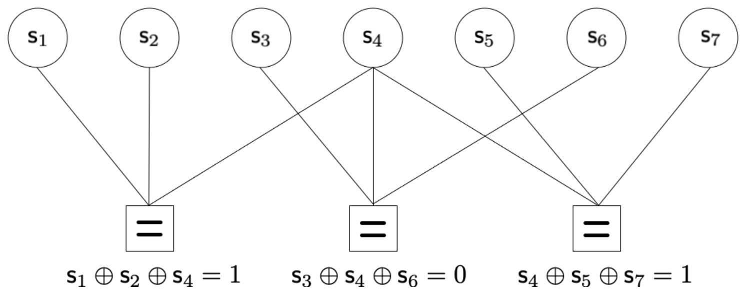

s_1
s_2
s_3
s_4
s_5
s_6
s_7
$=$
$=$
$=$
$s_1 \oplus s_2 \oplus s_4 = 1$
$s_3 \oplus s_4 \oplus s_6 = 0$
$s_4 \oplus s_5 \oplus s_7 = 1$

Figure 2-2: The Tanner graph representing the LDPC parity-check matrix from Example 2.1

$p_{\mathsf{y}_i \mid \mathsf{s}_i}$ about the observed value y_i of a symbol s_i, the MAP estimate of the symbol is

$$\hat{s}_i^{\text{MAP}} = \arg\max_{s_i \in \{0,1\}} p_{\mathsf{s}_i \mid \mathbf{y}^n}(s_i \mid \mathbf{y}^n) \tag{2.23a}$$

$$= \arg\max_{s_i \in \{0,1\}} \sum_{s_j,\, j \neq i} p_{\mathbf{s}^n \mid \mathbf{y}^n}(\mathbf{s}^n \mid \mathbf{y}^n)$$

$$= \arg\max_{s_i \in \{0,1\}} \sum_{s_j,\, j \neq i} p_{\mathbf{y}^n \mid \mathbf{s}^n}(\mathbf{y}^n \mid \mathbf{s}^n) p_{\mathbf{s}^n}(\mathbf{s}^n)$$

$$= \arg\max_{s_i \in \{0,1\}} \sum_{s_j,\, j \neq i} \prod_{i=1}^{n} p_{\mathsf{y}_i \mid \mathsf{s}_i}(y_i \mid s_i) \mathbb{1}_{\mathbf{c}^k = H\mathbf{s}^n}. \tag{2.23b}$$

Let's look at bit closer at the MAP decoder. We desire the maximum value of the marginal distribution of symbol i in (2.23a) and we show that we can rewrite the conditional distribution as marginalization over the joint distribution that factorizes according to (2.23b).

This resembles the factorization structure inherent to factor graphs. Thus, we can model $p_{\mathbf{y}^n, \mathbf{s}^n}(\mathbf{y}^n, \mathbf{s}^n)$ as factor graph $\mathcal{G} = (\mathcal{V}, \mathcal{F}, \mathcal{E})$ with one factor for each external belief and one factor for each row constraint of the LDPC parity-check matrix:

$$\mathcal{F} = \{f_i(s_i) \text{ for all } i \in \mathcal{V}\} \bigcup \{g_j(\mathbf{s}^n) \text{ for } j = 1, 2, \ldots, k\}, \tag{2.24}$$

where

$$f_i(s_i) \triangleq p_{\mathsf{y}_i \mid \mathsf{s}_i}(y_i \mid s_i), \tag{2.25}$$

$$g_j(\mathbf{s}^n) \triangleq \mathbb{1}_{\{c_j = H^j \mathbf{s}^n\}}. \tag{2.26}$$

The factor graph representation of an LDPC parity-check matrix is called a *Tanner graph*. With the Tanner graph representation, we can now easily compute the marginals for all nodes by running BP over the factor graph.

Example 2.1. Consider the parity-check matrix

$$H = \begin{pmatrix} 1 & 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 & 1 \end{pmatrix}.$$

Assume that the input source sequence is $\mathbf{s}^7 = (1, 1, 1, 1, 0, 0, 0)$. The factors representing the parity-check matrix constraints are

$$\begin{aligned} g_1(s_1, s_2, s_4) &= \mathbb{1}_{\{s_1 \oplus s_2 \oplus s_4 = 1\}} \\ g_2(s_3, s_4, s_6) &= \mathbb{1}_{\{s_3 \oplus s_4 \oplus s_6 = 0\}} \\ g_3(s_4, s_5, s_7) &= \mathbb{1}_{\{s_4 \oplus s_5 \oplus s_7 = 1\}}. \end{aligned}$$

The Tanner graph representing H is shown in Figure 2-2.

2.4 Prior Work

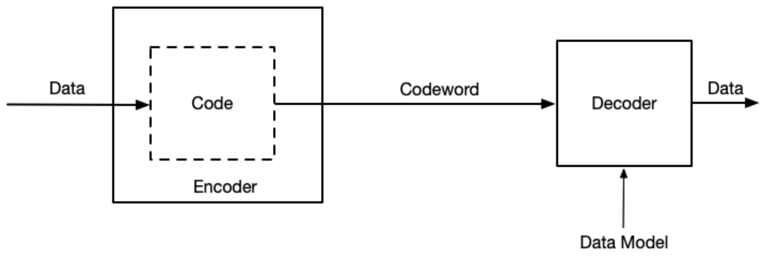


Figure 2-3: Model-code separation for compression architectures. The data is coded using a coding mechanism that is independent of the data model.

The problem of model-code separation in lossless compression architectures was studied in-depth by Huang and Wornell [26, 27]. The proposed architecture uses a compression pipeline with a model-free encoder and model-adaptive decoder, as shown in Figure 2-3. Note that the decoder must still be provided with some information about the coding mechanism to accurately decode the source. The architecture uses a low-complexity LDPC code for compressing binary sources. If the source symbols are from a larger alphabet, they are converted to a binary representation via a simple graycode transformation. The codeword is sent to the decoder which uses belief propagation to decode the source from the codeword.

Huang and Wornell showed that this architecture works well with binary sources and large alphabet sources for which the underlying PGM is known. However, even for simple datasets such as MNIST the method fails since we do not know the underlying PGM (which could be complicated).

It should be noted that structure learning algorithms for PGMs do exist [31, 15]. However, most algorithms make assumptions about the distribution of the nodes and/or try to induce sparsity in the graph. For example, the *Chow-Liu algorithm* [6] makes the assumption that the underlying graph is tree structured and the *graphical lasso algorithm* [16] assumes that the nodes follow a Gaussian distribution by optimizing for a sparse inverse covariance matrix. Other methods such as *neighborhood selection* [37] try to deduce conditional independencies between the variables of the input data to induce a sparse undirected PGM. As the data grows in complexity, these structure learning method either don't have sufficient representational power to efficiently describe the source or they fail to learn quickly.

Lee [36] and Lai [34] expanded on the separation architecture architecture by using a *restricted Boltzmann machine (RBM)* [48] to learn the source distribution. While this architecture is an improvement over its predecessor, an RBM still succumbs to the problems of structure learning as they are difficult to train on large datasets and often work better with Gaussian variables. Moreover, this architecture is more suitable for lossy compression.

In this thesis, we propose a novel data model based on the recently proposed deep probabilistic generative model called the sum-product network. We describe the sum-product network (SPN) in Chapter 3 (Sum-Product Networks) and describe our proposed SPN source-code BP algorithm for lossless image compression in Chapter 4 (Model-Code Separation Architecture).

2.5 Recent Work in Lossless Image Compression

Recent work in lossless image compression has focused on the use of deep generative models in the compression pipeline. Such systems use neural networks from input to output and are also called *neural image codecs*. One of the first neural image codecs that outperformed state-of-the-art lossless compression algorithms is the `L3C` system [38] introduced in 2019. The architecture uses an autoencoder framework to hierarchically compress an image into latent representations with decreasing spatial sizes. The latent representations are then quantized and fed to a PixelCNN [40], a state-of-the-art autoregressive model for modeling the likelihood of image sources. Autoregressive models are used to sample an image pixel-by-pixel by exploiting the chain rule of probability. PixelCNNs are capable of producing high resolution natural looking images. In `L3C`, PixelCNNs are used to learn a joint distribution over the latent maps that factorizes according to the chain rule. The learned distribution can be used to run length encode (RLE) the latent maps. Additional savings are made by compressing the distribution of each latent map above the bottommost layer using neural networks. Results show that `L3C` outperforms both `PNG` and `JPEG-2000`.

Recently, a new method based on normalizing flows called integer discrete flows (`IDF`) was introduced in [25]. Normalizing flows are a class of deep generative models that learn the likelihood over image sources by transforming a complex distribution into a tractable distribution, such as a Gaussian, by leveraging the variable transformation formula of probability. Normalizing flows are invertible neural networks that make use of specialized invertible transformations. Integer discrete flows are constructed by stacking together multiple invertible layers with quantizer modules in between. The `IDF` learns a tractable distribution over quantized latent maps that can be used to run length encode discrete bottleneck features. The images are losslessly reconstructed by decoding the latent maps and then recovering the input using the invertible layers. `IDF` demonstrates state-of-the-art results and it even outperforms `L3C` on various benchmarks.

In this thesis we do not directly compare our compression architecture against neural codecs. This is due to two related reasons — 1) the codecs considered in this section were trained on color images and hence to test them on new datasets would require a complete retraining of the architecture, and 2) training a neural codec on each of our experimental datasets would take at least a day each which is a much longer training time than our proposed architecture. Nerual codecs might also require vastly different structures for different data sources.

As discussed in Section 1.2.3, neural codecs are designed with joint model-code design and fall prey to the various drawbacks of such architectures. We hope that the work in this thesis will inspire future work in model-code separation within neural codecs.

Chapter 3

Sum-Product Networks

Probabilistic graphical models (Section 2.2) are powerful inference engines that can answer many probabilistic queries through marginalization via BP and sampling. However, the computational cost of exact inference is often very expensive for models representing complex distributions. Moreover, structure learning in graphical models is hard and is still an active field of research. In this section we introduce a new class of probabilistic generative models called sum-product networks (SPNs). We explain how SPNs solve the high complexity issue that we face with PGMs and show that by architectural modifications an SPN can be viewed as deep probabilistic graphical model.

3.1 Tractable Inference

Definition 3.1 (Tractable Model)**.** A tractable model is a probabilistic model in which exact inference requires a polynomial number of operations. Hence, inference can be performed with a tractable cost in both memory and time.

A classical example of a tractable model is a tree structured PGM where each marginal can be exactly computed using the BP algorithm with $\mathcal{O}(nT|\mathcal{S}|^2)$ complexity (see Section 2.2.3.1). When the graph has loops, BP inference is no longer exact and we instead have to resort to the *junction tree algorithm* [31, Chapter 10] to perform exact inference. The junction tree algorithm transforms the loopy graph into a tree structured graph with alphabet size per node upper bounded by $|\mathcal{S}|^{\mathrm{tw}(\mathcal{G})+1}$ resulting in a total inference cost of $\mathcal{O}(nT|S|^{2\mathrm{tw}(\mathcal{G})+2})$. Clearly the cost of exact inference can grow exponentially large for graphs with complex structures (i.e., large treewidths). Since exact inference is expensive on loopy graphs, approximate inference techniques such as loopy BP, variational approximations and MCMC [31, Chapters 11-12] are used to estimate probabilistic queries.

Consider the MNIST dataset of handwritten digits. To obtain probabilistic queries on an image from this dataset using the techniques discussed so far in this thesis, we would need to learn a PGM to represent the images. Structure learning in PGMs with discrete variables is not straightforward and moreover structure learning uses inference as a subroutine [31, Chapter 20]. While we could use an algorithm such as *Graphical lasso* [16] we are constrained to use the PGM as a Gaussian graphical model in this case, which is not ideal.

Sum-product networks, as we start to discuss next, help overcome many of these issues. They are probabilistic models that admit exact and tractable inference with complexity linear in the size of the model.

3.2 Architecture

A sum-product network is a deep probabilistic model introduced by Poon and Domingos [44]. A sum-product network represents a joint probability distribution over a set of random variables represented by a random vector $\mathbf{s}^n$. We state the formal definition below.

Definition 3.2 (Sum-Product Network)**.** A *sum-product network (SPN)* Φ is a rooted directed acyclic graph (DAG) whose leaves are tractable probability distributions and whose internal nodes are sums and products. Each outgoing edge (i, j) from a sum node is associated with a non-negative weight $w_{i,j}$. The value of an SPN $\Phi(\mathbf{s}^n)$ is the value of its root.

Definition 3.3 (Scope)**.** The *scope* of a node u is the set of variables that are descendants of u. The scope of a leaf node is the set of variables represented by the node.

Definition 3.4 (Evaluation of Sum-Product Network)**.** Let Φ be an SPN and let $\mathbf{s}^n$ be a source sequence whose likelihood we desire. The evaluation of the SPN proceeds from the leaves to the root with the following rules:

1. If u is a *leaf node* of the DAG, it is associated with a probability distribution ψ_u. The output of the node is $\psi_u(\mathbf{s}^n_{\mathrm{sc}(u)})$. Examples of leaf distributions are indicator distributions, categorical distributions and Gaussian distributions. If the leaf distribution is Gaussian and the scope of u is $\mathrm{sc}(u) = \{1, 3, 4\}$, then $\psi_u(\mathbf{s}^n_{\mathrm{sc}(u)}) = N(\mathbf{s}^n_{\mathrm{sc}(u)}; \mu_u, \Sigma_u)$ where μ_u is a 3-dimensional vector.

2. If u is a *product node*, the output is the product of its children's values,
$$\Phi_u(\mathbf{s}^n) = \prod_{v \in \mathrm{ch}(u)} \Phi_v(\mathbf{s}^n). \tag{3.1}$$

3. If u is a *sum node*, the output is the weighted sum of it children's values,
$$\Phi_u(\mathbf{s}^n) = \sum_{v \in \mathrm{ch}(u)} w_{u,v} \Phi_v(\mathbf{s}^n). \tag{3.2}$$

If we are provided with partial observations of the input sequence $\mathbf{s}^n_{\mathcal{A}}$, $\mathcal{A} \subset \{1, \ldots, n\}$ we can still compute the marginal probabilities of the input. The only modification that we need to make in the evaluation of the SPN is to marginalize out unobserved

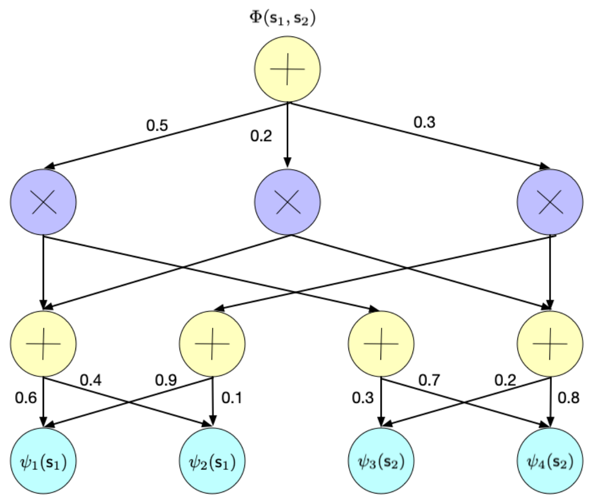


Figure 3-1: A normalized sum-product network over two random variables. The leaf nodes have singular scope and can contain any tractable distribution. If the leaf distribution is an indicator, the weights along the edges from the leaf to sum node are parameters for a categorical distribution. If the leaf distributions are, for example, Gaussian then the SPN can be interpreted as mixture model.

values in the leaf distribution, i.e., $\mathbf{s}^n_{\mathrm{sc}(u)\setminus\mathcal{A}}$. Hence, the new leaf node evaluation step is

$$\psi_u(\mathbf{s}^n_{\mathrm{sc}(u)\bigcap\mathcal{A}}) = \sum_{\mathbf{s}^n_{\mathrm{sc}(u)\setminus\mathcal{A}}} \psi_u(\mathbf{s}^n_{\mathrm{sc}(u)\setminus\mathcal{A}}, \mathbf{s}^n_{\mathrm{sc}(u)\bigcap\mathcal{A}}). \tag{3.3}$$

The definitions above provide us with a routine to evaluate the SPN. However, they don't tell us when an SPN correctly computes all the marginals of the distribution it models. This necessitates the following definitions from [44].

Definition 3.5 (Valid Sum-Product Network)**.** An SPN Φ that models some distribution $p_{\mathbf{s}^n}$ is *valid* if and only if $\Phi(\mathbf{s}^n_{\mathcal{A}}) = Zp_{\mathbf{s}^n_{\mathcal{A}}}(\mathbf{s}^n_{\mathcal{A}})$ for all $\mathcal{A} \subset \{1, \ldots, n\}$. In other words the SPN is valid if it always correctly computes the unnormalized marginal probabilities.

Definition 3.6 (Completeness)**.** An SPN is *complete* if all children of a sum node have identical scopes.

Definition 3.7 (Decomposability)**.** An SPN is *decomposable* if all children of the same product node have pairwise disjoint scopes.

Clearly, a valid SPN is desirable. The next property tells us how we can construct valid SPNs.

Property 3.1. *If an SPN is complete and decomposable it is valid.*

Imposing completeness and decomposability in an SPN architecture is very easy as we will see in Section 3.4. We can, in fact, make one more modification to the SPN architecture to ensure that it always outputs normalized probabilities.

Definition 3.8 (Normalized Sum-Product Network)**.** A *normalized sum-product network* is an SPN wherein the weights at every sum node add up to one. Such an SPN always outputs normalized probabilities.

Example 3.1. Consider the SPN in Figure 3-1 defined over binary random variables s_1 and s_2. We model the leaf distributions as indicators,

$$\begin{aligned}
\psi_1(s_1) &= \mathbb{1}_{\{s_1=1\}} \triangleq s_1 \\
\psi_2(s_1) &= \mathbb{1}_{\{s_1=0\}} \triangleq \bar{s}_1 \\
\psi_3(s_2) &= \mathbb{1}_{\{s_2=1\}} \triangleq s_2 \\
\psi_4(s_2) &= \mathbb{1}_{\{s_2=0\}} \triangleq \bar{s}_2
\end{aligned}$$

It is straightforward to verify that this SPN is complete and decomposable. Hence, this SPN is valid. The distribution represented by this SPN can be written out in *network polynomial* form as

$$\begin{aligned}
\Phi(s_1, s_2) = 0.5&(0.6s_1 + 0.4\bar{s}_1)(0.3s_2 + 0.7\bar{s}_2) \\
&+ 0.2(0.6s_1 + 0.4\bar{s}_1)(0.2s_2 + 0.8\bar{s}_2) \\
&+ 0.3(0.9s_1 + 0.1\bar{s}_1)(0.2s_2 + 0.8\bar{s}_2).
\end{aligned}$$

Let's compute the marginal probabilities of some queries.

1. If $s_1 = 1$ and $s_2 = 0$,

$$\begin{aligned}
p_{\mathsf{s}_1,\mathsf{s}_2}(1,0) &= \Phi(1,0) \\
&= 0.5 \times 0.6 \times 0.7 + 0.2 \times 0.6 \times 0.8 + 0.3 \times 0.9 \times 0.8 \\
&= 0.522.
\end{aligned}$$

2. If $s_1 = 1$ and $s_2 = 1$,

$$\begin{aligned}
p_{\mathsf{s}_1,\mathsf{s}_2}(1,1) &= \Phi(1,0) \\
&= 0.5 \times 0.6 \times 0.3 + 0.2 \times 0.6 \times 0.2 + 0.3 \times 0.9 \times 0.2 \\
&= 0.167\bar{9}.
\end{aligned}$$

3. If $s_1 = 1$, we must first marginalize out s_2 from every leaf distribution according to (3.3). Only the leaf distributions involving s_2 are modified:

$$\begin{aligned}
\psi_3 &= 0 + 1 = 1, \\
\psi_4 &= 0 + 1 = 1.
\end{aligned}$$

The network polynomial with these marginalized leaf distributions is

$$\begin{aligned} p_{\mathsf{s}_1}(1) = \Phi(s_1) &= 0.5 \cdot 0.6(0.3 + 0.7) \\ &+ 0.2 \cdot 0.6(0.2 + 0.8) \\ &+ 0.3 \cdot 0.9(0.2 + 0.8) \\ &= p_{\mathsf{s}_1,\mathsf{s}_2}(1,1) + p_{\mathsf{s}_1,\mathsf{s}_2}(1,0). \end{aligned} \tag{3.4}$$

Notice that in order to marginalize out s_2 all we needed to do was to set the leaf "distribution" to be a constant function that always evaluates to 1.

The result in (3.4) is extremely useful in computing marginals and we state it as a property below.

Property 3.2. *If all leaves in an SPN involving s_i have singular scope, we can marginalize out s_i by setting the leaf distribution functions involving this random variable to always evaluate to unity.*

3.3 Comparison with Other Architectures

SPNs bear resemblance to many other probabilistic and generative models. We look at a few related architectures and comment on the similarities and differences.

3.3.1 SPNs vs. PGMs

As stated in Section 3.1, exact inference is computationally expensive with PGMs that have high treewidth. Poon and Domingos [44] showed that via a simple algorithm, every junction tree (and hence every PGM) can be converted to an SPN and vice versa. Thus, the computation complexity of an SPN is similar to a junction tree. In general, a PGM has fewer edges than the corresponding SPN for the same distribution. So what is the benefit of using an SPN?

- **The key feature of an SPN is that it encodes all possible inference queries** whereas a PGM gives us the unnnormalized probability of a full source sequence. As we saw in Example 3.1, all marginals in an SPN can be computed in a single forward pass from leaves to nodes with complexity linear in the number of edges in the underlying DAG. While a naïve conversion from an undirected graph to an SPN would introduce possibly exponential number of edges, we can group together recurring computation by merging the computation subgraphs within the SPN DAG. This is akin to reusing messages in the BP algorithm. The sum and product operations represented as a DAG can be implemented very efficiently using parallel processors/GPUs. This allows SPNs to perform fast inference on high-treewidth graphs which would otherwise require approximate inference with PGMs.

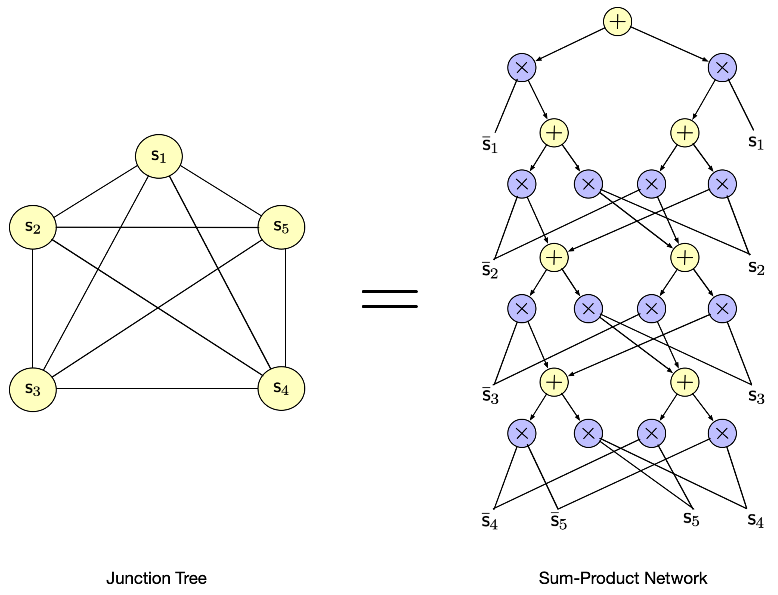


Figure 3-2: An SPN can represent a distribution over all sequences with an even number of 1s with $\mathcal{O}(n)$ edges whereas a junction tree requires a fully connected graph with $\mathcal{O}(n^2)$ edges.

- **Computation of the partition function is tractable in an SPN.** The limiting factor of PGMs is the computation of the partition function. With SPNs we circumvent the need for approximate inference algorithms to compute the partition function since it can be evaluated by marginalizing out all variables, i.e., setting all leaf distribution functions to one.

- **SPNs can represent non-factorizable high treewidth models more compactly**. Poon and Domingos [44] showed that an SPN can compactly represent the uniform distribution over all length n sequences with an even number of 1s whereas a PGM would require a fully connected graph to represent the same distribution, as shown in Figure 3-2. A fully connected polygon has $n(n-3)/2+n$ edges. Hence, the number of edges in the PGM representation grows as $\mathcal{O}(n^2)$ whereas the number of edges grow as $\mathcal{O}(n)$ in the SPN.

- **Context-specific independencies can help reduce the size of the SPN.** Context-specific independencies, independence relations that induce a different factorization structure in a distribution based on the value of random variable, create large cliques in a graphical model representation [50]. SPNs can compactly represent contextual-independencies with fast inference times by coalescing nodes and edges together. For example, in Figure 3-2, if the PGM had two cliques connected to each with a separator set of $\{\mathsf{s}_1, \mathsf{s}_2\}$, the upper portion of the SPN involving s_1 and s_2 would be shared between the SPN representation of both the cliques.

- **PGMs learned from data are often intractable unless the model size is kept small.** SPNs on the other hand can learn from large amounts of data with fast learning algorithms. SPNs can be made very deep and can be implemented very efficiently using tensorized operations. Moreover parameter learning is very efficient and hence the same SPN architecture can be used to model different sources of data.

- **PGMs convey factorization structure whereas SPNs do not**. The factorization structure of the distribution is lost in an SPN. This is a tradeoff that is made for faster inference.
- **PGMs encode conditional independencies whereas SPNs do not**. To bridge the gap between SPNs and PGMs, a new class of models called sum-product graphical models [14] was proposed to model independencies in the distribution while allowing for fast inference at the same time.

3.3.2 SPNs vs. Arithmetic Circuits

SPNs with indicator leaves are related to arithmetic circuits. An arithmetic circuit [10] uses the distributive law to express a distribution as a network polynomial (see Example 3.1). They are generally used to compile a directed graphical model over discrete variables into a network polynomial for faster inference. Unlike SPNs, where weights are present along the edges going into a sum node, an arithmetic circuit only includes weights in the leaves. An arithmetic circuit is a standard tool used to efficiently represent a polynomial. SPNs are more general because the leaf distribution need not be an indicator but rather any tractable distribution. Moreover, SPNs can be made quite deep and have weights along sum nodes to model complex distributions.

3.3.3 SPNs vs. Deep Generative Models

SPNs are closely related to deep generative models. The main difference between deep generative models and SPNs is that the latter has a probabilistic interpretation at each layer. Deep generative models use non-linear activations in their architecture and do not need to obey the architectural constraints of a valid SPN. Most deep generative models only output the likelihood of complete data whereas SPNs can answer marginal queries of incomplete data. SPNs have also been used as discriminative models [20, 45] to classify images into classes but deep neural networks (DNNs) still boast state-of-the-art results in this task.

Like normalizing flows [46] and autoregressive models [41, 29], SPNs can be trained via gradient descent to minimize the negative log-likelihood of data. They can also be trained using unsupervised learning methods such as the *expectation-maximization (EM)* algorithm [14].

Deep SPNs can be efficiently implemented on GPUs using convolutions [5, 61]. In fact, deeper sum-product networks with more layers and fewer sums and products per layer can model distributions better than shallow networks [12]. Random Tensorized SPN (RAT-SPN) [43] which use multiple copies of random grouping of the input variables have demonstrated that SPNs can capture rich structure about the source distribution even with randomized source connections.

There is long way to go before SPNs reach the performance of current generative models for image synthesis tasks and researchers are actively trying to bridge the gap between these models.

3.4 Deep Generalized Convolutional Sum-Product Networks

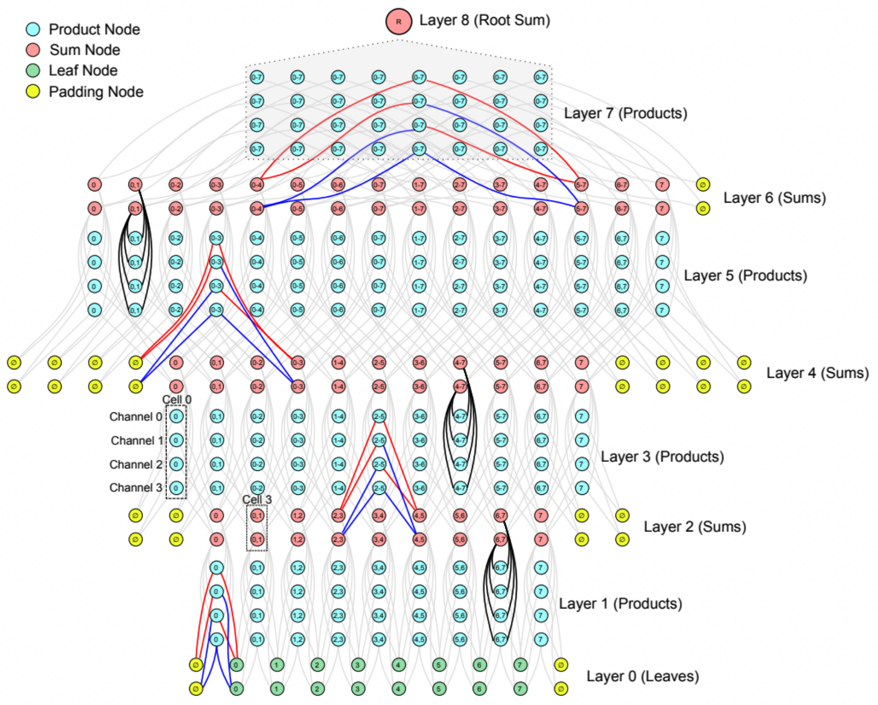


Figure 3-3: Example DGCSPN architecture in 1D taken from [61]. Layer 0 contains the leaf distributions. Every product layer doubles the dilation rate, starting with a rate of 1. The scopes are indicated by the numbers within each node. All children of the same sum node have the same scope.

As discussed in Section 3.3.3, SPNs can be implemented on GPUs using convolutions very efficiently. In this thesis we modify the deep generalized convolutional SPN (DGCSPN) proposed in [61] for lossless compression of images. The architectural details of DGCSPNs is summarized next.

- All leaf nodes have singular scope, i.e., they correspond to a single symbol in the input source.
- Sum layers and product layers are stacked in an alternating fashion as shown in Figure 3-3.
- Log-probabilities are propagated to avoid underflow issues.

- A sum node is implemented in the log-domain using the log-sum exponential trick.

$$\tilde{\Phi}_u(\mathbf{s}^n) = \log\left(\sum_{v\in \mathrm{ch}(u)} \exp\left(\tilde{w}_{u,v} + \tilde{\Phi}_v(\mathbf{s}^n) - c\right)\right) + \log c, \tag{3.5}$$

  where $\tilde{a} = \log a$ and c is some positive constant.

- S output channels are created at every sum node by adding P input channels, each having the same scope, from the previous product layer. For example, in Figure 3-3, $P = 4$ channels in Layer 1 are summed up twice to produce $S = 2$ output channels in Layer 2.

- Product layers are implemented as convolutions with increasing dilation rates at every subsequent layer. The convolutional patches overlap in every layer since the stride chosen is one. To ensure that the decomposability property is satisfied by the SPN, subsequent product layers must use a *dilation factor* that is twice that of the previous product to layer. A dilation factor of k in a convolution adds $k-1$ zeros between each element of the kernel. This is shown in Figure 3-3 by the increasing distance between inputs to product nodes as the layer increases.

- Like the sum nodes, P output channels are created at every product node by multiplying S instances/channels of a sum node with disjoint scopes from the previous sum layer. Padding nodes, which are set to 0 in the log domain, are used to ensure that GPU-optimized convolutions can be used correctly.

We modify the DGCSPN architecture to use **indicator leaves** instead of Gaussian leaves since this will allow us to compute marginal probabilities easily for use with BP on Tanner graphs. The implementation available online[1] only supports 2-dimensional signals. We modified the architecture so that it can support 1-dimensional convolutions for learning non-image sources. A routine for computing the MPE of a partially observed source was already implemented for Gaussian leaf distributions. We generalize this to other leaf distributions and implement a routine to compute all unary marginals in parallel as we describe in Section 3.6.

3.5 Parameter Learning

Parameter learning in SPNs can done via gradient descent or the expectation-maximization (EM) algorithm [13, Section 3.3.2]. Since we implement DGCSPNs using PyTorch [42], a deep learning framework, we use gradient descent for learning the parameters of our model. We only use indicator leaves in our experiments. Thus, the only weights that we need to learn are the weights along edges emanating from a sum node.

[1] We use the PyTorch implementation available at https://github.com/deeprob-org/deeprob-kit.

For numerical stability we propagating log-probabilities in the SPN. The optimization problem of the learning procedure is the minimization of the negative log-likelihood of the data,

$$\arg\min_{\mathbf{s}^n} - \log \Phi(\mathbf{s}^n). \tag{3.6}$$

To perform gradient descent we need the derivative of the log-probability with respect to the edge weight $w_{i,j}$ from a sum node i to a product node j:

$$\begin{aligned} \frac{\partial}{\partial w_{i,j}} \log \Phi(\mathbf{s}^n) &= \frac{1}{\Phi(\mathbf{s}^n)} \frac{\partial \Phi(\mathbf{s}^n)}{\partial w_{i,j}} && (3.7) \\ &= \frac{1}{\Phi(\mathbf{s}^n)} \frac{\partial \Phi(\mathbf{s}^n)}{\partial \Phi_i(\mathbf{s}^n)} \frac{\partial \Phi_i(\mathbf{s}^n)}{\partial w_{i,j}} && (3.8) \\ &= \frac{1}{\Phi(\mathbf{s}^n)} \frac{\partial \Phi(\mathbf{s}^n)}{\partial \Phi_i(\mathbf{s}^n)} \Phi_j(\mathbf{s}^n). && (3.9) \end{aligned}$$

Thus, the parameters can now be updated as

$$w_{i,j} \leftarrow w_{i,j} - \eta \frac{\partial}{\partial w_{i,j}}, \tag{3.10}$$

where η is the learning rate. Gradient computation is very easy in PyTorch using the optimized `autograd` function. What's more valuable is that we can use these gradients to compute the marginals of all the leaf node variables in a single forward and backward pass of the network.

3.6 Parallel Marginal Computation

We now describe a procedure to compute the marginals of all the variables in a single forward and backward pass of an SPN with singular scope indicator leaves. Darwiche [11] showed that in an arithmetic circuit, the posterior marginal of a leaf variable can be computed using gradients. Though arithmetic circuits use indicator leaves, the same formula gives us the *leaf distribution assignment probability* in a general SPN. The formula is,

$$p_{[\mathsf{s}_i]_k \,|\, \mathbf{s}^n_{\mathcal{A}}}([s_i]_k \,|\, \mathbf{s}^n_{\mathcal{A}}) = \frac{1}{\Phi(\mathbf{s}^n_{\mathcal{A}})} \frac{\partial \Phi(\mathbf{s}^n_{\mathcal{A}})}{\partial \psi_k}, \tag{3.11}$$

where $[\mathsf{s}_i]_k$ represents the k'th leaf distribution instance that involves s_i. For example, in Figure 3-1, there are two leaf distributions for s_i — ψ_1 and ψ_2. Thus, $p_{[\mathsf{s}_1]_2 \,|\, \mathbf{s}^n_{\mathcal{A}}}([s_1]_2 \,|\, \mathbf{s}^n_{\mathcal{A}})$ is the assignment probability of leaf distribution ψ_2 given the observed data $\mathbf{s}^n_{\mathcal{A}}$. For example, if the leaves are Gaussian, the probabilities can be interpreted as being similar to the soft mixture assignment probabilities, p_{z}, in Gaussian Mixture Models (GMMs):

$$p_{\mathsf{x}}(x) = \sum_i p_{\mathsf{x}|\mathsf{z}}(x|z) p_{\mathsf{z}}(z) \tag{3.12}$$

where $z \in \mathcal{Z}$ represents the number of clusters in the GMM. When the leaf distributions are indicators then (3.11) is equivalent to the categorical marginal probability of the discrete random variable s_i:

$$p_{[\mathsf{s}_i]_k \,|\, \mathbf{s}^n_{\mathcal{A}}}([s_i]_k \,|\, \mathbf{s}^n_{\mathcal{A}}) = p_{\mathsf{s}_i \,|\, \mathbf{s}^n_{\mathcal{A}}}(k \,|\, \mathbf{s}^n_{\mathcal{A}}). \tag{3.13}$$

Since we propagate log-probabilities in the network we need to relate the gradients in the log-domain to (3.11).

$$\begin{aligned}
\frac{\partial \log \Phi(\mathbf{s}^n_{\mathcal{A}})}{\partial \log \psi_k} &= \frac{\partial \log \Phi(\mathbf{s}^n_{\mathcal{A}})}{\partial \Phi(\mathbf{s}^n_{\mathcal{A}})} \frac{\partial \Phi(\mathbf{s}^n_{\mathcal{A}})}{\partial \log \psi_k} \\
&= \frac{1}{\Phi(\mathbf{s}^n_{\mathcal{A}})} \frac{\partial \Phi(\mathbf{s}^n_{\mathcal{A}})}{\partial \log \psi_k} \frac{\psi_k}{\psi_k} \\
&= \frac{1}{\Phi(\mathbf{s}^n_{\mathcal{A}})} \frac{\partial \Phi(\mathbf{s}^n_{\mathcal{A}})}{\partial \log \psi_k} \frac{\partial \log \psi_k}{\partial \psi_k} \psi_k \\
&= \frac{1}{\Phi(\mathbf{s}^n_{\mathcal{A}})} \frac{\partial \Phi(\mathbf{s}^n_{\mathcal{A}})}{\partial \psi_k} \psi_k && \text{(3.14a)} \\
&= p_{[\mathsf{s}_i]_k \,|\, \mathbf{s}^n_{\mathcal{A}}}([s_i]_k \,|\, \mathbf{s}^n_{\mathcal{A}}) \quad \text{since } \psi_k = 1. && \text{(3.14b)}
\end{aligned}$$

Note that in (3.14a) we have used the fact s_i is unobserved and hence was marginalized out in the forward pass of the SPN. From Property 3.2, we know that all leaf distributions containing s_i in their scope must be set to 1 in order to marginalize out s_i.

Thus, with a single forward pass and backward pass through the SPN we can compute all desired marginals for the provided evidence.

3.7 Inference with External Beliefs

In order to use an SPN along with existing BP algorithms the architecture must be capable of using externals beliefs about the input source to answer probabilistic queries. In a PGM, external beliefs b_i at a node s_i are incorporated by scaling the node potential ϕ_i by the value of the external belief,

$$\phi'_i(s_i) = b_i(s_i)\phi_i(s_i). \tag{3.15}$$

These external beliefs could come from some external graph that encodes other statistical properties about the source. The external belief is treated as an independent factor in the joint distribution. SPNs can consume external beliefs in a similar manner by multiplying the external belief distribution with the leaf distributions. Figure 3-4 illustrates this augmentation by modifying the SPN from Example 3.1.

Assume that the SPN has **indicator leaves**. Let the external belief to the k'th leaf distribution that has s_i in its scope be ψ^e_k. It is straightforward to verify that the posterior marginal formula must now be modified to remove the effect of scaling the leaf distribution when doing a forward pass through the model. Hence, (3.13)

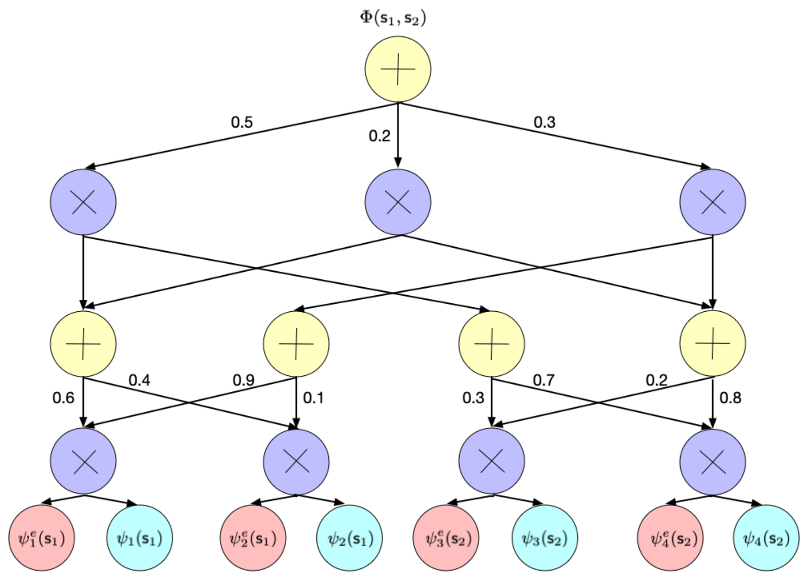


Figure 3-4: The SPN from Example 3.1 with external beliefs (colored in pink) provided by multiplying them with the leaf distribution.

becomes,

$$p_{\mathsf{s}_i \mid \mathbf{s}_{\mathcal{A}}^n}(k \mid \mathbf{s}_{\mathcal{A}}^n) = \frac{1}{\psi_k^e \Phi(\mathbf{s}_{\mathcal{A}}^n)} \frac{\partial \Phi(\mathbf{s}_{\mathcal{A}}^n)}{\partial \psi_k}. \tag{3.16}$$

We are now equipped with all the machinery to use SPNs for developing lossless compression architectures with model-code separation baked into the design.

Chapter 4

Model-Code Separation Architecture

Equipped with all the tools underlying the model-code separation architecture for lossless compression, we are now ready to describe its operation. We first describe its operation when given the PGM representation of the source and we then show how we can replace the PGM with an SPN to compress natural looking images from large datasets.

4.1 PGM-based Model-Code Separation Architecture

In this section, we describe the model-code separation architecture proposed by Huang and Wornell [27, 26] that uses a PGM as the data model. The architecture follows the design of Figure 2-3 with the encoder given no prior knowledge about the source. Though we could use a universal compression standard such as `GZIP`, the compression rate might be suboptimal since the system requires long input source sequences to learn the correct statistics.

In this architecture the decoder can be made as powerful as needed. In particular, we assume that the decoder is aware of the source structure in the form of a *source graph*, a PGM representing the statistical structure in the input signal. This suggests that the ideal code would be one that lends itself to optimal decoding using belief propagation. A suitable choice is the family of LDPC codes which we know can be represented as factor graphs (see Section 2.3.1).

We begin our discussion with the compression of binary sequences of length n, $\mathbf{s}^n \in \mathcal{S}^n$ where $\mathcal{S} = \{0, 1\}$.

4.1.1 Encoder

The encoder compresses the source via a simple projection with an LDPC parity-check matrix. Given a source sequence of length n and a target compression rate of $r_{\text{code}} = k/n$, the encoder generates a random LDPC parity-check matrix $H \in \{0,1\}^{k \times n}$. The compressed sequence is obtained as

$$\mathbf{c}^k = H\mathbf{s}^n. \tag{4.1}$$

4.1.2 Decoder

Since H is a tall matrix, the compressed sequence $\mathbf{c}^k$ corresponds to many possible source sequences. If we decode $\mathbf{s}^n$ using BP on the Tanner graph of H, the decoded output might not satisfy the statistical structure inherent to the source. The source graph captures this structure and we can run BP over this graph by providing the beliefs from the Tanner graph as external node potentials. This process is carried out iteratively until the source is decoded correctly.

4.1.2.1 Code Graph

The matrix H enforces k constraints which can be represented by a factor graph as demonstrated in Example 2.1. The resulting factor graph $\mathcal{G}_{\text{code}} = (\mathcal{V}, \mathcal{F}, \mathcal{E})$ has k factor nodes, $\mathcal{F} = \{1, \ldots, k\}$ and n variable nodes, $\mathcal{V} = \{1, \ldots, n\}$. The constraint represented by factor node a corresponds to row a of the LDPC matrix:

$$c_a = \sum_{i=1}^{n} H_{a,i} s_i. \tag{4.2}$$

The factor graph represents a uniform distribution over all codewords that satisfy the parity-check equations,

$$p^c_{\mathbf{s}_\mathcal{V}}(\mathbf{s}_\mathcal{V}) = \frac{1}{Z_c} \prod_{a \in \mathcal{F}} f_a(\mathbf{s}_{\mathcal{N}(a)}) = \frac{1}{Z_c} \prod_{a \in \mathcal{F}} \mathbb{1}_{\{c_a = \sum_{i=1}^{n} H_{a,i} s_i\}}. \tag{4.3}$$

This factor graph is called the *code graph.*

4.1.2.2 Source Graph

Suppose the data model is available to us in the form of an undirected PGM with n nodes, one for each symbol in the source sequence,

$$p^s_{\mathbf{s}_\mathcal{V}}(\mathbf{s}_\mathcal{V}) = \frac{1}{Z_s} \prod_{\mathcal{C} \in \text{cl}(\mathcal{G})} \phi_\mathcal{C}(\mathbf{s}_\mathcal{C}). \tag{4.4}$$

This graph is called the *source graph.* We can now easily run BP on this source graph by taking in external beliefs from the code graph which we describe next.

4.1.3 Source-Code Belief Propagation

The source graph and code graph are combined into a single graph as shown in Figure 4-1. The combined graph represents the distribution,

$$p^{\text{comb}}_{\mathbf{s}_\mathcal{V}}(\mathbf{s}_\mathcal{V}) = p^s_{\mathbf{s}_\mathcal{V}}(\mathbf{s}_\mathcal{V}) p^c_{\mathbf{s}_\mathcal{V}}(\mathbf{s}_\mathcal{V}). \tag{4.5}$$

The decoder runs BP on the combined graph to compute approximate marginals of the source sequence that satisfies both the source constraints and the code constraints.

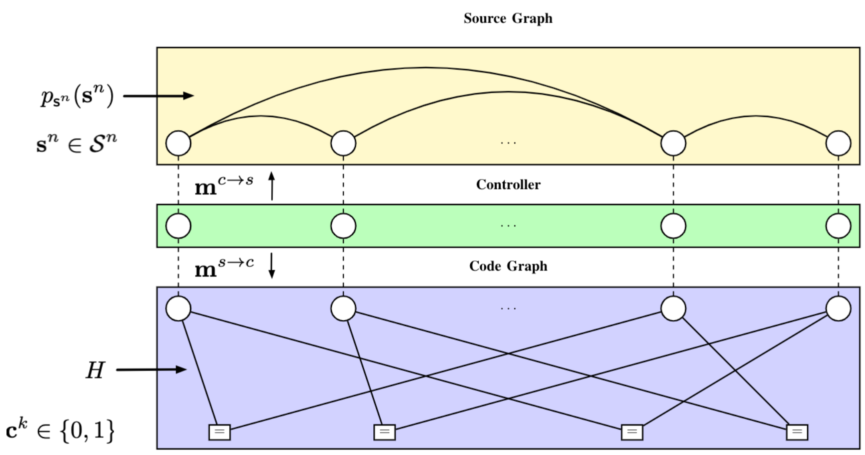

Source Graph
$p_{\mathbf{s}^n}(\mathbf{s}^n)$
$\mathbf{s}^n \in \mathcal{S}^n$
$\mathbf{m}^{c\to s}$
Controller
$\mathbf{m}^{s\to c}$
Code Graph
H
$\mathbf{c}^k \in \{0,1\}$

Figure 4-1: The combined source-code decoder. The source and code graph share the common nodes denoting the source symbols which is represented by the virtual controller.

Only the nodes representing the source symbols interact with the graphs on either side of the virtual controller. Assume that the source graph only has unary node potentials and pairwise potentials along edges. BP can be carried out efficiently using the following rules:

- Begin by initializing all messages in both graphs to $1/|\mathcal{S}|$. The controller accumulates the messages from the shared variables nodes of one graph and sends it to the other. Let the message at node u being sent from the code graph to source graph be $\mathbf{m}^{c\to s}_{u\to u}$ and let the message in the other direction be $\mathbf{m}^{s\to c}_{u\to u}$.

- If u is a node in the code graph, accumulate the messages from the controller, $\mathbf{m}^{s\to c}_{u\to u}$, by treating them as additional factor nodes,

$$g_u(s_u) \triangleq m^{s\to c}_{u\to u}(s_u). \tag{4.6}$$

- Using the updated node potentials, run BP on the code graph and compute the marginal of s_u by accumulating messages from all neighbors of u except the controller node. Send this message to the controller and call it $\mathbf{m}^{c\to s}_{u\to u}$.

- If u is a node in the source graph, accumulate the messages from the controller, $\mathbf{m}^{c\to s}_{u\to u}$, by multiplying them with the node potentials of the source graph,

$$\phi'_u(s_u) = \phi_u(s_u) m^{c\to s}_{u\to u}(s_u). \tag{4.7}$$

- Using the updated node potentials, run BP on the source graph and compute the marginal of s_u by accumulating messages from all neighbors of u except the controller node. Send this message to the controller and call it $\mathbf{m}^{s\to c}_{u\to u}$.

- Compute the unnormalized beliefs at every node by taking the product of the beliefs from both graphs.

$$b_u(s_u) = m^{c\to s}_{u\to u}(s_u) m^{s\to c}_{u\to u}(s_u). \tag{4.8}$$

Repeat the above steps until the unnormalized beliefs converge. The decompressed output can be recovered as

$$\hat{s}_u = \arg\max_{s\in\mathcal{S}} b_u(s). \tag{4.9}$$

4.1.4 Dealing with Large Alphabet Sources

We have so far described a model-code separation compression architecture that can compress binary sources. To handle non-binary sources a *translator* module is introduced in the architecture. Consider a source sequence $\mathbf{s}^n$ of length n where each symbol is drawn from an alphabet $\mathcal{S}$ of size $|\mathcal{S}| = M$. The objective of the translator is to translate the symbols into a representation that can be used by the architecture. Specifically, the code graph of the architecture that uses LDPC codes is constrained to use a bit-level representation of the source sequence.

We choose to use a graycode representation of the source sequence as done in [26]. Assume that $M = 2^B$ for some non-negative integer B. The graycoded representation of a source symbol s_i is a length $B = \log_2 |\mathcal{S}|$ sequence of binary digits

$$\mathbf{z}_i^B = (z_{i,1}, z_{i,2}, \ldots, z_{i,B}) \triangleq t_{|\mathcal{S}|\to 2}(s_i), \tag{4.10}$$

where $t_{|\mathcal{S}|\to 2} : \mathcal{S} \to \{0,1\}^B$ is the translator function that maps symbols over $\mathcal{S}$ to B-tuple strings over $\{0,1\}$. We define another translator function $T_{|\mathcal{S}|\to 2} : (\mathcal{S} \to \mathbb{R}^+) \to (\{0,1\} \to \mathbb{R}^+)^B$ that maps messages over $\mathcal{S}$, $\mathbf{m}^{(|\mathcal{S}|)}$, to messages over $\{0,1\}$, $\mathbf{m}^{(2)}$,

$$T_{|\mathcal{S}|\to 2}(\mathbf{m}^{(|\mathcal{S}|)}) = (\mathbf{m}_1^{(2)}, \ldots, \mathbf{m}_B^{(2)}) = \mathbf{m}^{(2)}, \tag{4.11}$$

where each message $\mathbf{m}_i^{(2)}$ is a 2-dimensional message over the alphabet $\{0,1\}$. Assuming that the messages are normalized probabilities, for $\omega \in \{1,\ldots,B\}$ and $\beta \in \{0,1\}$

$$T_{|\mathcal{S}|\to 2}(\mathbf{m}^{(|\mathcal{S}|)})_\omega(\beta) = m_\omega^{(2)}(\beta) \triangleq \sum_{s\in\mathcal{S}} m^{(|\mathcal{S}|)}(s) \mathbb{1}_{\{t_{|\mathcal{S}|\to 2} = \beta\}}. \tag{4.12}$$

The translation functions can be defined in the reverse direction in a similar manner.

$$t_{2\to|\mathcal{S}|}(\mathbf{z}_i^B) \triangleq \sum_{j=1}^{B} 2^{B-j} z_{i,j}, \tag{4.13}$$

and for $s \in \mathcal{S}$

$$T_{2\to|\mathcal{S}|}(\mathbf{m}^{(2)})(s) = \prod_{\omega=1}^{B} m_\omega^{(2)}(t_{|\mathcal{S}|\to 2}(s)_\omega). \tag{4.14}$$

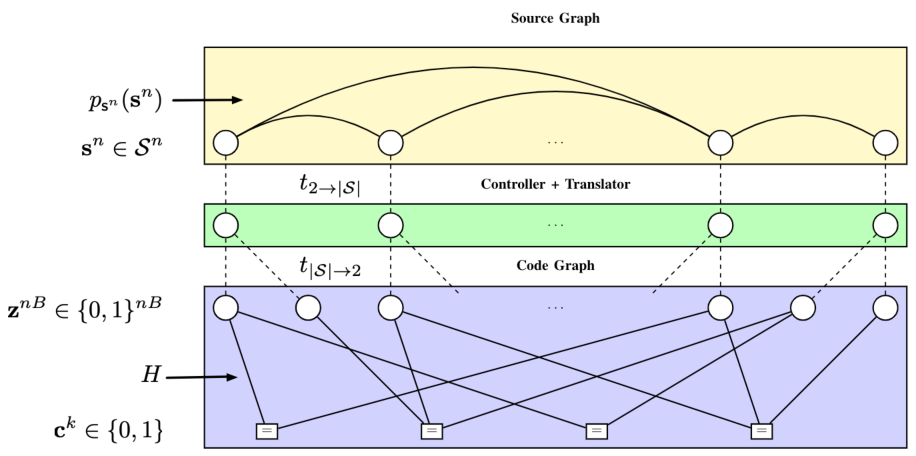

Source Graph
$p_{\mathbf{s}^n}(\mathbf{s}^n)$
$\mathbf{s}^n \in \mathcal{S}^n$
$t_{2\to|\mathcal{S}|}$
Controller + Translator
$t_{|\mathcal{S}|\to 2}$
Code Graph
$\mathbf{z}^{nB} \in \{0,1\}^{nB}$
H
$\mathbf{c}^k \in \{0,1\}$

Figure 4-2: The combined source-code decoder with a translator introduced to compress large alphabet sources.

The translation equations state that $\mathbf{m}^{|\mathcal{S}|}$ is a product of the marginals $\mathbf{m}_1^{(2)}, \ldots \mathbf{m}_B^{(2)}$ appropriately indexed at the indices specified by $t_{|\mathcal{S}|\to 2}$.

Fact 4.1. The graycode representation of a large alphabet symbol is heavily used in lossless compression, specifically in *run length encoding (RLE)*. The graycode transformation constrains large alphabet symbols with similar values to have bit representations with only a few bit flips. Consider the integer 3 with binary representation 011 and the integer 4 with binary representation 100. Though these numbers are similar, especially in the context of intensity of pixels, the binary representations are hard to compress due to three bit-flips. On the other hand, the graycode representations 010 and 110 for 3 and 4 respectively only differ by a single bit flip. Hence, graycodes are beneficial when using codes that leverage the differences in neighboring values.

If z is an integer and $\mathsf{bin}(z)$ is its binary representation, the graycode representation is obtained by the following transformation

$$\mathsf{gray}(z) = \mathsf{xor}(\mathsf{rightshift}(\mathsf{bin}(z), 1), \mathsf{bin}(z)). \tag{4.15}$$

4.1.5 Doping

Often the decoder does not converge without some non-trivial initialization of the messages. To ensure that decoding converges, we transmit a subset of the input source symbols uncompressed. This process is called *doping* and is analyzed in more detail in [26]. Doping acts as a seed for the decoding algorithm and helps in reducing the solution set of possible source sequence corresponding to a codeword. The overall compression rate of the architecture is

$$r_{\text{total}} = r_{\text{code}} + r_{\text{dope}}, \tag{4.16}$$

where r_{code} is defined in Section 4.1.1.

4.1.6 Overall Architecture

The overall model-code separation architecture for lossless compression of sources with arbitrary alphabet size is shown in Figure 4-2. The decoding algorithm follows the same procedure described in Section 4.1.3 with an extra step to translate the messages according to equations (4.12) and (4.14) in the controller.

4.1.7 Drawbacks of PGM-based Decoder

The main drawback of the current PGM based source-code decoder is highlighted in the difficulty of learning a PGM for an arbitrary source, as discussed in Section 2.4. As detailed in [26, Chapter 10], the current model is able to perform efficient decoding of sources with known graphical model structure. However, the model fails to decode simple images such as handwritten digits from the MNIST dataset since the true PGM is unknown. There have been advances in structure learning for PGMs but most work has concentrated on learning continuous models such as Gaussian graphical models. It is for this reason that we would like to replace the current PGM source graph with a flexible and powerful SPN.

4.2 SPN-based Model-Code Separation Architecture

We immediately see the benefits of using a model-code separation architecture when switching over to an SPN source model:

- Since we are only swapping out the PGM, the encoder stays unchanged!
- The PGM nodes representing the source symbols only interact with the controller. As long as we have a mechanism for the SPN to accept beliefs from the code graph and use these beliefs to return beliefs to the controller for translation (if required), we can use the source-code BP algorithm for decoding.
- We already have the required machinery. An SPN can easily accept external beliefs as described in Section 3.7 and moreover all marginals which need to be sent to the code graph can be computed in parallel as described in Section 3.6!

4.2.1 Architecture Details

We use a deep generalized convolutional SPN (DGCSPN) (see Section 3.4) in place of a PGM in our architecture. We experimented with other SPN architectures such as the random tensorized SPN (RAT-SPN) [43] and found that the DGCSPN admitted efficient parameter learning using gradient descent. For an input source $\mathbf{s}^n$ with $s_i \in \mathcal{S}$, we assign one indicator leaf with singular scope for each source symbol,

$$\psi_i(s) \triangleq \mathbb{1}_{\{s_i=s\}}. \tag{4.17}$$

Each sum and product node outputs S and P output channels respectively, with S typically equal to P (see Section 3.4 for more details). Common channel sizes are

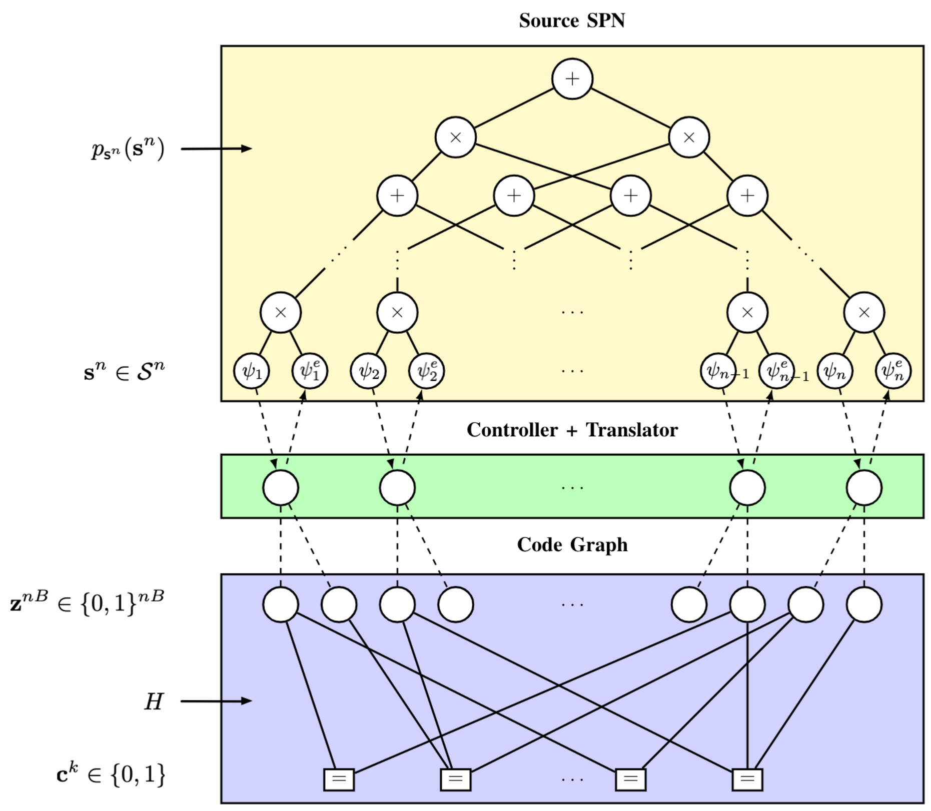


Figure 4-3: Combined SPN source-code decoder with arrows between the controller and SPN denoting the direction of message flow. The messages from the code graph are fed as external beliefs to the SPN. The messages from the source graph are accumulated from the gradients at the leaf nodes of the SPN and are then sent to the controller.

32, 64 and 128 depending on the dataset being used. All models are trained using the Adam optimizer [30] with a learning rate between 0.01 and 0.1 depending on the source data.

We replace the PGM in Figure 4-2 by an SPN as shown in Figure 4-3. Note the arrow directions between the controller and the SPN, specifically how the code graph messages are consumed by the SPN. The messages from the code graph are external beliefs to the SPN and hence they are multiplied with the leaf distributions as described in Section 3.7.

The messages sent from the code graph to the controller are over the alphabet $\{0,1\}$. Hence, we need to translate the message before providing it to the SPN as external beliefs. For a node u representing source symbol s_u, the external belief provided to the SPN is

$$\psi_u^e(s_u) \triangleq T_{2\to|\mathcal{S}|}(\mathbf{m}_{u\to u}^{c\to s})(s_u). \tag{4.18}$$

In order to compute the messages from the SPN to the code graph, we require the marginals of all the leaf nodes of the SPN. Recall that in order to compute the marginal of a node s_i, all other leaf distributions must be fixed to unity according to

Property 3.2, i.e., for all $j \neq i$, $\boldsymbol{\psi}_j = \mathbf{1}$. Since we want the marginals of all the nodes in parallel, we set $\boldsymbol{\psi}_i = \mathbf{1}$ for all $i \in \{1, \ldots, n\}$.

The marginals are accumulated at the leaf distribution nodes, hence the arrow points from the leaf distribution to the controller. In fact, the external beliefs though represented as nodes in Figure 4-3 should be interpreted as virtual connections. The marginals from the SPN are sent to the controller for translation. The code graph can now use the translated messages for code graph BP.

4.2.2 SPN Source-Code Belief Propagation

We call the decoding algorithm that runs BP with the SPN source model and the code graph as *SPN source-code belief propagation*. The entire algorithm is presented in Algorithm 2.

4.2.3 Benefits of Separation Architecture with SPNs

Having described the separation architecture, we are now ready to describe its benefits.

1. *Model-free coding*: The model-code separation architecture only requires the source sequence to have a bit-level representation in order to compress it. Moreover, by using an LDPC parity-check matrix, the code has low complexity and is agnostic of the source modality.

2. *Model-adaptive decoding*: Since the encoder has no knowledge of the source, the decoder requires knowledge about the source to accurately decode the source sequence, of which there can be many for a given codeword. The source data model can be swapped out easily in the decoder, for example, when our knowledge of the source changes. As the next chapter will demonstrate, SPNs can learn powerful statistical structure from large datasets and can losslessly compress natural images, a task which proved tough for PGMs due to their simple structure.

3. *Data model complexity*: By using an SPN as a source model, we are able to represent complex distributions using deep DAGs implemented efficiently through convolutions. Parameter learning is very easy with SPNs and hence we can use the same SPN architecture with different parameters to model a variety of complex distributions.

4. *Modularity*: The encoder, source graph, code graph and translator are all disjoint components. Hence, a change to one of these will not require significant changes to other components. For example, if we decide to use a code other than an LDPC code, all we need to do is to switch out the parity-check matrix in the encoder. The code graph can easily be updated to model different coding constraints by updating the factor nodes.

5. *Fast Decoding*: SPNs can compute all marginals in parallel. Even though an SPN has an underlying DAG much larger than that of a typical PGM, inference is extremely fast and most importantly tractable. On the other hand, BP on large PGMs is extremely slow. The SPN-based architecture can be efficiently implemented on fast inference engines such as GPUs with SPN-based architectures demonstrating decoding times under 0.05 seconds, much faster than a PGM-based architecture implemented on a GPU.

Algorithm 2: SPN Source-Code Belief Propagation.

```
Data: Source SPN $\Phi$, code graph $\mathcal{G} = (\mathcal{V}, \mathcal{F}, \mathcal{E})$, doped symbols $\mathbf{s}^n_{\mathcal{D}}$.
Result: Decoded source sequence $\hat{\mathbf{s}}^n$.

/* Initialize messages between source and code graphs.                */
$\mathbf{m}^{s\to c} \leftarrow \frac{1}{|\mathcal{S}|}\mathbf{1}$; $\mathbf{m}^{c\to s} \leftarrow \frac{1}{2}\mathbf{1}$;
/* Start iterative decoding.                                          */
$T \leftarrow 0$;
repeat
    $T \leftarrow T + 1$;
    /* Code Graph Belief-Propagation.                                 */
    1. Receive beliefs sent to the controller from the SPN and translate them
       for code graph BP,
           $T_{|\mathcal{S}|\to 2}(\mathbf{m}^{s\to c})$.
    2. Assimilate the beliefs from SPN as extra factor nodes and run BP on
    code graph.
    3. Return new beliefs to the controller,
           $\mathbf{m}^{c\to s}$;
    /* Parallel Marginal Computation in SPN.                          */
    1. Set all leaf node distributions to unity,
           $\boldsymbol{\psi}_u = \mathbf{1}$ for all $u$.
    2. Receive beliefs sent to the controller from the code graph, translate
       them for inclusion as external beliefs,
           $\boldsymbol{\psi}^e_u \triangleq T_{2\to|\mathcal{S}|}(\mathbf{m}^{c\to s})$.
    3. Run a forward pass of the SPN from leaves to root.
    4. Compute all marginals in parallel according to (3.16). Replace doped
    sites with indicator distributions. Send to the controller as new messages,
           $\mathbf{m}^{s\to c}$.
    /* Compute unnormalized probabilities over S                      */
           $\boldsymbol{b}_u \leftarrow \mathbf{m}^{s\to c}_{u\to u} t_{2\to|\mathcal{S}|}(\mathbf{m}^{c\to s}_{u\to u})$ for all $u$;
until unnormalized probabilities converge
/* Get decoded source sequence                                        */
           $\hat{s}_u = \arg\max_{s\in\mathcal{S}} b_u(s)$.
return $\hat{\mathbf{s}}^n$.
```

Chapter 5

Lossless Image Compression

In this chapter we apply our proposed SPN-based model-code separation architecture of Chapter 4 to compress binary and grayscale images. We call the class of systems that use a PGM-based decoder `PGM-SEP` and our proposed class of systems that use an SPN-based decoder `SPN-SEP`. Our experiments show that `SPN-SEP` does an excellent job compressing images and can match the decoding rate of `PGM-SEP` for sources where the true PGM is known. On datasets where the underlying PGM is unknown, we show that `SPN-SEP` outperforms other baseline universal and data-specific compressors.

5.1 Belief Propagation Decoding Threshold

We seek to characterize the utilizable rate of a code in the model-code separation architecture. Let H be a parity-check matrix of size $k \times n$ having Λ_i columns of weight i and P_j rows of weight j. Associated with this matrix are two *degree distributions* encoded by the generating functions,

$$\Lambda(x) \triangleq \sum_i \Lambda_i x^i, \tag{5.1a}$$

$$P(x) \triangleq \sum_j P_j x^j. \tag{5.1b}$$

We can also define the *normalized edge distributions*,

$$\lambda(x) \triangleq \frac{\Lambda'(x)}{\Lambda'(1)}, \tag{5.2a}$$

$$\rho(x) \triangleq \frac{P'(x)}{P'(1)}. \tag{5.2b}$$

The LDPC code that we use in our architecture has degree distributions $\Lambda(x) = x^3$ and $P(x) = (n - \lfloor n \operatorname{frac}(\overline{\rho}) \rfloor) x^{\lfloor \overline{\rho} \rfloor} + \lfloor n \operatorname{frac}(\overline{\rho}) \rfloor x^{\lfloor \overline{\rho} \rfloor}$, where $\overline{\rho}$ is the average sum of the row weights. Using the latter quantities we now define the *BP decoding threshold* as in [26, Chapter 5] to characterize the decoding capabilities of the chosen code.

Definition 5.1. Given a code with $\lambda(x)$ and $\rho(x)$, let $f(\epsilon, x) = \epsilon\lambda(1-\rho(1-x))$. The *BP decoding threshold*,

$$\epsilon^{\text{BP}} \triangleq \sup\{\epsilon : f(\epsilon, x) - x < 0, \forall x \in (0, 1]\}, \tag{5.3}$$

is the largest fraction of symbols that can be correctly decoding by BP with this code.

For more details about the decoding threshold and its relation to other channel thresholds please refer to [47, Chapter 3] and [26, Chapter 5].

5.2 Experimental Setup

In this chapter we use our proposed model to compress binary and grayscale images. To do so, we train an SPN on a dataset of source sequences and use the trained model for SPN source-code belief propagation. To compute the average compression rate for each parameter value of the source, we use a test set size of 1000 samples. This is a 50× increment in test set size compared to [26] to ensure that we report the most accurate compression rates.

We implement `SPN-SEP` in PyTorch using tensor operations[1]. The source SPNs are trained on 1-2 × NVIDIA RTX 3090 GPUs. To make fair comparisons against `PGM-SEP`, whenever the PGM representation of the source is known, we use our own GPU-based implementation of `PGM-SEP`. We summarize the various models that we use in our comparisons below.

- `SPN-SEP-prot`: An instance of the proposed model-code separation architecture that uses an SPN to model the source. An off-the-shelf library of binary LDPC codes is used. The doping rate is fixed to $r_{\text{dope}} = 0.04$ across all experiments. The total rate of the system is $r_{\text{total}} = r^*_{\text{code}} + r_{\text{dope}}$ where, r^*_{code} is the minimum coding rate for which decoding convergences within 100 iterations to the correct result.

- `SPN-SEP-thresh`: The rate returned by `SPN-SEP-prot` does not accurately describe the total utilizable rate of the code. For each minimal rate LDPC code found, we can associate with it a *BP decoding threshold*, (see Section 5.1) ϵ^{BP}, which is less than r^*_{code}. The gap between r^*_{code} and ϵ^{BP} is not due to the chosen architecture but is associated with the decoding performance of the LDPC code. For example, there might be a certain parity-check matrix that results in faster decoding than the one currently chosen. The *total threshold rate* $r^*_{\text{total}} = \epsilon^{\text{BP}} + r_{\text{dope}}$ is denoted by `SPN-SEP-thresh` and can be interpreted as the expected total rate with the best code chosen.

- `PGM-SEP-prot`: An instance of the baseline model-code separation architecture proposed by Huang and Wornell [27, 26] that uses a PGM to model the source.

[1]Our implementation is available at https://github.com/tkj516/deepgen_compress

- `PGM-SEP-thresh`: An instance of the baseline model-code separation architecture that uses a PGM to model the source where the expected total rate is reported for the best code.

- `GZIP`[2] : A universal compressor that uses Lempel-Ziv coding [59] based on dictionary learning from a bitstream. The source sequence is flattened into a binary bitstream before passing it to the compressor. The output length is the compressed file size minus the compressed file size of empty data to account for headers.

- `BZ2`[3]: A universal compressor that uses the Burrows-Wheeler [4] algorithm and Huffman coding [28]. The source sequence is flattened into a binary bitstream before passing it to the compressor, as is done for `GZIP`.

- `JBIG2`[4]: This state-of-the-art bi-level image compressor [39] based on 2D context dictionaries. We operate the encoder in lossless mode. The output length is the compressed file size minus the compressed file size of a 1-bit image to account for headers.

5.3 Compressing Binary Sources

We now report experimental results for the compression of binary sources.

5.3.1 Binary Ising Model

The homogenous 2D Ising model **Ising**(p, q) defined over an $n = h \times w$ grid of binary random variables is defined by the distribution

$$p_{\mathbf{s}^n}(\mathbf{s}^n) = p_{\mathbf{s}_\mathcal{V}}(\mathbf{s}_\mathcal{V}) = \frac{1}{Z} \prod_{i \in \mathcal{V}} \phi(s_i) \prod_{(i,j) \in \mathcal{E}} \psi(s_i, s_j), \tag{5.4}$$

where $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ is an undirected graph representing the distribution as shown in Figure 5-1 and

$$\phi(s_i) \triangleq p^{s_i}(1-p)^{(1-s_i)}, \tag{5.5a}$$

$$\psi(s_i, s_j) \triangleq q^{\mathbb{1}\{s_i = s_j\}}(1-q)^{1-\mathbb{1}\{s_i = s_j\}}. \tag{5.5b}$$

The source graph has singleton node potentials and pairwise edge potentials. The singleton potentials encode priors beliefs about the value of a node. A value closer to one represents a prior belief that the value of the node is more likely to be one. The value of the edge potentials quantify the amount of correlation between neighboring

[2]`GZIP` is available as the module `gzip` in Python.
[3]`BZ2` is available as the module `bz2` in Python.
[4]`JBIG2` is found at https://github.com/agl/jbig2enc

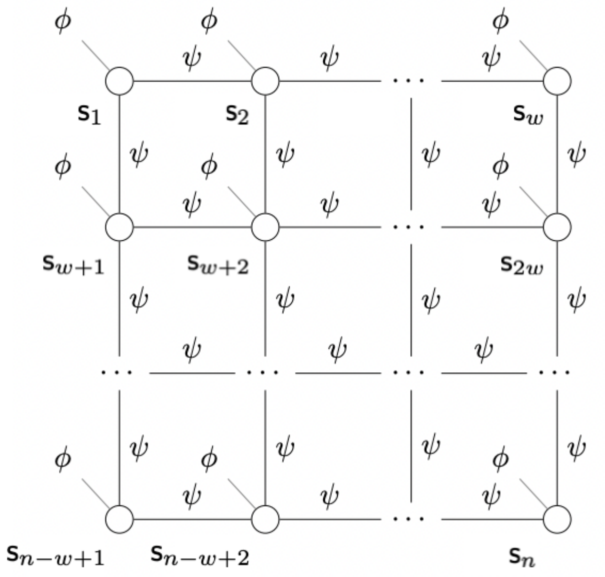


Figure 5-1: Binary Ising model $\textbf{Ising}(p, q)$.

nodes. A higher value of q conveys that neighboring nodes have an affinity to take on similar values and vice versa.

We are interested in *symmetric Ising models* where $p = 1/2$, i.e., a node has an equally likely chance of taking on the value zero or one. We can generate samples from this Ising model by the Gibbs sampling procedure (see Section 2.2.3.2). We use an extremely fast and optimized variant of the Gibbs sampler called the *chromatic Gibbs sampler* [21] that we implement in Python. At around $q = \sqrt{2}/2$, there is a phase transition and we start to observe long-range correlations as seen in Figures 5-2d and 5-2e .

Using $p = 1/2$, the distribution of the Ising model simplifies as

$$\begin{aligned} p_{\mathbf{s}_{\mathcal{V}}}(\mathbf{s}_{\mathcal{V}}) &= \frac{2^{-n}}{Z} \prod_{(i,j)\in\mathcal{E}} \psi(s_i, s_j) \\ &= \frac{2^{-n}}{Z} q^{\#\mathrm{SE}(\mathbf{s}_{\mathcal{V}})} (1-q)^{\#\mathrm{DE}(\mathbf{s}_{\mathcal{V}})} \\ &\propto \exp\left\{ \#\mathrm{DE}(\mathbf{s}_{\mathcal{V}}) \log\left(\frac{1-q}{q}\right)\right\} \\ &= \exp\left\{-2\theta\#\mathrm{DE}(\mathbf{s}_{\mathcal{V}})\right\} \\ &\triangleq \pi(\mathbf{s}_{\mathcal{V}}; \theta), \end{aligned}$$

where $\theta = \tanh^{-1}(2q-1)$ and $\#\mathrm{DE}(\mathbf{s}_{\mathcal{V}})$ is the number of edges in the graph that connect two nodes with different values.

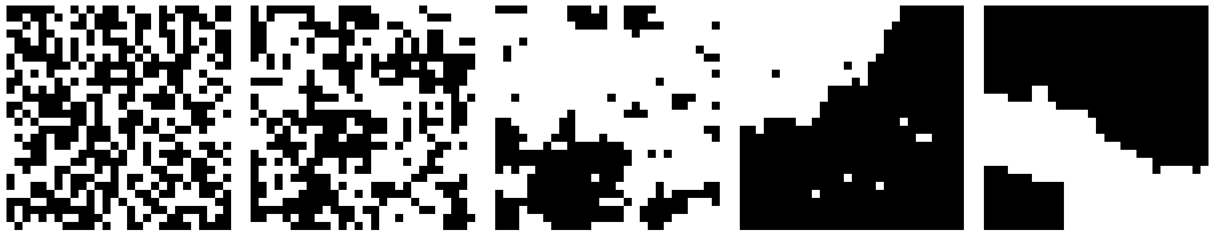


Figure 5-2: 28×28 Gibbs samples images from $\mathbf{Ising}(1/2, q)$ for different values of q.

The *finite entropy rate* of the model can be computed as

$$\begin{aligned} H_{h,w} &\triangleq \frac{1}{N}\mathbb{H}(\mathbf{s}_{\mathcal{V}}) \\ &= -\frac{1}{N}\sum_{j=1}^{N} p_{\mathbf{s}_{\mathcal{V}}}(\mathbf{s}_{\mathcal{V}}^{(j)}) \times \{\log \pi(\mathbf{s}_{\mathcal{V}}^{(j)};\theta) - \log Z\} \\ &= -\theta\frac{\mathbb{E}\{-2\#\mathrm{DE}(\mathbf{s}_{\mathcal{V}})\}}{N} + \frac{\log Z}{N}, \end{aligned}$$

where we have averaged over N samples from the distribution. We also report the asymptotic entropy rate (2.2) which can be obtained by taking $h, w \to \infty$.

5.3.1.1 Experiments

In our experiments we used $h = w = 28$. We collected 200,000 samples from the $\mathbf{Ising}(1/2, q)$ model using a chromatic Gibbs sampler, for each value of the parameter $q = 0.50 + 0.05t$ where $t \in \{0, \dots, 9\}$. Since we know the factorization of the distribution, the baseline systems `PGM-SEP-prot` and `PGM-SEP-thresh` are fully specified and can be used for comparisons.

To use our proposed system, we trained a separate DGCSPN (see Section 3.4) for each value of the parameter q. We used the same architecture and hyperparameters for all SPNs. Each sum and product node uses the same the number of output channels, $S = P = 32$. All indicator leaves have an alphabet size of 2 . All models were trained using the Adam optimizer with a learning rate of 0.01. We used a 70-20-10 train-validation-test split of the data and trained the model on a single NVIDIA 3090 GPU for 100 epochs. In most cases, *early stopping* was activated and the model was done training within 30 epochs.

We used a fixed doping rate of $r_{\text{dope}} = 0.04$ across all experiments.

Fact 5.1. *Early stopping* is a technique used in neural network training where the learning algorithm decides to terminate training when the decrease in validation loss is not significant anymore.

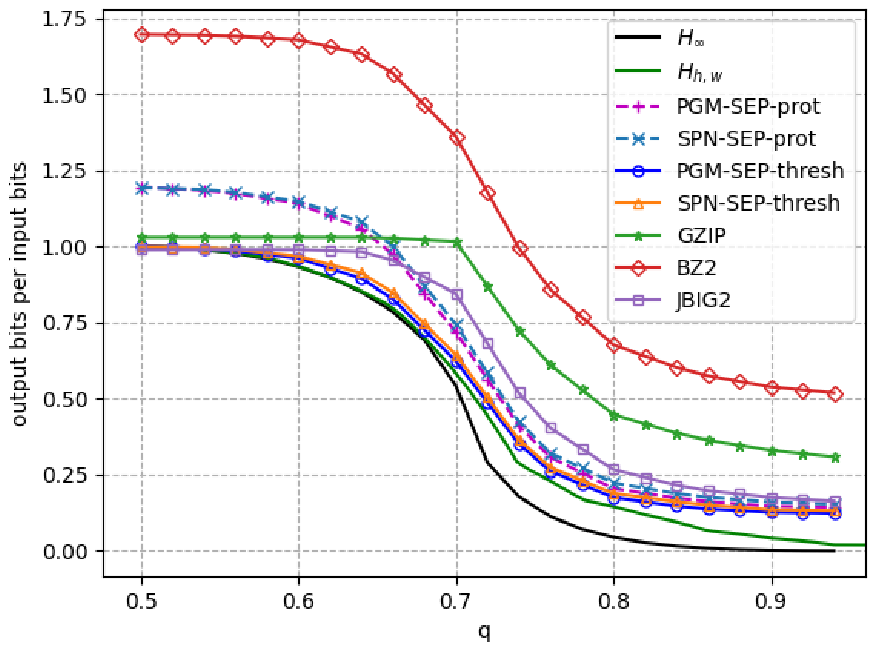


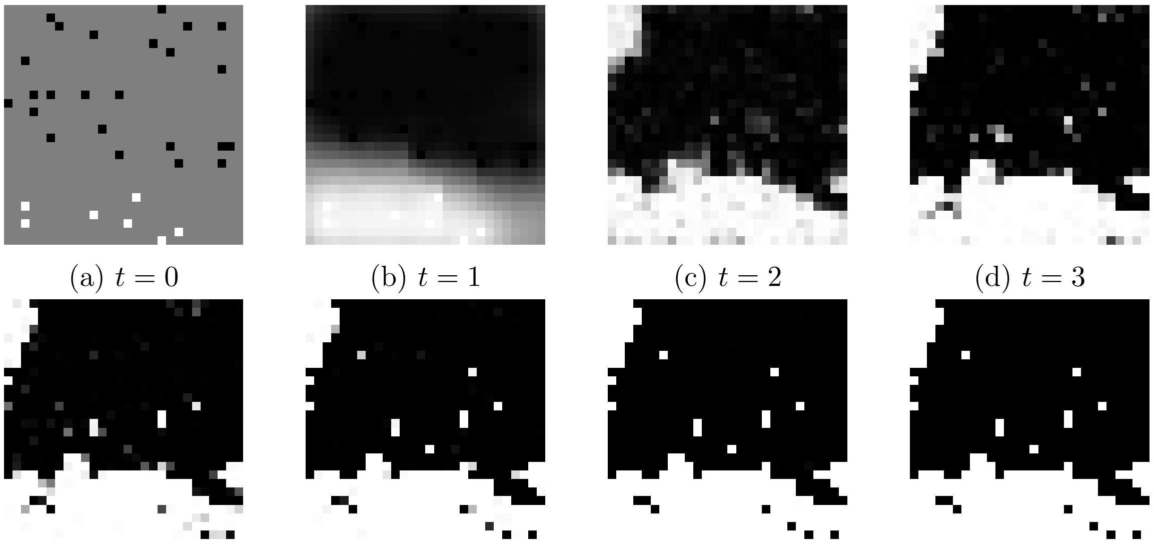
Figure 5-3: Compression performance for the **Ising**(p, q) source family for $p = 1/2$ and different values of q. The dimensions of the image were fixed to $h = w = 28$ and $r_{\text{code}} = 0.04$.

5.3.1.2 Results

The compression rate averaged over 1000 samples for each value of q using different compression architectures is shown in Figure 5-3. Both `SPN-SEP-prot` and `SPN-SEP-thresh` achieved nearly identical performance to the respective baseline systems `PGM-SEP-prot` and `PGM-SEP-thresh`. Moreover, `SPN-SEP-thresh` outperformed the state-of-the-art bi-level compressor `JBIG2` across all rates too. This demonstrates the powerful generative capabilities of SPNs and its ability to learn strong statistical structure from a large dataset of source samples. `SPN-SEP-prot` and `SPN-SEP-thresh` also outperformed the universal compressors `GZIP` and `BZ2` by a large margin, with the latter two systems demonstrating favorable results only in the low entropy regime (large q).

The average decoding runtime of `SPN-SEP` on a single GPU was 0.04 seconds in comparison to `PGM-SEP` which was around 0.1 seconds. While this may seem negligible, it is actually quite remarkable given that the SPN has thousands of parameters to specify the underlying DAG in comparison to the much fewer parameters required to specify the Ising model PGM.

Notice that the data was compressed better once we cross the phase transition value of q. This shows that better compression is achievable for sources with repetitive structures, i.e., lower entropy. Both `PGM-SEP-thresh` and `SPN-SEP-thresh` achieve rates close to the finite entropy of the model. `JBIG2` is consistently outperformed by


(a) $t = 0$ (b) $t = 1$ (c) $t = 2$ (d) $t = 3$

(e) $t = 4$ (f) $t = 5$ (g) $t = 6$ (h) $t = 7$

Figure 5-4: Decoding a sample from **Ising**$(1/2,\, 3/4)$ using `SPN-SEP`. Intensities α_i range from black=0 to white=1. Intensities denote the value of the estimated marginal $\hat{p}_{\mathsf{s}_i}(1) \triangleq \alpha_i$. We set $r_{\text{code}} = 0.5$ and $r_{\text{dope}} = 0.04$.

`SPN-SEP-thresh`, with similar performance only in the extremely low and extremely high end of the entropy spectrum.

The entire decoding process on a sample drawn from **Ising**$(1/2, 3/4)$ using `SPN-SEP` is shown in Figure 5-4. The SPN source-code BP algorithm is seeded with the initial doping bits at time $t = 0$. In just a single iteration, the algorithm starts to infer the general structure of the source from the doping bits. By the third iteration, the algorithm has decoded most of the long-range correlations within the source. The last three iterations correct decoding errors at edges within the image. It is difficult to decode the small scattered black and white patterns within the regions with large correlations with just the SPN. This is where the code graph comes into play to enforce the parity-check constraints and steer the decoder towards the correct solution.

We observed that when using a total decoding rate close to the finite entropy, both `SPN-SEP` and `PGM-SEP` took around the same number of iterations. At rates higher than the most optimal rate, `SPN-SEP` was able to recover the image in fewer iterations than `PGM-SEP`. Comparative decoding visuals using `PGM-SEP` can be found in Appendix A (Supplementary Images).

5.3.2 Binary MNIST

We transition towards natural datasets by first compressing binary MNIST images. The MNIST dataset [35] consists of 60,000 grayscale images of handwritten digits between 0 and 9, examples of which are shown in Figure 5-5. All the images have a shape of 28×28. Unlike the 2D Ising model, we don't know the statistical structure of MNIST digits and hence the PGM structure should be learned in order to use

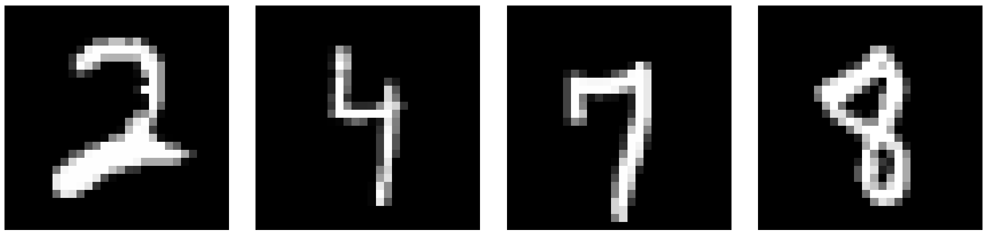


Figure 5-5: Sample images from the grayscale MNIST dataset. Images are binarized by using a mid-intensity threshold.

PGM-SEP. Structure learning is hard for PGMs when using large datasets. However, SPN-SEP can be easily used for compression since learning an SPN is extremely fast.

We used the same DGCSPN architecture from Section 5.3.1.1 to learn the source distribution for binary MNIST. Since MNIST images are grayscale, we thresholded the images at the mid-intensity level. We continued to use a 70-20-10 train test validation split for training. The DGCSPN trains extremely fast with early stopping activated in epoch 30. We used a single GPU and a batch size of 128 to train the model.

Dataset/Compressor	BZ2	GZIP	JBIG2	SPN-SEP-prot	SPN-SEP-thresh
Binary MNIST	0.78	0.59	0.34	0.24	0.20

Table 5.1: Average compression rate of the binary MNIST images using different compression strategies.

We compared our architecture against the universal compressors GZIP and BZ2, and the bi-level compressor JBIG2. Since the image is already binary, we flatten the image and pass it to the universal compressors for encoding. As shown in Table 5.1, both SPN-SEP-prot and SPN-SEP-thresh were able to achieve much better compression rates than BZ2, GZIP and JBIG2. This shows that the SPN is not only able able to learn strong statistical structure from the data but also an efficient structure. This is conveyed by the nearly 1.7× gain in compression rate by SPN-SEP over JBIG2. The difference in rates between SPN-SEP-prot and SPN-SEP-thresh indicates that via code optimization it is possible to achieve a considerable increase in compression.

Figure 5-6 shows the decoding steps involved in decoding the digit 7. Within two iterations the decoder was able to leverage the structure learned by the SPN to decode the digit as a 7. Notice that the decoded image at $t = 2$, is a perceptually accurate image of the digit 7. The code graph came into play towards the end by enforcing the parity-check constraints to produce the horizontal line. More decoding visuals can be found in Appendix A.

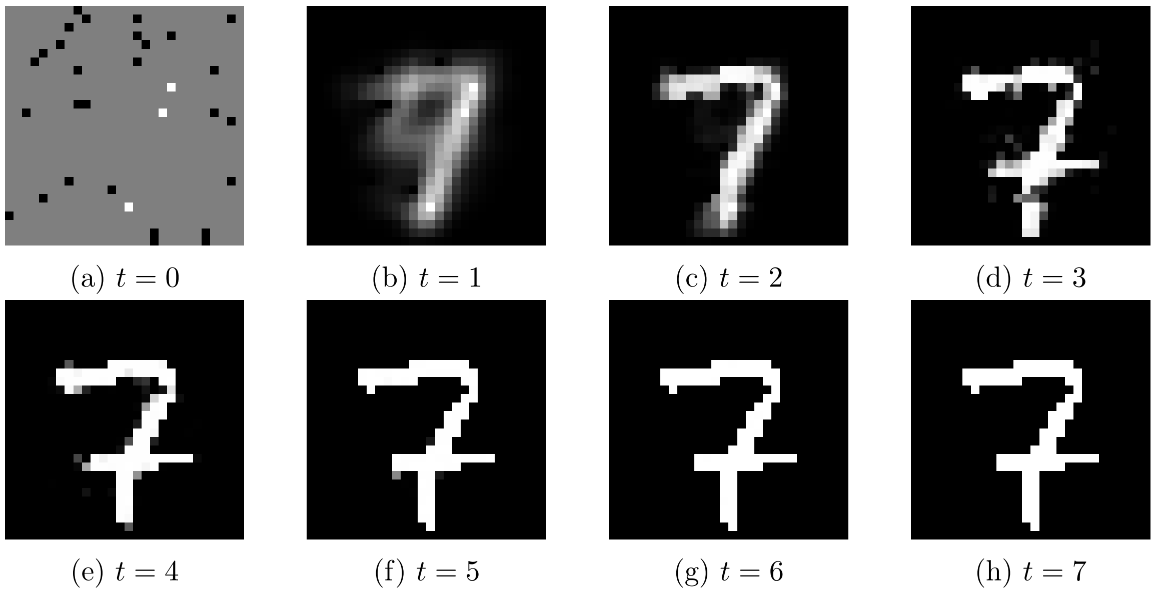

(a) $t = 0$ (b) $t = 1$ (c) $t = 2$ (d) $t = 3$ (e) $t = 4$ (f) $t = 5$ (g) $t = 6$ (h) $t = 7$

Figure 5-6: Decoding the digit 7 from the binary MNIST dataset using `SPN-SEP`. Intensities α_i range from black=0 to white=1. Intensities denote the value of the estimated marginal $\hat{p}_{\mathsf{s}_i}(1) \triangleq \alpha_i$. We set $r_{\text{code}} = 0.27$ and $r_{\text{dope}} = 0.04$

5.4 Compressing Grayscale Sources

Having compressed binary source we now use our architecture to compress grayscale sources.

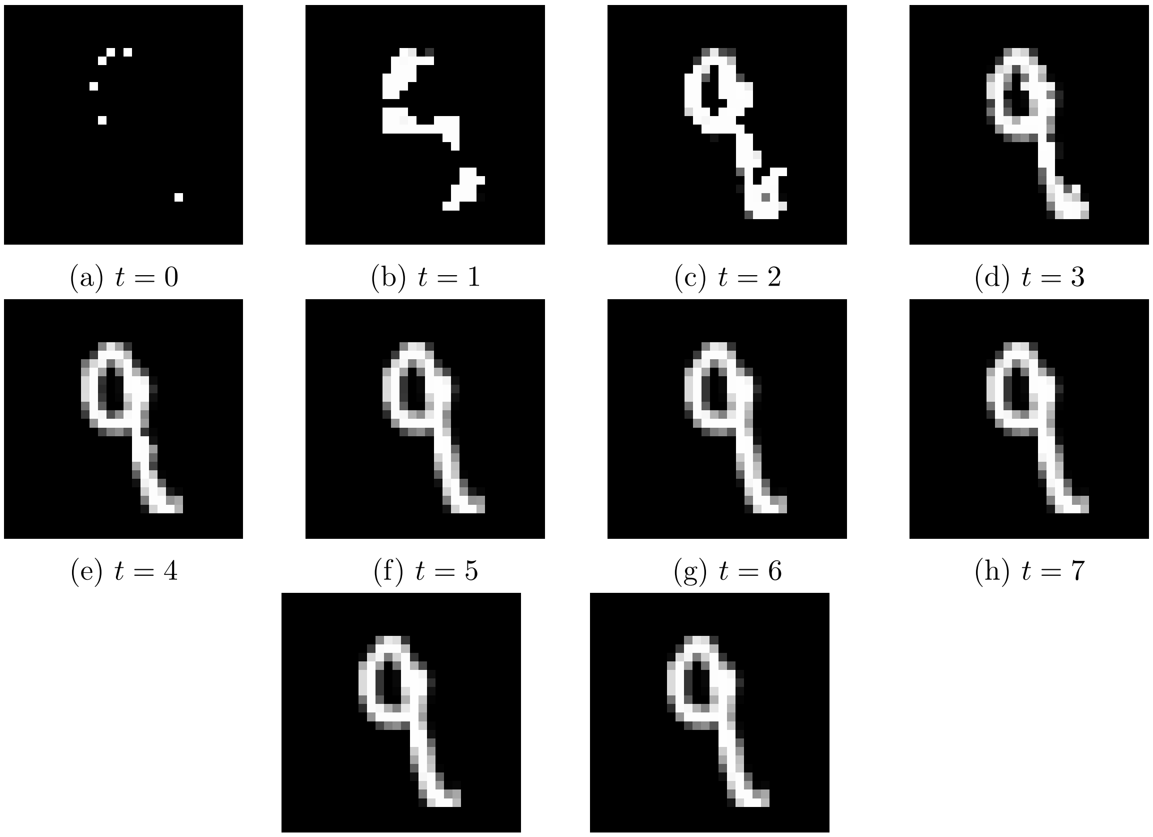
5.4.1 Grayscale MNIST

We now compress the MNIST dataset without binarization. We trained a DGCSPN with a slightly different architecture. The leaf distributions are now indicators with an alphabet size of 256. The sum nodes and product nodes have $S = P = 64$ output channels. The model was trained on 2 × NVIDIA RTX 3090 GPUs with a batch size of 128. We decreased the learning rate to 0.01 since the model is much larger than the one used in Section 5.3.2.

We ran multiple trials to find the best doping rate and we concluded that $r_{\text{dope}} = 0.04$ still works well for decoding. To speed up decoding we parallelized the SPN source-code decoding algorithm to use two GPUs. In practice we could continue to use one GPU but it adds extra strain on the GPU memory. Despite the model being almost twice as large as before, the decoding algorithm still runs under 0.1 seconds for decoding grayscale images.

Dataset/Compressor	`BZ2`	`GZIP`	`SPN-SEP-prot`	`SPN-SEP-thresh`
MNIST	0.27	0.22	0.23	0.20

Table 5.2: Average compression rate of grayscale MNIST images using different compression strategies.



(a) $t = 0$ (b) $t = 1$ (c) $t = 2$ (d) $t = 3$

(e) $t = 4$ (f) $t = 5$ (g) $t = 6$ (h) $t = 7$

(i) $t = 8$ (j) $t = 9$

Figure 5-7: Decoding the digit 9 from the MNIST dataset using `SPN-SEP`. Intensities α_i range from black=0 to white=1. Intensities denote the value of the max marginal $\alpha_i \triangleq \max_{s_i} \hat{p}_{\mathsf{s}_i}(s_i)$. We set $r_{\text{code}} = 0.23$ and $r_{\text{dope}} = 0.04$

We compared our architecture against `GZIP` and `BZ2`. Since the images are not binary anymore we get the binary representation for each pixel and then flatten the input image into a single bitstream. As shown in Table 5.2, we were able to achieve compression rates better than `BZ2`. `GZIP` did a good job compressing MNIST images and our `SPN-SEP-prot` architecture had decoding performance on par with `GZIP`. If we optimize for the best code it is possible to achieve even lower compression rates as indicated by a compression rate of 0.20 in the last column of Table 5.2.

We would like to comment that `GZIP` and `BZ2` are not necessarily compressing the inputs any better than the binary MNIST images. The large decrease in bitrate is a result of the 8-bit encoding of pixels which leads to long runs of 0s and 1s that can be easily compressed. While we could have indeed converted our binary MNIST images to 8-bits per pixel representations, doing so would also increase the size of the transmitted codeword which makes the system sub-optimal. Visuals of the decoding process are shown in Figures 5-7 with additional samples included in Appendix A.

5.4.2 Fashion MNIST

In 2017, the Fashion MNIST dataset [62] was released as a drop-in replacement for the MNIST dataset for the purposes of machine learning benchmarking. The dataset consists of grayscale images of clothing articles from 10 classes. Example images from the dataset are shown in Figure 5-10. We employed the same DGCSPN architecture from Section 5.4.1. The model was trained with a learning rate of 0.1 on 2 × NVIDIA RTX 3090 GPUs. The model trained within an hour. We continued to use a doping rate of 0.04.

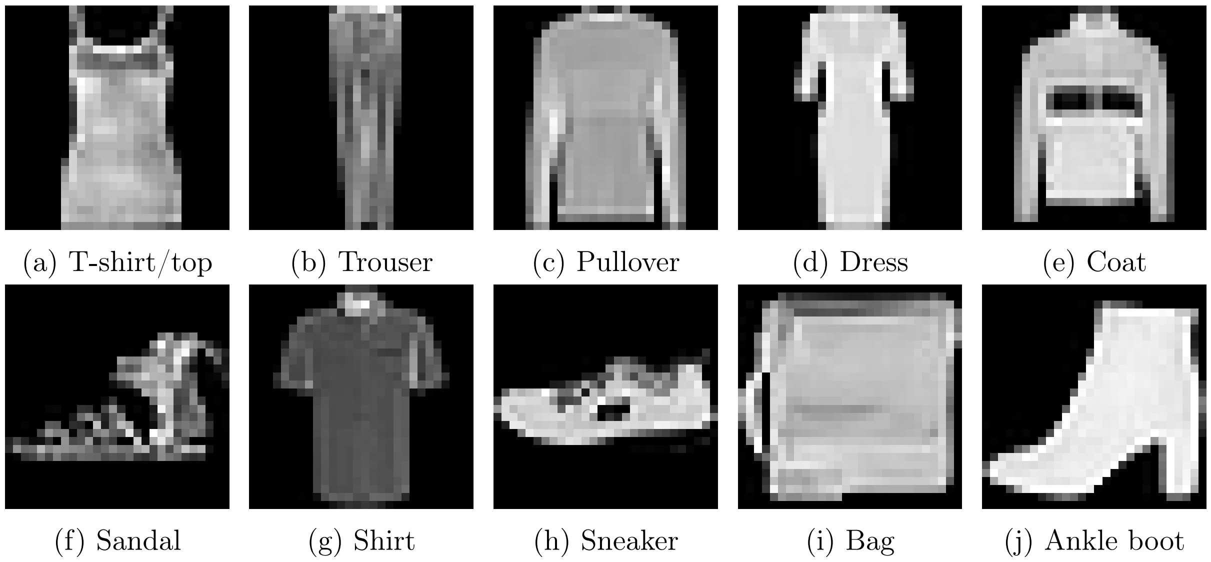


Figure 5-8: Samples from the Fashion MNIST dataset. Class labels are displayed below each image.

Dataset/Compressor	`BZ2`	`GZIP`	`SPN-SEP-prot`	`SPN-SEP-thresh`
Fashion MNIST	0.66	0.57	0.53	0.46

Table 5.3: Average compression rate of Fashion MNIST images using different compression strategies.

As shown in Table 5.3, we were able to achieve compression rates better than `BZ2` and `GZIP`. Figure 5-9 shows the steps involved in decoding a shoe. Notice how the decoder was able to infer a plausible shape in the very first iteration of decoding. The overall structure was inferred within the first five iterations. The remaining iterations served to reconstruct the correct intensity value for each pixel by ensuring constraint satisfaction via code graph BP. This shows that the SPN has efficiently learned the discerning features between different classes of the dataset even though it was trained as a generative model.

The articles of clothing have subtle intensity variations and different textures. Thus, to encode such variations the code requires more bits and hence the compression rate on Fashion MNIST images is almost twice that of MNIST images. Considerable compression can still be performed since the clothing articles have unique shapes.

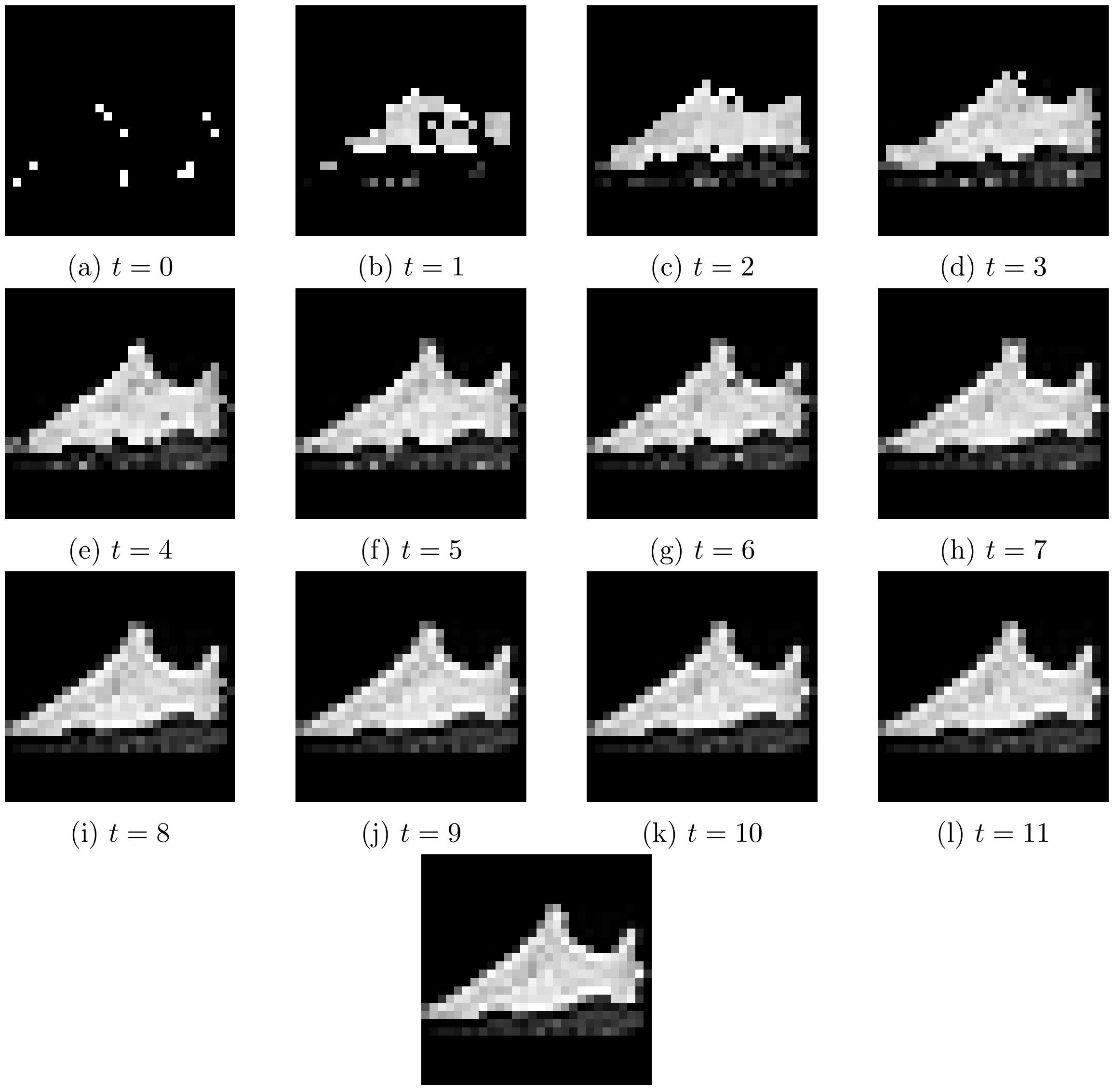
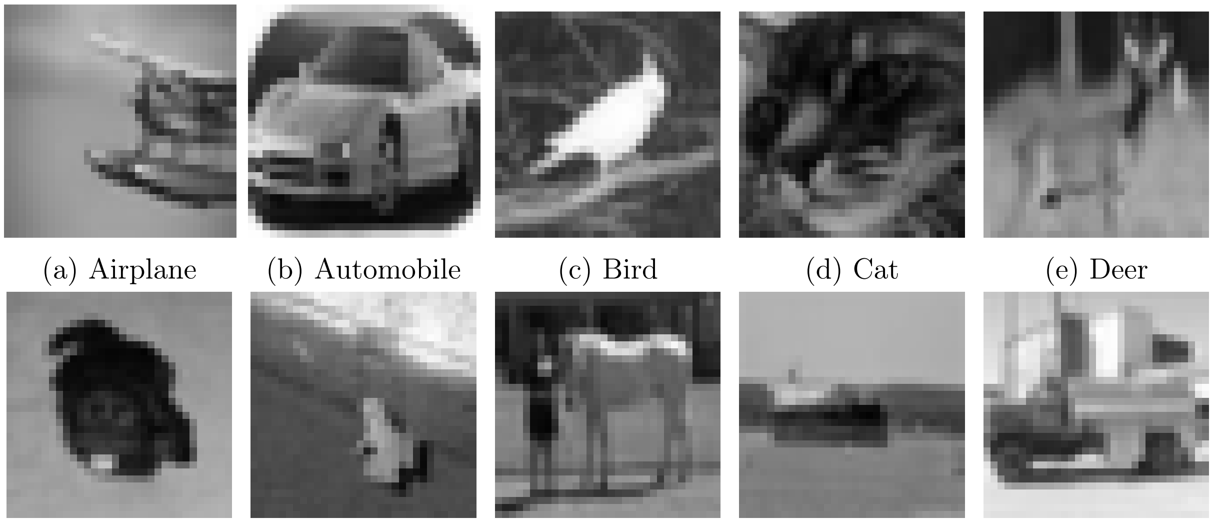


Figure 5-9: Decoding an image of a shoe from the Fashion MNIST dataset using `SPN-SEP`. Intensities α_i range from black=0 to white=1. Intensities denote the value of the estimated marginal $\hat{p}_{\mathsf{s}_i}(1) \triangleq \alpha_i$. We set $r_{\text{code}} = 0.45$ and $r_{\text{dope}} = 0.04$

5.4.3 CIFAR-10

We finally test our compression system on the CIFAR-10 dataset [32]. The dataset consists of 32 × 32 RGB images classified into 10 classes. There are 60,000 images with 6000 images per class. We convert the images to grayscale for the purposes of our experiments.



(a) Airplane (b) Automobile (c) Bird (d) Cat (e) Deer

(f) Dog (g) Frog (h) Horse (i) Ship (j) Truck

Figure 5-10: Samples from the CIFAR-10 dataset. Class labels are displayed below each image.

To determine a baseline that we would like to outperform, we first compressed 1000 test images using the universal compressors `GZIP` and `BZ2`. Not much compression was attained, with `GZIP` obtaining a compression rate of 0.93 and `BZ2` unable to compress the images at all as shown in Table 5.4. Thus, if we can attain better rates than the universal compressors, we might be able to conclude that the SPN is able to learn and exploit source structure for decoding.

Dataset/Compressor	`BZ2`	`GZIP`	`SPN-SEP-prot`	`SPN-SEP-thresh`
CIFAR-10	1.01	0.93	0.86	0.78

Table 5.4: Average compression rate of grayscale CIFAR-10 images using different compression strategies.

We trained a DGCSPN with 64 output channels at each sum and product node. Through several rounds of hyperparameter tuning we found that a learning rate of 0.01 and a batch size of 32 provided the best training results. As shown in Table 5.4, both `SPN-SEP-prot` and `SPN-SEP-thresh` beat the baseline universal compressors. This suggests that the SPN has learned a more efficient representation of the source structure than what was learned on-the-fly by `GZIP` and `BZ2`. This demonstrates that universal compressors could benefit from decoders that can incorporate various

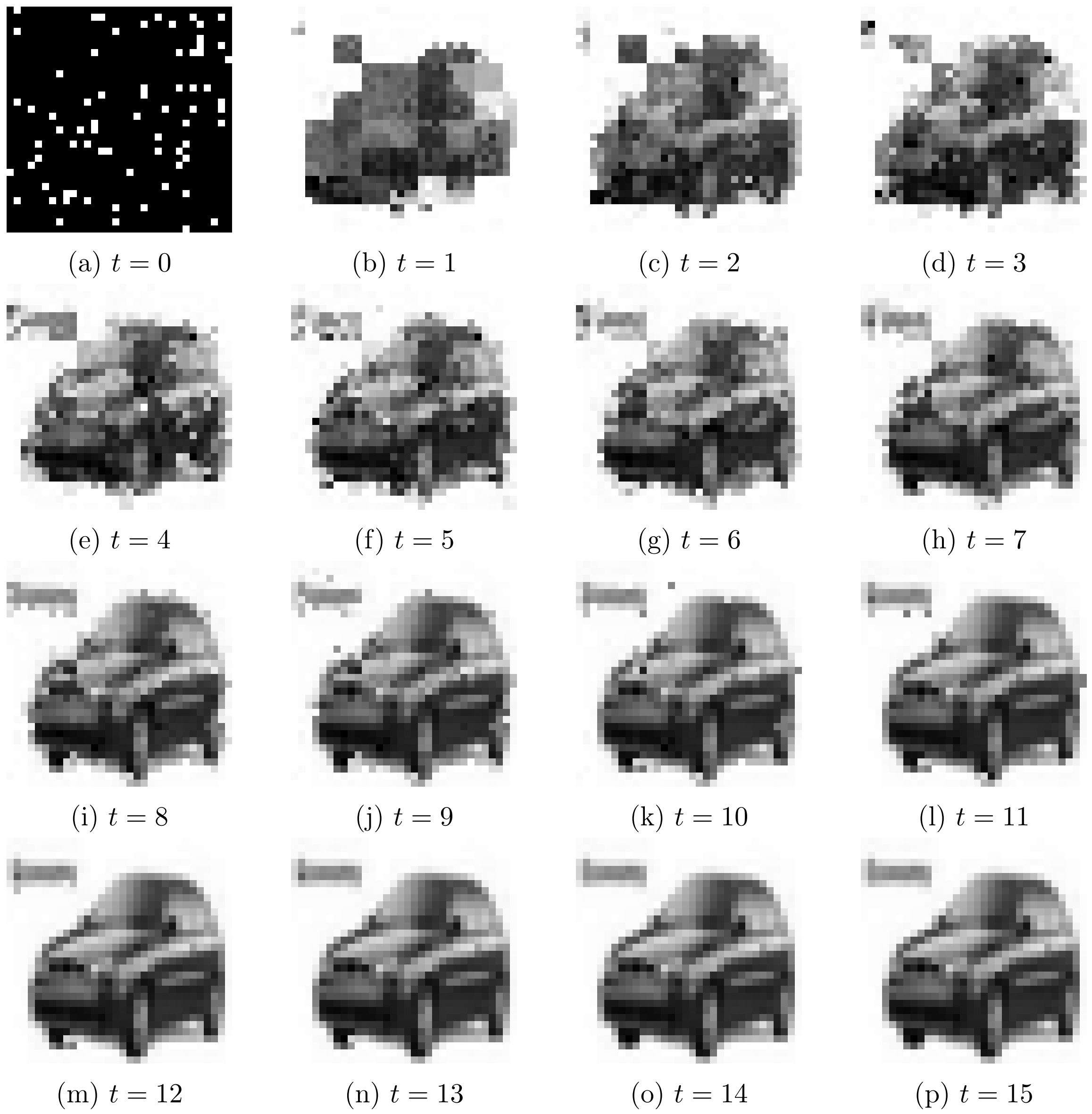


Figure 5-11: Decoding an image of a car from the CIFAR-10 dataset using `SPN-SEP`. Intensities α_i range from black=0 to white=1. Intensities denote the value of the estimated marginal $\hat{p}_{\mathsf{s}_i}(1) \triangleq \alpha_i$. We set $r_{\text{code}} = 0.85$ and $r_{\text{dope}} = 0.07$

different data models while leaving the encoder untouched. The large difference in rate between `SPN-SEP-prot` and `SPN-SEP-thresh` suggests that code optimization could play an important role in making the system more robust.

We continued to use $r_{\text{dope}} = 0.04$ for decoding. We found that increasing the doping rate did not lead to a significant reduction in coding rate r_{code}. This is a promising feature since we'd like to dope as few bits as necessary for practical applications.

Figures 5-11 shows the steps involved in decoding an image of a car. In general, at the best decoding rate, it took more than 20 iterations of SPN source-code BP to decode the image, with an average decoding time of 0.07 seconds. Notice how the reconstruction looks blocky in the beginning and then starts to resolve itself in later iterations. This suggests that the SPN source-code decoding process could be used for *progressive image reconstruction*.

Furthermore, note how the rectangular patch in the top left of the image was perfectly reconstructed. The last few iterations actually served to reconstruct this patch perfectly. Often we might not be interested in reconstructing perceptually irrelevant regions. Based on the nearly perfect reconstruction of the car in the first few iterations, this suggests that this architecture could also be used for lossy compression along the lines of [34] and [36].

5.5 Summary

The results from this chapter demonstrate the use of our proposed SPN-based model-code separation architecture, `SPN-SEP`, in decoding binary and grayscale images. Through various experiments we show that our architecture can be used not only to compress simple binary source such as Ising models with high compression rate, but also grayscale images with complex structure such as those from the CIFAR-10 dataset. The working of the SPN source-code BP algorithm is explained through numerous visual examples compiled in this chapter and Appendix A (Supplementary Images).

Chapter 6

Conclusion

6.1 Review

In this thesis, we propose a model-code separation architecture for lossless compression that uses a sum-product network (SPN) to model the source data. The architecture uses a model-free encoder based on LDPC codes and a model-adaptive decoder utilizing the SPN source-code belief propagation as described in Section 4.1.3. We build upon the architecture proposed by Huang and Wornell [27, 26] by using a powerful SPN instead of a PGM for modeling the data. The resulting architecture admits extremely fast lossless decoding on GPUs.

We develop the theory underlying our system in Chapters 2 and 3 before proceeding onto the system description in Chapter 4. Through experiments on different image datasets and comparisons against baselines models in Chapter 5 we demonstrate the effectiveness of our architecture in compressing binary and grayscale images.

6.2 Applications and Future Work

The results from previous chapters demonstrate that sum-product networks perform an excellent job at modeling the source distribution of signals. Moreover, lossless compression using the SPN source-code BP algorithm is fast, efficient and robust. We now comment on some additional applications of sum-product networks and the model-code separation architecture.

1. *Lossy Compression*: Lai [34] and Lee [36] showed that the PGM-based model-code separation architecture could be extended for the purposes of lossy image compression. In this extension, an extra quantizer module is introduced in the controller. The quantization of messages leads to loss in information (distortion) as measured by the mean-squared error reconstruction loss. By trading off between rate and distortion by varying the amount of quantization, a lossy compression architecture is realized. Our proposed SPN-based architecture can be extended using an SPN with Gaussian leaf distributions along similar lines for lossy compression.

2. *Autoencoders*: An autoencoder is a type of neural network that is used to learn compact representations of unlabeled data. Autoencoders are generative models that have two modules — the encoder and decoder. The encoder reduces the dimensionality of the input and the decoder reconstructs the input. The network is trained with reconstruction loss functions such as the mean-square error. Our proposed architecture can be used as a "true" autoencoder to perfectly reconstruct the input data. Given an input source sequence, a plausible autoencoder architecture could encode the source using an LDPC code and it could use the SPN source-code BP algorithm to decode the source. In addition to this we would require the autoencoder to be capable of sampling new data. This can be done as long we are able to sample codewords that correspond to source sequences with high probability. One idea is to use build on the LDPC code sampling method proposed in [65].

3. *Conditional SPNs and RGB Image Decoding*: In this thesis we extensively study binary and grayscale lossless image compression. The same methods when used on color images might not yield good results. The reasoning behind this hypothesis is that there is a strong correlation between the different color channels of an RGB image. In probability terms, given a channel of the image, the conditional distribution of another channel given the observed channel should be highly compressible. Image coding standards such as `PNG` and `JPEG` take advantage of this fact during compression. Recently there has been interest in conditional SPNs [54]. If conditional distributions can be learned efficiently then it is possible take advantage of these distributions during SPN source-code BP to compress color images.

4. *Compression for Lightweight Sensors*: If self-driving cars are indeed the future of travel, it will require numerous sensors to work together in harmony. It is vital that these sensors transmit sensory inputs to the central controller quickly and efficiently. There has been recent interest in the use of lightweight sensors to reconstruct complex scenes, an example of which is the *oversampled binary image sensor* [63]. This sensor quantizes the photon count at each pixel using a binary quantizer. Since the sensor is taking binary measurements, the imaging devices using many sensors in combination to reconstruct a complex scene from multiple measurements. Oversampled binary image sensors have been shown to produce images with large resolutions and higher dynamic range. It might be possible to leverage our model-code separation architecture to effectively compress the sensor measurements for storage and transmission, which is one of the limiting factors of oversampled image sensor architectures. This idea can be extended to signals from other modalities too.

5. *Alternative Coding Mechanisms*: In this thesis we use LDPC codes to compress source sequences. It would be interesting to experiment with other codes and conduct a comprehensive study on the dependence of our architecture on the coding mechanism.

6. *Probabilistic Discriminator in GANs*: Thus far, GANs have not been widely used for compressing images due to their inability to trade-off distortion, in the form of the discriminator, with rate. The idea behind this thesis was inspired by experiments on perceptual decoding of images using dither-quantized binary images (see Section 1.2). The results of this thesis show that an SPN can be used as a probabilistic decoder for compression. Hence, a possible next step is to experiment with SPNs as discriminators in GANs and leverage an SPN's probabilistic interpretation to use GANs for lossy compression.

7. *Modular Neural Codecs:* Current state-of-the-art neural image codecs such as `L3C` [38] and `IDF` [25] are trained end-to-end (see Section 1.2.3). Hence, a change to the coding mechanism requires a complete change to the architecture. Our model-code separation architecture using SPNs could inspire future work in lightweight and modular design of neural codecs. `IDF` can achieve an impressive 42% compression rate on color CIFAR-10 images beating all other compression strategies. If SPNs can be made more powerful by combining techniques from normalizing flows, autoregressive models and variational autoencoders it might be possible to model even more complex structure than what they currently can.

6.3 Final Remarks

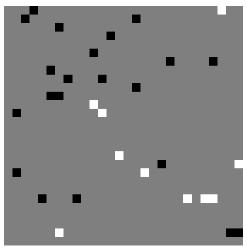
Statista estimates that 181 zettabytes of data will be consumed in the year 2025 [52], more than double of what is being consumed now. This demands an increase in storage capacity across all physical devices. We currently have smartphones with 1 TB of storage, something that we would not have foreseen 10 years back. However, there is a limit on the amount of storage that can be realized on devices. We ultimately need better compression algorithms to meet the ever increasing appetite for big data.

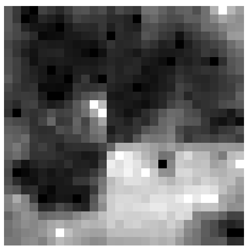
Each year we encounter new modalities of data that we would like to store with each new source of data having some inherent structure in it. It is not practical to establish a new compression standard for every source of data. While it is definitely a unique and rewarding opportunity to develop compression techniques for new sources of data, engineers and scientist should also address the twin questions of flexibility and modular design. Current systems still use joint model-code design in their architecture. As we have seen, such systems are often immutable, inflexible and non-reusable. This is especially important in neural codecs where it is infeasible to retrain complex deep architectures every time we encounter novel data.

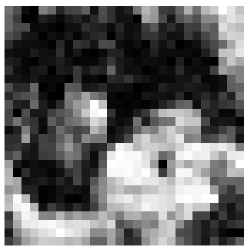
In this thesis we propose an architecture that starts to address some of these issues. We provide the basic tools and insights that can hopefully inspire the next generation of compression algorithms. We make use of modern AI and machine learning tools to develop a compression model rooted in information theory, source coding and statistical inference. As these fields advance, compression systems will undoubtedly too, ushering a new age of technology defined by modular and adaptive compression systems.

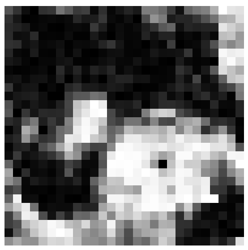
Appendix A

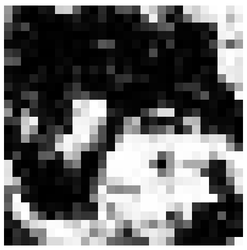
Supplementary Images

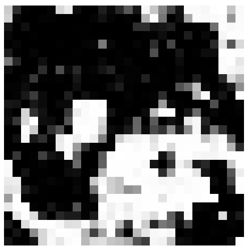

(a) $t = 0$

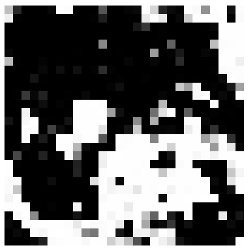

(b) $t = 2$

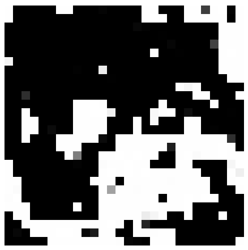

(c) $t = 4$


(d) $t = 6$


(e) $t = 8$


(f) $t = 10$


(g) $t = 12$


(h) $t = 14$

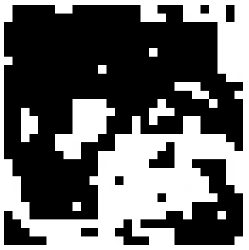
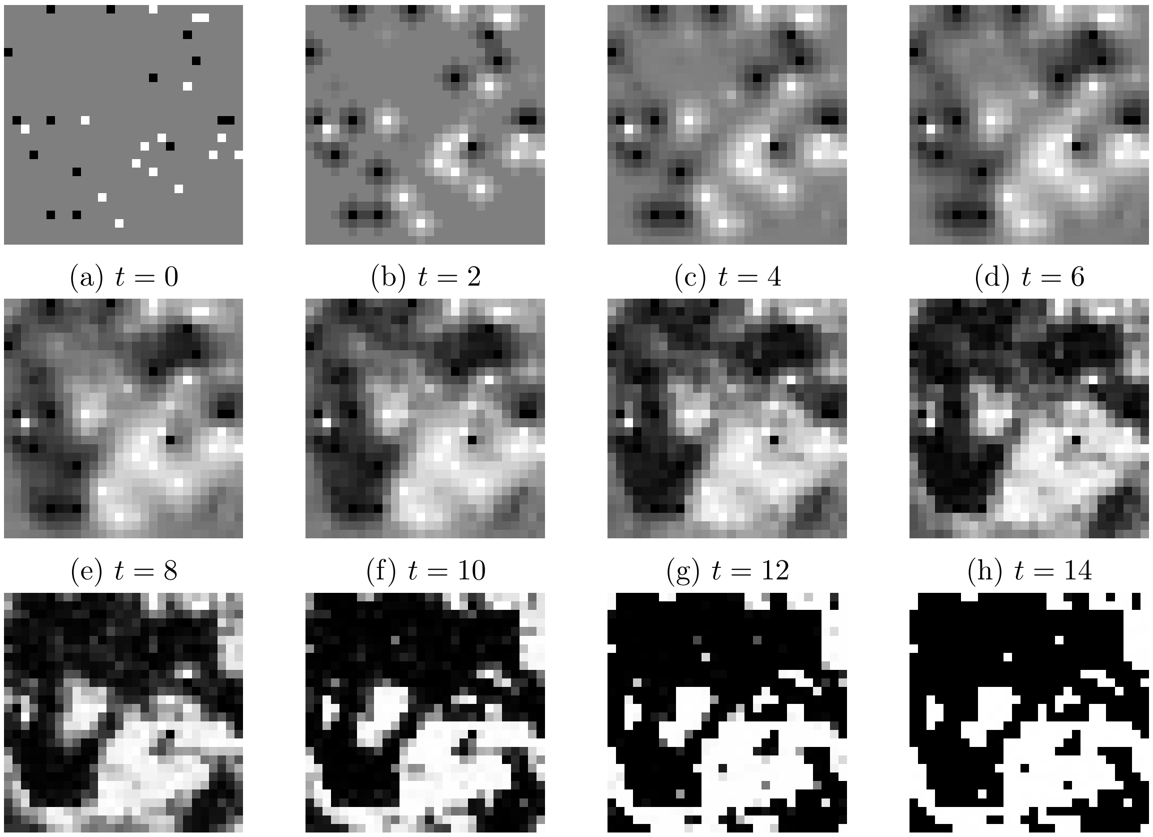

(i) $t = 4$

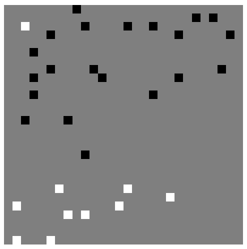
Figure A-1: Decoding a sample from **Ising**(1/2, 7/10) using `SPN-SEP`. Intensities α_i range from black=0 to white=1. Intensities denote the value of the estimated marginal $\hat{p}_{\mathsf{s}_i}(1) \triangleq \alpha_i$. We set $r_{\text{code}} = 0.70$ and $r_{\text{dope}} = 0.04$.

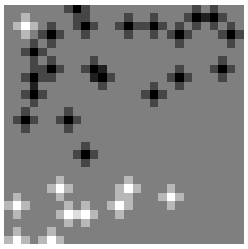

(a) $t = 0$ (b) $t = 2$ (c) $t = 4$ (d) $t = 6$

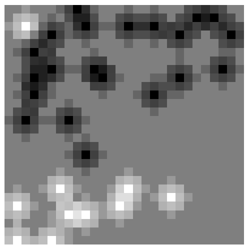
(e) $t = 8$ (f) $t = 10$ (g) $t = 12$ (h) $t = 14$

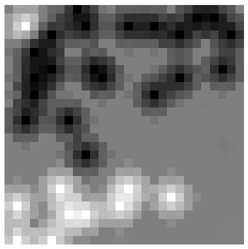
(i) $t = 16$ (j) $t = 18$ (k) $t = 20$ (l) $t = 22$

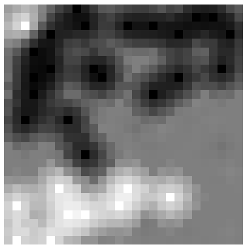
Figure A-2: Decoding the same sample from Figure A-1 using `PGM-SEP`. Intensities α_i range from black=0 to white=1. Intensities denote the value of the estimated marginal $\hat{p}_{\mathsf{s}_i}(1) \triangleq \alpha_i$. We set $r_{\text{code}} = 0.70$ and $r_{\text{dope}} = 0.04$.

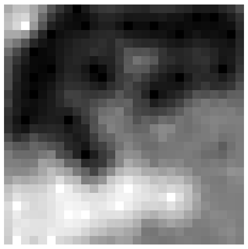

(a) $t = 0$

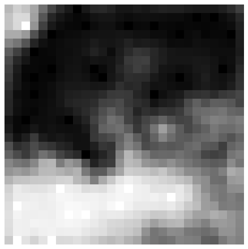

(b) $t = 1$

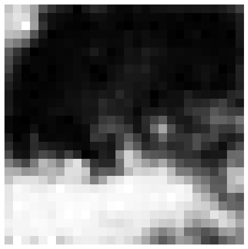

(c) $t = 2$

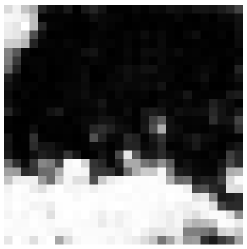

(d) $t = 3$


(e) $t = 4$


(f) $t = 5$


(g) $t = 6$

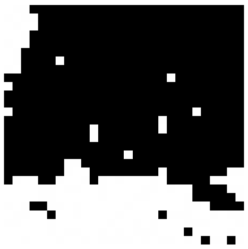

(h) $t = 7$


(i) $t = 8$

(j) $t = 9$

(k) $t = 10$

(l) $t = 11$


(m) $t = 12$

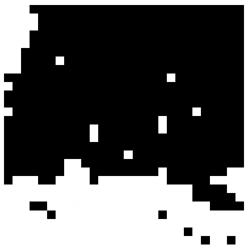
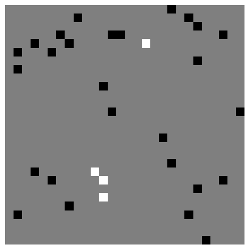
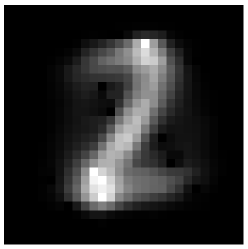
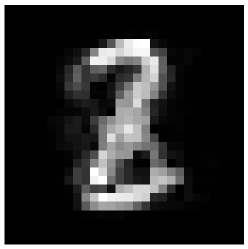
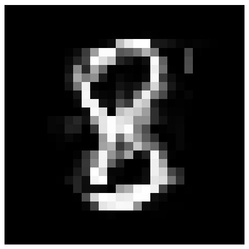

(n) $t = 13$

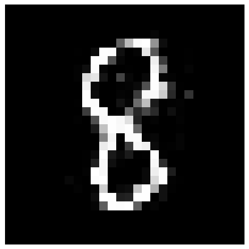
Figure A-3: Decoding the same sample in Figure 5-4 using `PGM-SEP`. Intensities α_i range from black=0 to white=1. Intensities denote the value of the estimated marginal $\hat{p}_{\mathsf{s}_i}(1) \triangleq \alpha_i$. We set $r_{\text{code}} = 0.5$ and $r_{\text{dope}} = 0.04$. Compared to 5-4, we see that `PGM-SEP` arrived at the solution differently. `PGM-SEP` started decoding by growing outwards from the doped sites whereas `SPN-SEP` started to look for global structure from the first iteration. This is because BP on a PGM uses local markov structure to decode the source pixels whereas SPN use the complete structure of the source.

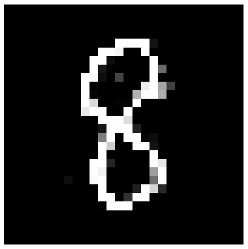

(a) $t = 0$

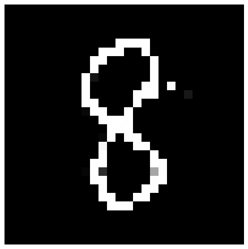

(b) $t = 1$

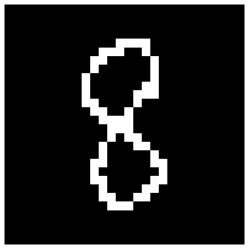

(c) $t = 2$


(d) $t = 3$


(e) $t = 4$


(f) $t = 5$


(g) $t = 6$


(h) $t = 7$

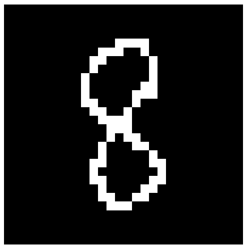
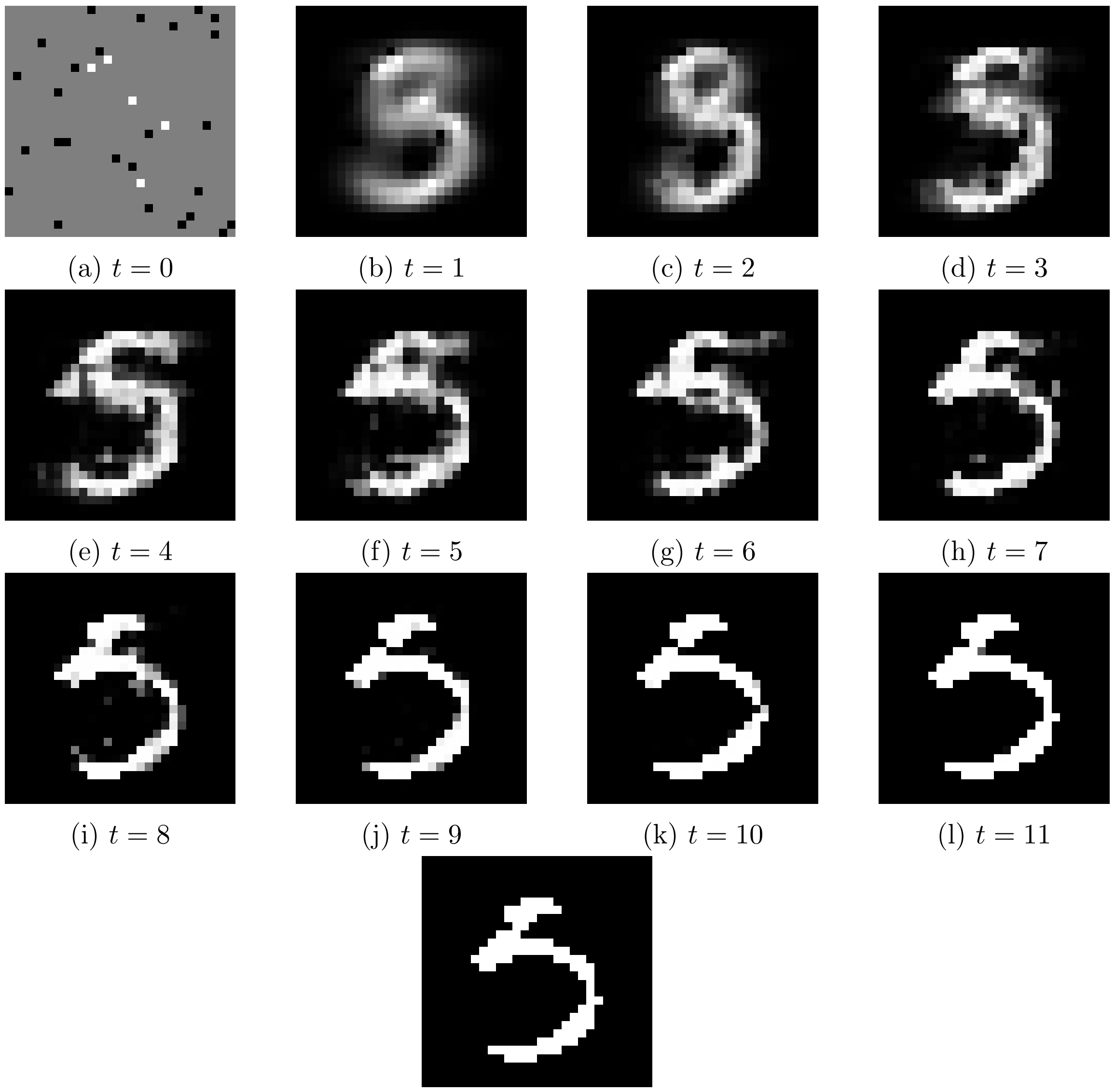

(i) $t = 8$

Figure A-4: Decoding the digit 8 from the binary MNIST dataset using `SPN-SEP`. Intensities α_i range from black=0 to white=1. Intensities denote the value of the estimated marginal $\hat{p}_{\mathsf{s}_i}(1) \triangleq \alpha_i$. We set $r_{\text{code}} = 0.3$ and $r_{\text{dope}} = 0.04$



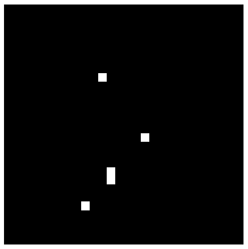
(a) $t = 0$ (b) $t = 1$ (c) $t = 2$ (d) $t = 3$

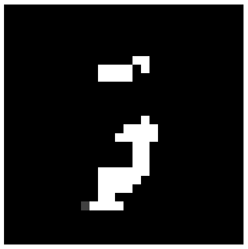
(e) $t = 4$ (f) $t = 5$ (g) $t = 6$ (h) $t = 7$

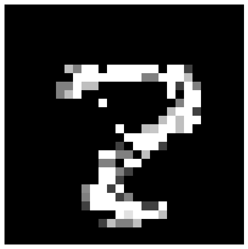
(i) $t = 8$ (j) $t = 9$ (k) $t = 10$ (l) $t = 11$

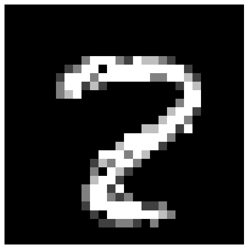
(m) $t = 12$

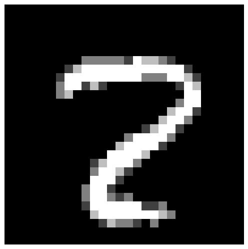
Figure A-5: Decoding the digit 5 from the binary MNIST dataset using `SPN-SEP`. Intensities α_i range from black=0 to white=1. Intensities denote the value of the estimated marginal $\hat{p}_{\mathsf{s}_i}(1) \triangleq \alpha_i$. We set $r_{\text{code}} = 0.27$ and $r_{\text{dope}} = 0.04$

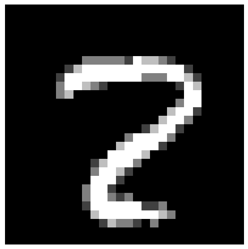

(a) $t = 0$


(b) $t = 1$


(c) $t = 2$


(d) $t = 3$


(e) $t = 4$


(f) $t = 5$

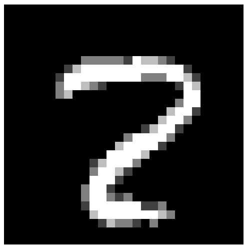
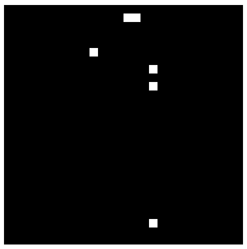
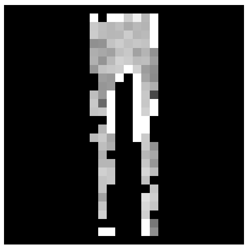
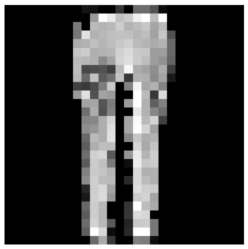
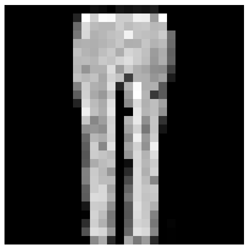

(g) $t = 6$

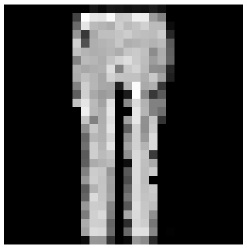
Figure A-6: Decoding the digit 2 from the MNIST dataset using `SPN-SEP`. Intensities α_i range from black=0 to white=1. Intensities denote the value of the max marginal $\alpha_i \triangleq \max_{s_i} \hat{p}_{\mathsf{s}_i}(s_i)$. We set $r_{\text{code}} = 0.23$ and $r_{\text{dope}} = 0.04$

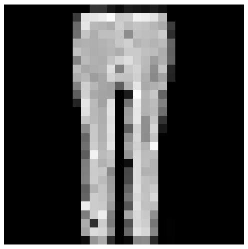

(a) $t = 0$

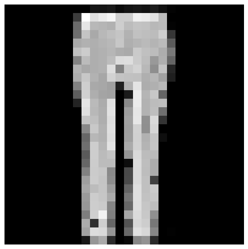

(b) $t = 1$

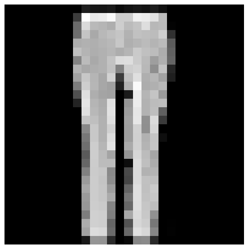

(c) $t = 2$

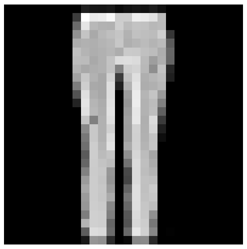

(d) $t = 3$

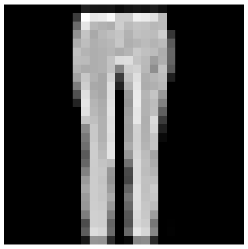

(e) $t = 4$

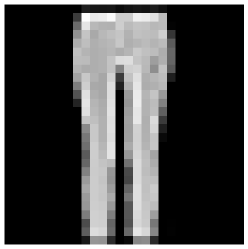

(f) $t = 5$

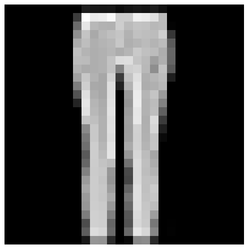

(g) $t = 6$


(h) $t = 7$


(i) $t = 8$


(j) $t = 9$


(k) $t = 10$


(l) $t = 11$

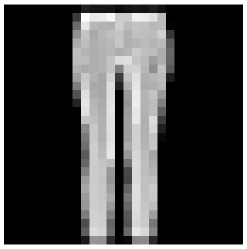
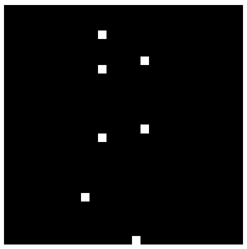
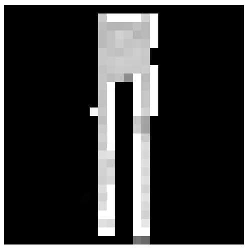
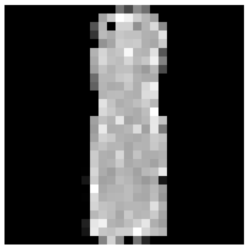
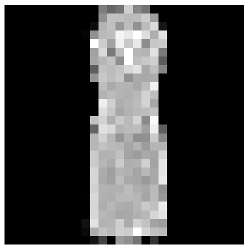

(m) $t = 12$

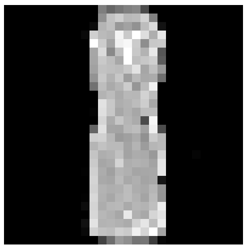
Figure A-7: Decoding an image of a pant from the Fashion MNIST dataset using `SPN-SEP`. Intensities α_i range from black=0 to white=1. Intensities denote the value of the estimated marginal $\hat{p}_{\mathsf{s}_i}(1) \triangleq \alpha_i$. We set $r_{\text{code}} = 0.45$ and $r_{\text{dope}} = 0.04$

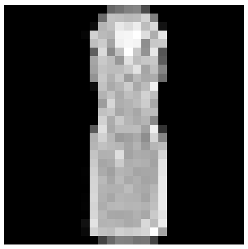

(a) $t = 0$

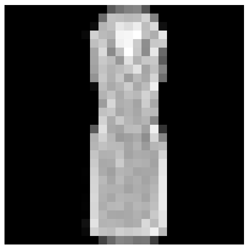

(b) $t = 1$


(c) $t = 2$


(d) $t = 3$


(e) $t = 4$


(f) $t = 5$


(g) $t = 6$

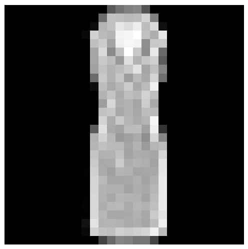
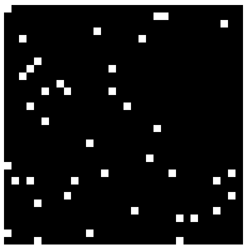
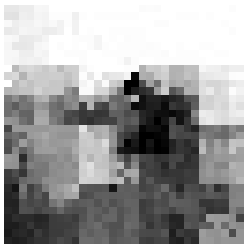
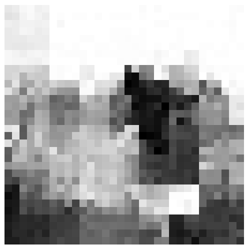
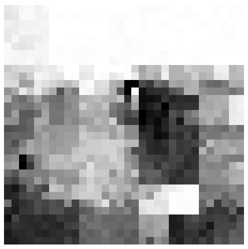

(h) $t = 7$

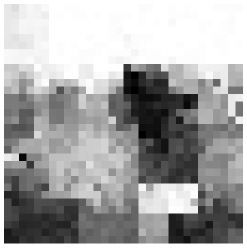
Figure A-8: Decoding an image of a dress from the Fashion MNIST dataset using `SPN-SEP`. Intensities α_i range from black=0 to white=1. Intensities denote the value of the estimated marginal $\hat{p}_{\mathsf{s}_i}(1) \triangleq \alpha_i$. We set $r_{\text{code}} = 0.48$ and $r_{\text{dope}} = 0.04$. Notice how the SPN initially drives the decoder towards decoding the image as a pant.

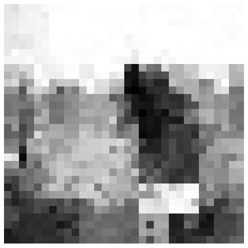

(a) $t = 0$

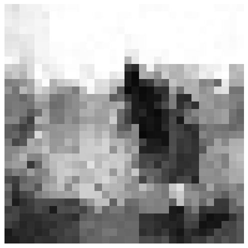

(b) $t = 2$

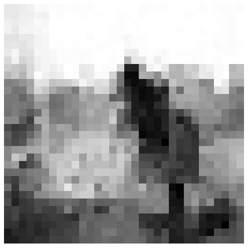

(c) $t = 4$

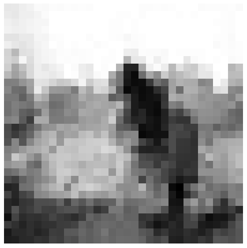

(d) $t = 6$

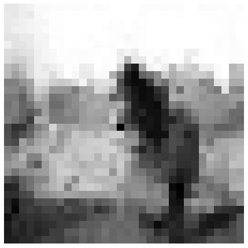

(e) $t = 8$

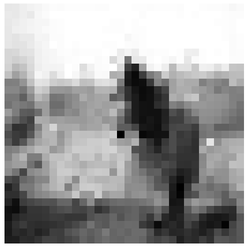

(f) $t = 10$

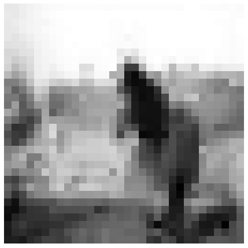

(g) $t = 12$

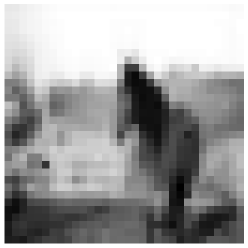

(h) $t = 14$

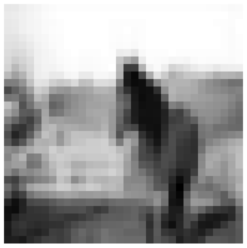

(i) $t = 16$


(j) $t = 18$


(k) $t = 20$


(l) $t = 22$


(m) $t = 24$


(n) $t = 26$

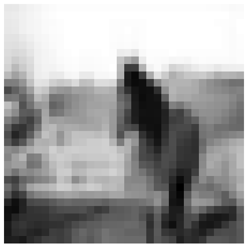
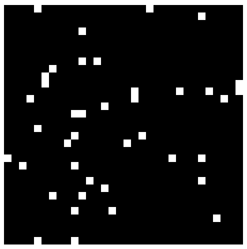
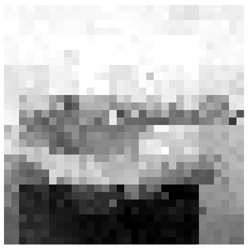
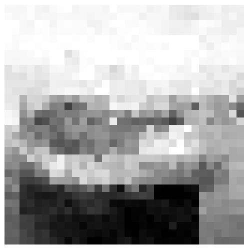
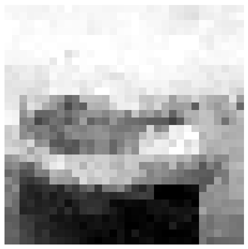

(o) $t = 28$

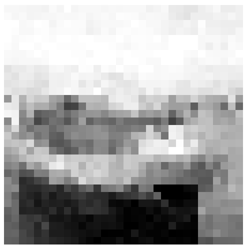
Figure A-9: Decoding an image of a horse from the CIFAR-10 dataset using `SPN-SEP`. Intensities α_i range from black=0 to white=1. Intensities denote the value of the estimated marginal $\hat{p}_{\mathsf{s}_i}(1) \triangleq \alpha_i$. We set $r_{\text{code}} = 0.80$ and $r_{\text{dope}} = 0.04$.

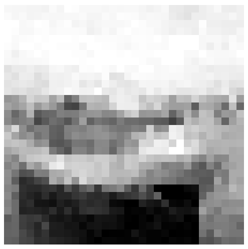

(a) $t = 0$

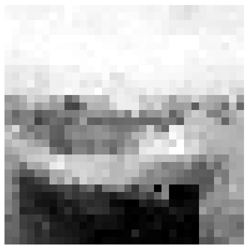

(b) $t = 5$

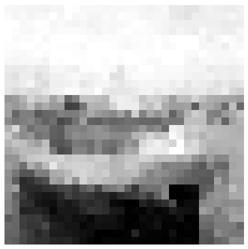

(c) $t = 10$

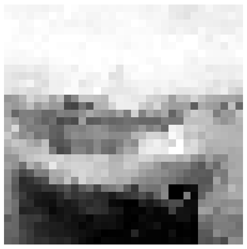

(d) $t = 15$

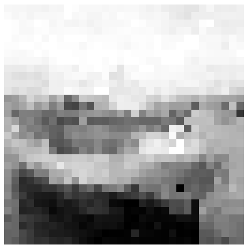

(e) $t = 20$

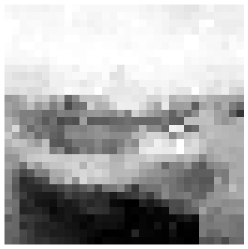

(f) $t = 25$

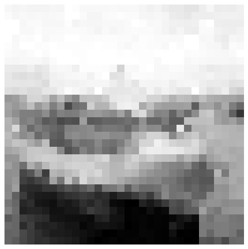

(g) $t = 30$

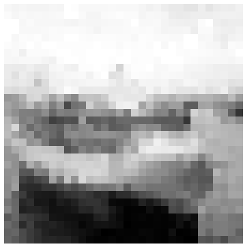

(h) $t = 35$

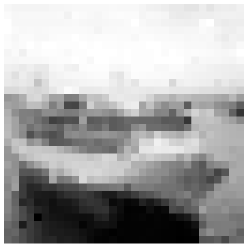

(i) $t = 40$

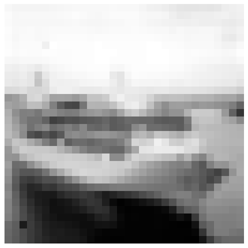

(j) $t = 45$


(k) $t = 50$


(l) $t = 55$


(m) $t = 60$


(n) $t = 65$


(o) $t = 70$

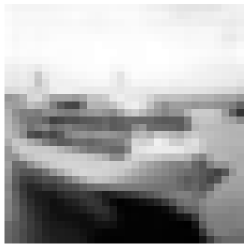

(p) $t = 74$

Figure A-10: Decoding an image of a boat from the CIFAR-10 dataset using `SPN-SEP`. Intensities α_i range from black=0 to white=1. Intensities denote the value of the estimated marginal $\hat{p}_{\mathsf{s}_i}(1) \triangleq \alpha_i$. We set $r_{\text{code}} = 0.80$ and $r_{\text{dope}} = 0.04$.

Bibliography

[1] N. Ahmed, T. Natarajan, and K.R. Rao. Discrete cosine transform. *IEEE Transactions on Computers*, C-23(1):90–93, 1974.

[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] Thomas Boutell and T Lane. PNG (portable network graphics) specification version 1.0. *Network Working Group*, pages 1–102, 1997.

[4] Michael Burrows and David Wheeler. A block-sorting lossless data compression algorithm. In *Digital SRC Research Report*. Citeseer, 1994.

[5] Cory J Butz, Jhonatan S Oliveira, André E dos Santos, and André L Teixeira. Deep convolutional sum-product networks. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33, pages 3248–3255, 2019.

[6] CKCN Chow and Cong Liu. Approximating discrete probability distributions with dependence trees. *IEEE transactions on Information Theory*, 14(3):462–467, 1968.

[7] Peter Clifford. Markov random fields in statistics. *Disorder in physical systems: A volume in honour of John M. Hammersley*, pages 19–32, 1990.

[8] Imre Csiszár and János Körner. *Information theory: coding theorems for discrete memoryless systems*. Cambridge University Press, 2011.

[9] Imre Csiszár and Paul C Shields. Information theory and statistics: A tutorial. 2004.

[10] Adnan Darwiche. A logical approach to factoring belief networks. *KR*, 2:409–420, 2002.

[11] Adnan Darwiche. A differential approach to inference in Bayesian networks. *Journal of the ACM (JACM)*, 50(3):280–305, 2003.

[12] Olivier Delalleau and Yoshua Bengio. Shallow vs. deep sum-product networks. *Advances in neural information processing systems*, 24:666–674, 2011.

[13] Mattia Desana. *Sum-Product Graphical Models: a Graphical Model Perspective on Sum-Product Networks*. PhD thesis, 2018.

[14] Mattia Desana and Christoph Schnörr. Sum-product graphical models. *Machine Learning*, 109(1):135–173, 2020.

[15] Mathias Drton and Marloes H Maathuis. Structure learning in graphical modeling. *Annual Review of Statistics and Its Application*, 4:365–393, 2017.

[16] Jerome Friedman, Trevor Hastie, and Robert Tibshirani. Sparse inverse covariance estimation with the graphical lasso. *Biostatistics*, 9(3):432–441, 2008.

[17] Mark Gales and Steve Young. The application of hidden Markov models in speech recognition. 2008.

[18] Robert Gallager. Low-density parity-check codes. *IRE Transactions on information theory*, 8(1):21–28, 1962.

[19] Stuart Geman and Donald Geman. Stochastic relaxation, Gibbs distributions, and the Bayesian restoration of images. *IEEE Transactions on pattern analysis and machine intelligence*, (6):721–741, 1984.

[20] Robert Gens and Pedro Domingos. Discriminative learning of sum-product networks. *Advances in Neural Information Processing Systems*, 25:3239–3247, 2012.

[21] Joseph Gonzalez, Yucheng Low, Arthur Gretton, and Carlos Guestrin. Parallel Gibbs sampling: From colored fields to thin junction trees. In *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics*, pages 324–332. JMLR Workshop and Conference Proceedings, 2011.

[22] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial nets. *Advances in neural information processing systems*, 27, 2014.

[23] Horst Hampel, Ronald B Arps, Christos Chamzas, David Dellert, Donald L Duttweiler, Toshiaki Endoh, William Equitz, Fumitaka Ono, Richard Pasco, Istvan Sebestyen, et al. Technical features of the JBIG standard for progressive bi-level image compression. *Signal Processing: Image Communication*, 4(2):103–111, 1992.

[24] W Keith Hastings. Monte Carlo sampling methods using Markov chains and their applications. 1970.

[25] Emiel Hoogeboom, Jorn WT Peters, Rianne van den Berg, and Max Welling. Integer discrete flows and lossless compression. *arXiv preprint arXiv:1905.07376*, 2019.

[26] Ying-zong Huang. *Model-code separation architectures for compression based on message-passing*. PhD thesis, Massachusetts Institute of Technology, 2015.

[27] Ying-zong Huang and Gregory W Wornell. A class of compression systems with model-free encoding. In *2014 Information Theory and Applications Workshop (ITA)*, pages 1–7. IEEE, 2014.

[28] David A. Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the IRE*, 40(9):1098–1101, 1952.

[29] Nal Kalchbrenner, Erich Elsen, Karen Simonyan, Seb Noury, Norman Casagrande, Edward Lockhart, Florian Stimberg, Aaron Oord, Sander Dieleman, and Koray Kavukcuoglu. Efficient neural audio synthesis. In *International Conference on Machine Learning*, pages 2410–2419. PMLR, 2018.

[30] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[31] Daphne Koller and Nir Friedman. *Probabilistic graphical models: principles and techniques.* MIT press, 2009.

[32] Alex Krizhevsky, Geoffrey Hinton, et al. Learning multiple layers of features from tiny images. 2009.

[33] Frank R Kschischang, Brendan J Frey, and H-A Loeliger. Factor graphs and the sum-product algorithm. *IEEE Transactions on information theory*, 47(2):498–519, 2001.

[34] Wai Lok Lai et al. *A probabilistic graphical model based data compression architecture for Gaussian sources.* PhD thesis, Massachusetts Institute of Technology, 2016.

[35] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[36] Joshua Ka-Wing Lee. *A model-adaptive universal data compression architecture with applications to image compression.* PhD thesis, Massachusetts Institute of Technology, 2017.

[37] Nicolai Meinshausen and Peter Bühlmann. High-dimensional graphs and variable selection with the lasso. *The annals of statistics*, 34(3):1436–1462, 2006.

[38] Fabian Mentzer, Eirikur Agustsson, Michael Tschannen, Radu Timofte, and Luc Van Gool. Practical full resolution learned lossless image compression. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 10629–10638, 2019.

[39] F. Ono, W. Rucklidge, R. Arps, and C. Constantinescu. JBIG2-the ultimate bi-level image coding standard. In *Proceedings 2000 International Conference on Image Processing (Cat. No.00CH37101)*, volume 1, pages 140–143 vol.1, 2000.

[40] Aaron van den Oord, Nal Kalchbrenner, Oriol Vinyals, Lasse Espeholt, Alex Graves, and Koray Kavukcuoglu. Conditional image generation with PixelCNN decoders. *arXiv preprint arXiv:1606.05328*, 2016.

[41] Aaron van den Oord, Oriol Vinyals, and Koray Kavukcuoglu. Neural discrete representation learning. *arXiv preprint arXiv:1711.00937*, 2017.

[42] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. PyTorch: An imperative style, high-performance deep learning library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019.

[43] Robert Peharz, Antonio Vergari, Karl Stelzner, Alejandro Molina, Xiaoting Shao, Martin Trapp, Kristian Kersting, and Zoubin Ghahramani. Random sum-product networks: A simple and effective approach to probabilistic deep learning. In *Uncertainty in Artificial Intelligence*, pages 334–344. PMLR, 2020.

[44] Hoifung Poon and Pedro Domingos. Sum-product networks: A new deep architecture. In *2011 IEEE International Conference on Computer Vision Workshops (ICCV Workshops)*, pages 689–690. IEEE, 2011.

[45] Abdullah Rashwan, Pascal Poupart, and Chen Zhitang. Discriminative training of sum-product networks by extended Baum-Welch. In *International Conference on Probabilistic Graphical Models*, pages 356–367. PMLR, 2018.

[46] Danilo Rezende and Shakir Mohamed. Variational inference with normalizing flows. In *International conference on machine learning*, pages 1530–1538. PMLR, 2015.

[47] Tom Richardson and Ruediger Urbanke. *Modern coding theory.* Cambridge university press, 2008.

[48] Ruslan Salakhutdinov, Andriy Mnih, and Geoffrey Hinton. Restricted Boltzmann machines for collaborative filtering. In *Proceedings of the 24th international conference on Machine learning*, pages 791–798, 2007.

[49] David Salomon and Giovanni Motta. *Handbook of data compression.* Springer, 2010.

[50] Raquel Sánchez-Cauce, Iago París, and Francisco Javier Díez. Sum-product networks: A survey. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2021.

[51] Khalid Sayood. *Introduction to data compression.* Morgan Kaufmann, 2017.

[52] Arne von See. Total data volume worldwide 2010-2025, Jun 2021.

[53] C. E. Shannon. A mathematical theory of communication. *The Bell System Technical Journal*, 27(3):379–423, 1948.

[54] Xiaoting Shao, Alejandro Molina, Antonio Vergari, Karl Stelzner, Robert Peharz, Thomas Liebig, and Kristian Kersting. Conditional sum-product networks: Imposing structure on deep probabilistic architectures. In *International Conference on Probabilistic Graphical Models*, pages 401–412. PMLR, 2020.

[55] A. Skodras, C. Christopoulos, and T. Ebrahimi. The JPEG 2000 still image compression standard. *IEEE Signal Processing Magazine*, 18(5):36–58, 2001.

[56] G.K. Wallace. The JPEG still picture compression standard. *IEEE Transactions on Consumer Electronics*, 38(1):xviii–xxxiv, 1992.

[57] Dong Wang, Xiaodong Wang, and Shaohe Lv. An overview of end-to-end automatic speech recognition. *Symmetry*, 11(8):1018, 2019.

[58] Xintao Wang, Ke Yu, Shixiang Wu, Jinjin Gu, Yihao Liu, Chao Dong, Yu Qiao, and Chen Change Loy. ESRGAN: Enhanced super-resolution generative adversarial networks. In *Proceedings of the European conference on computer vision (ECCV) workshops*, pages 0–0, 2018.

[59] Welch. A technique for high-performance data compression. *Computer*, 17(6):8–19, 1984.

[60] Greg Welch, Gary Bishop, et al. An introduction to the Kalman filter. 1995.

[61] Jos Wolfshaar and Andrzej Pronobis. Deep generalized convolutional sum-product networks. In *International Conference on Probabilistic Graphical Models*, pages 533–544. PMLR, 2020.

[62] Han Xiao, Kashif Rasul, and Roland Vollgraf. Fashion-MNIST: a novel image dataset for benchmarking machine learning algorithms, 2017.

[63] Feng Yang, Yue M Lu, Luciano Sbaiz, and Martin Vetterli. Bits from photons: Oversampled image acquisition using binary Poisson statistics. *IEEE Transactions on image processing*, 21(4):1421–1436, 2011.

[64] Jonathan S Yedidia, William T Freeman, Yair Weiss, et al. Understanding belief propagation and its generalizations. *Exploring artificial intelligence in the new millennium*, 8:236–239, 2003.

[65] Xuhong Zhang et al. *A sampling technique based on LDPC codes.* PhD thesis, Massachusetts Institute of Technology, 2015.