BAYESIAN NETWORKS FOR PATTERN CLASSIFICATION, DATA COMPRESSION, AND CHANNEL CODING

Brendan J. Frey

A thesis submitted in conformity with the requirements

for the degree of Doctor of Philosophy,

Graduate Department of Electrical and Computer Engineering,

in the University of Toronto

Bayesian Networks for Pattern Classification, Data Compression, and Channel Coding

Brendan J. Frey

A thesis submitted in conformity with the requirements
for the degree of Doctor of Philosophy,
Graduate Department of Electrical and Computer Engineering,
in the University of Toronto
May 1997

Abstract

Pattern classification, data compression, and channel coding are tasks that usually must deal with complex but structured natural or artificial systems. Patterns that we wish to classify are a consequence of a causal physical process. Images that we wish to compress are also a consequence of a causal physical process. Noisy outputs from a telephone line are corrupted versions of a signal produced by a structured man-made telephone modem. Not only are these tasks characterized by complex structure, but they also contain random elements. Graphical models such as Bayesian networks provide a way to describe the relationships between random variables in a stochastic system.

In this thesis, I use Bayesian networks as an overarching framework to describe and solve problems in the areas of pattern classification, data compression, and channel coding. Results on the classification of handwritten digits show that Bayesian network pattern classifiers outperform other standard methods, such as the k-nearest neighbor method. When Bayesian networks are used as source models for data compression, an exponentially large number of codewords are associated with each input pattern. It turns out that the code can still be used efficiently, if a new technique called "bits-back coding" is used. Several new error-correcting decoding algorithms are instances of "probability propagation" in various Bayesian networks. These new schemes are rapidly closing the gap between the performances of practical channel coding systems and Shannon's 50-year-old channel coding limit. The Bayesian network framework exposes the similarities between these codes and leads the way to a new class of "trellis-constraint codes" which also operate close to Shannon's limit.

Nomenclature

Vectors, matrices, high-dimensional matrices and sets of variables are written in boldface Roman type. (Vectors are usually written in lower-case type.) Sets are quite different from vectors, but this abuse of notation permits set operations (*e.g.*, "$\subseteq$", "$\backslash$") while at the same time permiting cardinal access to the set members (*e.g.*, weighted sum of the elements via indexing). I use curly braces $\{\ldots\}$ to write the elements in a set or vector of variables. $\{z_k\}$ is the set containing a singleton variable z_k, whereas $\{z_k\}_{k=1}^K = \{z_1, \ldots, z_K\}$. Extra labels on variables usually appear as upper-case Roman in superscripts (*e.g.*, θ^{V}), whereas vector, matrix, and high-dimensional matrix indices usually appear as subscripts (*e.g.*, θ_{ik}^{V}). For example, we can write the following with respect to the set of parameters $\boldsymbol{\theta}$: $\{\theta_{ik}^{\mathrm{V}}\}_{k=1}^K = \boldsymbol{\theta}_i^{\mathrm{V}} \subseteq \boldsymbol{\theta}^{\mathrm{V}} \subseteq \boldsymbol{\theta}$, and $\sum_j \theta_{ij}^{\mathrm{V}} h_j$. Some types of index (notably training case indices) appear as superscripts in braces (*e.g.*, $\mathbf{v}^{(t)}$).

Probability mass functions are usually written in upper-case Roman italics type (*e.g.* $P(\cdot)$, $Q(\cdot)$) whereas probability density functions are usually written in lower-case Roman italics type (*e.g.* $p(\cdot)$, $q(\cdot)$). The distribution is identified by the random variable, so the distribution $P(\mathbf{v})$ is different from $P(\mathbf{x})$. Also, to keep the formulas short, the symbols for a random variable and its value are usually the same. So, $P(u_k|\mathbf{y})$ sometimes refers to the probability that the random variable U_k takes on the value u_k, and at other times refers to the set of probabilties corresponding to the values that U_k can take on. In cases where a random variable and its value must be distinguishable, I write an assignment. So, $P(u_k = u'_k|\mathbf{y})$ means $P_{U_k|\mathbf{Y}}(u'_k|\mathbf{y})$. A distribution subscripted with "r" refers to the correct, or "real" distribution. For example, if $P(\mathbf{v})$ is a model distribution, we hope that $P(\mathbf{v}) \approx P_{\mathrm{r}}(\mathbf{v})$.

Here is a list of symbols:

a	represents the binary "alarm" variable in the burlar alarm problem; or represents an entire signalling waveform defined on $0 \leq t \leq T$
$a(\cdot)$	signalling (channel input) waveform for channel coding
$\mathbf{a}_k$	set of variables containing the *parents* of z_k
b	represents the binary "burglar" variable in the burlar alarm problem
BER	bit error rate
$B_{Q\|\|P}$	lower bound on $\log P(\mathcal{D}\|\boldsymbol{\theta})$, $B_{Q\|\|P} = -F_{Q\|\|P} = \log P(\mathbf{v}) - D_{Q\|\|P}$
C	communication capacity of a channel in bits/second or bits/usage
$\mathbf{c}_k$	set of variables containing the *children* of z_k
D	delay buffer in an LFSR; or mathematical delay operator
$\mathcal{D}$	collection of data
$D_{Q\|\|P}$	Kullback-Leibler divergence (relative entropy) between $Q(\cdot)$ and $P(\cdot)$

$\mathbf{d}_k$	set of variables containing the *descendents* of z_k
$d_{\min}$	Hamming distance between the closest two codewords in a channel code
$\mathrm{E}[\cdot]$	expectation with respect to the stated distribution
$\mathcal{E}(\cdot)$	$\mathcal{E}(\mathbf{v})$ is the expected length of the codeword for $\mathbf{v}$, in a multi-valued source code
E_b	the energy transmitted per information bit in channel coding
e	represents the binary "earthquake" variable in the burlar alarm problem
$\mathcal{F}(\cdot)$	$\mathcal{F}(\mathbf{v})$ is the theoretical bits-back codeword length for input pattern $\mathbf{v}$
$F_{Q\|\|P}$	free energy between $Q(\cdot)$ and $P(\cdot)$, $F_{Q\|\|P} = D_{Q\|\|P} - \log P(\mathbf{v})$
$G(\cdot)$	$G(D)$ is the generator polynomial in the delay operator D, for a convolutional code
$g(\cdot)$	logistic function: $g(x) = 1/(1+\exp[-x])$; or output bit generating function for an LFSR
$\mathcal{H}$	entropy measured in bits
$H(\cdot)$	binary entropy function, $H(p) = -p\log p - (1-p)\log(1-p)$
$\mathbf{h}$	set of hidden (unobserved) variables; or codeword index in a multi-valued source code
$\mathbf{h}^{(t)}$	vector of hidden variables for training case t
K	number of hidden variables; or number of information variables
$L(\cdot)$	log-likelihood ratio for a random variable given some observations — *e.g.*, $L(z=z') = \log[P(z=z'\|\mathbf{y})/P(z \neq z'\|\mathbf{y})]$
$\hat{L}^i(\cdot)$	approximation to $L(\cdot)$ produced at iteration i of iterative decoding
$\ell(\cdot)$	$\ell(\mathbf{v})$ is the length of the source codeword for $\mathbf{v}$
$\ell(\cdot,\cdot)$	$\ell(\mathbf{v},\mathbf{h})$ is the length of the $\mathbf{h}$th codeword for $\mathbf{v}$, in a multi-valued source code
$\log(\cdot)$	natural logarithm
$\log_x(\cdot)$	logarithm to the base x
N	number of variables in a network; or number of visible variables; or number of codeword variables
N_{t}	number of constituent trellises in an interleaved trellis-constraint code
N_0	single-sided spectral density for a white Gaussian process
$\mathbf{n}_k$	set of variables containing the *nondescendents* of z_k (excluding z_k)
P	transmitter power for channel coding
$\mathcal{P}$	set of all distributions that can be represented by a Bayesian network
$P(\cdot)$	probability mass function for a Bayesian network; or the probability function for a mixed (discrete and continuous) set of variables
$\hat{P}(\cdot)$	estimate of $P(\cdot)$
$\mathbf{P}^{\mathbf{A}_i Z_i}$	conditional probability matrix for variable z_i, $\mathbf{P}^{\mathbf{A}_i Z_i}_{\mathbf{a}_i z_i} = P(z_i\|\mathbf{a}_i)$

$p(\cdot)$	probability density function for a Bayesian network
$Q(\cdot)$	general variational distribution; or general recognition network distribution
$\mathbf{q}$	vector of parity-check variables
$q(\cdot)$	variational probability density; or recognition network probability density
R	rate of a binary channel code in information bits / codeword bits
$S(\cdot,\cdot)$	state transition function for a LFSR: $s_k = S(s_{k-1}, u_k)$
$\mathbf{s}$	set of discrete LFSR state variables, where s_i is the state at time i
t	training case index, $1 \leq t \leq T$
$\mathbf{u}$	vector of binary information variables
$\hat{\mathbf{u}}$	an estimate of the true information vector $\mathbf{u}$
$\mathbf{v}$	vector of visible (observed) variables
$\mathbf{v}^{(t)}$	vector of visible (observed) variables for training case t
$\mathbf{x}$	vector of binary codeword variables; or dummy vector variable
$x_{i,j}$	the jth branch variable that participates in constraint i of an interleaved trellis-constraint code
$\mathbf{y}$	vector of real channel output variables; or dummy vector variable
$y(\cdot)$	channel output waveform for channel coding
y^{o}	observed value of a variable $y \in \mathbf{v}$ in a Bayesian network
$\mathbf{z}$	set of variables used to discuss properties of Bayesian networks in general
α	normalization constant
$\delta(\cdot,\cdot)$	delta function: $\delta(x,y) = 1$ if $x = y$ and 0 otherwise
η	learning rate for steepest descent parameter estimation
$\boldsymbol{\theta}$	set of all parameters for a parameterized Bayesian network
$\boldsymbol{\theta}_i$	set of parameters associated with variable i
θ_{ij}	parameter associated with the connection from variable j to variable i
θ_{i0}	constant (bias) parameter associated with variable i
$\boldsymbol{\theta}^{\mathrm{H}}$	set of parameters associated with the set of hidden variables $\mathbf{h}$
$\boldsymbol{\theta}^{\mathrm{V}}$	set of parameters associated with the set of visible variables $\mathbf{v}$
$\boldsymbol{\lambda}^{Z_1 Z_2}$	child-parent message sent from z_1 to z_2 — has $\|z_2\|$ elements
$\boldsymbol{\xi}$	variational parameters
$\pi(\cdot)$	permutation function that maps integers in $[1,N]$ to $[1,N]$, for some integer N
$\boldsymbol{\pi}^{Z_1 Z_2}$	parent-child message sent from z_1 to z_2 — has $\|z_1\|$ elements
τ	time in a Markov chain Monte Carlo simulation
$\Phi(\cdot)$	cummulative distribution for a standard normal p.d.f.
$\boldsymbol{\phi}$	recognition network parameters (see $\boldsymbol{\theta}_i$ and $\boldsymbol{\theta}_{ij}$ for refined details)

$\lvert\cdot\rvert$	$\lvert\mathbf{z}\rvert$ = number of variables in $\mathbf{z}$; or $\lvert z_k\rvert$ = number of values z_k can take on
$\sum_{\mathbf{x}}$	summation over all possible configurations of $\mathbf{x}$
$\sum_{\mathbf{x}':x'_i=x_i}$	summation over the configurations of $\mathbf{x}'$ for which element $x'_i = x_i$
$\{\ldots\}$	set or vector of variables, $\{z_k\}$ means the set containing z_k alone
$\oplus$	addition modulo 2

Acknowledgements

During my Ph.D. program here at the University of Toronto, I have been fortunate to benefit from interactions with several excellent researchers. I thank my thesis advisor Geoffrey Hinton for his guidance. I greatly value his open-mindedness, his inspirational discussions, and his honest criticisms. I also thank Radford Neal, whose creativity and persistent persuit of precision has certainly enhanced my research and this dissertation. I am grateful to Frank Kschischang for valuable discussions on error-correcting coding and for introducing me to some of the premier communications researchers. I greatly appreciate recent energetic collaborations with David MacKay, who I find is an engaging collaborator. In addition, I thank the other members of my thesis committee: Glenn Gulak, Michael Jordan, Subbarayan Pasupathy, and Tas Venetsanopoulos.

Thanks also go to the following former and present members of the Neural Networks Research Group for valuable conversations: Peter Dayan, Zoubin Ghahramani, Carl Rasmussen, Virginia de Sa, Brian Sallans, and Chris Williams.

My love goes to my wife Utpala Purohit-Frey and my son Shardul Frey, for being supportive of my interest in research and indeed for motivating me to get some work done!

I was financially supported by a Natural Sciences and Engineering Research Coucil 1967 Science and Engineering Scholarship, a Walter C. Sumner Memorial Fellowship, and a University of Toronto Open Fellowship. My work was financially supported by grants from the Natural Sciences and Engineering Research Council and the Institute for Robotics and Intelligent Systems.

Contents

Chapter 1

Introduction

In this thesis, I explore algorithms for pattern classification, data compression, and channel coding. At first, it may be hard to imagine how three so different research areas can be brought into focus under a single theme that is both novel and of practical value. My hope is to convince the reader that these three problems can be attacked in an interesting and fruitful manner using a recently developed class of algorithms that make use of *probabilistic structure*. These algorithms take advantage of a graphical description of the dependencies between random variables in order to compute, or estimate, probabilities derived from a joint distribution. As simple and well-known examples, the forward-backward algorithm [Baum and Petrie 1966] and the Viterbi algorithm [Forney 1973] make use of a chain-like Markovian relationship between random variables.

The roots of probabilistic structure reach far back to the beginning of the 20th century. In 1921, Sewall Wright developed "path analysis" as a means to study statistical relationships in biological data. Few new developments were made until the 1960's when statisticians began using graphs to describe restrictions in statistical models called "log-linear models" [Vorobev 1962; Goodman 1970]. In 1963, the idea of hierarchical probabilistic structure briefly reared its head in the engineering research community when Gallager invented an error-correcting decoding algorithm based on a graphical description of the probabilistic relationships between variables involved in channel coding. Most likely because of the primitive computers available at the time, his algorithm was quickly overlooked by his peers, only to be rediscovered nearly 35 years later independently by at least three research groups, and to be shown to yield unprecedented performance in error-correcting coding applications [Berrou, Glavieux and Thitimajshima 1993; Wiberg, Loeliger and Kötter 1995; MacKay and Neal 1995]. A simpler chain-type Markovian graphical structure later became popular and very useful in the engineering community, largely due to an excellent tutorial paper by Forney [1973], in which the notion of a "trellis" was introduced. Probabilistic structure has been most extensively developed in the artificial intelligence literature, with applications ranging from taxonomic hierarchies [Woods 1975; Schubert 1976] to medical diagnosis [Spiegelhalter 1990]. In the late 1980's, Pearl [1986] and Lauritzen and Spiegelhalter [1988] independently published a general algorithm for computing probabilities based on a graphical representation of probabilistic structure. This algorithm is practical and exact for only a special type of probabilistic structure. Over the last decade, there has also been a tremendous increase in interest in estimating the parameters of models with fixed graphical structure. In the mid 1980's, Hinton and Sejnowski [1986] introduced a maximum likelihood algorithm for learning the parameters of a graph-based log-linear model called a "Markov random field". More recently, approximate algorithms for general models based on directed graphs have been introduced. These include Markov chain Monte Carlo methods [Pearl 1987; Neal 1992], "Helmholtz machines" [Hinton et al. 1995; Dayan et al. 1995], and variational techniques [Saul, Jaakkola and Jordan 1996; Jaakkola, Saul and Jordan 1996; Frey 1997b].

1.1 A probabilistic perspective

Offhand, it is not obvious that sophisticated probability models are needed to solve problems in pattern classification, data compression, and channel coding. Given a segment of speech, a classifier outputs a decision, say, as to whether or not the speaker has security clearance. It appears there are no random variables in this model. The classifier may also output a measure of reliability regarding the decision it makes. In this case, it appears there is just one binary random variable that captures the variability in the decision. The mean of this Bernoulli random variable must somehow be related to the input, and this task can be viewed as some sort of function approximation. Similarly, given a highly-redundant image, a data compression algorithm usually produces a unique sequence of codeword symbols. Given the output of a noisy telephone line, a channel decoder (telephone modem) makes a deterministic decision about the contents of the transmitted data file.

While the above modelling approaches either require only very low-dimensional probability models or do not use random variables at all, in doing so, they are clearly not representing the true causal structure in each problem. For example, in reality each speaker has a unique glottis that interacts in a random way with a unique shape of vocal tract and a unique random style of articulation to produce a speech segment. It seems like a fruitful approach to speaker identification would involve representing these random variables and the probabilistic relationships between them. In the following three sections, I attempt to reveal some of the probabilistic structure present in pattern classification, data compression, and channel coding problems.

1.1.1 Pattern classification

A soft-decision *classifier* estimates the probability that a given pattern $\mathbf{v}$ belongs to each class $j \in \{0, \ldots, J-1\}$. That is, the classifier estimates

$$P_{\mathrm{r}}(j|\mathbf{v}), \tag{1.1}$$

where the subscript "r" in P_{r} indicates a true (real) probability (as opposed to one produced by a model). If these probabilities can be accurately estimated, Bayes decision theory tells us that a minimum rate of error can be achieved by choosing the class j that maximizes $P_{\mathrm{r}}(j|\mathbf{v})$ [Chow 1957; Duda and Hart 1973].

We could use a logistic regression model to estimate $P_{\mathrm{r}}(j|\mathbf{v})$. For example, regression using a flexible model has been successfully used to classify individual digits extracted from handwritten United States ZIP codes [Le Cun et al. 1989]. However, this approach ignores the *causal structure* of the physical process of producing handwritten digits.

In order to faithfully capture the actual physical process that produces each digit, we first ought to specify an *a priori* distribution $P(j)$ over the digit classes $j \in \{0, \ldots, 9\}$ — maybe some digits are more common than others. Next, for a given class of digit j, we expect there to be a distribution $P(\mathbf{h}|j)$ over a set of digit attributes $\mathbf{h}$. These attributes are called "hidden variables", because they are not part of the classifier inputs or outputs. Each element of $\mathbf{h}$ might specify the presence or absence of a particular line segment or flourish. Given a set of features $\mathbf{h}$, we expect there to be a distribution $P(\mathbf{v}|\mathbf{h})$ over possible images — this distribution models the way in which features combine to make an image, as well as noise such as ink spots. The joint distribution given by this model of the real world can be written

$$P(j, \mathbf{h}, \mathbf{v}) = P(j)P(\mathbf{h}|j)P(\mathbf{v}|\mathbf{h}), \tag{1.2}$$

and the distribution over classes given a pattern can be obtained by marginalizing out $\mathbf{h}$ and using Bayes' rule:

$$P(j|\mathbf{v}) = \frac{\sum_{\mathbf{h}} P(j, \mathbf{h}, \mathbf{v})}{\sum_{j'} \sum_{\mathbf{h}} P(j', \mathbf{h}, \mathbf{v})}. \tag{1.3}$$

So, it appears that to properly model the structure of the problem, we need a more sophisticated probabilistic description than (1.1). In general, a *correct* model of this sort, where $P(j|\mathbf{v}) \approx P_{\mathrm{r}}(j|\mathbf{v})$ will perform optimally in terms of classification error.

1.1.2 Data compression

A *source code* maps each input pattern $\mathbf{v}$ to a codeword $\mathbf{u}$, such that for each valid $\mathbf{u}$ there is a unique pattern. I will consider sources where the patterns are i.i.d. (independent and identically drawn) from $P_{\mathrm{r}}(\mathbf{v})$. The purpose of noiseless source coding, or lossless data compression, is to losslessly represent the source patterns by codewords, so that the expected codeword length is as low as possible. Shannon's noiseless source coding theorem [Shannon 1948] states that the average codeword length per source pattern cannot be less than the entropy of the source:

$$\mathrm{E}[\ell(\mathbf{v})] \geq \mathcal{H}, \tag{1.4}$$

where $\ell(\mathbf{v})$ is the length of the codeword for $\mathbf{v}$ in bits, and $\mathcal{H}$ is the entropy of the source in bits:

$$\mathcal{H} = -\sum_{\mathbf{v}} P_{\mathrm{r}}(\mathbf{v}) \log_2 P_{\mathrm{r}}(\mathbf{v}). \tag{1.5}$$

Arithmetic coding [Rissanen and Langdon 1976; Witten, Neal and Cleary 1987] is a practical algorithm for producing near-optimal codewords when the source distribution $P_{\mathrm{r}}(\mathbf{v})$ is known. Sometimes, *e.g.*, if $\mathbf{v}$ is binary-valued, these probabilities can be easily estimated from the source. Often, however, the distribution is too complex, and so a more sophisticated parametric model or flexible model must be used to estimate the probabilities. For example, consider a high-dimensional binary image $\mathbf{v}$ that is produced by the physical process described above, so that

$$P(j, \mathbf{h}, \mathbf{v}) = P(j)P(\mathbf{h}|j)P(\mathbf{v}|\mathbf{h}), \tag{1.6}$$

The probabilities used by the arithmetic encoder are obtained by marginalizing out j and $\mathbf{h}$:

$$P(\mathbf{v}) = \sum_{j} \sum_{\mathbf{h}} P(j, \mathbf{h}, \mathbf{v}). \tag{1.7}$$

We see that a probabilistic description can also be very useful for source coding.

1.1.3 Channel coding

A block *channel code* maps a vector of information symbols $\mathbf{u}$ to a vector of codeword symbols $\mathbf{x}$. This mapping adds redundancy to $\mathbf{u}$ in order to protect the block against channel noise. (As a simple example, the codeword might consist of three repetitions of the information vector.) After $\mathbf{x}$ is transmitted across the channel, the decoder receives a noise-corrupted version $\mathbf{y}$ and produces an estimate of the information block $\hat{\mathbf{u}}$. We say that a *block error* or a *word error* has occurred if $\hat{\mathbf{u}} \neq \mathbf{u}$. In it's simplest form, Shannon's channel coding theorem [Shannon 1948] states that for any given channel, there *exists*[1] a code that can achieve an arbitrarily low probability of block error when the signal-to-noise ratio is greater than a channel-dependent threshold called the Shannon limit. Roughly speaking, the codewords are kept far apart in codeword symbol space, so that when a moderately noise-corrupted codeword is received, it is still possible to determine with high probability which codeword was transmitted.

From a probabilistic perspective, the decoder can minimize the word error rate by choosing an estimate $\hat{\mathbf{u}}$ that maximizes $P_{\mathrm{r}}(\hat{\mathbf{u}}|\mathbf{y})$, or minimize the symbol error rate by choosing an estimate $\hat{\mathbf{u}}$ that maximizes $\prod_k P_{\mathrm{r}}(\hat{u}_k|\mathbf{y})$. A probabilistic model can be constructed by examining the encoding process and the channel. We first specify a (usually uniform) distribution for the information blocks, $P(\mathbf{u})$. Often, the encoder uses a set of *state* variables, $\mathbf{s}$, in order to produce the codeword. These variables are determined from the information block using a distribution $P(\mathbf{s}|\mathbf{u})$ — although this relationship is usually deterministic, this probabilistic description will come in handy later on when we study probabilistic decoding. The transmitted codeword is determined from the information block and state variables by $P(\mathbf{x}|\mathbf{u}, \mathbf{s})$. Finally, the real-valued channel outputs are related to the transmitted codeword by a probability density function $p(\mathbf{y}|\mathbf{x})$ that models the channel. The joint distribution given by the model is

$$P(\mathbf{u}, \mathbf{s}, \mathbf{x}, \mathbf{y}) = P(\mathbf{u})P(\mathbf{s}|\mathbf{u})P(\mathbf{x}|\mathbf{u}, \mathbf{s})p(\mathbf{y}|\mathbf{x}), \tag{1.8}$$

and the distribution over information symbol u_k given the channel output can be obtained by marginalizing out $\mathbf{s}$, $\mathbf{x}$ and u_j, for all $j \neq k$, and using Bayes' rule:

$$P(u_k|\mathbf{y}) = \frac{\sum_{u_j \forall j \neq k} \sum_{\mathbf{s}} \sum_{\mathbf{x}} P(\mathbf{u}, \mathbf{s}, \mathbf{x}, \mathbf{y})}{\sum_{\mathbf{u}'} \sum_{\mathbf{s}} \sum_{\mathbf{x}} P(\mathbf{u}', \mathbf{s}, \mathbf{x}, \mathbf{y})}. \tag{1.9}$$

Although this probabilistic formulation may seem strange compared to many of the strongly

[1] Shannon was quite the tease. He proved the code exists, but did not show us a practical way to encode or decode it.

algebraic traditional approaches, it is this formulation that I view as the foundation of the recently proposed high-performance *turbo-codes* [Berrou and Glavieux 1996].

1.1.4 Probabilistic inference

As presented above, pattern classification, data compression, and channel coding are all similar in that some type of marginal (and possibly conditioned) distribution is sought for a given joint distribution. Consider a set of random variables $\mathbf{z} = \{z_1, z_2, \ldots, z_N\}$ that co-vary according to a joint distribution $P(z_1, z_2, \ldots, z_N)$. For any two subsets of variables $\mathbf{z}^1 \subseteq \mathbf{z}$ and $\mathbf{z}^2 \subseteq \mathbf{z}$, I will refer to the computation or estimation of $P(\mathbf{z}^1|\mathbf{z}^2)$, or a decision based on $P(\mathbf{z}^1|\mathbf{z}^2)$, as *probabilistic inference*.

Examples of probabilistic inference include the computation of the class probabilities for pattern classification (1.3), the computation of the input probability for data compression (1.7), and the information symbol decisions based on the information symbol probabilities for channel coding (1.9). Notice that in these different cases of probabilistic inference, the joint distributions can be decomposed in different ways. In fact, if we decompose the joint distributions at the level of individual variables instead of vector variables, we can envision a wide variety of rich structures. In the next section, I describe Bayesian networks, which can be used to describe this structure.

1.2 Bayesian networks

Often, the joint distribution associated with a probabilistic inference problem can be decomposed into locally interacting factors. For example, the joint distributions involved in the applications of Bayes' rule in (1.3), (1.7), and (1.9) can be expressed in the forms given in (1.2), (1.6), and (1.8). By taking advantage of such *probabilistic structure*, we can design inference algorithms that are more efficient than the blind application of Bayes' rule.

1.2.1 Probabilistic structure

Probabilistic structure can be characterized by a set of *conditional independence* relationships. (This structural description does not fix the values of the probabilities.) For example, in the case of channel coding, we can use the chain rule of probability to write out the joint distribution:

$$P(\mathbf{u}, \mathbf{s}, \mathbf{x}, \mathbf{y}) = P(\mathbf{u})P(\mathbf{s}|\mathbf{u})P(\mathbf{x}|\mathbf{u}, \mathbf{s})p(\mathbf{y}|\mathbf{u}, \mathbf{s}, \mathbf{x}). \tag{1.10}$$

The probability density function (the last factor) can be simplified, since the received vector $\mathbf{y}$ is conditionally independent of the information vector $\mathbf{u}$ and the state vector $\mathbf{s}$, given the transmitted codeword $\mathbf{x}$:

$$p(\mathbf{y}|\mathbf{u}, \mathbf{s}, \mathbf{x}) = p(\mathbf{y}|\mathbf{x}). \tag{1.11}$$

By substituting this conditional independency relationship into (1.10), we obtain the more structured form of the joint distribution given in (1.8).

The general idea is to express the joint distribution as a product of factors, where each factor depends on a subset of the random variables. In the simplest case, each factor depends on a single random variable, making marginalization easy. Most distributions that describe practical problems cannot be broken up in this way, and the subsets overlap. Within this richer set of models, some structures lead to highly efficient exact algorithms (*e.g.*, the forward-backward algorithm for a chain-type structure). Other structures are not tractable and lead to approximate algorithms.

It turns out that graph theory provides a succinct way to represent probabilistic structure. A graphical representation for probabilistic structure, along with functions that can be used to derive the joint distribution, is called a *graphical model*. Examples of graphical models include *Markov random fields* [Kinderman and Snell 1980], *Bayesian networks* [Pearl 1988], and *chain graphs* [Lauritzen and Wermuth 1989]. Here, I consider Bayesian networks.

1.2.2 Definition of a Bayesian network

The conditional independency relationships for a distribution can be described graphically. Not only does the graphical representation concisely capture probabilistic structure, but it forms a framework for computing useful probabilities. Bayesian networks are specified in terms of *directed acyclic graphs*, in which all edges are directed and in which there are no closed paths when edge directions are followed. A *Bayesian network* for a set of random variables $\mathbf{z} = (z_1, z_2, \dots, z_N)$ consists of a directed acyclic graph with one vertex for each variable, and a set of probability functions $P(z_k|\mathbf{a}_k)$, $k = 1, \dots, N$, where the *parents* $\mathbf{a_k}$ of z_k are the variables that have directed edges connecting *to* z_k. (For simplicity of prose, I will often refer to a vertex by its variable name.) If z_k has no parents, then $\mathbf{a}_k = \emptyset$. For now, we can think of each function $P(z_k|\mathbf{a}_k)$ as an exhaustive list of probabilities corresponding to the possible configurations of z_k and $\mathbf{a}_k$. (In the case of a density $p(z_k|\mathbf{a}_k)$, the entire density function must be specified.) Together, the graph and the probability functions are referred to as the *network specification*.

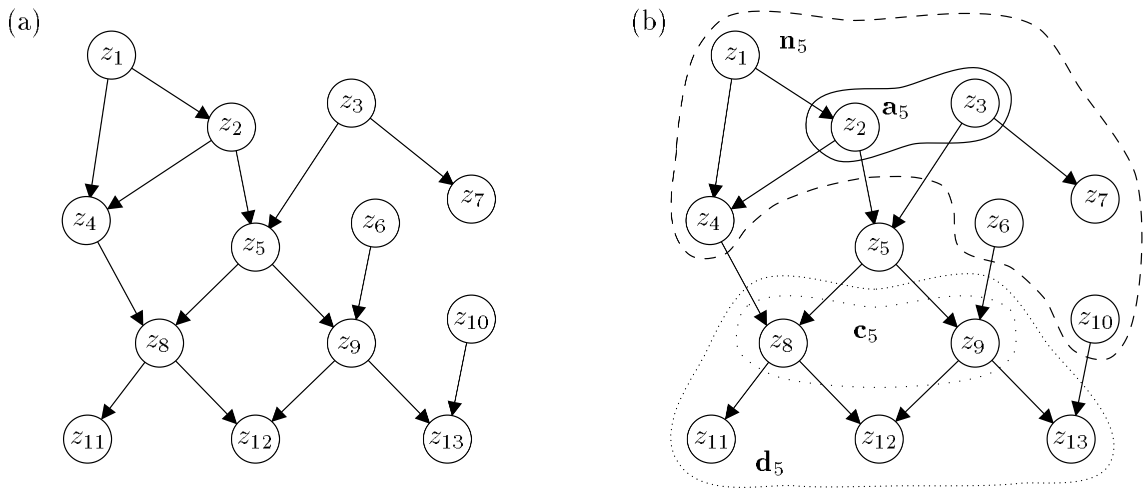

Figure 1.1: (a) An example of a Bayesian network. (b) The parents $\mathbf{a}_5$ of z_5 are shown by a solid loop; the children $\mathbf{c}_5$ are shown by a sparse dotted loop; the descendents $\mathbf{d}_5$ are shown by a dense dotted loop; the nondescendents $\mathbf{n}_5$ are shown by a dashed loop.

Several definitions will help to understand how a Bayesian network describes the probabilistic structure for a joint distribution $P(\mathbf{z})$. The *children* $\mathbf{c}_k$ of z_k are the variables that have directed edges connecting *from* z_k. The *descendents* $\mathbf{d}_k$ of z_k are its children, its children's children, *etc.* The *nondescendents* $\mathbf{n}_k$ of z_k are the variables in $\{z_1, z_2, \dots, z_{k-1}, z_{k+1}, \dots, z_N\}$ that are not in $\mathbf{d}_k$, *i.e.*, $\mathbf{n}_k = \mathbf{z} \setminus (\mathbf{d}_k \cup \{z_k\})$. Note that $\mathbf{n}_k \neq \mathbf{z} \setminus \mathbf{d}_k$, since z_k is *not* included in the nondescendents. From these definitions, it follows that $\mathbf{a}_k \subseteq \mathbf{n}_k$.

Figure 1.1 shows an example of a Bayesian network, along with the parents, children, descendents and nondescendents of variable z_5.

The meaning of a Bayesian network is that given the parents of z_k, the distribution over z_k will not change if any combination of the nondescendents of z_k are also given:

$$P(z_k|\mathbf{a}_k, \mathbf{w}) = P(z_k|\mathbf{a}_k), \quad \forall\ \mathbf{w} \subseteq \mathbf{n}_k. \tag{1.12}$$

In other words, z_k is conditionally independent of any combination of its nondescendents, given its parents. To take the family hierarchy (not necessarily a tree) analogy further, given the genetic code of Susan's parents, determining the genes of her siblings, her grandparents, her grandparents' children, her children's other parents or any combination of the above does not influence our prediction of Susan's genetic make-up. This is not true for descendents. For example, determining the genes of Susan's grandchildren *does* influence our prediction of her genetic make-up, even though determining the genes of those parents of Susan's grandchildren who are not Susan's children *does not* (notice that the latter are nondescendents).[2]

The joint distribution for a Bayesian network can be written in a structured form simply by writing out the product of the distributions for individual variables, where each variable is conditioned on its parents:

$$P(\mathbf{z}) = \prod_{k=1}^{N} P(z_k|\mathbf{a}_k). \tag{1.13}$$

This form follows from (1.12) in the following way. Since a Bayesian network contains no directed cycles, it is always possible to choose an *ancestral ordering* $z_{\pi(1)}, z_{\pi(2)}, \ldots, z_{\pi(N)}$, where $\pi(\cdot)$ is a permutation map, so that the descendents of each variable come later in the ordering: $\mathbf{d}_{\pi(k)} \subseteq \{z_{\pi(k+1)}, \ldots, z_{\pi(N)}\}$. For example, the variables in the network shown in Figure 1.1 were assigned so that $z_1, z_2, \ldots, z_{13}$ is an ancestral ordering. Using the general chain rule of probability applied to the ancestral ordering, we have

$$P(\mathbf{z}) = \prod_{k=1}^{N} P(z_{\pi(k)}|z_{\pi(1)}, \ldots, z_{\pi(k-1)}). \tag{1.14}$$

From the definition of the ancestral ordering, it follows that the set of variables that precede $z_{\pi(k)}$ is a subset of its nondescendents and that the parents of $z_{\pi(k)}$ are a subset of the variables that precede $z_{\pi(k)}$:

$$\mathbf{a}_{\pi(k)} \subseteq \{z_{\pi(1)}, \ldots, z_{\pi(k-1)}\} \subseteq \mathbf{n}_{\pi(k)}. \tag{1.15}$$

For this reason, $\{z_{\pi(1)}, \ldots, z_{\pi(k-1)}\} = \mathbf{a}_{\pi(k)} \cup (\{z_{\pi(1)}, \ldots, z_{\pi(k-1)}\} \setminus \mathbf{a}_{\pi(k)})$, and taking $\mathbf{w} = \{z_{\pi(1)}, \ldots, z_{\pi(k-1)}\} \setminus \mathbf{a}_{\pi(k)}$ in (1.12), we have

$$\begin{aligned} P(z_{\pi(k)}|z_{\pi(1)}, \ldots, z_{\pi(k-1)}) &= P(z_{\pi(k)}|\mathbf{a}_{\pi(k)}, \{z_{\pi(1)}, \ldots, z_{\pi(k-1)}\} \setminus \mathbf{a}_{\pi(k)}) \\ &= P(z_{\pi(k)}|\mathbf{a}_{\pi(k)}). \end{aligned} \tag{1.16}$$

Inserting this result into (1.14), we obtain the form given in (1.13).

If the probability functions for a Bayesian network are not specified, the network is meant to represent *all* distributions that can be written in the form given in (1.13). For the network with ancestral ordering $z_1, z_2, \ldots, z_{13}$ shown in Figure 1.1, (1.13) gives a joint

[2] Interestingly, if we have previously determined the genes of Susan's grandchildren, then determining the genes of those parents of Susan's grandchildren who are not Susan's children *does* influence our prediction of Susan's genetic make-up. See Section 1.2.4 for more details.

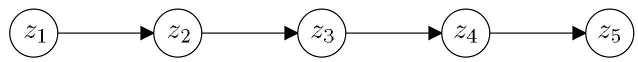

Figure 1.2: An example of a chain-type Bayesian network, or a *Markov chain*.

distribution

$$\begin{aligned} P(\mathbf{z}) &= P(z_1)P(z_2|z_1)P(z_3)P(z_4|z_1,z_2)P(z_5|z_2,z_3)P(z_6)P(z_7|z_3) \\ &\quad \cdot P(z_8|z_4,z_5)P(z_9|z_5,z_6)P(z_{10})P(z_{11}|z_8)P(z_{12}|z_8,z_9)P(z_{13}|z_9,z_{10}). \end{aligned} \tag{1.17}$$

This product could have been written in any order, but using an ancestral ordering helps clarify the dependencies. In this equation, each variable z_k is conditioned on variables whose distributions appear to the *left* of the distribution for z_k. Note that a Bayesian network may have more than one ancestral ordering. In this case, $z_{10}, z_6, z_3, z_7,\ z_1, z_2, z_4, z_5, z_8, z_{11}, z_9, z_{13}, z_{12}$ is also an ancestral ordering.

An interesting special case of a Bayesian network is the chain-type network shown in Figure 1.2, also known as a first-order *Markov chain*. Applying (1.13) to this network, we obtain

$$P(\mathbf{z}) = P(z_1)P(z_2|z_1)P(z_3|z_2)P(z_4|z_3)P(z_5|z_4). \tag{1.18}$$

This type of structure is frequently used to model time series data, where it is often assumed that the next state of a physical system depends only on the previous state. Comparing this network to the more complex networks that appear later in this thesis, the Bayesian network can be thought of as a generalization of the Markov chain.

1.2.3 Ancestral simulation

It is often practically impossible to simulate vectors $\mathbf{z}$ that are distributed according to $P(\mathbf{z})$. However, if the joint distribution can be described by a Bayesian network, and if a value for each z_k can be drawn from its conditional probability $P(z_k|\mathbf{a}_k)$ in a practical manner, then the ancestral ordering can be used to draw an entire vector. Starting with $k = 1$, we draw $z_{\pi(1)}$ from $P(z_{\pi(1)})$. We continue to draw $z_{\pi(k)}$ from $P(z_{\pi(k)}|\mathbf{a}_{\pi(k)})$ for $k = 2, \ldots, N$ until an entire vector $\mathbf{z}$ has been drawn. In this way, the probabilistic structure implied by the graph allows us to decompose the simulation problem into local pieces.

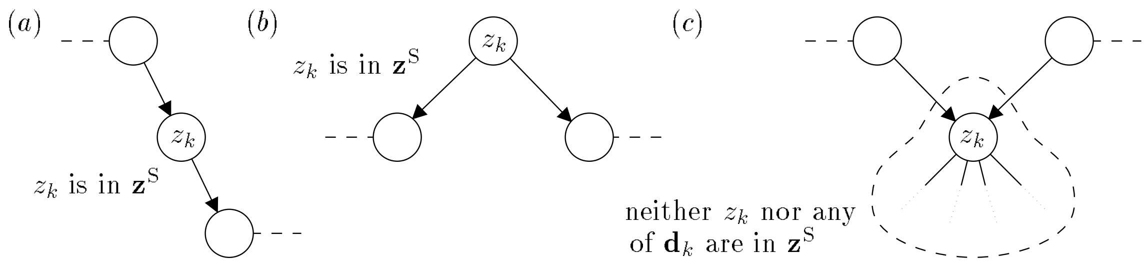

Figure 1.3: The three ways in which a path may be blocked. If all paths from all variables in $\mathbf{z}^{\mathrm{A}}$ to all variables in $\mathbf{z}^{\mathrm{B}}$ are blocked, then $\mathbf{z}^{\mathrm{A}}$ and $\mathbf{z}^{\mathrm{B}}$ are dependency-separated by $\mathbf{z}^{\mathrm{S}}$.

1.2.4 Dependency separation

In Section 1.2.2, I used the notion of conditional independence to define a Bayesian network. In particular, a Bayesian network implies that variable z_k is conditionally independent of any subset of its nondescendents, given its parents. This is expressed mathematically by (1.12). A convenient way to describe this scenario is to say that "z_k is dependency-separated from any combination of its nondescendents by its parents".

Consider the uncountable set of distributions $\mathcal{P}$ that can be described by a given Bayesian network. In general, I will say that $\mathbf{z}^{\mathrm{A}}$ is dependency-separated from $\mathbf{z}^{\mathrm{B}}$ by $\mathbf{z}^{\mathrm{S}}$ ("S" for separation), if and only if

$$P(\mathbf{z}^{\mathrm{A}}|\mathbf{z}^{\mathrm{S}}, \mathbf{z}^{\mathrm{B}}) = P(\mathbf{z}^{\mathrm{A}}|\mathbf{z}^{\mathrm{S}}), \quad \text{or, equivalently} \quad P(\mathbf{z}^{\mathrm{A}}, \mathbf{z}^{\mathrm{B}}|\mathbf{z}^{\mathrm{S}}) = P(\mathbf{z}^{\mathrm{A}}|\mathbf{z}^{\mathrm{S}})P(\mathbf{z}^{\mathrm{B}}|\mathbf{z}^{\mathrm{S}}), \tag{1.19}$$

for all $P \in \mathcal{P}$. (See [Pearl 1988] for an extensive discussion of dependency-separation.) Notice that dependency-separation is symmetric with respect to $\mathbf{z}^{\mathrm{A}}$ and $\mathbf{z}^{\mathrm{B}}$. The case of dependency-separation that I used to define a Bayesian network is special, in that $\mathbf{z}^{\mathrm{S}}$ was the set of parents of the single variable $\mathbf{z}^{\mathrm{A}}$, and $\mathbf{z}^{\mathrm{B}}$ was a subset of the nondescendents of $\mathbf{z}^{\mathrm{A}}$.

It is possible to ascertain dependency separation in the general case simply by inspecting the Bayesian network. If $\mathbf{z}^{\mathrm{A}}$, $\mathbf{z}^{\mathrm{B}}$, and $\mathbf{z}^{\mathrm{S}}$ are three disjoint subsets of $\mathbf{z}$, then $\mathbf{z}^{\mathrm{S}}$ dependency-separates $\mathbf{z}^{\mathrm{A}}$ from $\mathbf{z}^{\mathrm{B}}$ if, in every path connecting any variable in $\mathbf{z}^{\mathrm{A}}$ to any variable in $\mathbf{z}^{\mathrm{B}}$, there is at least one variable z_k that satisfies one or more of the following three conditions:

1. z_k acts as both a parent and a child in the path and $z_k \in \mathbf{z}^{\mathrm{S}}$ (Figure 1.3a), or

2. z_k acts as the parent of two variables in the path and $z_k \in \mathbf{z}^{\mathrm{S}}$ (Figure 1.3b), or

3. z_k acts as the child of two variables in the path and neither z_k nor any of its descendents

are in $\mathbf{z}^{\mathrm{S}}$ (Figure 1.3c).

(Note that the identification of a path does not depend on edge directions.) A path for which one of these conditions is met is said to be *blocked*. In order to ascertain dependency-separation, we need to consider only paths that do not intersect themselves, since those conditions that hold for any given path will also hold for the path when extra variables that form a loop are considered.

For example, in Figure 1.1a we have the following dependency separation relationships. z_6 is dependency-separated from z_{10} (by nothing), since the sole path z_6, z_9, z_{13}, z_{10} is blocked by z_{13} in condition 3. What if z_9 is observed? Then, z_6 is still dependency-separated from z_{10}, by condition 1 applied to the sole path z_6, z_9, z_{13}, z_{10}. In contrast, if only z_{13} is observed, then z_6 is *not* dependency-separated from z_{10}, since there exists a path z_6, z_9, z_{13}, z_{10} for which none of the three conditions can be met. This means that once z_{13} is observed, z_6 and z_{10} *may* become dependent. Note that there *may* exist a distribution in $\mathcal{P}$ where z_6 and z_{10} are *independent* given z_{13}, but there exists at least one distribution in $\mathcal{P}$ where z_6 and z_{10} are dependent given z_{13}.

Here are some more complicated examples. z_2 is dependency-separated from z_9 by z_5, since path z_2, z_5, z_9 is blocked by z_5 in condition 1, paths $z_2, z_4, z_8, z_{12}, z_9$ and $z_2, z_1, z_4, z_8, z_{12}, z_9$ are blocked by z_{12} in condition 3, paths z_2, z_4, z_8, z_5, z_9 and $z_2, z_1, z_4, z_8, z_5, z_9$ are blocked by z_5 in condition 2 *and* by z_8 in condition 3, path $z_2, z_5, z_8, z_{12}, z_9$ is blocked by z_5 in condition 1 *and* by z_{12} in condition 3. z_2 is dependency-separated from $\{z_3, z_7\}$ (by nothing), since the paths z_2, z_5, z_3 and z_2, z_5, z_3, z_7 are blocked by z_5 in condition 3. This means that in the absence of observations z_2 and $\{z_3, z_7\}$ are independent. z_2 is not dependency-separated from $\{z_3, z_7\}$ by z_{12}, since there exists a path z_2, z_5, z_3 for which none of the conditions can be met. Condition 3 applied to z_5 fails, because z_{12} is a descendent of z_5. This means that once z_{12} is observed, z_2 and $\{z_3, z_7\}$ may *become* dependent.

1.2.5 Example 1: Recursive convolutional codes and turbo-codes

Recall from Section 1.1.3 that the purpose of channel coding is to communicate over a noisy channel in an error-free (or nearly error-free) fashion. To do this, we encode a given binary information vector $\mathbf{u}$ as a longer codeword vector $\mathbf{x}$, which contains extra bits whose purpose is to "protect" the information from the channel noise. (An example is a repetition code, where each information bit is simply transmitted several times.) The codeword is converted to a physical form (*e.g.*, radio waves) and then sent over a channel. A vector of noisy signals $\mathbf{y}$ is received at the output of the channel. Given $\mathbf{y}$, the decoder must make a guess $\hat{\mathbf{u}}$ at what the original $\mathbf{u}$ was.

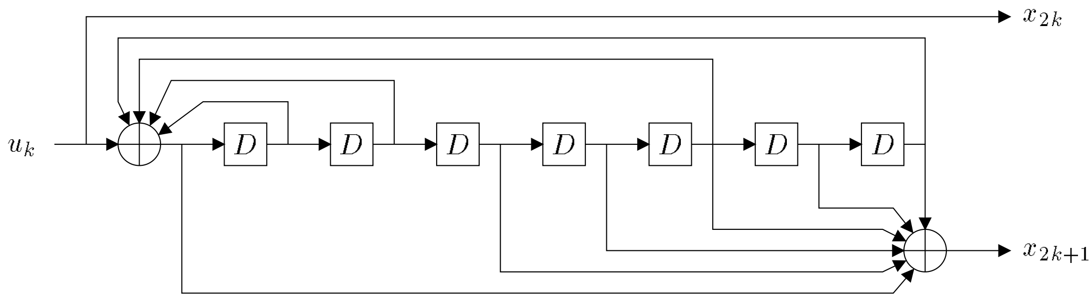

Figure 1.4: The LFSR for a systematic recursive convolutional code that has a minimum distance of 10.

One very popular class of channel codes (you probably have one of these in your telephone modem) can be described using Bayesian networks. The encoder for a *recursive convolutional code* is simply a linear feedback shift register (LFSR) that takes in information bits and generates codeword bits. See [Lin and Costello 1983] for an extensive treatment of convolutional codes. Figure 1.4 shows the LFSR for the convolutional code that is described below. Each box represents a 1-bit memory element and D indicates a delay buffer. The discs represent addition modulo 2 (XOR). For this particular convolutional code, every second output is actually just a copy of the input bit. This type of code is called *systematic*. Notice that for each input bit, two output bits are produced, so this is a rate 1/2 code. If there are K information bits, then there will be $N = 2K$ codeword bits. The device shown in Figure 1.4 is called a *linear* feedback shift register because the output sequence generated by the sum of two input sequences is equal to the sum of the two output sequences that are generated by the individual input sequences (where summation is modulo 2). The details of how to choose the feedback delay taps and the output taps in order to produce a good code can be found in [Lin and Costello 1983; Berrou and Glavieux 1996]. However, the operation of an encoder of this type is quite simple. The LFSR is initialized so that all memory elements contain 0's. Then, the information bits u_k are fed into the LFSR, producing codeword bits x_k. Signals that represent the codeword bits are then transmitted over the channel. For example, on a twisted pair of wires, we might apply +1 volts if $x_k = 1$ and −1 volts if $x_k = 0$.

Figure 1.5a shows the Bayesian network for a recursive systematic convolutional code. Normally, the number of information bits K is much larger than 6 (typical numbers range from 100 to 100,000 bits). s_k is the state of the LFSR at time k, extended to include the input bit (this makes the network simpler). To fully specify the Bayesian netowork, we must also provide the conditional distributions. Assuming the information bits are uniformly distributed,

$$P(u_k) = 0.5, \text{ for } u_k \in \{0, 1\}. \tag{1.20}$$

Let $S(s_{k-1}, u_k)$ be a function (determined from the LFSR) that maps the previous state and current input to the next state, and let $g(s_k)$ be a function that maps the state to the nonsystematic output bit. Then, the deterministic conditional distributions for the states and state outputs are

$$\begin{aligned} P(s_0|u_0) &= \delta(s_0, S(0, u_0)) \\ P(s_k|u_k, s_{k-1}) &= \delta(s_k, S(s_{k-1}, u_k)) \\ P(x_{2k}|u_k) &= \delta(x_{2k}, u_k) \\ P(x_{2k+1}|s_k) &= \delta(x_{2k+1}, g(s_k)), \end{aligned} \tag{1.21}$$

where $\delta(a, b) = 1$ if $a = b$ and 0 otherwise. Assuming that the channel simply adds independent Gaussian noise with variance σ^2 to the +1/-1 signals described above, the conditional distributions for the received channel output signals are

$$p(y_k|x_k) = \begin{cases} \frac{1}{\sqrt{2\pi\sigma^2}} e^{-(y_k - 1.0)^2/2\sigma^2} & \text{if } x_k = 1 \\ \frac{1}{\sqrt{2\pi\sigma^2}} e^{-(y_k + 1.0)^2/2\sigma^2} & \text{if } x_k = 0. \end{cases} \tag{1.22}$$

Given an information vector $\mathbf{u}$, encoding and channel transmission can be simulated by one sweep of ancestral simulation. For example, we first directly copy u_0 into x_0, which is then used to draw a noisy channel output value y_0. Then, we use u_0 to determine s_0, which is then used to determine x_1, which is then used to draw a noisy channel output value y_1. Then, we directly copy u_1 into x_2 and so on until the entire channel output vector $\mathbf{y}$ has been obtained.

The decoder sees only the vector $\mathbf{y}$, and ideally would infer the most likely value of each information bit, *i.e.*, determine for each k the u_k that maximizes $P(u_k|\mathbf{y})$. In general such a probabilistic inference is very difficult, but if we take advantage of the graphical structure of the code it turns out it can be done quite easily. In fact, it is possible to compute $P(u_k|\mathbf{y})$ $k = 0, \ldots, K-1$ *exactly* using the forward-backward (a.k.a. BCJR) algorithm [Baum and Petrie 1966; Bahl et al. 1974], which is just a special case of the general *probability propagation* algorithm discussed in Section 2.1. Once the block is decoded, we can compare the decoded information bit values with the true ones to determine the number of bit errors made for the block transmission. If we simulate the transmission of

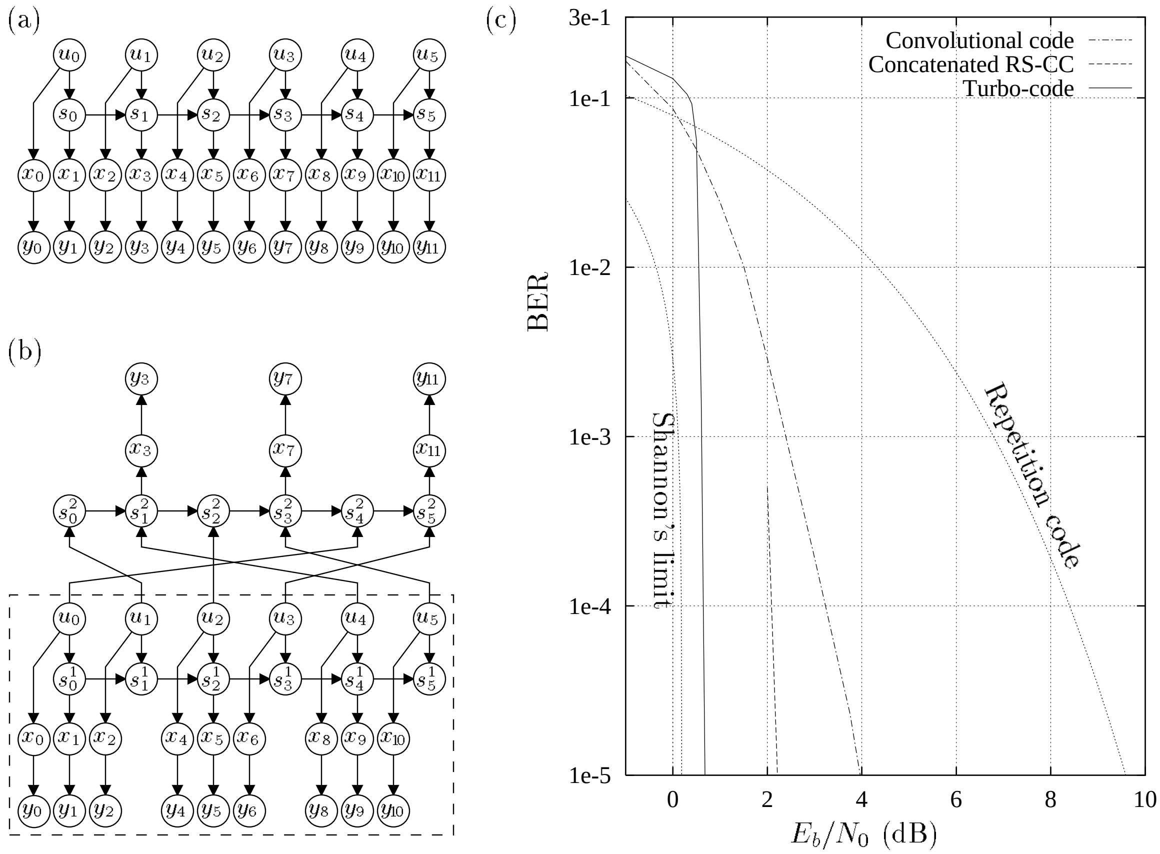

Figure 1.5: (a) The Bayesian network for a recursive systematic convolutional code. (b) The Bayesian network for an example of the recently proposed turbo-code. (c) The bit error rate (BER) performance for instances of these two codes.

many blocks, we can obtain an estimate of the bit error rate (BER).

This procedure was carried out using the convolutional code shown in Figure 1.4. This recursive systematic convolutional code was designed to maximize the minimum Hamming distance between all pairs of codewords [Viterbi and Omura 1979; Lin and Costello 1983] ($d_{\text{min}} = 10$). The information vector length was $K = 5000$ (giving a codeword length of $N = 10000$), and 5000 vectors (25×10^6 information bits in all) were transmitted for a fixed noise variance. It is common practice to give BER results as a function of the noise level measured by a signal-to-noise ratio E_b/N_0 in decibels. For any system with $N = 2K$ and transmission power (variance) of unity, E_b/N_0 is related to σ^2 by $E_b/N_0 = -10 \log_{10} \sigma^2$. Figure 1.5c shows the BER as a function of E_b/N_0 for this recursive systematic convolutional code[3]. Notice that as E_b/N_0 increases (σ^2 decreases), the BER drops.

[3] A technical detail: Trellis termination was used to improve the performance of the code.

In the same figure, I also give the BER curve for a simple repetition code, where each information bit is transmitted twice, maintaining $N = 2K$. If the information bit is 0, a pair of -1's are sent; if the information bit is 1, a pair of +1's are sent. Each pair of received noisy signals is then averaged before a threshold of 0.0 is applied to detect the information bit. The curve on the far left shows Shannon's limit; for a given E_b/N_0, it is impossible to communicate with a BER below this curve. (See Sections 5.1.6 and 5.1.7 for a derivation of this curve.) So, systems of practical interest give performance points that lie between the Shannon limit curve and the curve for the repetition code. Performance points to the left of the Shannon limit are impossible, and performance points to the right of the curve for the repetition code are not of practical interest.

Recently, a code and decoding algorithm were discovered that give unprecedented BER performance. It turns out that the *turbo-decoding* algorithm for these *turbo-codes* [Berrou and Glavieux 1996] is just the probability propagation algorithm discussed in Section 2.1 applied to a code network like the one shown in Figure 1.5b [Frey and Kschischang 1996; Kschischang and Frey 1997; MacKay, McEliece and Cheng 1997]. This Bayesian network contains two recursive convolutional code networks that are connected to the information bits in different ways. The information bits feed directly into one of the chains ($\mathbf{s}^1$), but feed into the second chain ($\mathbf{s}^2$) in a *permuted order* as shown. In order to produce the same number of codeword bits per codeword as would be produced by the recursive systematic convolutional encoder described above, every second output of each LFSR is alternately not transmitted (a procedure called *puncturing*).

Figure 1.5c shows the BER performance for a turbo-code system with $K = 65,536$ and $N = 131,072$. 530 vectors ($\sim 35 \times 10^6$ information bits) were transmitted to determine the BER for each noise level. Each of the two LFSRs had 4 bits of memory and used identical feedback and output delay taps. All four delayed bits were fed back to the input of the LFSR. Only the bit entering the first delay element and the most-delayed bit were fed forward to the output. (This block length and these constituent LFSR's were proposed in [Berrou, Glavieux and Thitimajshima 1993]). The decoding complexity per information bit for the turbo-code was roughly twice that for the convolutional code described above. The information bit permuter was chosen at random. The turbo-code system clearly outperforms the computationally comparable single convolutional code system. At a BER of 10^{-5}, the turbo-code system is tolerant to 3.3 dB more noise than the single convolutional code system, and is only 0.5 dB from the Shannon limit. Also shown on this graph is the performance of a concatenated Reed-Solomon convolutional code described in [Lin and Costello 1983], which had been considered to be the best practical code until the proposal of turbo-codes. The turbo-code system is tolerant to 1.5 dB more noise than the concatenated system.

In Chapter 5, I explore some of the exciting new applications of Bayesian networks to channel coding problems, with a focus on using the probability propagation algorithm discussed in Section 2.1 for inference.

1.2.6 Parameterized Bayesian networks

It is sometimes convenient to represent the conditional distributions $P(z_k|\mathbf{a}_k)$ in parametric form. That is, the distribution over z_k given its parents $\mathbf{a}_k$ is specified not by an exhaustive list of probability masses, but by a function of z_k, $\mathbf{a}_k$, and a set of parameters $\boldsymbol{\theta}_k$. (The subscript k indicates that $\boldsymbol{\theta}_k$ is a set of parameters associated with z_k.) In this case, we write the conditional distribution as $P(z_k|\mathbf{a}_k, \boldsymbol{\theta}_k)$. The total set of parameters is $\boldsymbol{\theta} = \{\boldsymbol{\theta}_1, \ldots, \boldsymbol{\theta}_N\}$, and the parameterized joint distribution is expressed as $P(\mathbf{z}|\boldsymbol{\theta})$. Such a parametric form can be useful in applications such as density estimation, pattern classification, and data compression, where the distribution $P(\mathbf{z}|\boldsymbol{\theta})$ is to be estimated from a data set. In this case, the parametric form can act as a regularizer. Since the number of possible configurations of each z_k and $\mathbf{a}_k$ is usually quite large, we would need an extremely large data set to estimate all probabilities accurately. Using the parametric form, however, we need only estimate each parameter. As described in Section 2.3, a parametric form is also useful when formulating variational inference algorithms.

A common parametric Bayesian network is the sigmoidal Bayesian network [Neal 1992; Jordan 1995; Saul, Jaakkola and Jordan 1996], whose random variables are all binary. The conditional probability function $P(z_k|\mathbf{a}_k, \boldsymbol{\theta}_k)$ can be viewed as a regression model that is meant to predict z_k from a set of attributes $\mathbf{a}_k$. A standard statistical method for predicting a binary-valued variable is *logistic regression* [McCullagh and Nelder 1983], in which the conditional probability for z_k given $\mathbf{a}_k$ is

$$P(z_k|\mathbf{a}_k, \boldsymbol{\theta}_k) = \begin{cases} 1/(1+\exp[-\theta_{k0} - \sum_{\forall j: z_j \in \mathbf{a}_k} \theta_{kj} z_j]) & \text{if } z_k = 1, \\ 1 - 1/(1+\exp[-\theta_{k0} - \sum_{\forall j: z_j \in \mathbf{a}_k} \theta_{kj} z_j]) & \text{if } z_k = 0, \end{cases} \tag{1.23}$$

where the parameter θ_{k0} represents a constant *bias* in the exponent. The logistic function $g(x) = 1/(1+\exp[-x])$ is used to restrict the probability to lie between 0 and 1. (This function is shown in Figure 1.6.) In terms of log-odds,

$$\log \frac{P(z_k = 1|\mathbf{a}_k, \boldsymbol{\theta}_k)}{P(z_k = 0|\mathbf{a}_k, \boldsymbol{\theta}_k)} = \theta_{k0} + \sum_{\forall j: z_j \in \mathbf{a}_k} \theta_{kj} z_j, \tag{1.24}$$

which shows how each parent $z_j \in \mathbf{a}_k$ independently increases or decreases the log-odds for z_k, depending on the sign of θ_{kj}.

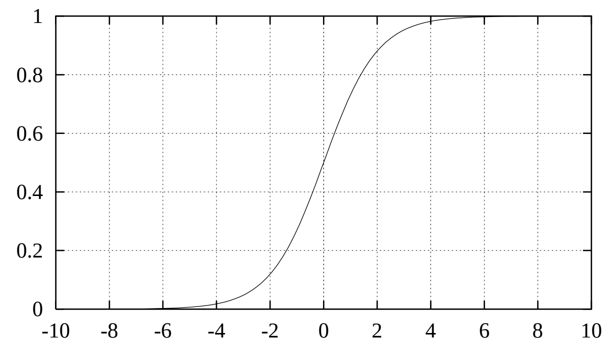

Figure 1.6: The logistic function $g(x) = 1/(1 + \exp[-x])$.

Sometimes, for the sake of notational simplicity, I will assume that the set of parents for each variable z_k is specified by some parameter constraints. Assume without loss of generality that for a given network, the random variables $\mathbf{z}$ have an ancestral ordering $z_1, z_2, \ldots, z_N$. I take $P(z_k|\mathbf{a}_k, \boldsymbol{\theta}_k) = P(z_k|\{z_j\}_{j=1}^{k-1}, \boldsymbol{\theta}_k)$, where in the second expression the parameters are constrained so that the function does not depend on nonparents. Also, in order to succinctly account for the bias, I will usually assume that there is a dummy variable z_0 that is set to $z_0 = 1$. (Thus the notation θ_{k0} for the bias in the summations above.) Using these notational simplifications and using $g(\cdot)$ for the logistic function, the sigmoidal model described above can be written

$$P(z_k|\{z_j\}_{j=1}^{k-1}, \boldsymbol{\theta}_k) = z_k g(\textstyle\sum_{j=0}^{k-1} \theta_{kj} z_j) + (1 - z_k)(1 - g(\textstyle\sum_{j=0}^{k-1} \theta_{kj} z_j)), \tag{1.25}$$

where θ_{kj} is set to 0 for each nonparent z_j.

1.2.7 Example 2: The bars problem

Bayesian networks provide a useful framework for specifying *generative models*. A generative model can be used to generate data vectors that exhibit interesting structure. The generative models discussed in this thesis can also be used for pattern classification and data compression, in the fashion described in Sections 1.1.1 and 1.1.2. If the Bayesian network is parameterized, we can estimate the parameters of the network from a training set by making the generative distribution "close" (say, in the Kullback-Leibler pseudo-distance) to the training set distribution. We hope that in this fashion, we can extract the "true" underlying generative process, or at least one that is equally efficient at describing the data.

For example, the 4×4 binary images shown in Figure 1.7a were generated by first selecting an orientation (horizontal or vertical) with equal probability, and then randomly

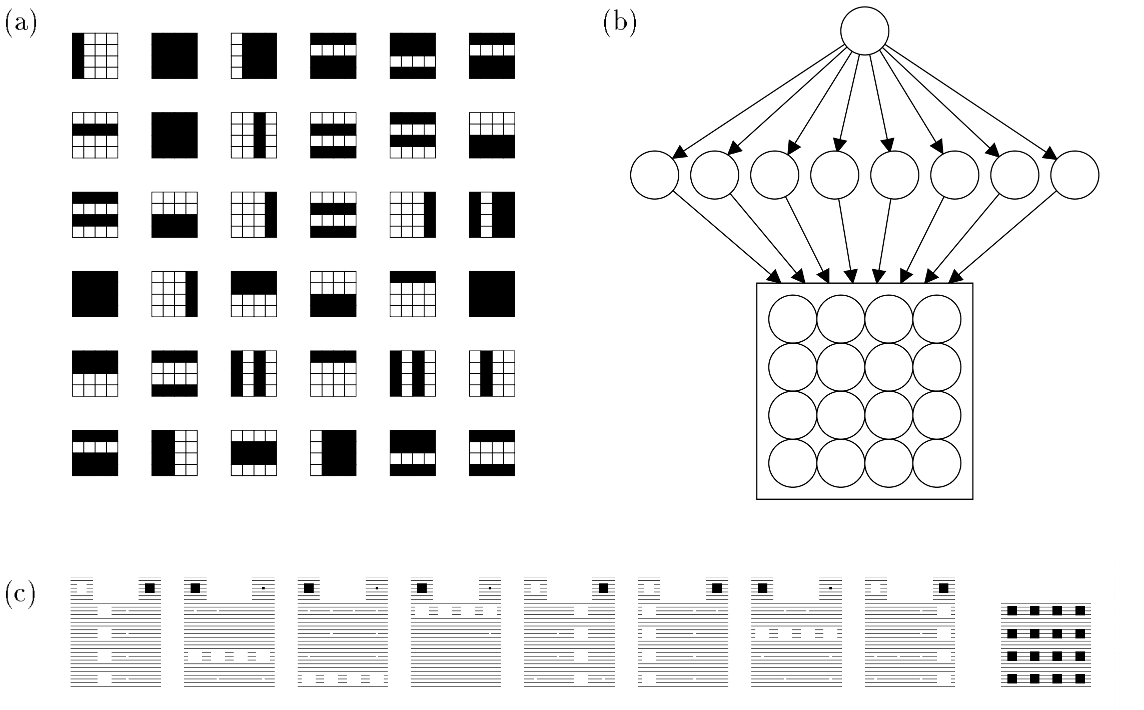

Figure 1.7: (a) Examples of training images typical of the "bars problem". (b) The graph for a parameterized Bayesian network that was estimated from a large training set using the wake-sleep algorithm. Edges that terminate on the box are connected to all vertices within the box. (c) The parameters of the Bayesian network clearly show that the network has learned the notion of horizontal and vertical bars (see the main text for a more complete description).

instantiating each of the four possible bars with that orientation with probability 0.5. (The all-on images were removed from the training set, since the orientation of the bars in an all-on image is ambiguous.) Helmholtz machine and the wake-sleep algorithm (described in Sections 2.4 and 3.4.3), I fit the parameterized network shown in Figure 1.7b to a large training set of 2×10^6 images produced in this way. The network has three layers of binary variables: 1 in the top layer, 8 in the middle layer, and 16 in the visible layer (the image). The variables in adjacent layers are fully-connected, and the conditional distributions are modelled using logistic regression, as described in the previous section. After parameter estimation (see [Hinton et al. 1995] for details), ancestral simulation of the network produces output images that are indistinguishable from the training images.

After learning, the bias for the top-layer variable is nearly zero, so that under the joint distribution it has the value 1 as often as it has the value 0. The values of the other parameters are depicted in Figure 1.7c. The eight large blocks on the left show the parameters associated with the connections that feed into and out of the middle-layer variables. The bias for a variable is shown by the small black or white square on the top right of the block for each middle-layer variable. Positive parameters are white, negative parameters are black, and the area of the square is proportional to the magnitude of the parameter. (The largest parameter shown in the figure is 14.1.) The parameter associated with the connection from the top-layer variable to a middle-layer variable is shown by the small square on the top left of the block for each middle-layer variable. Finally, the parameters associated with the connections from a middle-layer variable to the visible variables are shown by the 4×4 grid of squares in the block for each middle-layer variable. The biases for the 16 visible variables are shown by the 4×4 grid of squares on the far right of Figure 1.7c.

It is clear from these parameters that each middle-layer variable represents the presence (value of 1) or absence (value of 0) of a particular horizontal or vertical bar. If the top-layer variable is 1, the probability that a horizontal bar is present is nearly zero, since the biases for these variables are nearly zero and the parameters that connect these variables to the top-layer variable are large and negative. On the other hand, if the top-layer variable is 0, the probability that a horizontal bar is present is 0.5. In this way, the network captures the true generative model that produced the training data.

In Chapters 3 and 4, I show how Monte Carlo inference, variational inference, and Helmholtz machines can be used to fit Bayesian networks to training data for the purposes of pattern classification, unsupervised learning, and data compression.

1.3 Organization of this thesis

In the remainder of this thesis, I use Bayesian networks as a platform to develop algorithms for pattern classification, data compression, and channel coding. The last of these problems is quite different from the former two, since we will usually *design* an error-correcting code using a Bayesian network and then use probabilistic inference to perform decoding. On the other hand, for pattern classification and data compression, we will usually *estimate* a parameterized Bayesian network from some training data and then use probabilistic inference to classify a new pattern or produce a source codeword for a new pattern.

In Chapter 2, I discuss different ways to perform probabilistic inference, including probability propagation, Markov chain Monte Carlo, variational optimization, and the Helmholtz machine.

Several types of Bayesian networks that are suitable for pattern classification are presented in Chapter 3. I show how Markov chain Monte Carlo, variational optimization, and the Helmholtz machine wake-sleep algorithm can be used for probabilistic inference and parameter estimation in these networks. Based on a digit classification problem, I compare the performances of these systems with several standard algorithms, including the k-nearest neighbor method and classification and regression trees (CART). Learning to extract structure from data *without* using a supervised signal such as class identity is another interesting parameter estimation problem. At the end of this chapter, I examine unsupervised learning in Bayesian networks that have binary-valued and real-valued variables.

In Chapter 4, I consider the problem of how to efficiently compress data using Bayesian networks with hidden variables. When there are hidden variables, a Bayesian network may assign many source codewords of similar length to a particular input pattern. I present the "bits-back" coding algorithm that can be used to efficiently communicate patterns, despite this redundancy in the source code.

In Chapter 5, I present several published error-correcting codes in terms of Bayesian networks and show that their corresponding iterative decoding algorithms can be derived as special cases of probability propagation. In particular, the recently proposed turbo-decoding algorithm, which brought researchers a leap closer to (and almost up against) the Shannon limit, is an instance of probability propagation. Motivated by these results and the breadth in perspective offered by Bayesian networks, I present a new class of "interleaved trellis-constraint codes", which when iteratively decoded are competitive with iteratively decoded turbo-codes. I also present two approaches for speeding up a popular class of computationally burdensome iterative decoding algorithms.

Chapter 2

Probabilistic Inference in Bayesian Networks

In this chapter, I discuss methods for probabilistic inference that make use of the Bayesian network description of the joint distribution. Many readers may be aware of how probabilistic inference in a Markov chain is simplified by a chain-type graphical structure. A generalized form of this simplification holds for those Bayesian networks that have only a single path (when edge directions are ignored) between any two vertices. In Section 2.1, I review an algorithm for "probability propagation", which can be used to infer the *exact* distributions over individual variables or small groups of variables in such networks. For networks that have multiple paths between one or more pairs of vertices, this algorithm is not exact. Although there are procedures for attempting to convert an original network to one that is appropriate for probability propagation [Spiegelhalter 1986; Lauritzen and Spiegelhalter 1988], these procedures are not practically fruitful when the number of multiple paths is large. In these cases, approximate inference methods must be used. In Section 2.2, I discuss a Monte Carlo approach to inference, where we attempt to produce a sample from the desired distribution. Histograms based on the sample can then be used to approximate the true marginal distributions of interest. In Section 2.3, I present a variational method for approximate inference. Here, we construct a parameterized approximation to the true distribution and then attempt to optimize the parameters of this variational approximation in order to make it as close as possible to the true distribution. This technique requires that the distribution specified by the Bayesian network can be expressed in a form suitable for mathematical analysis. Finally, in Section 2.4 I present the Helmholtz machine. This method can be very efficient, and is tailored to inference in Bayesian networks whose parameters are estimated from data.

2.1 Exact inference in singly-connected Bayesian networks

In the late 1980's, Pearl [1986; 1988] and Lauritzen and Spiegelhalter [Spiegelhalter 1986; Lauritzen and Spiegelhalter 1988] independently published the exact *probability propagation* algorithm for inferring the distributions over individual variables in singly-connected Bayesian networks. A *singly-connected* network has only a single path (ignoring edge directions) connecting any two vertices (see Figure 2.1a for an example). Not only does the algorithm make use of the probabilistic structure implied by a Bayesian network, but it also uses the network as a circuit that specifies *message passing* channels for inference computations. In a singly-connected network, cutting an edge breaks the network into two pieces. So, each edge acts as a message passing bottle-neck for communicating information regarding one side of the network to the other.

By passing short real-valued vectors between neighboring vertices in the singly-connected Bayesian network for a set of variables $\mathbf{z} = \{z_1, \ldots, z_{|\mathbf{z}|}\}$, the probability propagation algorithm computes $P(z_i|\mathbf{v})$ $i = 1, \ldots, |\mathbf{z}|$ for an arbitrary subset $\mathbf{v}$ of observed elements in $\mathbf{z}$. One flavor of probability propagation is the *generalized forward-backward algorithm*, in which messages are passed in a highly regular way. Since this regularity simplifies the description of the algorithm, I will present the generalized forward-backward algorithm first. The more general probability propagation algorithm can then quite easily be described by relaxing the regularity in the way messages are passed. A proof that probability propagation computes $P(z_i|\mathbf{v})$, $i = 1, \ldots, |\mathbf{z}|$ can be found in Appendix A.1. A simple Tcl-based probability propagation software package is described in Appendix B.

2.1.1 The generalized forward-backward algorithm

To begin with, the singly-connected Bayesian network is arranged as a horizontal tree with an arbitrarily chosen "root" vertex on the far right. For example, if the circled vertex z_9 in Figure 2.1a is chosen as the root, we obtain the tree shown in Figure 2.1b. (Imagine the network sits in a viscous fluid and we grasp the root vertex and pull it down and then to the right.) Beginning with the leaves of the tree (*i.e.*, the vertices on the left), messages are passed level by level forward to the root. Each vertex "fuses" its incoming messages in order to produce an outgoing message, and also stores the incoming messages for later use. Then, messages are passed level by level backward from the root to the leaves. The total number of messages passed in this fashion is $2(|\mathbf{z}| - 1)$, since each edge passes a message in both directions. Once both passes are complete, each vertex z_i fuses all stored incoming messages to obtain $P(z_i|\mathbf{v})$. This algorithm differs from the standard forward-backward (a.k.a. "BCJR") algorithm [Baum and Petrie 1966; Bahl et al. 1974] in two ways. First, the

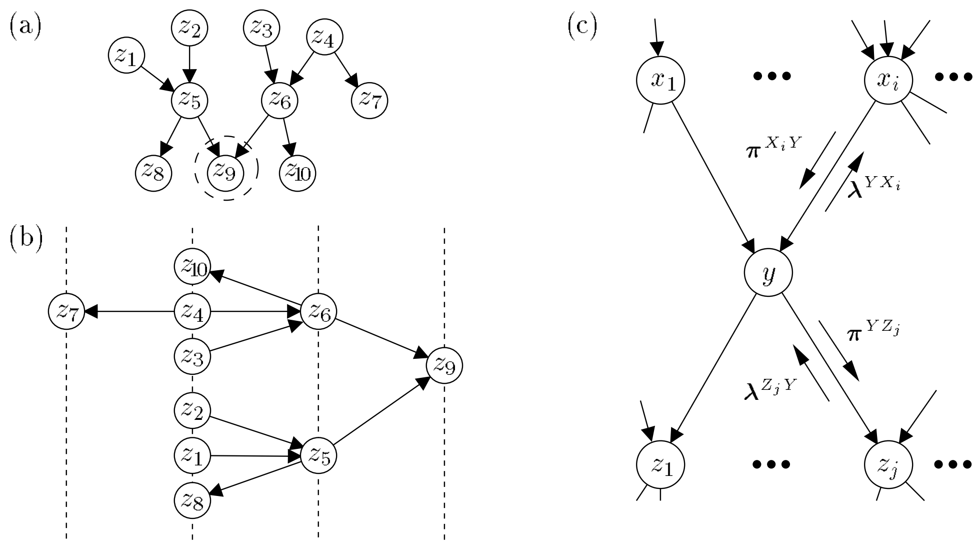

Figure 2.1: (a) shows a singly-connected Bayesian network. A tree is obtained by choosing z_9 as a root vertex, as shown in (b). (c) shows a network fragment with the λ- and π-messages that are passed to and from the parents $\mathbf{x}$ and the children $\mathbf{z}$ of y, during the two passes of the generalized forward-backward algorithm.

underlying graph is a tree, not a chain. Second, some edges may be directed forward while others may be directed backward, whereas in the standard forward-backward algorithm, all edges are directed forward.

During each of the forward and backward passes, two types of vector messages are passed. A π-*message* is passed from a parent to a child in the direction of the edge, and represents the *probability* distribution over the parent given the observed variables connected to the parent through all paths that do not go through the child. A λ-*message* is passed from a child to a parent in the opposite direction of the edge, and represents the *likelihood* for the observed variables connected to the child through all paths that do not go through the parent, given the parent variable. Both types of vector have lengths that are equal to the number of values the *parent variable* can take on. In fact, since a normalization operation can be applied at a later stage, one fewer elements may be passed by assuming that the first element is always 1. For example, in a Bayesian network with binary variables, each message may consist of just a single real value, since all parent variables are binary.

Consider the network fragment shown in Figure 2.1c, where $\mathbf{x}$ is the set of parents of y, and $\mathbf{z}$ is the set of children of y. Let $|y|$ be the number of discrete values that y can take on. Without loss of generality, I assume $y \in \{1, \ldots, |y|\}$. (The following equations hold for real-valued variables as well, if summations are replaced by integrals.) In this case, the conditional probability $P(y|\mathbf{x})$ can be viewed as a high-dimensional matrix $\mathbf{P}^{\mathbf{X}Y}$,

$$P^{\mathbf{X}Y}_{\mathbf{x}y} = P(y|\mathbf{x}). \tag{2.1}$$

In this section I use capitalized variable names in superscripts to *label* vectors and matrices, and lower-case variable names in subscripts to index the elements of vectors and matrices. If y has no parents, then we take $\mathbf{x} = \varnothing$ so that

$$P_{\varnothing y}^{\varnothing Y} = P(y|\varnothing) = P(y). \tag{2.2}$$

To compute an outgoing message, a vertex must take into account the incoming messages on all other edges. Let $|\mathbf{x}|$ be the number of parents of y (number of variables in $\mathbf{x}$) and let $|x_i|$ be the number of values that parent x_i can take on. Variable y may receive a vector message from each parent x_i $i = 1, \ldots, |\mathbf{x}|$:

$$\boldsymbol{\pi}^{X_iY} = (\pi_1^{X_iY}, \ldots, \pi_{|x_i|}^{X_iY}), \tag{2.3}$$

and a vector message from each child z_j $j = 1, \ldots, |\mathbf{z}|$:

$$\boldsymbol{\lambda}^{Z_jY} = (\lambda_1^{Z_jY}, \ldots, \lambda_{|y|}^{Z_jY}). \tag{2.4}$$

To compute an outgoing π-message (*e.g.*, in Figure 2.1b, from z_4 to z_6 in the forward pass, and from z_6 to z_{10} in the backward pass), a variable must fuse the incoming λ- and π-messages on the other edges. For example, in the network fragment shown in Figure 2.1c, y computes the elements of a π-message sent to z_j in the following way:

$$\pi_y^{YZ_j} = \Big[\prod_{\substack{k=1\\k\neq j}}^{|\mathbf{z}|} \lambda_y^{Z_kY}\Big] \Big[\sum_{\mathbf{x}} P_{\mathbf{x}y}^{\mathbf{X}Y} \prod_{i=1}^{|\mathbf{x}|} \pi_{x_i}^{X_iY}\Big], \quad y = 1, \ldots, |y|. \tag{2.5}$$

(The sum is over all possible configurations of the parents, $\mathbf{x}$.) If y has no parents, $|\mathbf{x}| = 0$ and the second term in (2.5) evaluates to $P_{\varnothing y}^{\varnothing Y}$, which is equal to the probability $P(y)$ given in the network specification. In the special case that y is connected by only one edge, the first term in (2.5) evaluates to 1, so that

$$\pi_y^{YZ_j} = P(y), \quad y = 1, \ldots, |y|. \tag{2.6}$$

If y is not a free variable, but has the observed value y^{o}, it computes the elements of the π-message in a different way:

$$\pi_y^{YZ_j} = \delta(y, y^{\mathrm{o}}), \quad y = 1, \ldots, |y|, \tag{2.7}$$

where $\delta(y, y^{\mathrm{o}}) = 1$ if $y = y^{\mathrm{o}}$ and 0 otherwise.

To compute an outgoing λ-message (*e.g.*, in Figure 2.1b, from z_{10} to z_6 in the forward pass, and from z_6 to z_4 in the backward pass), a variable must again fuse the incoming λ- and π-messages on the other edges. For example, in the network fragment shown in Figure 2.1c, y computes the elements of a λ-message sent to x_i in the following way:

$$\lambda_{x_i}^{YX_i} = \sum_{y=1}^{|y|} \Big[\prod_{j=1}^{|\mathbf{z}|} \lambda_y^{Z_jY}\Big] \Big[\sum_{\mathbf{x}':x_i'=x_i} P_{\mathbf{x}'y}^{\mathbf{X}Y} \prod_{\substack{k=1\\k\neq i}}^{|\mathbf{x}|} \pi_{x_k'}^{X_kY}\Big], \quad x_i = 1, \ldots, |x_i|. \tag{2.8}$$

If y has no children, $|\mathbf{z}| = 0$ and the first term in the summation in (2.8) evaluates to 1. In the special case that y is connected by only one edge, the second term in the summation evaluates to $P_{x_iy}^{X_iY} = P(y|x_i)$, so that

$$\lambda_{x_i}^{YX_i} = 1, \quad x_i = 1, \ldots, |x_i|. \tag{2.9}$$

If y is observed and has the value y^{o}, the elements of the λ-message are

$$\lambda_{x_i}^{YX_i} = \sum_{\mathbf{x}':x_i'=x_i} P_{\mathbf{x}'y^{\mathrm{o}}}^{\mathbf{X}Y} \prod_{\substack{k=1\\k\neq i}}^{|\mathbf{x}|} \pi_{x_k'}^{X_kY}. \tag{2.10}$$

After the forward pass and the backward pass are complete, each unobserved vertex y computes $P(y|\mathbf{v})$ by fusing the stored incoming messages as follows:

$$P(y|\mathbf{v}) = \alpha\Big[\prod_{k=1}^{|\mathbf{z}|} \lambda_y^{Z_kY}\Big] \Big[\sum_{\mathbf{x}} P_{\mathbf{x}y}^{\mathbf{X}Y} \prod_{i=1}^{|\mathbf{x}|} \pi_{x_i}^{X_iY}\Big], \quad y = 1, \ldots, |y|, \tag{2.11}$$

where α is a normalizing constant, which is computed to ensure that $\sum_{y=1}^{|y|} P(y|\mathbf{v}) = 1$.

2.1.2 The burglar alarm problem

In order to illustrate how the generalized forward-backward algorithm works, I now introduce a variant of the simple "burglar alarm" network described by Pearl [1988]. The network describes a shoddy burglar alarm that is sensitive not only to burglars, but also to earthquakes. The three binary random variables in the network are b for "burglary", e for "earthquake", and a for "alarm". A value of 0 for one of these variables indicates that the corresponding event *has not* occured, whereas a value of 1 indicates that the corresponding event *has* occured. Figure 2.2a shows the network, which has the following conditional

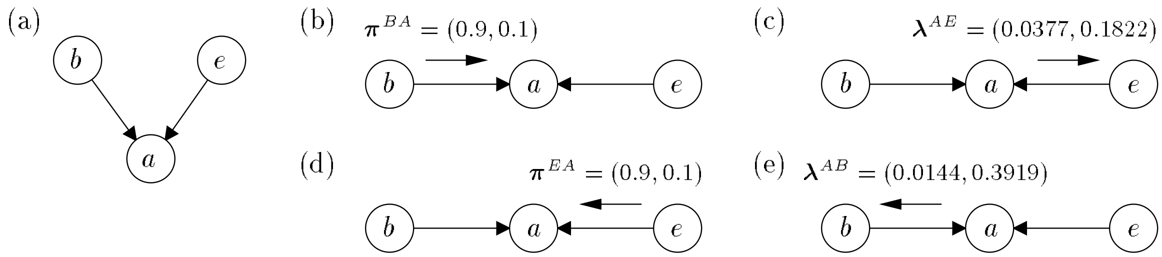

Figure 2.2: (a) The Bayesian network for the burglar alarm problem, with variables b ("burglary"), e ("earthquake") and a ("alarm"). (b) to (e) show the messages passed during the generalized forward-backward algorithm.

probability relationships:

$$\begin{gathered}
P(b\!=\!1) = 0.1, \qquad P(e\!=\!1) = 0.1, \\
P(a\!=\!1|b\!=\!0, e\!=\!0) = 0.001, \quad P(a\!=\!1|b\!=\!1, e\!=\!0) = 0.368, \\
P(a\!=\!1|b\!=\!0, e\!=\!1) = 0.135, \quad P(a\!=\!1|b\!=\!1, e\!=\!1) = 0.607.
\end{gathered} \tag{2.12}$$

Suppose that while you are away at a conference, the burglar alarm contacts you by cell phone and informs you that the alarm is ringing ($a = 1$). We would like to infer the distribution over the two causes to make a well-informed decision about whether or not you should be concerned about a burglary. Since this network is quite simple, we can apply Bayes rule $P(b, e|a) = P(a|b, e)P(b)P(e)/\sum_{b',e'} P(a|b', e')P(b')P(e')$ to obtain the exact solution,

$$\begin{gathered}
P(b\!=\!0, e\!=\!0|a\!=\!1) = 0.016, \quad P(b\!=\!1, e\!=\!0|a\!=\!1) = 0.635, \\
P(b\!=\!0, e\!=\!1|a\!=\!1) = 0.233, \quad P(b\!=\!1, e\!=\!1|a\!=\!1) = 0.116.
\end{gathered} \tag{2.13}$$

The most likely explanation for the ringing alarm is that a burglary took place. Notice, however, that although an earthquake is also a likely explanation, it is relatively *unlikely* that both a burglar and an earthquake were simultaneously the cause of the alarm.

Now, consider using probability propagation for probabilistic inference in this network. (See Appendix B.2 for a description of how this network can be processed using the BNC software package). After we arbitrarily select e as the root, the generalized forward-backward algorithm proceeds by sending a message from the leaf b to a, as shown in Figure 2.2b. Since b is a parent of a, this vector will be a π-message, and since b is connected by only one edge, we use (2.6):

$$\boldsymbol{\pi}^{BA} = \big(P(b=0), P(b=1)\big) = (0.9, 0.1). \tag{2.14}$$

Next, a sends a message to the root e, as shown in Figure 2.2c. Since a is a child of e, this vector will be a λ-message, and since a is observed ($a^{\mathsf{v}} = 1$), we use (2.10):

$$\begin{aligned}\boldsymbol{\lambda}^{AE} &= \Big(\big[\textstyle\sum_{b=0}^{1} P(a=1|b, e\!=\!0)\pi_b^{BA}\big],\ \ \big[\textstyle\sum_{b=0}^{1} P(a=1|b, e\!=\!1)\pi_b^{BA}\big]\Big) \\ &= (0.001 \cdot 0.9 + 0.368 \cdot 0.1,\ 0.135 \cdot 0.9 + 0.607 \cdot 0.1) = (0.0377, 0.1822). \end{aligned}\tag{2.15}$$

Next, e sends a message to a, as shown in Figure 2.2d. Since e is a parent of a, this vector will be a π-message, and since e is connected by only one edge, we use (2.6):

$$\boldsymbol{\pi}^{EA} = \big(P(e=0), P(e=1)\big) = (0.9, 0.1). \tag{2.16}$$

Finally, a sends a message to b, as shown in Figure 2.2e. Since a is a child of b, this vector will be a λ-message, and since a is observed, we use (2.10):

$$\begin{aligned}\boldsymbol{\lambda}^{AB} &= \Big(\big[\textstyle\sum_{e=0}^{1} P(a=1|b\!=\!0, e)\pi_e^{EA}\big],\ \ \big[\textstyle\sum_{e=0}^{1} P(a=1|b\!=\!1, e)\pi_e^{EA}\big]\Big) \\ &= (0.001 \cdot 0.9 + 0.135 \cdot 0.1,\ 0.368 \cdot 0.9 + 0.607 \cdot 0.1) = (0.0144, 0.3919). \end{aligned}\tag{2.17}$$

Now, b and e can compute their marginal distributions using (2.11):

$$\begin{aligned}\big(P(b\!=\!0|a\!=\!1), P(b\!=\!1|a\!=\!1)\big) &= \big(\alpha\lambda_0^{AB} P(b=0), \alpha\lambda_1^{AB} P(b=1)\big) \\ &= (0.01296\alpha, 0.03919\alpha) = (0.249, 0.751), \quad \text{and} \\ \big(P(e\!=\!0|a\!=\!1), P(e\!=\!1|a\!=\!1)\big) &= \big(\alpha\lambda_0^{AE} P(e=0), \alpha\lambda_1^{AE} P(e=1)\big) \\ &= (0.03393\alpha, 0.01822\alpha) = (0.651, 0.349). \end{aligned}\tag{2.18}$$

These distributions are exactly equal to the marginal posterior distributions computed from (2.13).

2.1.3 Probability propagation

The highly regular way in which messages are passed in the generalized forward-backward algorithm can be relaxed to obtain a more general *probability propagation* algorithm. It turns out that as long as a few simple rules are followed, messages may be passed in any order (even in parallel) to obtain the probabilities $P(z_i|\mathbf{v})$ $i = 1, \ldots, |\mathbf{z}|$. These rules prescribe how the network is to be initialized for propagation, and how messages are created, propagated, absorbed, and buffered. Aside from these rules, the formulas for propagating messages are identical to those in (2.5) to (2.10).

Before propagation begins, the network must be *initialized*. This procedure computes the *a priori* incoming messages for each vertex, and corresponds to a generalized forward-backward pass without any observations. It is easy to show that in this case *all* λ-messages will be equal to $\mathbf{1}$, and so initialization consists of passing π-messages using an ancestral ordering. (To see this, imagine performing a forward-backward pass on the network in Figure 2.1a, without any observations.) After initialization, each vertex z_i has available its *a priori* probability $P(z_i)$. In some networks (such as those used for channel coding) these probabilities are uniform and so the initialization procedure can be skipped.

Messages are now *created* in response to observations. If variable y is observed to have the value y^o, then a message must be sent out on each of the edges connected to y, using (2.7) for y's children and (2.10) for y's parents.

Messages are *propagated* in response to other messages. If variable y receives a message on an edge, y must send out messages on all *other* edges.

Messages are *absorbed* by vertices that are connected by only a single edge. This rule follows naturally from the propagation rule, since if such a vertex receives a message on its only edge, the vertex is not required to propagate it back.

It is not necessary that messages be propagated without delay. In fact, a vertex may *buffer* one or more outgoing messages and pass them at any time. (It is usually most convenient to *compute* them at a later time, too.) For example, if a vertex has just received a message and is about to receive another one, computations can often be saved by waiting for the second message before computing and sending out a set of messages.

At any time during propagation, vertex y can compute a current estimate $\hat{P}(y|\mathbf{v})$ of $P(y|\mathbf{v})$ using (2.11). If the above rules are followed and propagation continues until there are no buffered messages remaining in the network, then the estimates will equal the exact probabilities: $\hat{P}(y|\mathbf{v}) = P(y|\mathbf{v})$.

Instead of a complete initialization, it is possible to simply buffer the initial messages leaving each parentless vertex. Since these messages will be propagated eventually, this has the same final result as the initialization procedure described above, although the intermediate probabilities may differ.

The generalized forward-backward algorithm described in the previous section can be viewed as a special case of probability propagation. First, the network is arranged as a tree. Then, the π-messages leaving each parentless vertex in response to network initialization are buffered. Next, the messages created in response to observations are all buffered. (At this point, no computations have been performed.) During the forward pass, each right-going message induces a set of buffered left-going messages and a single right-going message. The latter right-going message is passed to the next level, where it too induces a set of buffered left-going messages and a single right-going message. So, once the forward pass is complete, there are no more buffered right-going messages in the network. During the backward pass, each vertex receives a left-going message from its only right-hand edge. Since an incoming message to a vertex never induces an outgoing message on the same edge, the left-going message will induce only a set of buffered left-going messages. So, there will be no buffered messages remaining in the network once the backward pass is complete. Finally, each vertex z_i can compute the exact value for $P(z_i|\mathbf{v})$.

2.1.4 Grouping variables and duplicating variables

Often, it is possible to convert a multiply-connected Bayesian network to a singly-connected Bayesian network, so that probability propagation can then be applied in a practical manner. To do this, we *group* variables, until there are no more multiple paths in the network. Graphically, two variables z_j and z_k are grouped by removing from the graph z_j and z_k as well as the edges to which they are connected, and then introducing a new vector variable $\{z_j, z_k\}$. The set of parents of the new vector variable is the union of the sets of parents of the two old variables. The set of parents of each child of z_j and z_k is extended to include *both* z_j and z_k. New edges are introduced to reflect these relationships. This grouping operation will produce a valid Bayesian network as long as z_k is not an indirect descendent of z_j and *vice versa*. Otherwise, a directed cycle will result from the grouping, violating the requirement that a Bayesian network have no directed cycles. Note that if z_k is a child of z_j, and at the same time *not* an indirect descendent, the grouping is still valid, since no directed cycles are produced.

As shown in Appendix A.2, this grouping operation also preserves the representational capacity of the network. Any distribution represented by the old network can be represented by the new one. In fact, all of the conditional probabilities $P(z_k|\mathbf{a}_k)$ in the new network will be the same as in the old network, except the ones that involve either of the grouped variables. The latter conditional probabilities can quite easily be derived from the old ones.

Note that although grouping variables may help to produce a singly-connected network to which probability propagation can be applied, the grouping operation also hides the structure that makes probability propagation an attractive inference method in the first

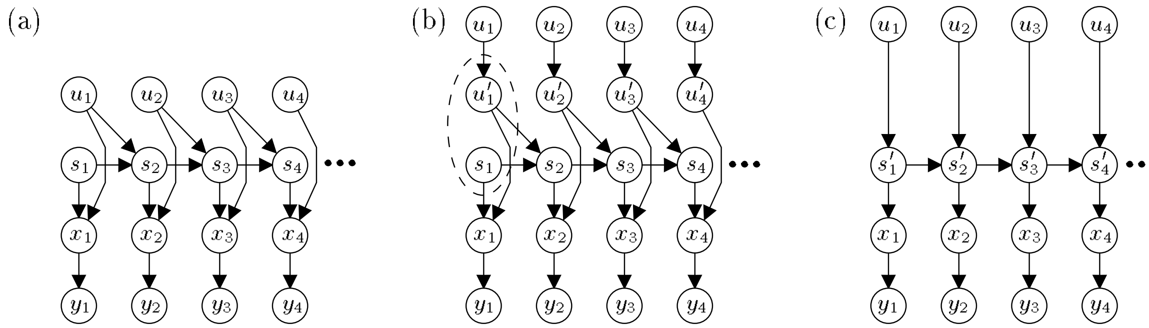

Figure 2.3: Transforming the multiply-connected Bayesian network for a recursive convolutional code (a) into a singly-connected network (c), involves variable *duplication* (a)-(b) and *grouping* (b)-(c).

place. So, it is important to produce minimal groupings and retain as much of the structure as possible. Not surprisingly, by grouping variables, *any* network can be made singly-connected — simply group all variables together into a single vertex. However, extreme groupings of this sort usually eliminate too much structure. "Probability propagation" for the single vertex is equivalent to manipulating the full joint distribution, which in most practical cases in unwieldy.

Another useful operation is *duplicating* variables. A variable z_j can be duplicated by adding an extra variable z_{N+1} to the network, and creating the following new parent-child relationships: $\mathbf{a}_{N+1} = \mathbf{a}_k^{\text{old}}$ and $\mathbf{a}_k^{\text{new}} = \{z_{N+1}\}$. This procedure is especially useful in combination with grouping, since although we may wish to group z_j and z_k in order to make the network singly-connected, we may also wish to graphically distinguish z_j from the vector variable $\{z_j, z_k\}$.

For example, the recursive convolutional code network shown in Figure 1.5a can be derived from the more natural recursive convolutional code network shown in Figure 2.3a. The latter network explicitly shows the dependence of the encoder state variable s_k on the previous information symbol u_{k-1} and the previous state s_{k-1}, as well as the dependence of the encoder output x_k on u_k and s_k. This network is multiply-connected, so probability propagation cannot be used to compute $P(u_k|\mathbf{y})$ for maximum *a posteriori* information symbol decoding. To convert the network to a singly-connected one, we first duplicate the information symbols (so that they are graphically distinguished in the final network) as shown in Figure 2.3b. Then, we group pairs of information symbols and state variables as shown by the dashed loop, producing the singly-connected network shown in Figure 2.3c. Note that by grouping variables in this way, the number of values that each new state s'_k can take on is increased by a factor of two.

Although in many cases grouping can be used to produce a tractable network, there are cases where it is impossible to find an appropriate grouping. In fact, it turns out that in general, inference in multiply-connected networks is a very difficult problem.

2.1.5 Exact inference in multiply-connected networks is NP-hard

Probability propagation is an exact method of inference for *singly-connected* Bayesian networks. Cooper [1990] has shown that probabilistic inference in Bayesian networks is *in general* NP-hard. Summations relevant to inference, such as the ones in (1.3), (1.7), and (1.9), contain an exponential number of terms and it appears that in general these summations cannot be simplified. Researchers have thus focused on developing exact inference algorithms for restricted classes of networks (*e.g.*, probability propagation for singly-connected networks), and on developing approximate inference algorithms for networks that are intractable (assuming P $\neq$ NP). In fact, Dagum [1993] (see also [Dagum and Chavez 1993]) has shown that for general Bayesian networks, approximate inference to a desired number of digits of precision is NP-hard. (*I.e.*, the time needed to obtain an approximate inference that is accurate to n digits is believed to be exponential in n.)

One obvious approach to approximate inference in a multiply-connected Bayesian network is to use the probability propagation algorithm while ignoring the fact that the network is multiply-connected. Each vertex propagates messages as if the network were singly-connected. In this case, the propagation procedure will never terminate, because there will be loops in which messages will endlessly circulate. Although this method has provided excellent results in the area of channel coding, it is frowned upon in other areas (such as medical diagnosis) because there is little theoretical understanding of the behavior of this iterative procedure.

Another disadvantage of the probability propagation algorithm is that it is cumbersome for inferring the *joint* distribution over several variables (*e.g.*, u_1 and u_4 in Figure 2.3c). This inference is accomplished by first computing the distribution over u_1 given the observations, using one forward-backward sweep. Then, the distribution over u_4 given the observations *and* each of the possible values for u_1 is computed using one forward-backward sweep for each possible value for u_1. (Notice that these sweeps may be partial, since they need only take into account the effects of clamping u_1 to different values.) If the variables of interest have n possible configurations, roughly n (possibly partial) forward-backward sweeps are needed. If we cannot afford the time to perform all of these sweeps, a faster approximate algorithm may be more appropriate.

In the following sections, I describe several more principled approaches to approximate probabilistic inference, including Monte Carlo, variational inference, and Helmholtz machines.

2.2 Monte Carlo inference

The Monte Carlo method [Hammersley and Handscomb 1964; Kalos and Whitlock 1986; Ripley 1987] makes use of pseudo-random numbers in order to perform computations. Monte Carlo inference uses random numbers in order to perform inference in a Bayesian network that describes a joint distribution $P(\mathbf{z})$. If we can somehow obtain a reasonably large sample from the distribution $P(\mathbf{h}|\mathbf{v})$ over some unobserved *hidden* variables $\mathbf{h} \subseteq \mathbf{z}$ in a Bayesian network, given some observed *visible* variables $\mathbf{v} \subseteq \mathbf{z}$ then relative frequencies can be used for approximate inference.

2.2.1 Inference by ancestral simulation

One brute force Monte Carlo approach is to simply simulate the network using the ancestral ordering. Then, we extract from the sample all those vectors that have the desired value for the component $\mathbf{v}$. Next, we compile a frequency histogram for the different values that $\mathbf{h}$ can take on. Although this approach is sometimes useful (notably, when using a small network to verify that a more sophisticated inference method works), in general it is not computationally efficient. The problem is that the value of $\mathbf{v}$ that we wish to condition on may occur *extremely* rarely, so that an inordinate sample size must be used in order to obtain results.

If we are interested in a subset $\mathbf{h}^{\mathrm{I}} \subseteq \mathbf{h}$ of the hidden variables, it so happens that in some cases ancestral simulation can be used to obtain a sample from $P(\mathbf{h}^{\mathrm{I}}|\mathbf{v})$ in an efficient manner. In general, if the parents of the visible variables are dependency-separated (see Section 1.2.4) from the hidden variables of interest by the visible variables, then ancestral simulation can be used to obtain a sample from $P(\mathbf{h}^{\mathrm{I}}|\mathbf{v})$. (See Appendix A.3 for a proof.) If the visible variables have no parents, then it follows trivially that the variables in this null set are dependency-separated from the hidden variables. Using the ancestral ordering, a value is drawn for each *hidden* variable given its parents. After one complete sweep, the value for $\mathbf{h}^{\mathrm{I}}$ will be an unbiased draw from $P(\mathbf{h}^{\mathrm{I}}|\mathbf{v})$.

For example, suppose that in the multiply-connected network with ancestral ordering $z_1, z_2, z_3, z_4, z_5, z_6, z_7$ shown in Figure 2.4, the set of visible variables is $\{z_1, z_4\}$. Suppose also that we would like to infer the distribution over the subset of hidden variables $\{z_6, z_7\}$. Since $\{z_6, z_7\}$ is dependency-separated by $\{z_1, z_4\}$ from the parents $\{z_2\}$ of $\{z_1, z_4\}$, we can estimate $P(z_6, z_7|z_1, z_4)$ by ancestral simulation. We draw a value for z_2, then for z_3 given z_1, then for z_5, then for z_6 given z_3 and z_4, then for z_7 given z_4 and z_5. We can estimate $P(z_6, z_7|z_1, z_4)$ by repeating this procedure over and over while building up a histogram for $\{z_6, z_7\}$.

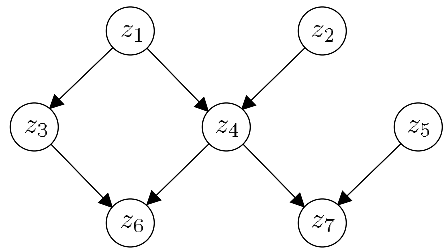

Figure 2.4: An example of a Bayesian network.

Notice that in the above example, it was not really necessary to draw a value for z_2, since for the ancestral simulation method to work, it was *required* that z_2 be dependency-separated from $\{z_6, z_7\}$ by $\{z_1, z_4\}$; *i.e.*, given $\{z_1, z_4\}$, z_2 does not influence $\{z_6, z_7\}$. In general, values need only be drawn for those variables in the ancestral ordering that are *not* dependency-separated from $\mathbf{h}^{\mathrm{I}}$ by $\mathbf{v}$. If these can be easily isolated, then simulation computations can be saved.

2.2.2 Gibbs sampling

When inference by ancestral simulation is not possible, Markov chain Monte Carlo is often used (see an excellent review of these methods by Neal [1993]). Given $\mathbf{v}$, a temporal sequence $\mathbf{h}^{(1)}, \mathbf{h}^{(2)}, \ldots$ of the hidden variable values is produced by simulating a Markov chain whose stationary distribution is carefully constructed (*e.g.*, as described below) to be equal to $P(\mathbf{h}|\mathbf{v})$. By collecting these values over time, an approximate sample is obtained. Ideally, the chain is run long enough so that equilibrium is reached. In practice, the Markov chain may be terminated before equilibrium is reached, so that the simulation time can be kept within a reasonable limit. Once collected, the sample can be used to produce a frequency histogram of the variables of interest in $\mathbf{h}$.

The *Gibbs sampling* algorithm is the simplest of the Markov chain Monte Carlo methods, and has been successfully applied to Bayesian networks [Pearl 1987; Pearl 1988; Neal 1992] as well as other graphical models [Geman and Geman 1984; Hinton and Sejnowski 1986]. In this algorithm, each successive state $\mathbf{h}^{(\tau)}$ is chosen by modifying only a single variable in the previous state $\mathbf{h}^{(\tau-1)}$. The variables are usually modified in sequence. If at time τ, we have decided to modify $z_k \in \mathbf{h}$, then we draw a value $z_k^{(\tau)}$ from

$$P(z_k|\{z_j = z_j^{(\tau-1)}\}_{j=1, j\neq k}^{N}). \tag{2.19}$$

Usually, we cannot obtain this distribution directly, but instead must first compute the joint

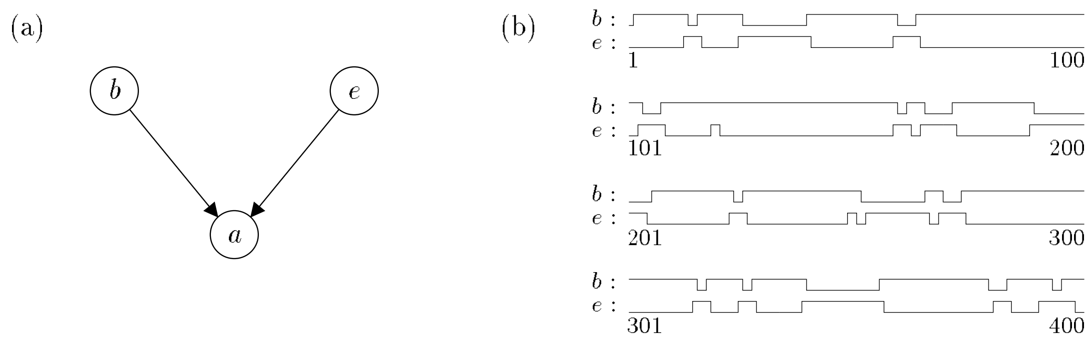

Figure 2.5: (a) The Bayesian network for the burglar alarm problem, with variables b ("burglary"), e ("earthquake") and a ("alarm"). (b) 400 steps of Gibbs sampling for the variables b and e when the alarm is observed to be ringing ($a = 1$).

probability which is proportional to the conditional probability:

$$P(z_k | \{z_j = z_j^{(\tau-1)}\}_{j=1, j\neq k}^{N}) \propto P(z_k, \{z_j = z_j^{(\tau-1)}\}_{j=1, j\neq k}^{N}), \tag{2.20}$$

where the constant of proportionality does not depend on z_k. The joint probability can usually be computed easily from (1.13). If z_k is discrete, we compute the joint probability for each value that it can take on, normalize these values, and then randomly draw a value $z_k^{(\tau)}$ from this normalized distribution. When z_k is a continuous random variable, it can be quite difficult to draw a value from its distribution. Efficient sampling methods for several special types of continuous parametric distribution are given in [Devroye 1986] and [Ripley 1987]. In order to draw values from other types of distribution, more sophisticated techniques such as adaptive rejection sampling [Gilks and Wild 1992] must be used.

2.2.3 Gibbs sampling for the burglar alarm problem

In order to illustrate how Gibbs sampling works, I use the simple burglar alarm problem presented in Section 2.1.2, whose Bayesian network is shown in Figure 2.5a.

In order to perform Gibbs sampling, we need the probabilities for each of the hidden variables conditioned on *all* the other variables. Since these conditional probabilities are proportional to the joint probabilities, we can compute them in the following way, using

$P(b=1|e=0, a=1)$ as an example:

$$\begin{aligned} P(b=1|e=0,a=1) &= \frac{P(b=1,e=0,a=1)}{P(e=0,a=1)} \\ &= \frac{P(b=1,e=0,a=1)}{P(b=0,e=0,a=1)+P(b=1,e=0,a=1)} \\ &= \frac{0.03312}{0.00081+0.03312} = 0.976, \end{aligned} \tag{2.21}$$

where $P(b, e, a) = P(a|b, e)P(b)P(e)$ is the joint distribution determined from the network specification in (2.12). Similarly,

$$\begin{aligned} P(b=1|e=1,a=1) &= \frac{0.00607}{0.01215+0.00607} = 0.333, \\ P(e=1|b=0,a=1) &= \frac{0.01215}{0.00081+0.01215} = 0.938, \\ P(e=1|b=1,a=1) &= \frac{0.00607}{0.03312+0.00607} = 0.155. \end{aligned} \tag{2.22}$$

Gibbs sampling proceeds by alternately visiting b and e, while sampling from $P(b|e, a=1)$ and $P(e|b, a=1)$ using the above formulas. Figure 2.5b shows the values of b and e for 400 steps of Gibbs sampling, starting from an initial configuration $(b=0, e=0)$. (In each step, one variable is updated.) The Markov chain shows that the configurations $(b = 0, e = 0)$ and $(b = 1, e = 1)$ are unlikely compared to $(b = 1, e = 0)$ and $(b = 0, e = 1)$. The correct probabilities in (2.13) can be approximated using the relative frequencies computed from this chain:

$$\begin{aligned} \hat{P}(b=0,e=0|a=1) &= 0.010, \\ \hat{P}(b=1,e=0|a=1) &= 0.674, \\ \hat{P}(b=0,e=1|a=1) &= 0.228 \\ \hat{P}(b=1,e=1|a=1) &= 0.088. \end{aligned} \tag{2.23}$$

These are quite close to the correct values given in (2.13). Usually, an initial segment of the Markov chain is discarded when computing these statistics. The motivation for this procedure is that we would like to have samples that are typical of the *equilibrium* distribution, not the initial configuration.

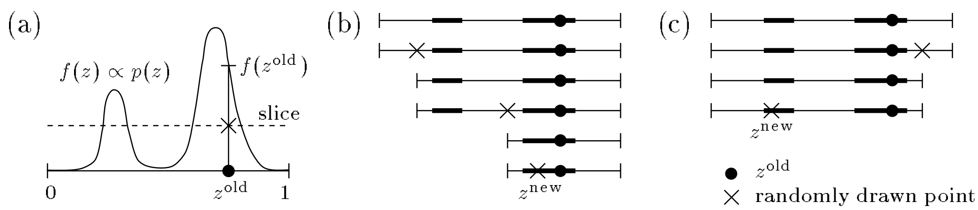

Figure 2.6: Slice sampling. After obtaining a random slice from the density (a), random values are drawn until one is accepted. (b) and (c) show two such sequences.

2.2.4 Slice sampling

In networks with continuous variables, it is often not an easy task to sample from the conditional distribution of each hidden variable, as Gibbs sampling requires. Unlike the case for discrete variables, it is usually not possible to compute the joint distribution for every configuration of a hidden variable. There are infinitely many configurations, and it is often practically impossible to determine an effective discretization. Methods for sampling from continuous distributions include the Metropolis algorithm [Metropolis et al. 1953; Neal 1993] and hybrid methods that use "momentum" in order to help search the configuration space [Duane et al. 1987; Neal 1993]. In this section, I review a technique called *slice sampling* [Neal 1997; Frey 1997a], that can be used for drawing a value z from a univariate probability density $p(z)$ — in the context of inference, $p(z)$ is the conditional distribution $p(z_k|\{z_j = z_j^{(\tau-1)}\}_{j=1,j\neq k}^N)$. Slice sampling does not directly produce values distributed according to $p(z)$, but instead produces a Markov chain that is guaranteed to converge to $p(z)$. At each step in the sequence, the old value z^{old} is used as a guide for where to pick the new value z^{new}. When used in a system with many variables, these updates may be interleaved for greatly improved efficiency.

To perform slice sampling, all that is needed is an efficient way to evaluate a function $f(z)$ that is *proportional* to $p(z)$ — in this application, the easily computed joint probability $p(z_k, \{z_j = z_j^{(\tau-1)}\}_{j=1,j\neq k}^N)$ is appropriate. Figure 2.6a shows an example of a univariate distribution, $p(z)$. The version of slice sampling discussed here requires that all of the probability mass lies within a bounded *interval* as shown. To obtain z^{new} from z^{new}, $f(z^{\text{new}})$ is first computed and then a uniform random value is drawn from $[0, f(z^{\text{new}})]$. The distribution is then horizontally "sliced" at this value, as shown in Figure 2.6a. Any z for which $f(z)$ is greater than this value is considered to be part of the slice, as indicated by the bold line segments in the picture shown at the top of Figure 2.6b. Ideally, z^{new} would now be drawn uniformly from the slice. However, determining the line segments that comprise the slice is not easy, for although it is easy to determine whether a particular z is in the slice, it is much more difficult to determine the line segment boundaries, especially if the distribution is multimodal. Instead, a uniform value is drawn from the original interval as shown in the second picture of Figure 2.6b. If this value is in the slice it is accepted as z^{new} (note that this decision requires an evaluation of $f(z)$). Otherwise either the left or the right interval boundary is moved to this new value, while keeping z^{new} in the interval. This procedure is repeated until a value is accepted. For the sequence in Figure 2.6b, the new value is in the same mode as the old one, whereas for the sequence in Figure 2.6c, the new value is in the other mode. Once z^{new} is obtained, it is used as z^{new} for the next step. As shown in Appendix A.4, this procedure satisfies detailed balance and therefore gives the desired stationary distribution $p(z)$.

2.3 Variational inference

In contrast to both the rather unprincipled approach of applying probability propagation to multiply-connected networks, and the computationally intensive stochastic approach of Monte Carlo, variational inference is a nonstochastic technique that directly addresses the quality of inference. In the Bayesian network literature, variational inference methods [Saul, Jaakkola and Jordan 1996; Ghahramani and Jordan 1996; Jaakkola, Saul and Jordan 1996] were introduced as an alternative variation on the central theme of Helmholtz machines [Hinton et al. 1995; Dayan et al. 1995], which are described in Section 2.4. However, I will present variational inference first, because it is simpler to understand.

Suppose we are given a set of visible variables $\mathbf{v} \subseteq \mathbf{z}$. (This set may includes different variables on different occasions.) In order to solve the inference problem of estimating $P(\mathbf{h}|\mathbf{v})$, we introduce a parameterized variational distribution $Q(\mathbf{h}|\boldsymbol{\xi})$ that is meant to approximate $P(\mathbf{h}|\mathbf{v})$. The most appropriate form of this distribution will depend on many factors, including the network specification and the quality of inference desired. Next, the distance between $P(\mathbf{h}|\mathbf{v})$ and $Q(\mathbf{h}|\boldsymbol{\xi})$ (*e.g.*, Euclidean, relative entropy) is minimized with respect to $\boldsymbol{\xi}$, either directly or by using an optimization technique such as a Newton-like method or a conjugate gradient method [Fletcher 1987]. Once optimized, the distribution $Q(\mathbf{h}|\boldsymbol{\xi})$ is used as an approximation to $P(\mathbf{h}|\mathbf{v})$.

The main advantage of variational inference over probability propagation in multiply-connected networks is the explicit choice of a distance measure that is minimized. Although probability propagation is optimal for singly-connected networks, there is very little known theoretically about the quality of inference that results when the network is multiply-connected. On the other hand, there is no general guarantee that in multiply-connected networks, variational methods will perform better than probability propagation. An example where probability propagation in multiply-connected networks works very well for practical purposes is the celebrated turbo-decoding algorithm for error-correcting coding [Berrou, Glavieux and Thitimajshima 1993; Frey and Kschischang 1996].

Compared to Monte Carlo, variational inference may provide the designer with a more structured approach to choosing a computationally tolerable approximation to $P(\mathbf{h}|\mathbf{v})$. However, variational methods do not usually provide a means to obtain exact inference. Also, variational inference can only be applied when the network is well-tailored to a sensible distance measure along with a fruitful form of variational distribution. (For example, the majority of work on variational methods for Bayesian networks to date has focussed on networks that are parameterized.) In contrast, Monte Carlo methods can be applied to any Bayesian network, and can be designed so that they are guaranteed to converge to the correct solution.

2.3.1 Choosing the distance measure

Depending on the particular problem, different measures of distance may be appropriate. For example, in the case of hard-decision classification and hard-decision channel coding, a binary distance is ideal. Under this distance, the distributions are identical if they lead to the same decisions. Otherwise, the distance is incremented for each incorrect decision. In practice, this distance must be softened in order to use continuous optimization methods.

As another example, we will see in Chapters 3 and 4 that for pattern classification and data compression, the appropriate "distance" is the Kullback-Leibler divergence, or relative entropy, between $Q(\mathbf{h}|\boldsymbol{\xi})$ and $P(\mathbf{h}|\mathbf{v})$:

$$D_{Q||P} = \sum_{\mathbf{h}} Q(\mathbf{h}|\boldsymbol{\xi}) \log \frac{Q(\mathbf{h}|\boldsymbol{\xi})}{P(\mathbf{h}|\mathbf{v})}. \tag{2.24}$$

Notice that this is not a true distance since it is not symmetric: $D_{Q||P} \neq D_{P||Q}$, where

$$D_{P||Q} = \sum_{\mathbf{h}} P(\mathbf{h}|\mathbf{v}) \log \frac{P(\mathbf{h}|\mathbf{v})}{Q(\mathbf{h}|\boldsymbol{\xi})}. \tag{2.25}$$

(For density functions, the summations are replaced by integrals.)

The choice of whether to use $D_{Q||P}$ or $D_{P||Q}$ depends on our objective. The former places emphasis on not inferring unlikely values of $\mathbf{h}$ at the cost of not inferring some of the likely values, whereas the latter places emphasis on inferring all likely values of $\mathbf{h}$ at the cost of inferring some of the unlikely values. For example, consider a real-valued univariate probability density $p(z)$ over z that has two modes, as shown in Figure 2.7. Suppose the

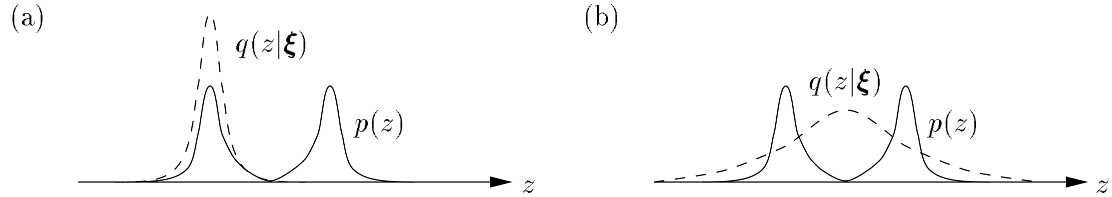

Figure 2.7: The effect of using (a) $D_{q||p}$ versus (b) $D_{p||q}$ when fitting a variational distribution $q(z|\boldsymbol{\xi})$ that is unimodel to a bimodal distribution $p(z)$.

variational distribution $q(z|\boldsymbol{\xi})$ is a Gaussian with $\boldsymbol{\xi}$ consisting of a mean and a variance. Figure 2.7a shows the optimum variational distribution that is obtained by minimizing $D_{q||p}$, whereas Figure 2.7b shows the optimum variational distribution that is obtained by minimizing $D_{p||q}$.

Notice that in order to compute $D_{Q||P}$ in (2.24), we need $P(\mathbf{h}|\mathbf{v})$, which is what we were after in the first place. So, in practice, we usually minimize the following *free energy* function:

$$F_{Q||P} = D_{Q||P} - \log P(\mathbf{v}) = \sum_{\mathbf{h}} Q(\mathbf{h}|\boldsymbol{\xi}) \log \frac{Q(\mathbf{h}|\boldsymbol{\xi})}{P(\mathbf{h}, \mathbf{v})}. \tag{2.26}$$

Notice that minimizing $F_{Q||P}$ with respect to $\boldsymbol{\xi}$ gives the same set of parameters as minimizing $D_{Q||P}$, since $\log P(\mathbf{v})$ does not depend on $\boldsymbol{\xi}$. In order to compute $F_{Q||P}$, we only need $P(\mathbf{h}, \mathbf{v})$, which is readily available in Bayesian networks. ($P(\mathbf{h}, \mathbf{v})$ is *not* easy to compute in other types of graphical models, such as Markov random fields.)

2.3.2 Choosing the form of $Q(\mathbf{h}|\boldsymbol{\xi})$

The form of $Q(\mathbf{h}|\boldsymbol{\xi})$ will strongly influence the quality of the variational inference as well as the tractability of computing the distance and its derivatives (which may be needed for the optimization procedure). Exact inference can be achieved in principle by associating one parameter $\xi_{\mathbf{h}}$ with each state of the hidden variables $\mathbf{h}$, where $\xi_{\mathbf{h}}$ is meant to be an estimate of $P(\mathbf{h}|\mathbf{v})$. However, computing the distance will require an explicit summation over all possible states of the hidden variables. The number of terms in this sum equals the number of possible configurations of the hidden variables, so this approach will only be tractable when there are not many configurations of the hidden variables. In fact, in most cases the above procedure will not be any more computationally efficient than directly computing $P(\mathbf{h}|\mathbf{v})$ using Bayes rule.

We Would like to choose $Q(\mathbf{h}|\boldsymbol{\xi})$ so that the effect of the hidden variables $\mathbf{h}$ in the distance measure can be integrated out either analytically or using a reasonably small number of computations. In this way, the distance and its gradients can be determined without having to numerically examine each possible state of the hidden variables $\mathbf{h}$.

2.3.3 Variational inference for the burglar alarm problem

In this section, I illustrate variational inference using the burglar alarm network described in Section 2.2.3. One type of variational distribution that is often used is the product form distribution. Under this variational distribution, the hidden variables are independent. For continuous variables, further assumptions may be needed regarding the distributions for each hidden variable (*e.g.*, see Section 3.7). For the binary burglar b and earthquake e variables in the burglar alarm network, we can specify an arbitrary product-form distribution using the parameters ξ_1 and ξ_2 for the probabilities that $b=1$ and $e=1$ respectively. That is,

$$Q(b,e|\boldsymbol{\xi}) = Q(b|\boldsymbol{\xi})Q(e|\boldsymbol{\xi}) = \xi_1^b(1-\xi_1)^{1-b}\xi_2^e(1-\xi_2)^{1-e}. \tag{2.27}$$

Inserting this variational distribution into (2.26), and using the simple *binary entropy function* $H(\xi_1) = -\xi_1 \log \xi_1 - (1-\xi_1)\log(1-\xi_1)$, we get

$$\begin{aligned}
F_{Q\|P} &= \sum_{\substack{b=0,1\\e=0,1}} Q(b,e|\boldsymbol{\xi}) \log \frac{Q(b,e|\boldsymbol{\xi})}{P(b,e,a=1)} \\
&= \sum_{\substack{b=0,1\\e=0,1}} Q(b,e|\boldsymbol{\xi})[b\log\xi_1 + (1-b)\log(1-\xi_1) + e\log\xi_2 + (1-e)\log(1-\xi_2)] \\
&\qquad - \sum_{\substack{b=0,1\\e=0,1}} Q(b,e|\boldsymbol{\xi}) \log P(b,e,a=1) \\
&= -H(\xi_1) - H(\xi_2) - \sum_{\substack{b=0,1\\e=0,1}} Q(b,e|\boldsymbol{\xi}) \log P(b,e,a=1).
\end{aligned} \tag{2.28}$$

Notice that the product form of $Q(b,e|\boldsymbol{\xi})$ was used to simplify the first term of the second equality.

At this point, without any further restrictions, we have not gained any computational advantage by using the variational approach. To compute $F_{Q\|P}$ and its derivatives, we must still examine all possible configurations of the hidden variables to compute the expectation of $\log P(b,e,a=1)$. In order to make profitable use of variational inference, $\log P(b,e,a=1)$

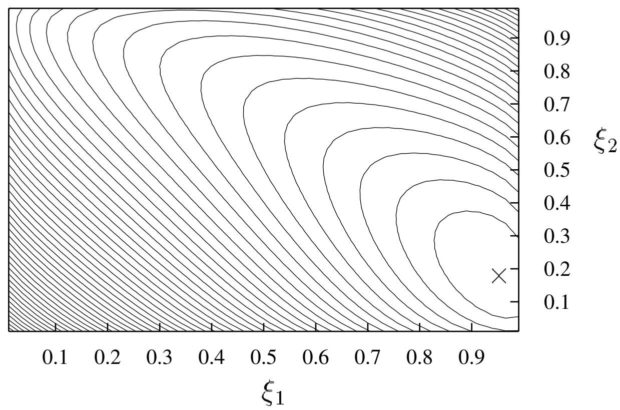

Figure 2.8: The contours of $F_{Q\|P}$ for a variational technique applied to the burglar alarm problem. The global minimum uccurs at $\xi_1 = 0.951$, $\xi_2 = 0.186$.

must have a form that makes the computation of $F_{Q\|P}$ easy. It turns out that the conditional probabilities (2.12) for the burglar alarm network were obtained from

$$P(a=1|b,e) = \exp[6b + 5e - 4.5be - 7]. \tag{2.29}$$

So, the joint distribution $P(b,e,a=1)$ can be written

$$P(b,e,a=1) = P(a=1|b,e)P(b)P(e) = \exp[6b + 5e - 4.5be - 7]0.1^b 0.9^{1-b} 0.1^e 0.9^{1-e}. \tag{2.30}$$

Substituting this into (2.28), we get

$$\begin{aligned} F_{Q\|P} &= -H(\xi_1) - H(\xi_2) \\ &- \sum_{\substack{b=0,1 \\ e=0,1}} Q(b,e|\boldsymbol{\xi})[6b + 5e - 4.5be - 7 + b\log 0.1 + (1-b)\log 0.9 + e\log 0.1 + (1-e)\log 0.9] \\ &= -H(\xi_1) - H(\xi_2) - 3.8\xi_1 - 2.8\xi_2 + 4.5\xi_1\xi_2 + 7.21. \end{aligned} \tag{2.31}$$

Notice that the hidden variables b and e do not appear in this final expression. Because of the product form of $Q(b,e|\boldsymbol{\xi})$ and the exponential form of $P(b,e,a=1)$, we were able to integrate them out.

Figure 2.8 shows a plot of the contours of $F_{Q\|P}$ as a function of ξ_1 and ξ_2. The global minimum occurs at $\xi_1 = 0.951$, $\xi_2 = 0.186$, which means the inference estimates are $\hat{P}(b = 1|a=1) = 0.951$ and $\hat{P}(e = 1|a=1) = 0.186$. These estimates clearly favor a burglar as the cause of the alarm. Recall that Gibbs sampling allowed us to estimate covariance statistics between the two hidden variables. Variational inference does not readily produce those estimates. However, compared to the marginal probabilities $P(b=1|a=1) = 0.751$ and $P(e=1|a=1) = 0.349$ produced by the probability propagation algorithm, the variational method places more emphasis on the more likely cause b. In this sense, the variational technique produces a product form distribution that reveals covariance better than the marginals, produced, say, by probability propagation. For example, using only the marginal probabilities produced by propagation, we might conclude that the probability that *both* a burglar and an earthquake occurred is $0.751 \times 0.349 = 0.262$. In contrast, using the probabilities produced by the variational method gives $0.951 \times 0.186 = 0.177$, which is closer to the correct value of 0.116 given in (2.13).

In this case, because the burglar alarm network is so small, the analytic form of $F_{Q||P}$ in (2.31) is not much simpler than the expression that would be obtained if (2.27) were substituted into (2.28) and explicit summation over all values of b and e were performed. However, for larger networks, exponential computational savings may be made by using conditional distributions that lead to simple forms of $\log P(\mathbf{h}, \mathbf{v})$.

2.3.4 Bounds and extended representations

In practice, the form of $\log P(\mathbf{h}, \mathbf{v})$ is often not simple, so that a straight-forward variational approach cannot be attempted. In these cases, it may be possible to derive an upper *bound* on the distance that does not depend on $\mathbf{h}$, and then try to minimize the bound instead of the distance itself [Saul, Jaakkola and Jordan 1996]. Effectively, we approximate $\log 1/P(\mathbf{h}, \mathbf{v})$ with an upper found that *can* be integrated analytically.

Alternatively, we may express each conditional distribution $P(z_k|\mathbf{a}_k)$ in terms of conditional distributions over an extended set of variables [Jaakkola, Saul and Jordan 1996]. For example, $P(z_k|\mathbf{a}_k)$ might be the marginal distribution of $P(z_k, y_k|\mathbf{a}_k)$, where y_k is part of the extended representation. Let $\mathbf{y}^{\mathrm{H}}$ be the extension variables associated with the variables in $\mathbf{h}$. It is sometimes possible to introduce a variational distribution $Q(\mathbf{h}, \mathbf{y}^{\mathrm{H}}|\boldsymbol{\xi})$ over the extended representation for which $\mathbf{h}$ and $\mathbf{y}^{\mathrm{H}}$ *can* be integrated out in the distance measure.

2.4 Helmholtz machines

One of the main drawbacks of Markov chain Monte Carlo inference and variational inference is that for complex networks, each time a set of variables is observed, either a computationally taxing Markov chain must be simulated, or a high-dimensional optimization must be performed to find the best variational distribution. The essential problem, of course, is that the optimal distribution over $\mathbf{h}$ is different for different values $\mathbf{v}$ of the visible variables. A *Helmholtz machine* [Dayan et al. 1995; Hinton et al. 1995] tackles this problem by coupling the original *generative* network with a *recognition* Bayesian network that is meant to be capable of quickly producing an estimate of, or an approximate sample from, $P(\mathbf{h}|\mathbf{v})$. This recognition network essentially replaces the variational optimization needed for variational inference. It is called a "recognition" network because it is meant to recognize the hidden variable values, or "causes", that are responsible for the values of the visible variables.

As described above, the job of the recognition network is to quickly produce an approximation to $P(\mathbf{h}|\mathbf{v})$. Obviously, the recognition network must be different from the generative network, or the inference could be done directly on the generative network. I will highlight this difference by labeling the recognition distribution with Q. So, the recognition network is used to compute $Q(\mathbf{h}|\mathbf{v})$, which is an approximation to $P(\mathbf{h}|\mathbf{v})$ as given by the generative network. Various types of recognition network are described below, but they all share a common property. Since the recognition network is a Bayesian network, we cannot expect to be able to quickly compute $Q(\mathbf{h}|\mathbf{v})$ for arbitrary sets $\mathbf{h}$ and $\mathbf{v}$. In fact, I will usually assume that the set of visible variables is the same for each inference case, although, of course, the values for the visible variables may differ from case to case. This restriction is the main disadvantage of the Helmholtz machine compared to Monte Carlo inference and variational inference, which usually place no restrictions on which variables are observed.

2.4.1 Factorial recognition networks

To ensure that the inference process is fast, we ought to design the recognition network so that the computation of $Q(\mathbf{h}|\mathbf{v})$ can be carried out efficiently. The simplest recognition network in this sense is one for which each variable in $\mathbf{h}$ is dependency-separated from each other variable in $\mathbf{h}$ by the visible variables $\mathbf{v}$. In other words, given the visible variables, the hidden variables are independent. I will refer to such a network as a *factorial* recognition network, since given the visible variables, the distribution over the hidden variables can be factored into a product of probabilities:

$$Q(\mathbf{h}|\mathbf{v}) = \prod_{z_k \in \mathbf{h}} Q(z_k|\mathbf{v}). \tag{2.32}$$

A factorial recognition network with $\mathbf{h} = \{z_1, z_2, z_3\}$ is shown in Figure 2.9a. Note that by condition 2 in Section 1.2.4, variables in $\mathbf{h}$ are dependency-separated by $\mathbf{v} = \{z_4, z_5, z_6, z_7\}$.

In many cases, the product form approximation given in (2.32) is not very close to $P(\mathbf{h}|\mathbf{v})$. However, it is the easiest network to design or estimate, and because the hidden variables

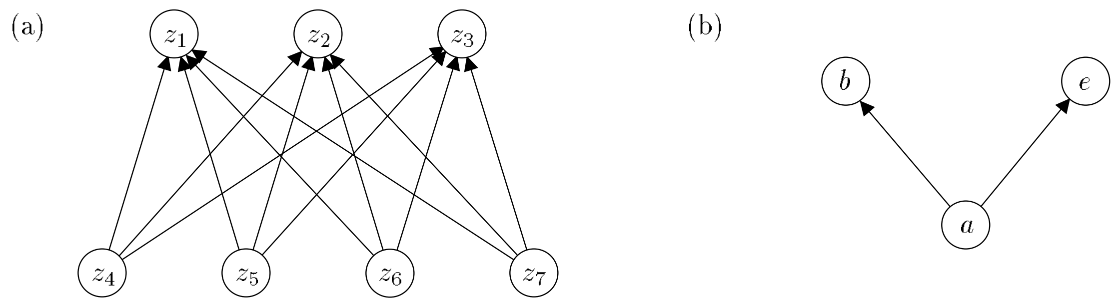

Figure 2.9: (a) An example of a factorial recognition network. (b) A factorial recognition network for the burglar alarm problem.

are independent given the visible variables, it is computationally efficient for inference.

Figure 2.9b shows a factorial recognition network for the burglar alarm problem. The recognition distribution is given by $Q(b, e|a) = Q(b|a)Q(e|a)$, and so is limited to the sane inference estimates as the variational technique described in Section 2.3.3. Namely, the factorial recognition network cannot capture the covariance between the two causes. Can we design a recognition network that can give better estimates? The answer is "yes", by using a nonfactorial recognition network.

2.4.2 Nonfactorial recognition networks

Although it is easy to imagine situations where a factorial recognition network will suffice, for the burglar alarm problem discussed above we saw that a factorial recognition network could not capture the covariance between the two causes of the alarm. In this section, I describe nonfactorial recognition networks that are more powerful than factorial ones.

A *nonfactorial* recognition network can represent a distribution where at least one variable in $\mathbf{h}$ is *not* dependency-separated from at least one other variable in $\mathbf{h}$ by the visible variables $\mathbf{v}$. Of course, there are many ways to make a network nonfactorial. For example, a nonfactorial recognition network is obtained by making some hidden variables depend on other ones in addition to the visible variables. Figure 2.10a shows a fully-connected nonfactorial recognition network, which can be contrasted with the factorial network in Figure 2.9a.

Another way to produce a nonfactorial recognition network is through the use of *auxiliary variables* or *dangling units* [Dayan and Hinton 1996]. These variables do not influence the output of the generative model, but help facilitate inference in the recognition network. For example, an auxiliary variable in the recognition network can be used to choose between two or more modes.

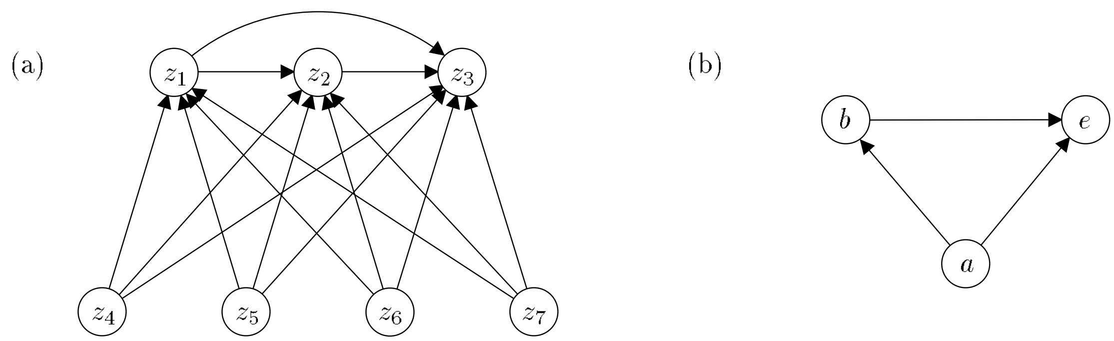

Figure 2.10: (a) An example of a nonfactorial recognition network. (b) A nonfactorial recognition network for the burglar alarm problem.

2.4.3 The stochastic Helmholtz machine

Suppose we are interested in only one of the hidden variables, and we would like to obtain its distribution given the visible variables, after marginalizing out the other hidden variables. For a factorial recognition network, each hidden variable is independent given the visible variables. So, the marginal distribution is obtained simply by ignoring the other hidden variables. In fact, the marginal distribution for $z_k \in \mathbf{h}$ in this case is $Q(z_k|\mathbf{v})$, which is part of the recognition network specification.

Such a simple procedure for marginalization is not in general available for nonfactorial recognition networks. In these networks, the hidden variables are not independent given the visible variables. However, Monte Carlo provides an easy way to estimate marginal statistics. If we can obtain a sufficiently large sample from the recognition network, the distribution for z_k can be approximated by constructing a histogram for z_k alone. Of course, we could directly apply Monte Carlo methods such as Gibbs sampling (Section 2.2.2) to the generative network. However, the hope is that we can carefully design the nonfactorial recognition network so that it is better suited to Monte Carlo than the generative network. In fact, we can avoid complicated Markov chain Monte Carlo by using a recognition network for which ancestral simulation (see Section 1.2.3) can be used.

In general, recognition networks can be either factorial or nonfactorial and stochastic or nonstochastic. Here, "nonstochastic" refers to the way the recognition network is used, not to what the network represents. All Bayesian networks represent a stochastic phenomena, but not all networks are used with Monte Carlo. A factorial recognition network can easily be operated stochastically, simply by choosing each hidden variable z_k from its distribution $Q(z_k|\mathbf{h})$. A nonfactorial recognition network is operated stochastically using a Monte Carlo method (preferably ancestral simulation). A factorial recognition network can easily be operated nonstochastically, since the joint distribution over the hidden variables factors and the marginal distribution for each hidden variable is readily available. However, a nonfactorial recognition network *usually* cannot be operated nonstochastically. As described above, the dependencies between the hidden variables makes this difficult. However, there are special cases where nonfactorial recognition networks can be operated nonstochastically. In particular, recognition networks that can be viewed as a mixture of factorial networks can be operated nonstochastically with relative ease.

2.4.4 A nonfactorial recognition network for the burglar alarm problem

In many cases, a simple nonfactorial recognition network can be used to represent covariances between hidden variables. A nonfactorial recognition network for the burglar alarm problem is shown in Figure 2.10b. The difference between this network and the factorial one in Figure 2.9b, is that e now depends on b as well as a. The conditional distributions for a recognition network that performs exact inference are

$$Q(b{=}1|a{=}1) = 0.751, \qquad Q(e|b, a{=}1) = \begin{cases} 0.154 & \text{if } b = 1, \\ 0.936 & \text{if } b = 0. \end{cases} \tag{2.33}$$

Sampling hidden variables using ancestral simulation in this network is actually more efficient than using Gibbs sampling in the generative network, as described in Section 2.2.3. (The computational savings are quite low in this case, because there are only two hidden variables.)

The joint distribution over the hidden variables given $a = 1$ can be computed from $Q(b, e|a{=}1) = Q(e|b, a{=}1)Q(b|a{=}1)$:

$$\begin{aligned} Q(b{=}0, e{=}0|a{=}1) &= 0.016, \\ Q(b{=}1, e{=}0|a{=}1) &= 0.635, \\ Q(b{=}0, e{=}1|a{=}1) &= 0.233, \\ Q(b{=}1, e{=}1|a{=}1) &= 0.116. \end{aligned} \tag{2.34}$$

These probabilities are identical to the probabilities in (2.13) for exact inference.

Chapter 3

Pattern Classification

Automated methods for making decisions based on inputs play a very important role both in engineering applications and in helping us understand how biological systems respond to their environments. As many engineers and cognitive scientists will attest, the terms "input" and "decision" for this *pattern classification* problem are not clearly defined in theory. In practice, the problem is usually decomposed through design and analysis. The input to the classifier is provided by a preprocessor that transcribes part of the physical state of the world. Different preprocessors are appropriate for different classifiers, and often an iterative process is used to find the optimal preprocessor-classifier pair for a given problem. In general, the preprocessor uses simple statistical and signal processing techniques, whereas the classifier is left with the "hard" problem of coming up with decisions.

A very simple method for making hard decisions is the *nearest neighbor classifier*. This classifier keeps a database of labeled training patterns. Given a test pattern, the nearest neighbor classifier outputs the class of the pattern in its database that is "closest" to the test pattern. Any distance metric may be used, but typically Euclidean distance or one of its generalizations are used. Figure 3.1 shows a selection of normalized and quantized 8×8 binary images of hand-written digits made available by the US Postal Service Office of Advanced Technology. A database with a total of 7000 patterns was constructed with 700 patterns from each digit class. Using nearest neighbor classification, a misclassification rate of 6.7% was obtained on a test set of 4000 patterns. Slightly better results can be achieved by using the *k-nearest neighbor* method. This method picks the most common class of the k training patterns that are closest to the test pattern.

One interesting property of the k-nearest neighbor method is that it is a *consistent* classifier. That is, as the number of training cases T tends to infinity, the decisions produced by the k-nearest neighbor method (with, *e.g.*, $k = \sqrt{T}$) become Bayes optimal. However,

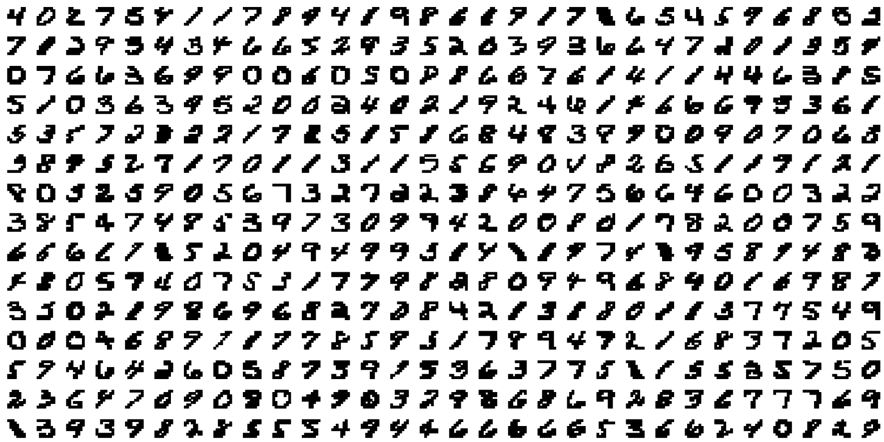

Figure 3.1: 450 examples of 8×8 binary images of hand-written digits.

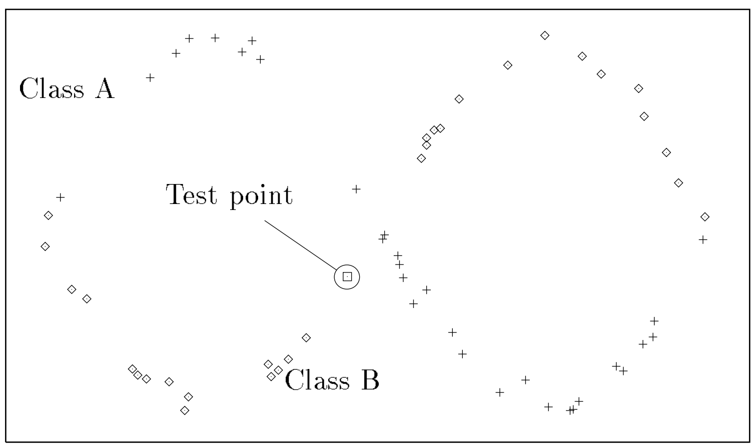

Figure 3.2: Two classes of 2-dimensional training data and a test point.

although k-nearest neighbor classification works quite well when a large training set is available, it performs poorly when training data is limited. Figure 3.2 shows a training set consisting of two classes with 30 2-dimensional real-valued patterns in each class. Suppose we wish to classify the indicated test point. The nearest neighbor method will choose class A. In fact, just as our intuition tells us, the test point was drawn from class B. If a k-nearest neighbor classifier is used, class A will consistently be erroneously chosen for sensible values of k.

The above example illustrates a fundamental flaw with the nearest neighbor approach — namely, that it does not use global structure. Viewing the data from class B with a local (narrow) "window", the test pattern seems very unlikely. However, a more global examination of the data from class B leads us to believe that the data comes from a roughly sinusoidal manifold, and that just by *chance* there isn't any training data for this class in the central region of the figure. Under this view, the test pattern is much more likely. An even more global examination indicates that the two classes of data are probably similar, except for the fact that they lie on manifolds that are relatively inverted. As a result, by inverting one class of data, we actually have 60 points available for estimating the prototypical manifold. In this way, we obtain even more evidence that the test point is from class B.

One way to endow methods with the ability to extract global structure is to use parameterized models that can *generalize* in nontrivial ways. In Bayesian terms, we have prior expectations about certain properties of the data. For example, we expect the probability density function for the data within a given class to be smooth on some scale. The class of distributions that our model can represent should reflect these prior expectations. By fitting the model to the data, this prior knowledge is then modified to obtain a more data-driven set of posterior expectations. In the above example, we decide that a sinusoidal manifold is a reasonable compromise between our prior expectations regarding continuity and the observed data within class B.

Simple parametric models, such as multidimensional Gaussian density functions, can be used to obtain some degree of generalization. However, overly simple models of this sort are inflexible in that they cannot generalize in complex ways. Also, for real-world problems, such inflexible models are often *inconsistent*, since they often cannot represent the complexity in natural data sets. In this chapter, I examine the use of more flexible Bayesian network models for pattern classification.

I begin this chapter with a description of how Bayesian networks can be used for pattern classification. Then, in Section 3.2, I present the "autoregressive" network which is quite simple, but performs surprisingly well as a pattern classifier. In Section 3.3, I describe maximum likelihood estimation and "maximum likelihood-bound" estimation for models with latent (hidden) variables. Latent variables are not part of the input pattern, but are meant to represent higher-order structure in the data (*e.g.*, handwriting style). In Section 3.4, I review three techniques for estimating the parameters of sigmoidal Bayesian networks with latent variables: Gibbs sampling, variational inference, and the wake-sleep algorithm. Then, in Section 3.5, all of these models are compared with the k-nearest neighbor classifier and a tree-based classifier when classifying handwritten digits.

An area which is closely related to estimating probability models for pattern classification is unsupervised learning. I view unsupervised learning as the process of estimating a probability model for a class of data. The hope is that some of the latent variables in the model will come to represent interesting features, and that these features can then be automatically extracted for novel input patterns. In Section 3.6 I present results for the Helmholtz machine, when it is given the task of trying to extract structure from noisy 16×16 images of horizontal and vertical bars. Finally, in Section 3.7, I present a new type of parameterized Bayesian network that can be used to simultaneously extract continuous and categorical structure in an unsupervised manner.

3.1 Bayesian networks for pattern classification

Bayesian networks provide a means of producing structured probabilistic models with arbitrary complexity. In this sense, they are *flexible* models. The majority of this chapter is devoted to using Bayesian networks to produce one model for *each* class of training data. A new test pattern is classified by choosing the class of the model that is best suited to the test pattern. In contrast, it is certainly possible to construct a Bayesian network that has one set of pattern variables $\mathbf{v}$, a variable that represents the class j, plus other variables that represent important physical effects. An inference method can then be used to compute $P(j|\mathbf{v})$ using the network. An advantage of this approach is that the model may make efficient use of the similarities and differences between all of the classes. For example, if each class of data in Figure 3.2 is modelled separately, then the similarity between the two classes cannot be exploited as described above. In practice, however, a parameter estimation algorithm may fail to find such similarities and in the process of trying to model both classes fail to properly extract the features from any one class. Another disadvantage of the single-model approach is that a new class of data cannot be introduced without refitting the model. Despite these disadvantages, the single-model approach is seductively interesting. In Sections 3.6 and 3.7, I study networks that are estimated from *unlabeled* data, where the hidden variables automatically come to represent data classes. Although estimation methods for this *unsupervised learning* problem are currently not highly competitive with other practical engineering techniques, they are potentially very powerful and help shed light on how natural neural systems might work.

The multiple-model approach to pattern classification consists of estimating one model for each of the J classes of data. In this sense, each model is conditioned on a class number. For the sake of generality, I will assume that the jth model has a set of features or hidden attributes $\mathbf{h}_j$ that help model the pattern variables $\mathbf{v}$. Network j thus represents a distribution $P(\mathbf{v}, \mathbf{h}_j|j)$. Finally, the class probabilities $P(j)$ must be specified; these are simply determined from the relative sizes of the classes of data and any prior knowledge we have at hand. (For example, even though a training set contains 10 patterns from class 0 and 14 patterns from class 1, if we know ahead of time that the classes are equally likely then we set $P(j=0)=P(j=1)$.)

Ideally, the model estimate for class j will yield a distribution that is close to the true distribution $P_{\mathrm{r}}(\mathbf{v}|j)$ of the data from class j:

$$P(\mathbf{v}|j)=\sum_{\mathbf{h}_j} P(\mathbf{v},\mathbf{h}_j|j)\approx P_{\mathrm{r}}(\mathbf{v}|j). \tag{3.1}$$

However, even if the approximation is good, the sum in the above expression is exponential in the number of feature variables, and so cannot be computed directly. Instead, for a given test pattern $\mathbf{v}$, one of the inference methods described in Chapter 1 can be used to produce class likelihood estimates,

$$\hat{P}(\mathbf{v}|j), \qquad j\in\{0,\ldots,J-1\}. \tag{3.2}$$

Finally, Bayes rule is used to produce soft classification decisions,

$$\hat{P}(j|\mathbf{v})=\frac{\hat{P}(\mathbf{v}|j)P(j)}{\sum_{j'}\hat{P}(\mathbf{v}|j')P(j')}, \qquad j\in\{0,\ldots,J-1\}, \tag{3.3}$$

and a hard decision j^* can be made by choosing the best class,

$$j^*=\mathrm{argmax}_j\hat{P}(j|\mathbf{v}). \tag{3.4}$$

The technique used to estimate the class models and the inference method used to estimate $P(j|\mathbf{v})$ depend on the structure of the networks. Before examining intractable models for which inference and parameter estimation must be approximated, I discuss an interesting class of tractable systems. For the sake of notational simplicity, the following sections present models and algorithms for estimating $P(\mathbf{v})$, with the class index j left off. It should be kept in mind that one such density model must be estimated for each class.

3.2 Autoregressive networks

There are a variety of Bayesian network architectures for which inference and parameter estimation can be performed exactly within a reasonable amount of time. An architecture of this type that I discuss here is easy to implement and works surprisingly well on some problems. I define an *autoregressive network* as a fully-connected parameterized Bayesian network without any latent variables. The graph for the network is thus specified com-

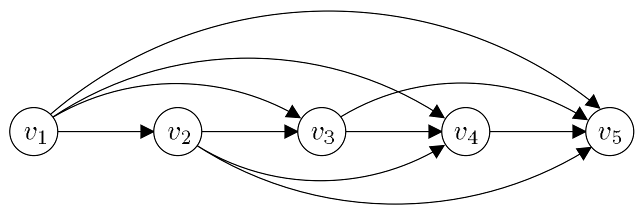

Figure 3.3: An autoregressive network with ancestral ordering v_1, v_2, v_3, v_4, v_5.

pletely by an ancestral ordering. Unless I am considering different ancestral orderings of the variables, I will usually assume that the variables are labeled in the ancestral order. Then, the parameterized distribution $P(\mathbf{v}|\boldsymbol{\theta})$ for the data can be written

$$P(\mathbf{v}|\boldsymbol{\theta}) = \prod_{i=1}^{N} P(v_i|\{v_k\}_{k=1}^{i-1}, \boldsymbol{\theta}_i), \tag{3.5}$$

where $\boldsymbol{\theta}$ is the entire set of parameters, and $\boldsymbol{\theta}_i$ is the set of parameters associated with input v_i. Each of the conditional probability distributions in this expression is represented using some sort of parametric or flexible model. Figure 3.3 shows an example of an autoregressive network with five variables.

3.2.1 The logistic autoregressive network

If the pattern consists of binary variables ($v_i \in \{0, 1\}$) logistic regression [McCullagh and Nelder 1983] (see Section 1.2.6) may be used:

$$P(v_i|\{v_k\}_{k=1}^{i-1}, \boldsymbol{\theta}) = v_i g(\textstyle\sum_{k=0}^{i-1} \theta_{ik} v_k])) + (1 - v_i)(1 - g(\textstyle\sum_{k=0}^{i-1} \theta_{ik} v_k)), \tag{3.6}$$

where $g(x) = 1/(1 + \exp[-x])$ is the logistic function, and a dummy variable $v_0 = 1$ is used to account for a constant in the arguments of the exponent.

For this *logistic autoregressive network*, $P(\mathbf{v}|\boldsymbol{\theta})$ can be computed in $\mathcal{O}(N^2)$ time in the following way. For each variable v_i, the sum $\sum_{k=0}^{i-1} \theta_{v_i v_k} v_k$ is determined from the values of $v_1, \ldots, v_{i-1}$, and then $P(v_i|\{v_k\}_{k=1}^{i-1}, \boldsymbol{\theta}_{v_i})$ is determined from the value of v_i using (3.6). $P(\mathbf{v}|\boldsymbol{\theta})$ is then computed using (3.5).

3.2.2 MAP estimation for autoregressive networks

An autoregressive network can be fit to a class of training patterns $\mathbf{v}^{(1)}, \ldots, \mathbf{v}^{(T)}$ using MAP parameter estimation. To do so, we need to specify both a prior distribution over the parameters $\boldsymbol{\theta}$, and also the training set likelihood given the parameters. Assuming that each training case is independent and identically drawn (i.i.d.), the log-likelihood of the training set is

$$\log P(\{\mathbf{v}^{(t)}\}_{t=1}^{T}|\boldsymbol{\theta}) = \log \prod_{t=1}^{T} P(\mathbf{v}^{(t)}|\boldsymbol{\theta}) = \sum_{t=1}^{T} \log P(\mathbf{v}^{(t)}|\boldsymbol{\theta}) = \sum_{t=1}^{T} \log \left[\prod_{i=1}^{N} P(v_i^{(t)}|\{v_k^{(t)}\}_{k=1}^{i-1}, \boldsymbol{\theta}) \right]$$

$$= \sum_{i=1}^{N} \left[\sum_{t=1}^{T} \log P(v_i^{(t)}|\{v_k^{(t)}\}_{k=1}^{i-1}, \boldsymbol{\theta}) \right] = \sum_{i=1}^{N} \log \left[\prod_{t=1}^{T} P(v_i^{(t)}|\{v_k^{(t)}\}_{k=1}^{i-1}, \boldsymbol{\theta}) \right]. \tag{3.7}$$

If the parameters are independent under the prior, then given a training set, the ith term in the sum of the last expression depends on a set of parameters $\boldsymbol{\theta}_i = \{\theta_{i1}, \ldots, \theta_{i,i-1}\}$ that are independent of all the other sets of parameters $\boldsymbol{\theta}_{i'}$, $i' \neq i$. So, MAP estimation can be broken down into N subproblems, where subproblem i is to estimate the parameters $\boldsymbol{\theta}_i$ for the model that predicts v_i from $\{v_k\}_{k=1}^{i-1}$.

Here, I derive a gradient-based MAP estimation method for the logistic regression model used in the logistic autoregressive network. Let the data for subproblem i be denoted $\mathcal{D}_i = \{\{v_k^{(t)}\}_{k=1}^{i}\}_{t=1}^{T}$. Up to a constant of proportionality that does not depend on $\boldsymbol{\theta}_i$, the likelihood of the training data for subproblem i is

$$P(\mathcal{D}_i|\boldsymbol{\theta}_i) \propto \prod_{t=1}^{T} P(v_i^{(t)}|\{v_k^{(t)}\}_{k=1}^{i-1}, \boldsymbol{\theta}_i)$$

$$= \prod_{t=1}^{T} [v_i^{(t)} g(\textstyle\sum_{k=0}^{i-1} \theta_{ik} v_k^{(t)}])) + (1 - v_i^{(t)})(1 - g(\textstyle\sum_{k=0}^{i-1} \theta_{ik} v_k^{(t)}))], \tag{3.8}$$

where the last expression is obtained from (3.6).

I use a prior distribution where the parameters in $\boldsymbol{\theta}_i$ are independent and normally distributed with mean 0 and a fixed variance σ_i^2:

$$P(\boldsymbol{\theta}_i) = \prod_{k=0}^{i-1} \frac{1}{\sqrt{2\pi\sigma_i^2}} e^{-\theta_{ik}^2/2\sigma_i^2}. \tag{3.9}$$

Up to a constant of proportionality that is independent of $\boldsymbol{\theta}_i$, the posterior distribution over the model parameters $\boldsymbol{\theta}_i$ given the data $\mathcal{D}_i$ is

$$P(\boldsymbol{\theta}_i|\mathcal{D}_i) \propto P(\boldsymbol{\theta}_i, \mathcal{D}_i) = P(\mathcal{D}_i|\boldsymbol{\theta}_i)P(\boldsymbol{\theta}_i)$$
$$\propto \prod_{t=1}^{T}[v_i^{(t)} g(\sum_{k=0}^{i-1}\theta_{ik}v_k^{(t)}])) + (1-v_i^{(t)})(1-g(\sum_{k=0}^{i-1}\theta_{ik}v_k^{(t)}))]\prod_{k=0}^{i-1}\frac{1}{\sqrt{2\pi\sigma_i^2}}e^{-\theta_{ik}^2/2\sigma_i^2}. \tag{3.10}$$

Taking the logarithm of this expression and leaving out constants that do not effect the optimization procedure, MAP estimation for subproblem i entails maximizing

$$\mathcal{L}^i = \sum_{t=1}^{T}[v_i^{(t)}\log g(\textstyle\sum_{k=0}^{i-1}\theta_{ik}v_k^{(t)}) + (1-v_i^{(t)})\log(1-g(\textstyle\sum_{k=0}^{i-1}\theta_{ik}v_k^{(t)}))] \;-\; \displaystyle\frac{1}{2\sigma_i^2}\sum_{k=0}^{i-1}\theta_{ik}^2. \tag{3.11}$$

I use the conjugate gradient optimization method [Fletcher 1987] which requires the derivatives of $\mathcal{L}^i$:

$$\frac{\partial\mathcal{L}^i}{\partial\theta_{ik}} = \sum_{t=1}^{T} v_k^{(t)}[v_i^{(t)} - g(\textstyle\sum_{k'=0}^{i-1}\theta_{ik'}v_{k'}^{(t)})] \;-\; \theta_{ik}/\sigma_i^2. \tag{3.12}$$

Both $\mathcal{L}^i$ and its derivatives can be computed in $\mathcal{O}(iT)$ time.

3.2.3 Scaled priors in logistic autoregressive networks

In the prior distribution over the parameters (3.9), how should the variance σ_i^2 of the parameters for the ith input depend on i? That is, before seeing any training data, how do we expect the variance of the parameters for a variable to depend on how many inputs that variable receives?

Assume we don't have prior knowledge of a preferred ordering of the variables. By symmetry, it makes sense to assume a uniform *prior* distribution over the variables; *i.e.*, under the prior each variable is equally likely to have each of the values 0 and 1. The dummy variable $v_0 = 1$ is exempt from this prior, of course. Now, consider the prior probability predictions made for v_i. This prior distribution has two sources of variability: a Gaussian prior over the parameters $\boldsymbol{\theta}_i$, and a uniform distribution over the inputs $\{v_k\}_{k=1}^{i-1}$. By symmetry, this prior distribution over the probability predictions made for v_i should *not* depend on i. As shown below, this restriction determines how to set the variance for the parameters $\boldsymbol{\theta}_i$ for each variable v_i.

Since the probability prediction for v_i is determined by its total input $\sum_{k=0}^{i-1}\theta_{ik}v_k$, I will enforce the above restriction on the total inputs. Averaging over the two sources of variability, we get a mean value of the total input for v_i of

$$\mathrm{E}\left[\sum_{k=0}^{i-1}\theta_{ik}v_k\right] = \sum_{k=0}^{i-1}\mathrm{E}[\theta_{ik}v_k] = \sum_{k=0}^{i-1}\mathrm{E}[\theta_{ik}]E[v_k] = 0. \tag{3.13}$$

We can take $\mathrm{E}[\theta_{ik}v_k] = \mathrm{E}[\theta_{ik}]E[v_k]$, since the parameters and the inputs are independent under the prior. The final step holds since the parameters have mean 0.

Since the mean total input is 0, the variance of the total input for v_i is

$$\begin{aligned}\mathrm{E}\left[\left(\sum_{k=0}^{i-1}\theta_{ik}v_k\right)^2\right] &= \mathrm{E}\left[\sum_{k=0}^{i-1}\sum_{j=0}^{i-1}\theta_{ik}\theta_{ij}v_k v_j\right] \\ &= \sum_{k=0}^{i-1}\sum_{j=0}^{i-1}E[\theta_{ik}\theta_{ij}]E[v_k v_j]. \end{aligned}\tag{3.14}$$

Now, since θ_{ik} and θ_{ij}, $k \neq j$ are independent under the prior, $E[\theta_{ik}\theta_{ij}]$ is nonzero only if $j = k$. So, the variance of the total input for v_i is

$$\sum_{k=0}^{i-1}E[\theta_{ik}^2]E[v_k^2] = \sigma_i^2 + \sum_{k=1}^{i-1}\sigma_i^2\frac{1}{2} = \frac{1}{2}(i+1)\sigma_i^2. \tag{3.15}$$

Under the prior, the probability predictions for v_i should not depend on i. So, the variances of the total inputs for v_1 and v_i should not differ:

$$\sigma_1^2 = \frac{1}{2}(i+1)\sigma_i^2, \text{ and so } \sigma_i^2 = \frac{2}{i+1}\sigma_1^2. \tag{3.16}$$

Note that σ_1^2 is the variance of the total input for v_1, which has no input variables. All of the variances can be set by picking a reasonable value for σ_1^2. In my simulations, I chose $\sigma_1^2 = 4$. This value allows for probabilities near 0 and 1 at the output of the logistic function, without favoring them too much (see Figure 1.6 on page 18).

It may be a good idea to let the biases in the network have a *separate* Gaussian prior, although I have not yet explored this possibility experimentally.

3.2.4 Ensembles of autoregressive networks

An autoregressive network is specified by chosing an order for the variables $v_1, \ldots, v_N$. Leaving computational considerations aside, if the subproblem models $P(v_i|\{v_k\}_{k=1}^{i-1})$ are consistent (*i.e.*, they converge to the correct distribution as the number of training examples tends to infinity) and there is a sufficiently large training set, then the particular ordering chosen is not important. The model for subproblem i will correctly represent the real conditional distribution $P_r(v_i|\{v_k\}_{k=1}^{i-1})$, and so the product of the subproblem distributions will give the true joint distribution. However, the data sets considered here are small, and the parametric subproblem models considered here (*e.g.*, logistic regression) are inconsistent for many distributions of data. In this case, the order of the variables is important in two contrasting ways. Certain orderings may give rise to simpler true conditional distributions $P_r(v_i|\{v_k\}_{k=1}^{i-1})$ that can be more accurately represented by the model distributions $P(v_i|\{v_k\}_{k=1}^{i-1}, \boldsymbol{\theta}_i)$. In contrast, for a given training set, different orderings may lead to different amounts of overfitting.

I do not address here the difficult issue of how to select an ordering that optimally balances these two effects. This problem is difficult both because the discrete ordering cannot be optimized by a gradient-based method and because for the training sets I will consider here, there is not enough data available to get a reliable estimate of which ordering is best. Instead of searching for an optimal ordering, I estimate an *ensemble* of autoregressive networks, where each network uses a randomly selected ordering of the variables. The probability prediction for a given vector $\mathbf{v}$ is then taken to be the average of the predictions over the ensemble of networks.

3.3 Estimation of models with unobserved variables

The notion of unobserved or hidden variables arises in many model estimation contexts. For example, due to mechanical failure, training data derived from physical measurements may sometimes lack values for some variables in some cases. In contrast, it is often useful to build hidden variables into a model by design. These variables are meant to represent latent causes that influence the visible variables. Several of the Bayesian network models discussed in the remainder of this chapter are latent variable models (*e.g.*, see Section 3.4). For the sake of notational simplicity, I will use $\mathbf{v}$ to refer to the observed variables and $\mathbf{h}$ to refer to the unobserved variables. This is a slight abuse of notation, since it can happen that some visible variables are unobserved. For example, several of the photo-sensors in a digital camera may be burned out, so that some of the variables in the image pattern $\mathbf{v}$ are unobserved.

We would like to estimate a probabilistic model $P(\mathbf{z})$ for a training set cosisting of T patterns $\mathbf{v}^{(1)}, \mathbf{v}^{(2)}, \ldots, \mathbf{v}^{(T)}$, where each pattern specifies the values of an observed subset $\mathbf{v}$ of the variables in $\mathbf{z}$. In general, each training case may specify a *different* subset of visible variables.

Let $\mathbf{h}^{(t)} = \mathbf{z} \backslash \mathbf{v}^{(t)}$ be the set of hidden (unobserved) variables for training case t. Assuming that the training cases are i.i.d., the log-likelihood of the training data is

$$\log P(\mathcal{D}|\boldsymbol{\theta}) = \log \prod_{t=1}^{T} P(\mathbf{v}^{(t)}|\boldsymbol{\theta}) = \sum_{t=1}^{T} \log P(\mathbf{v}^{(t)}|\boldsymbol{\theta}) = \sum_{t=1}^{T} \log \left[\sum_{\mathbf{h}^{(t)}} P(\mathbf{v}^{(t)}, \mathbf{h}^{(t)}|\boldsymbol{\theta}) \right]. \tag{3.17}$$

To maximize this log-likelihood, we set its derivative with respect to each parameter θ in $\boldsymbol{\theta}$ to zero:

$$\begin{aligned}
\frac{\partial \log P(\mathcal{D}|\boldsymbol{\theta})}{\partial \theta} &= \sum_{t=1}^{T} \frac{1}{\sum_{\mathbf{h}^{(t)\prime}} P(\mathbf{v}^{(t)}, \mathbf{h}^{(t)\prime}|\boldsymbol{\theta})} \frac{\partial}{\partial \theta} \left[\sum_{\mathbf{h}^{(t)}} P(\mathbf{v}^{(t)}, \mathbf{h}^{(t)}|\boldsymbol{\theta}) \right] \\
&= \sum_{t=1}^{T} \sum_{\mathbf{h}^{(t)}} \frac{1}{\sum_{\mathbf{h}^{(t)\prime}} P(\mathbf{v}^{(t)}, \mathbf{h}^{(t)\prime}|\boldsymbol{\theta})} \frac{\partial}{\partial \theta} P(\mathbf{v}^{(t)}, \mathbf{h}^{(t)}|\boldsymbol{\theta}) \\
&= \sum_{t=1}^{T} \sum_{\mathbf{h}^{(t)}} \frac{P(\mathbf{v}^{(t)}, \mathbf{h}^{(t)}|\boldsymbol{\theta})}{\sum_{\mathbf{h}^{(t)\prime}} P(\mathbf{v}^{(t)}, \mathbf{h}^{(t)\prime}|\boldsymbol{\theta})} \frac{\partial}{\partial \theta} \log P(\mathbf{v}^{(t)}, \mathbf{h}^{(t)}|\boldsymbol{\theta}) \\
&= \sum_{t=1}^{T} \sum_{\mathbf{h}^{(t)}} P(\mathbf{h}^{(t)}|\mathbf{v}^{(t)}, \boldsymbol{\theta}) \frac{\partial}{\partial \theta} \log P(\mathbf{v}^{(t)}, \mathbf{h}^{(t)}|\boldsymbol{\theta}) = 0, \quad \forall\, \theta \in \boldsymbol{\theta}.
\end{aligned} \tag{3.18}$$

The relation $\partial \log f(\theta)/\partial\theta = (1/\theta)\partial f(\theta)/\partial\theta$ was used in the first and third line of the derivation. Even though $\partial \log P(\mathbf{v}^{(t)}, \mathbf{h}^{(t)}|\boldsymbol{\theta})/\partial\theta$ is quite often easy to compute, in many cases of practical interest the *system of equations* obtained by setting $\partial \log P(\mathcal{D}|\boldsymbol{\theta})/\partial\theta$ to zero for each θ is highly nonlinear and cannot be solved in closed-form. One approach is to perform gradient descent in $\log P(\mathbf{v}^{(t)}, \mathbf{h}^{(t)}|\boldsymbol{\theta})$, while sampling from $P(\mathbf{h}^{(t)}|\mathbf{v}^{(t)}, \boldsymbol{\theta})$ using Markov chain Monte Carlo. This gives a Monte Carlo approximation to gradient descent in $\log P(\mathcal{D}|\boldsymbol{\theta})$ as given in (3.18). Another approach is to solve the system of nonlinear equations iteratively. Although in principle any method for solving a nonlinear system of equations can be used (*e.g.*, Newton's method [Fletcher 1987]), the structure of (3.18) gives rise to a particularly simple two-phase iterative method, called the *expectation-maximization* (EM) algorithm [Baum and Petrie 1966; Dempster, Laird and Rubin 1977].

3.3.1 ML estimation by expectation-maximization (EM)

Often, we have available an efficient method for estimating the model when all of the variables are visible. That is, the system of equations obtained by setting

$$\sum_{t=1}^{T} \frac{\partial}{\partial \theta} \log P(\mathbf{v}^{(t)}, \mathbf{h}^{(t)} | \boldsymbol{\theta}) = 0, \quad \forall\, \theta \in \boldsymbol{\theta} \tag{3.19}$$

for arbitrary $\mathbf{h}^{(t)}$ can be solved quite easily. Notice that it is *essentially* this system of equations that is obtained if the dependence of $P(\mathbf{h}^{(t)}|\mathbf{v}^{(t)}, \boldsymbol{\theta})$ on $\boldsymbol{\theta}$ in (3.18) is ignored. The summation over $\mathbf{h}^{(t)}$ in (3.18) has the effect of replicating training case t once for each configuration of the hidden variables for that case, and weighting each replication by $P(\mathbf{h}^{(t)}|\mathbf{v}^{(t)}, \boldsymbol{\theta})$. This observation leads to the following iterative two-phase EM algorithm:

1. E-step: Compute $P(\mathbf{h}^{(t)}|\mathbf{v}^{(t)}, \boldsymbol{\theta})$ for each configuration $\mathbf{h}^{(t)}$ of the hidden variables for each training case, and set $Q(\mathbf{h}^{(t)}) \leftarrow P(\mathbf{h}^{(t)}|\mathbf{v}^{(t)}, \boldsymbol{\theta})$.

2. M-step: Solve the following system of equations for $\boldsymbol{\theta}$,

$$\sum_{t=1}^{T} \sum_{\mathbf{h}^{(t)}} Q(\mathbf{h}^{(t)}) \frac{\partial}{\partial \theta} \log P(\mathbf{v}^{(t)}, \mathbf{h}^{(t)} | \boldsymbol{\theta}) = 0, \quad \forall\, \theta \in \boldsymbol{\theta}. \tag{3.20}$$

   Stop if a convergence criterion is satisfied; otherwise go to 1.

In practice, the values of $Q(\mathbf{h}^{(t)})$ for each training case are usually not stored during the E-step. Instead, statistics that are sufficient for the M-step are accumulated while processing the training set. There are several proofs that each EM iteration is guaranteed to increase the likelihood of the training data [Baum and Petrie 1966; Dempster, Laird and Rubin 1977; Meng and Rubin 1992; Neal and Hinton 1993]. After presenting a more general algorithm for maximizing *lower bounds* on the data likelihood $P(\mathcal{D}|\boldsymbol{\theta})$, I will show that each iteration of EM is guaranteed to increase the data likelihood.

3.3.2 Maximum likelihood-bound (MLB) estimation

Neal and Hinton [1993] introduced a new view of the EM algorithm as a method for maximizing a lower bound on the likelihood of a training set. This interpretation opened the door to tractable approximations to EM for models that were clearly intractable. I will refer to the new approach as maximum likelihood-bound (MLB) estimation in order to highlight its relationship to ML estimation. MLB estimation is an approximation to ML estimation that follows from using the wrong distribution $Q(\mathbf{h}^{(t)})$ in the E-step of the EM algorithm;

i.e., $Q(\mathbf{h}^{(t)}) \neq P(\mathbf{h}^{(t)}|\mathbf{v}^{(t)}, \boldsymbol{\theta})$. There are practical reasons for using a suboptimal distribution $Q(\mathbf{h}^{(t)})$, the most obvious being that in some cases it is computationally infeasible to compute $Q(\mathbf{h}^{(t)})$ for *every* configuration of the hidden variables $\mathbf{h}^{(t)}$ for each training case. For example, some of the Bayesian network models discussed below have over one million configurations per training case.

The bound used in MLB estimation is obtained using the following form of Jensen's inequality [Cover and Thomas 1991]:

$$\log \sum_i q_i a_i \geq \sum_i q_i \log a_i, \tag{3.21}$$

where $\sum_i q_i = 1$, and a_i are arbitrary scalars. Applying this inequality to the log-likelihood of the training data (3.17), we get

$$\begin{aligned} \log P(\mathcal{D}|\boldsymbol{\theta}) &= \sum_{t=1}^{T} \log \left[\sum_{\mathbf{h}^{(t)}} P(\mathbf{v}^{(t)}, \mathbf{h}^{(t)}|\boldsymbol{\theta}) \right] = \sum_{t=1}^{T} \log \left[\sum_{\mathbf{h}^{(t)}} Q(\mathbf{h}^{(t)}) \frac{P(\mathbf{v}^{(t)}, \mathbf{h}^{(t)}|\boldsymbol{\theta})}{Q(\mathbf{h}^{(t)})} \right] \\ &\geq \sum_{t=1}^{T} \sum_{\mathbf{h}^{(t)}} Q(\mathbf{h}^{(t)}) \log \frac{P(\mathbf{v}^{(t)}, \mathbf{h}^{(t)}|\boldsymbol{\theta})}{Q(\mathbf{h}^{(t)})} = B_{Q||P}. \end{aligned} \tag{3.22}$$

The goal of MLB estimation is to jointly estimate a distribution $Q(\mathbf{h}^{(t)})$ (which may or may not be parameterized) and a distribution $P(\mathbf{v}^{(t)}, \mathbf{h}^{(t)}|\boldsymbol{\theta})$, so as to maximize this lower bound on the likelihood. This leads to the following generalized EM algorithm:

1. Generalized E-step: Increase the bound $B_{Q||P}$ with respect to a distribution $Q(\mathbf{h}^{(t)})$.

2. Generalized M-step: Increase the bound $B_{Q||P}$ with respect to $\boldsymbol{\theta}$.

Note that unlike the E-step of the EM algorithm, the generalized E-step may produce a Q-distribution for which $Q(\mathbf{h}^{(t)}) \neq P(\mathbf{h}^{(t)}|\mathbf{v}^{(t)}, \boldsymbol{\theta})$.

The EM algorithm can be viewed as a special case of the generalized EM algorithm, where we alternately maximize the bound $B_{Q||P}$ with respect to an *unconstrained* distribution $Q(\mathbf{h}^{(t)})$, and then with respect to $P(\mathbf{v}^{(t)}, \mathbf{h}^{(t)}|\boldsymbol{\theta})$ via $\boldsymbol{\theta}$. If the bound is maximized with respect to $Q(\mathbf{h}^{(t)})$ during the generalized E-step, while enforcing $\sum_{\mathbf{h}^{(t)}} Q(\mathbf{h}^{(t)}) = 1$ using a Lagrange multiplier, we obtain $Q(\mathbf{h}^{(t)}) = P(\mathbf{h}^{(t)}|\mathbf{v}^{(t)}, \boldsymbol{\theta})$. This form of the generalized EM algorithm is identical to the standard EM algorithm presented in the previous section. Also, in this case the inequality in (3.22) becomes an equality: $B_{Q||P} = \log P(\mathcal{D}|\boldsymbol{\theta})$. It follows that the EM algorithm is a maximum likelihood estimation method.

Note that in general, MLB estimation *does not* give the same estimates as ML estimation. As a degenerate example, imagine that we use ML estimation to obtain a model $P(\mathbf{v}^{(t)}, \mathbf{h}^{(t)}|\boldsymbol{\theta})$ from a training set, and that we then apply MLB estimation with $Q(\mathbf{h}^{(t)})$ fixed at a uniform distribution. In this case, the bound can be increased by moving $P(\mathbf{v}^{(t)}, \mathbf{h}^{(t)}|\boldsymbol{\theta})$ *away* from the ML estimate (unless $P(\mathbf{v}^{(t)}|\mathbf{h}^{(t)}, \boldsymbol{\theta})$ happens to be uniform, in which case a uniform $Q(\mathbf{h}^{(t)})$ makes the bound tight so that $P(\mathbf{v}^{(t)}, \mathbf{h}^{(t)}|\boldsymbol{\theta})$ will not change). However, as long as we are able produce estimates of $Q(\mathbf{h}^{(t)})$ that are "close" to $P(\mathbf{h}^{(t)}|\mathbf{v}^{(t)}, \boldsymbol{\theta})$, MLB estimation will be close to ML estimation. Of course, in most cases, if we have the computational resources available to obtain an ML estimate, MLB estimation should not be used. In Section 3.4, I introduce a class of Bayesian networks that have many latent (hidden) variables. For these networks, it is computationally intractable to perform ML estimation, and so MLB estimation is used.

3.4 Multiple-cause networks

In many cases, it makes sense to postulate that a data vector $\mathbf{v}$ naturally arises from the consequences of a set of hidden *causes* $\mathbf{h}$. For example, an image may be nicely described as a two-dimensional rendition of a combination of objects. If h_k is a binary variable indicating the presence of object k in the image, then the model distribution $P(\mathbf{v}|\mathbf{h}, \boldsymbol{\theta}^{\mathrm{V}})$ is the distribution over images given which objects are present. ($\boldsymbol{\theta}^{\mathrm{V}}$ is a set of parameters associated with the distribution over $\mathbf{v}$). This distribution is meant to capture the way in which the objects interact to form the image as well as any inexplicable noise.

The model $P(\mathbf{v}|\mathbf{h}, \boldsymbol{\theta}^{\mathrm{V}})$ may be simplified by assuming that the K causes dependency-separate the image pixels. That is, once we know which causes are present, each pixel is independent of the others. In this case, $P(\mathbf{v}|\mathbf{h}, \boldsymbol{\theta}^{\mathrm{V}}) = \prod_{i=1}^{N} P(v_i|\mathbf{h}, \boldsymbol{\theta}_i^{\mathrm{V}})$. If the visible variables are binary, each conditional distribution can be implemented using, for example, logistic regression. In contrast to the logistic autoregressive network where each visible variable is regressed on a subset of the other visible variables (see (3.5)), in the multiple-cause network each visible variable is regressed on the hidden cause variables $\mathbf{h}$:

$$P(v_i|\mathbf{h}, \boldsymbol{\theta}_i^{\mathrm{V}}) = v_i g(\textstyle\sum_{k=0}^{K} \theta_{ik}^{\mathrm{V}} h_k) + (1 - v_i)(1 - g(\sum_{k=0}^{K} \theta_{ik}^{\mathrm{V}} h_k)), \tag{3.23}$$

where $\boldsymbol{\theta}_i^{\mathrm{V}} = \{\theta_{i0}^{\mathrm{V}}, \ldots, \theta_{iK}^{\mathrm{V}}\}$, and we take $h_0 = 1$ in order to account for a constant in the summations. Binary Bayesian networks which use logistic regression for the conditional distributions are often called binary sigmoidal networks [Neal 1992] and are sometimes called stochastic multi-layer perceptrons.

To complete the model, we provide a distribution $P(\mathbf{h}|\boldsymbol{\theta}^{\mathrm{H}})$ over the set of causes. Al-

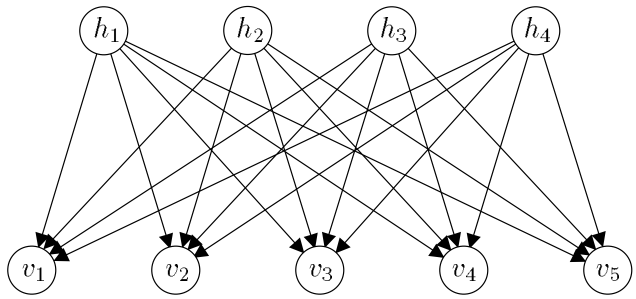

Figure 3.4: A multiple-cause network with five visible variables $\mathbf{v}$ and four hidden cause variables $\mathbf{h}$.

though it seems natural that in many cases the hidden variables h_k might be interdependent, for the sake of simplicity, I will assume for now that they are not:

$$P(\mathbf{h}|\boldsymbol{\theta}^{\mathrm{H}}) = \prod_{k=1}^{K} P(h_k|\theta_k^{\mathrm{H}}), \tag{3.24}$$

where $\boldsymbol{\theta}^{\mathrm{H}} = \{\theta_1^{\mathrm{H}}, \ldots, \theta_K^{\mathrm{H}}\}$. These probabilities can be nicely parameterized using the logistic function:

$$P(h_k|\theta_k^{\mathrm{H}}) = h_k g(\theta_k^{\mathrm{H}}) + (1 - h_k)(1 - g(\theta_k^{\mathrm{H}})). \tag{3.25}$$

An example of this type of *multiple-cause* Bayesian network is shown in Figure 3.4. Notice that the dependency-separation of the variables $\mathbf{v}$ by the set of hidden variables $\mathbf{h}$ is ensured by condition 2 described in Section 1.2.4.

Supposing that we have somehow obtained an accurate model of the true causal process for each class of data (*e.g.*, using a method described below), in order to perform classification we would like to compute the marginal probability $P(\mathbf{v}|\boldsymbol{\theta})$ for each class model. This can be computed exactly using

$$P(\mathbf{v}|\boldsymbol{\theta}) = \sum_{\mathbf{h}} P(\mathbf{v}|\mathbf{h}, \boldsymbol{\theta}^{\mathrm{V}}) P(\mathbf{h}|\boldsymbol{\theta}^{\mathrm{H}}), \tag{3.26}$$

where $\boldsymbol{\theta} = \{\boldsymbol{\theta}^{\mathrm{H}}, \boldsymbol{\theta}^{\mathrm{V}}\}$ is the entire set of parameters. However, this sum is exponential in the number of causes K and so in practice, we must use another approach. It is obvious from Figure 3.4 that probability propagation cannot be used to obtain an exact result, since the Bayesian network contains many cycles. In fact, we must use an approximate inference method.

Since exact probabilistic inference is needed for ML and MAP parameter estimation, these estimation methods are also intractable. For example, in order to perform the E-step of the EM algorithm, we must compute $P(\mathbf{h}|\mathbf{v},\boldsymbol{\theta})$, which has an exponential number (2^K) of terms.

In the next three sections, I show how Gibbs sampling, variational inference, and the stochastic Helmholtz machine can be used to approximate $P(\mathbf{v}|\boldsymbol{\theta})$ and perform maximum likelihood-bound (MLB) parameter estimation in multiple-cause networks.

3.4.1 Estimation by Gibbs sampling

In order to estimate a multiple-cause network from a set of training examples $\mathbf{v}^{(1)},\ldots,\mathbf{v}^{(T)}$, we can perform on-line gradient descent in $\log P(\mathbf{v},\mathbf{h}|\boldsymbol{\theta})$ while sampling from $P(\mathbf{h}|\mathbf{v},\boldsymbol{\theta})$ using the Gibbs sampling method described in Section 2.2.2. For the current training case $\mathbf{v}^{(t)}$, we simulate a Markov chain to obtain a configuration $\mathbf{h}^{(t)}$ of the hidden variables. (Notice that in general $\mathbf{h}^{(t)}$ will be different each time $\mathbf{v}^{(t)}$ is processed — ideally, $\mathbf{h}^{(t)}$ will have a distribution $P(\mathbf{h}|\mathbf{v}^{(t)},\boldsymbol{\theta})$.) In order to perform Gibbs sampling for the logistic multiple-cause network described above, we need to be able to sample from the distribution for each hidden variable h_k given the other cause variables and the visible variables. Since the cause variables are binary, we only need to compute a function that is proportional to $P(h_k|\{h_j\}_{j=1,j\neq k}^K,\mathbf{v},\boldsymbol{\theta})$. The two values can then be normalized to obtain $P(h_k|\{h_j\}_{j=1,j\neq k}^K,\mathbf{v},\boldsymbol{\theta})$. Since the total joint probability for $\mathbf{h}$ and $\mathbf{v}$ can be easily computed in $\mathcal{O}(KN)$ time using the ancestral ordering, the joint probability can be used to compute the conditional distribution as follows:

$$P(h_k=1|\{h_j\}_{j=1,j\neq k}^K,\mathbf{v},\boldsymbol{\theta}) = \frac{P(h_k=1,\{h_j\}_{j=1,j\neq k}^K,\mathbf{v}|\boldsymbol{\theta})}{P(h_k=0,\{h_j\}_{j=1,j\neq k}^K,\mathbf{v}|\boldsymbol{\theta}) + P(h_k=1,\{h_j\}_{j=1,j\neq k}^K,\mathbf{v}|\boldsymbol{\theta})}. \tag{3.27}$$

For a given training case $\mathbf{v}^{(t)}$, the latent variables are visited in a specified order while drawing a new value for each variable from its conditional distribution. The entire set of latent variables $\mathbf{h}$ is processed in this fashion for a specified number of times before the Markov chain is terminated and some configuration $\mathbf{h}^{(t)}$ of the latent variables is produced.

The hidden variable biases and the parameters connecting the hidden variables to the visible variables are adjusted by following the derivatives of $\log P(\mathbf{v}^{(t)},\mathbf{h}^{(t)}|\boldsymbol{\theta})$ as follows:

$$\Delta\theta_k^{\mathrm{H}} = \eta[h_k^{(t)} - g(\theta_k^{\mathrm{H}})], \tag{3.28}$$

and

$$\Delta\theta_{ik}^{\mathrm{V}} = \eta h_k^{(t)}[v_i^{(t)} - g(\textstyle\sum_{j=0}^{K}\theta_{ij}^{\mathrm{V}} h_j^{(t)})], \tag{3.29}$$

where η is a learning rate.

In order to perform classification, we would like to compute $P(\mathbf{v}|\boldsymbol{\theta})$ for a given visible vector:

$$P(\mathbf{v}|\boldsymbol{\theta}) = \sum_{\mathbf{h}} P(\mathbf{v}, \mathbf{h}|\boldsymbol{\theta}). \tag{3.30}$$

This problem can be viewed as a form of *free energy estimation* [Sheykhet and Simkin 1990; Neal 1993]. I use a very simple approximation that is quite fast and works well in practice for classification purposes. Since the number of terms in the above sum is exponential in the number of causes, I approximate it by assuming that the majority of the total probability mass is contributed by a small number of clusters in $\mathbf{h}$-space. These clusters are found by simulating a Markov chain as described above. At multiple points in the chain, the configuration of $\mathbf{h}$ and all neighboring configurations (*i.e.*, those configurations within a Hamming distance of 1) are added to a list of "significant terms". Only the neighboring states of $\mathbf{h}$ are considered because once $P(\mathbf{v}, \mathbf{h}|\boldsymbol{\theta})$ has been computed, it is easy to compute the probabilities for configurations that differ from $\mathbf{h}$ by only one bit. After a specified number of clusters have been visited in this manner, the above sum is approximated by adding up the terms for the tabulated configurations. This method for estimating $P(\mathbf{v}|\boldsymbol{\theta})$ will not work well when there is a large number of clusters in $\mathbf{h}$-space that contribute significantly to the sum. However, I have found that in practice the Gibbs sampling learning algorithm tends to favor a small number of clusters, making this approximation reasonable.

3.4.2 MLB estimation by variational inference

In this section, I review the variational method developed by Saul *et al.* [1996] for MLB estimation in sigmoidal Bayesian networks. It turns out that a product-form variational distribution leads to an intractable bound, and so the bound itself must be bounded by a tractable function.

For MLB estimation by variational inference, the Q-distribution in the likelihood bound (3.22) depends on some variational parameters $\boldsymbol{\xi}$. For the sake of simplicity, consider the bound for *one* training case $\mathbf{v}$:

$$\log P(\mathcal{D}|\boldsymbol{\theta}) \geq B_{Q\|P} = \sum_{\mathbf{h}} Q(\mathbf{h}|\boldsymbol{\xi}) \log \frac{1}{Q(\mathbf{h}|\boldsymbol{\xi})} + \sum_{\mathbf{h}} Q(\mathbf{h}|\boldsymbol{\xi}) \log P(\mathbf{h}, \mathbf{v}|\boldsymbol{\theta}). \tag{3.31}$$

MLB estimation entails iteratively maximizing this bound, first by varying $\boldsymbol{\xi}$ (the generalized E-step), and second by adjusting the model parameters $\boldsymbol{\theta}$ (the generalized M-step). The first term in this bound is the entropy of the variational distribution $Q(\mathbf{h}|\boldsymbol{\xi})$, and the second term is the expected log-probability of $\mathbf{h}$ and $\mathbf{v}$ under the variational distribution.

Here, I consider a product-form variational distribution over the K latent variables $h_1, \ldots, h_K$ (notice that h_0 is not included since it is fixed to $h_0 = 1$):

$$Q(\mathbf{h}|\boldsymbol{\xi}) = \prod_{k=1}^{K} Q(h_k|\xi_k) = \prod_{k=1}^{K} \xi_k^{h_k} (1 - \xi_k)^{(1-h_k)}, \tag{3.32}$$

where ξ_k is the probability under the variational distribution that $h_k = 1$. Using this variational distribution, the entropy term in $B_{Q\|P}$ simplifies to

$$\sum_{\mathbf{h}} Q(\mathbf{h}|\boldsymbol{\xi}) \log \frac{1}{Q(\mathbf{h}|\boldsymbol{\xi})} = \sum_{k=1}^{K} \big\{ \xi_k \log \xi_k + (1 - \xi_k) \log(1 - \xi_k) \big\}. \tag{3.33}$$

The second term in (3.31) is

$$\begin{aligned} \sum_{\mathbf{h}} Q(\mathbf{h}|\boldsymbol{\xi}) \log P(\mathbf{h}, \mathbf{v}|\boldsymbol{\theta}) &= \sum_{\mathbf{h}} Q(\mathbf{h}|\boldsymbol{\xi}) \log \big\{ \textstyle\prod_{k=1}^{K} P(h_k|\theta_k^{\mathrm{H}}) \prod_{i=1}^{N} P(v_i|\mathbf{h}, \boldsymbol{\theta}_i^{\mathrm{V}}) \big\} \\ &= \sum_{k=1}^{K} \sum_{h_k=0}^{1} Q(h_k|\xi_k) \log P(h_k|\theta_k^{\mathrm{H}}) + \sum_{i=1}^{N} \sum_{\mathbf{h}} Q(\mathbf{h}|\boldsymbol{\xi}) \log P(v_i|\mathbf{h}, \boldsymbol{\theta}_i^{\mathrm{V}}). \end{aligned} \tag{3.34}$$

Since the conditional probabilities are given by logistic regression, this term contains many expectations of nonlinear functions. The first step to simplifying these expectations is to express the conditional probability $P(h_k|\theta_k^{\mathrm{H}})$ given in (3.25) in the following way:

$$P(h_k|\theta_k^{\mathrm{H}}) = \frac{\exp(h_k \theta_k^{\mathrm{H}})}{1 + \exp(\theta_k^{\mathrm{H}})}. \tag{3.35}$$

The expectation of $\log P(h_k|\theta_k^{\mathrm{H}})$ is then

$$\sum_{h_k=0}^{1} Q(h_k|\xi_k) \log P(h_k|\theta_k^{\mathrm{H}}) = \xi_k \theta_k^{\mathrm{H}} - \log\big\{1 + \exp(\theta_k^{\mathrm{H}})\big\}. \tag{3.36}$$

Similarly, the conditional probability $P(v_i|\mathbf{h}, \boldsymbol{\theta}_i^{\mathrm{V}})$ can be written

$$P(v_i|\mathbf{h}, \boldsymbol{\theta}_i^{\mathrm{V}}) = \frac{\exp\left(v_i \sum_{k=0}^{K} \theta_{ik}^{\mathrm{V}} h_k\right)}{1 + \exp\left(\sum_{k=0}^{K} \theta_{ik}^{\mathrm{V}} h_k\right)}. \tag{3.37}$$

The expectation of $\log P(v_i|\mathbf{h}, \boldsymbol{\theta}_i^{\mathrm{V}})$ is then

$$\sum_{\mathbf{h}} Q(\mathbf{h}|\boldsymbol{\xi}) \log P(v_i|\mathbf{h}, \boldsymbol{\theta}_i^{\mathrm{V}}) = v_i \sum_{k=0}^{K} \theta_{ik}^{\mathrm{V}} \xi_k - \sum_{\mathbf{h}} Q(\mathbf{h}|\boldsymbol{\xi}) \log\left\{1 + \exp\left(\sum_{k=0}^{K} \theta_{ik}^{\mathrm{V}} h_k\right)\right\}. \tag{3.38}$$

The overall bound for $\mathbf{v}$ is

$$\begin{aligned} B_{Q\|P} = {} & \sum_{k=1}^{K} \left\{\xi_k \log \xi_k + (1-\xi_k)\log(1-\xi_k)\right\} + \sum_{k=1}^{K} \xi_k \theta_k^{\mathrm{H}} - \sum_{k=1}^{K} \log\left\{1 + \exp(\theta_k^{\mathrm{H}})\right\} \\ & + \sum_{i=1}^{N} v_i \sum_{k=0}^{K} \theta_{ik}^{\mathrm{V}} \xi_k - \sum_{\mathbf{h}} Q(\mathbf{h}|\boldsymbol{\xi}) \log\left\{1 + \exp\left(\sum_{k=0}^{K} \theta_{ik}^{\mathrm{V}} h_k\right)\right\}. \end{aligned} \tag{3.39}$$

Except for the last term, the values of these terms and their derivatives (with respect to the variational parameters) can quite easily be computed. The explicit summation over $\mathbf{h}$ in the last term cannot be reduced to a tractable form. However, the last term can be bounded by introducing some extra variational parameters $\boldsymbol{\nu}$. (See [Saul, Jaakkola and Jordan 1996] for details.) MLB estimation for the new bound $B'_{Q\|P} \leq B_{Q\|P}$ entails iteratively maximizing this bound, first by varying $\boldsymbol{\xi}$ and $\boldsymbol{\nu}$ (the generalized E-step), and second by adjusting the model parameters $\boldsymbol{\theta}$ (the generalized M-step).

3.4.3 The stochastic Helmholtz machine

A stochastic Helmholtz machine consists of a pair of Bayesian networks that are fit to training data using an algorithm that approximates MLB parameter estimation, where the bound on the likelihood may be very complex. In addition to the multiple-cause network that describes $P(\mathbf{v}, \mathbf{h}|\boldsymbol{\theta})$ (the *generative* network), there is a *recognition* network that describes $Q(\mathbf{h}|\mathbf{v}, \boldsymbol{\phi})$. The stochastic Helmholtz machine requires that the recognition network have an ancestral ordering such that it is easy to draw samples from $Q(\mathbf{h}|\mathbf{v}, \boldsymbol{\phi})$. The advantage of the stochastic Helmholtz machine over Markov chain Monte Carlo is that each sample from the recognition network is independent, as opposed to dependent on the last sample. The advantage of the stochastic Helmholtz machine over variational inference is that more complicated (*e.g.*, nonfactorial) distributions can be represented by the inference process used for MLB estimation. The main disadvantage of the stochastic Helmholtz machine is that a recognition network that is compatible with the generative network must somehow be estimated, and this can be a very difficult task when a complex recognition network is used. An example of inference in the stochastic Helmholtz machine is described in Section 2.4.4. Here, I describe the *wake-sleep algorithm* for on-line estimation of both the generative and recognition parameters ($\boldsymbol{\theta}$ and $\boldsymbol{\phi}$) [Hinton et al. 1995].

Suppose we have a current generative network (which may or may not be a good model of the data) and a current recognition network. For a parameterized recognition network, the likelihood bound in (3.22) is

$$\log P(\mathcal{D}|\boldsymbol{\theta}) \geq B_{Q||P} = \sum_{t=1}^{T} \sum_{\mathbf{h}} Q(\mathbf{h}|\mathbf{v}^{(t)}, \boldsymbol{\phi}) \log \frac{P(\mathbf{h}, \mathbf{v}^{(t)}|\boldsymbol{\theta})}{Q(\mathbf{h}|\mathbf{v}^{(t)}, \boldsymbol{\phi})}. \tag{3.40}$$

This bound can be estimated by averaging $\log P(\mathbf{h}, \mathbf{v}^{(t)}|\boldsymbol{\theta})/Q(\mathbf{h}|\mathbf{v}^{(t)}, \boldsymbol{\phi})$ over multiple recognition sweeps for each input pattern. In each recognition sweep, the recognition network is stochastically simulated to obtain a configuration $\mathbf{h}$ of the latent variables.

We would like to maximize $B_{Q||P}$ with respect to the recognition network parameters $\boldsymbol{\phi}$ for all $\mathbf{v}$, if possible. As discussed in Section 3.3.2, the unconstrained recognition distribution that maximizes this likelihood bound is

$$Q(\mathbf{h}|\mathbf{v}, \boldsymbol{\phi}) = P(\mathbf{h}|\mathbf{v}, \boldsymbol{\theta}). \tag{3.41}$$

However, except for very simple recognition networks, this optimization is intractable for the same reason that exact inference is intractable. Instead, we optimize a different function whose global maxima give identical recognition networks in certain limits to those produced by maximizing $B_{Q||P}$. The limits may not apply in practice, so that the recognition network may be slightly suboptimal.

Assume for the moment that the recognition network is consistent with the distribution $P(\mathbf{h}|\mathbf{v}, \boldsymbol{\theta})$. In this case, the parameters $\boldsymbol{\phi}$ that maximize

$$B_{P||Q} = \sum_{\mathbf{h},\mathbf{v}} P(\mathbf{h}, \mathbf{v}|\boldsymbol{\theta}) \log \frac{Q(\mathbf{h}|\mathbf{v}, \boldsymbol{\phi})}{P(\mathbf{h}, \mathbf{v}|\boldsymbol{\theta})} \tag{3.42}$$

will also maximize $B_{Q||P}$ in (3.40). (Note the reversed order of the distributions). So, for a given generative network, the optimum recognition network can be found by maximizing $B_{P||Q}$ with respect to the recognition parameters $\boldsymbol{\phi}$. The derivative of $B_{P||Q}$ with respect to a recognition network parameter ϕ is

$$\frac{\partial B_{P||Q}}{\partial \phi} = \frac{\partial}{\partial \phi} \sum_{\mathbf{h},\mathbf{v}} P(\mathbf{h},\mathbf{v}|\boldsymbol{\theta}) \log \frac{Q(\mathbf{h}|\mathbf{v},\boldsymbol{\phi})}{P(\mathbf{h},\mathbf{v}|\boldsymbol{\theta})} = \sum_{\mathbf{h},\mathbf{v}} P(\mathbf{h},\mathbf{v}|\boldsymbol{\theta}) \frac{\partial \log Q(\mathbf{h}|\mathbf{v},\boldsymbol{\phi})}{\partial \phi}. \tag{3.43}$$

So, the recognition network can be estimated using stochastic gradient descent by sampling $\mathbf{h}$ and $\mathbf{v}$ from $P(\mathbf{h},\mathbf{v}|\boldsymbol{\theta})$ using ancestral simulation, and then adjusting the recognition network parameters so as to increase the log-likelihood of the hidden variables given the visible variables. This procedure is called *sleep-phase* learning, since the recognition network is adjusted to be better suited to the "fantasies" produced by the generative network.

In practice, sleep-phase learning is only an approximation to the generalized E-step of iterative MLB estimation (Section 3.3.2) for one main reason. An ideal recognition network produces a good approximation to $P(\mathbf{h}|\mathbf{v},\boldsymbol{\theta})$ even for a vector $\mathbf{v}$ that has a very small probability under $P(\mathbf{h},\mathbf{v}|\boldsymbol{\theta})$. (This corresponds to a plausible real-world pattern that the generative network has not yet learned.) For sleep-phase learning to produce such a recognition network, an extremely large sample size must be drawn from $P(\mathbf{h},\mathbf{v}|\boldsymbol{\theta})$ in order to get an example of the unlikely vector. For the sake of tractability, a relatively small sample size must be used, which implies that the ideal recognition network cannot be found. This means that in practice, maximizing $B_{P||Q}$ does *not* give the same recognition network as would be obtained by maximizing $B_{Q||P}$. In fact, in order to prevent overfitting of the recognition network, an inconsistent parametric recognition network is used, so that the global maxima of the two functions may not even coincide.

For a given recognition network, the generative network is adjusted in the *wake-phase* using a Monte Carlo implementation of the generalized M-step of iterative MLB estimation. That is, on-line stochastic gradient descent in the likelihood bound $B_{Q||P}$ is performed with respect to the generative network parameters $\boldsymbol{\theta}$. The derivative of the bound with respect to a generative network parameter θ is

$$\begin{aligned}\frac{\partial B_{Q||P}}{\partial \theta} &= \frac{\partial}{\partial \theta} \sum_{t=1}^{T} \sum_{\mathbf{h}} Q(\mathbf{h}|\mathbf{v}^{(t)},\boldsymbol{\theta}) \log \frac{P(\mathbf{h},\mathbf{v}^{(t)}|\boldsymbol{\theta})}{Q(\mathbf{h}|\mathbf{v}^{(t)},\boldsymbol{\phi})} \\ &= \sum_{t=1}^{T} \sum_{\mathbf{h}} Q(\mathbf{h}|\mathbf{v}^{(t)},\boldsymbol{\phi}) \frac{\partial \log P(\mathbf{h},\mathbf{v}^{(t)}|\boldsymbol{\theta})}{\partial \theta}. \end{aligned} \tag{3.44}$$

For a training vector $\mathbf{v}^{(t)}$, the recognition network is used to sample values for the latent variables $\mathbf{h}$. Then, the generative parameters are adjusted so as to increase the log-likelihood of the latent variables *and* the visible variables.

The two phases of learning are usually applied in alternation. A training pattern is presented; the recognition network is used to obtain a random $\mathbf{h}$; and the generative network is adjusted. Next, the generative network is used to obtain a random $\mathbf{h}$ and then a random $\mathbf{v}$; and the recognition network is adjusted. The result of this constant mixing of the two phases is that the generative network becomes better at modelling the training data and *at the same time* tries to produce causes for the training data that can be properly inferred by the restricted recognition network. This can be seen mathematically by breaking $B_{Q||P}$ into two pieces:

$$B_{Q||P} = \log P(\mathcal{D}|\boldsymbol{\theta}) - \sum_{t=1}^{T} \sum_{\mathbf{h}} Q(\mathbf{h}|\mathbf{v}^{(t)}, \boldsymbol{\phi}) \log \frac{Q(\mathbf{h}|\mathbf{v}^{(t)}, \boldsymbol{\phi})}{P(\mathbf{h}|\mathbf{v}^{(t)}, \boldsymbol{\theta})}. \tag{3.45}$$

The first term encourages the generative network to model the data, whereas the second term (a negative Kullback-Leibler pseudo-distance) encourages it to be compatible with the recognition network. As a result of the latter, for a generative network that is estimated using the wake-sleep algorithm, the global maxima of $B_{Q||P}$ and $B_{P||Q}$ often *do* coincide.

Assuming that under the recognition distribution, the latent variables are independent given the visible variables, we have:

$$Q(\mathbf{h}|\mathbf{v}, \boldsymbol{\phi}) = \prod_{k=1}^{K} Q(h_k|\mathbf{v}, \boldsymbol{\phi}_k), \tag{3.46}$$

Also, consider modelling each of these components using logistic regression:

$$Q(h_k|\mathbf{v}, \boldsymbol{\phi}_k) = h_k g(\textstyle\sum_{i=0}^{N} \phi_{ki} v_i) + (1 - h_k)(1 - g(\sum_{i=0}^{N} \phi_{ki} v_i)), \tag{3.47}$$

where we take $v_0 = 1$ in order to account for a constant in the exponents. This recognition network is shown in Figure 3.5.

For this logistic recognition network, the recognition parameters are adjusted as follows during the sleep phase, in order to increase $\log Q(\mathbf{h}|\mathbf{v}, \boldsymbol{\phi})$:

$$\Delta\phi_{ki} = v_i(h_k - g(\textstyle\sum_{j=0}^{N} \phi_{kj} v_j)). \tag{3.48}$$

It turns out that in many practical cases this recognition network is sufficient for producing good density estimates. However, if it is estimated in conjunction with a fixed generative network that describes the simple burglar alarm problem (see Section 2.2.3), the likelihood bound $B_{Q||P}$ may actually *decrease*. Consider how the recognition network is modified for fantasies where the burglar alarm is ringing. We simulate the generative network, obtaining

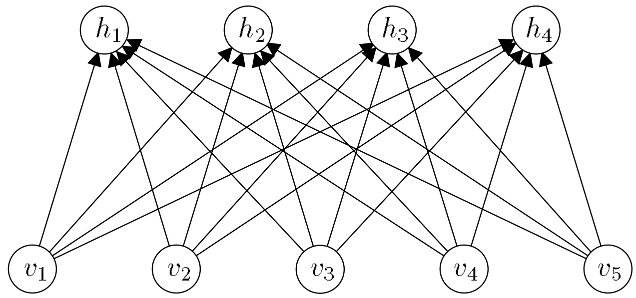

Figure 3.5: A recognition network that implements $Q(\mathbf{h}|\mathbf{v},\boldsymbol{\phi})$ for the generative network shown in Figure 3.4.

values for b, e, and a, and discard those samples for which $a \neq 1$. For the recognition network parameters ϕ^{BA} and ϕ^{EA} that connect the common consequence a to the two causes b and e, the expected learning updates become

$$\begin{aligned} \mathrm{E}\big[\Delta\phi^{BA}\big] &= \mathrm{E}\big[b - g(\phi^{BA} + \phi^{B0})\big] = P(b=1|a=1) - g(\phi^{BA} + \phi^{B0}) \\ \mathrm{E}\big[\Delta\phi^{EA}\big] &= \mathrm{E}\big[e - g(\phi^{EA} + \phi^{E0})\big] = P(e=1|a=1) - g(\phi^{EA} + \phi^{E0})), \end{aligned} \tag{3.49}$$

where ϕ^{B0} and ϕ^{E0} are the recognition biases for b and e. Each connection is modified so as to predict as closely as possible the *marginal* posterior distributions $P(b=1|a=1)$ and $P(e=1|a=1)$ over the corresponding causes b and e. After training, the recognition distribution over b and e given $a=1$ will be the product of the marginals. For the configuration $b=1$, $a=1$,

$$Q(b=1, e=1|a=1, \boldsymbol{\phi}) = P(b=1|a=1)P(e=1|a=1) = 0.751 \times 0.349 = 0.262, \tag{3.50}$$

where the values for $P(b=1|a=1)$ and $P(e=1|a=1)$ were computed from (2.13). This value is quite a bit higher than the correct value of $P(b=1, e=1|a=1) = 0.116$. In fact, if we assume that b and e are independent given $a=1$, the recognition distribution that maximizes the likelihood bound $B_{Q||P}$ has $\hat{P}(b=1, e=1|a=1) = 0.177$. This is an example where maximizing $B_{P||Q}$ is a poor approximation to maximizing $B_{Q||P}$. Notice, however, that the problem arises because we are using an inconsistent recognition network.

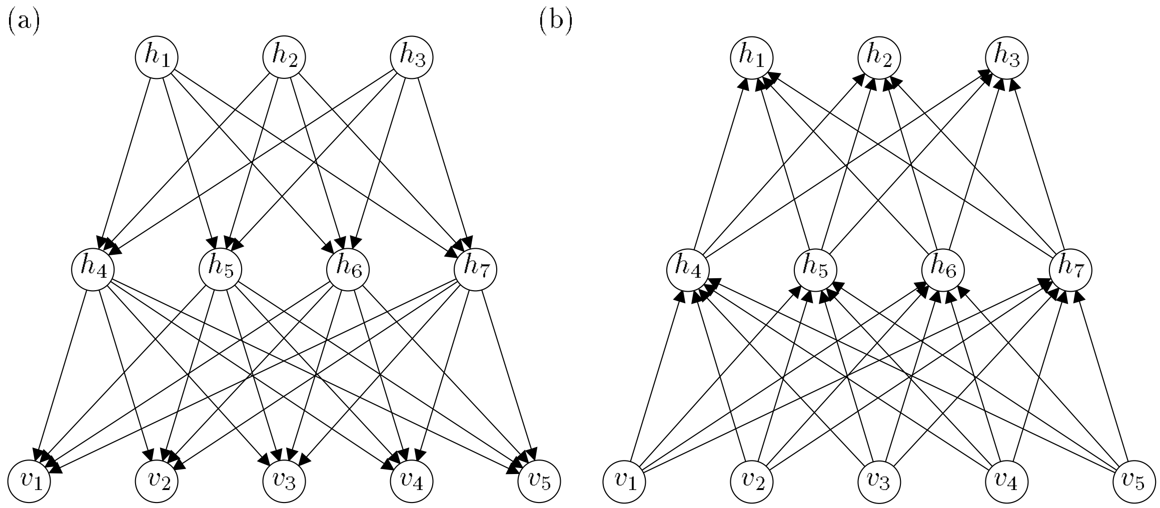

Figure 3.6: (a) A hierarchical network with three extra "meta-cause" variables which produce correlations between the cause variables (middle layer). (b) A similarly layered recognition network.

3.4.4 Hierarchical networks

Earlier in this section, I presented multiple-cause networks with the assumption that the causes were independent in the generative network (see (3.24) on page 63). Even with the assumption that the causes are independent, it is still possible to represent quite complex correlations in the visible variables $\mathbf{v}$. However, in many cases the causes are certainly interdependent. For example, if the causes variables represent the presence or absence of various objects in facial images, we expect that both a toque and a top-hat are not present simultaneously.

In order to model interdependent causes, we can simply add another layer of "meta-causes" at a higher level in the network. Even if we assume that the "meta-causes" are independent, the network can still represent fairly complex relationships between the causes. Such a *hierarchical* network is shown in Figure 3.6a. We have already seen examples of hierarchical networks, such as the network used in the bars problem example (Figure 1.7).

The parameter estimation methods already described in this chapter are applicable to hierarchical networks with any reasonable depth. Of particular interest, however, is the recognition model for the Helmholtz machine. Figure 3.6b shows a recognition model that is appropriate for the network in Figure 3.6a. Note that the top layer of hidden variables could receive input from the bottom layer of visible variables, not just the middle layer of hidden variables. However, this introduces extra parameters into the Helmholtz machine, which may worsen the effect of overfitting. In my experiments, I use layered generative and recognition networks like the ones shown in Figure 3.6. In some cases, adding extra connections may help.

3.4.5 Ensembles of networks

According to the Bayesian doctrine for prediction, when using a density model $P(\mathbf{v}|\boldsymbol{\theta})$ to estimate the probability of $\mathbf{v}$, we ought to integrate out the model parameters $\boldsymbol{\theta}$. If we specify a prior $P(\boldsymbol{\theta})$ and measure the likelihood $P(\mathcal{D}|\boldsymbol{\theta})$ of the training data $\mathcal{D}$, Bayesian inference uses the posterior $P(\boldsymbol{\theta}|\mathcal{D}) \propto P(\boldsymbol{\theta})P(\mathcal{D}|\boldsymbol{\theta})$ (where the constant of proportionality does not depend on $\boldsymbol{\theta}$) to obtain a probability

$$P(\mathbf{v}|\mathcal{D}) = \int_{\boldsymbol{\theta}} P(\mathbf{v}|\boldsymbol{\theta})P(\boldsymbol{\theta}|\mathcal{D})d\boldsymbol{\theta}. \tag{3.51}$$

For example, this integral can be approximated using Laplace's approximation [Spiegelhalter and Lauritzen 1990], Markov chain Monte Carlo methods [Neal 1993; Neal 1996] or variational techniques [Jaakkola and Jordan 1997]. Here, I consider the ensemble method, which is less sophisticated than the above approaches, but is also easier to implement and and in practice usually gives a significant improvement over MAP parameter estimation.

Suppose we perform MAP parameter estimation using multiple restarts (different random initial parameters) so that we have an *ensemble* of M models, where model m has parameters $\boldsymbol{\theta}_m$. Each model may correspond to a different local maximum of the posterior $P(\boldsymbol{\theta}|\mathcal{D})$, and we assume that each model is equally likely in the posterior. We then approximate the above integral by

$$\hat{P}(\mathbf{v}|\mathcal{D}) = \frac{1}{M}\sum_{m=1}^{M} P(\mathbf{v}|\boldsymbol{\theta}_m). \tag{3.52}$$

If $P(\mathbf{v}|\boldsymbol{\theta})$ does not change much over the width of each mode in the posterior, then as long as the modes are properly represented by the ensemble of models, $\hat{P}(\mathbf{v}|\mathcal{D})$ will be very close to the correct value $P(\mathbf{v}|\mathcal{D})$ given by integration. On the other hand, if there is a mode in the posterior that is so wide that $P(\mathbf{v}|\boldsymbol{\theta})$ *does* vary significantly across the mode, then $\hat{P}(\mathbf{v}|\mathcal{D})$ may be quite different from $P(\mathbf{v}|\mathcal{D})$. This is because only a peak in the posterior is being included in the sum, while the mass surrounding the peak is being ignored, even though the corresponding predictions are quite variable.

Since the Bayesian networks described above are flexible models, we expect that with limited training data they may have multiple data likelihood optima (corresponding to multiple peaks in the posterior, if we assume a uniform prior over network parameters). For this reason, when time permits, a significant classification rate improvement can be obtained by using an ensemble of networks for each class of data. The classification decision is then based on the average probabilities computed from the ensemble for each class.

3.5 Classification of hand-written digits

An interesting and useful pattern classification problem is the classification of hand-written digits. In this section, I present results on the classification of 8×8 binary images of hand-written digits made available by the US Postal Service Office of Advanced Technology. I compare the following Bayesian network methods: logistic autoregressive classifier (LARC-1), a stochastic Helmholtz machine with one hidden layer (SHM-1), a stochastic Helmholtz machine with two hidden layers (SHM-2), and an ensemble of stochastic Helmholtz machines with one hidden layer (ESHM-1). In order to place the performance of these networks in context, I include the following methods: classification and regression trees (CART-1), the naive Bayes classifier (NBAYESC-1), and the k-nearest neighbor method (KNN-CLASS-1). The performances of these classifiers are assessed using 5 different training set sizes (120, 240, 480, 960 and 1920 cases) so that the effect of the number of training cases on each method can be studied. After describing the classifiers and the methods used to estimate them, I present and discuss the performance results.

3.5.1 Logistic autoregressive classifiers (LARC-1,ELARC-1)

LARC-1 models each of the 10 classes of data using a logistic autoregressive network (see Section 3.2), where the variables are ordered in a raster-scan fashion. Once each of the 10 networks have been estimated from the training data, a test pattern is classified by outputing the class corresponding to the network that gives the greatest likelihood to the pattern.

Before estimating each network from its respective class of training patterns, the double precision parameters $\boldsymbol{\theta}$ were initialized to uniformly random values on $[-0.01, 0.01)$. Over-fitting was prevented by using MAP estimation with a scaled Gaussian parameter prior. The prior variance of the first input was set to $\sigma_1^2 = 4.0$. A conjugate gradient algorithm was used for MAP estimation.

ELARC-1 uses an ensemble of 8 logistic autoregressive networks, where each element in the ensemble uses a different ordering of the variables in $\mathbf{v}$. One of the elements uses the raster-scan ordering, whereas the other 7 elements use a randomly selected ordering. The probability of a test pattern for a given class is estimated by averaging the probability estimates from each of the 8 networks in the ensemble for that class.

3.5.2 The Gibbs Machine (GM-1)

GM-1 models the distribution of each of the 10 classes of data using a logistic multiple-cause network of the type shown in Figures 3.4, that is trained using Gibbs sampling. Each network has 64 visible binary (0/1) variables (8×8) and one hidden layer of 16 binary (0/1) variables. Once the 10 networks have been estimated, classification of a test pattern proceeds by estimating the probability of the pattern under each network, using the method described in Section 3.4.1. The class corresponding to the network that gives the highest probability is output as the prediction.

Before estimating a network using Gibbs sampling, all of its double precision parameters ($\boldsymbol{\theta}$) were initialized to uniformly random values on $[-0.01, 0.01)$. For each training pattern, a single configuration of the hidden variables was obtained by performing 10 sweeps of Gibbs sampling, while annealing the network from a temperature of 5.0 to 1.0 using a $1/\tau$ schedule, where τ is the sweep count. Then, the network parameters were adjusted using a learning rate of 0.01. For a training set of T patterns, a randomly chosen set of $\lfloor T/3 \rfloor$ cases were set aside as a "validation" set. By monitoring the probability estimate for this validation data, early-stopping was used to prevent overfitting. After every 10 epochs of learning (1 epoch = one sweep through the remaining $\lceil 2T/3 \rceil$ training cases), for each validation pattern, 10 sweeps of Gibbs sampling with annealing were performed as described above, and then 20 sweeps of Gibbs sampling at unity temperature were performed to obtain 20 configurations. Then, the probability of the validation pattern was estimated by computing the probability mass associated with each configuration and its 1-nearest neighbors. Each network was trained for a minimum of 100 epochs (the validation probability estimate was still computed every 10 epochs in this interval). Then, learning was stopped after the current epoch n, if the epoch $n_{\max}$ at which the maximum validation probability estimate occurred took place no less than $n/3$ epochs ago. Also, in order to terminate learning runs where the validation probability estimate continued to increase asymptotically towards a limit, a maximum of 2000 training epochs were performed. In summary, learning was stopped at epoch n if $n \geq 2000$ or if $n_{\max} \leq 2n/3$ and $n \geq 100$. (A similar early stopping technique has been used with regression models [Rasmussen 1996].)

3.5.3 The mean field (variational) Bayesian network (MFBN-1)

MFBN-1 models the distribution of each of the 10 classes of data using a logistic multiple-cause network of the type shown in Figures 3.4. Each network is fit using the variational technique described in Section 3.4.2. Each network has 64 visible binary (0/1) variables (8×8) and one hidden layer of 16 binary (0/1) variables. Once the 10 networks have been estimated, classification of a test pattern proceeds by computing the likelihood bound for each network, using the method described in Section 3.4.2. The class corresponding to the network that gives the highest bound is output as the prediction.

Before estimating a network using the variational method, all of its double precision parameters ($\boldsymbol{\theta}$) were initialized to uniformly random values on $[-0.01, 0.01)$. For each training case, the variational parameters were initialized to uniformly random values on $[-0.01, 0.01)$. The variational bound was increased at each generalized E-step using the following iterative method [Saul, Jaakkola and Jordan 1996]. After each iteration, if the bound did not increase by more than 1% then no more iterations were performed for the current training case. A maximum of 10 iterations was performed. These algorithm parameters were suggested by Jaakkola (personal communication). The variational bound was increased at each generalized M-step using batch gradient descent with a learning rate of 0.01.

The validation procedure used to train each network was identical to the one used for GM-1, except that the variational bound was used instead of an estimate of the validation case probability. The validation bound was computed every 5 epochs. No fewer than 100 epochs were performed, and no more than 1000 epochs were performed.

3.5.4 Stochastic Helmholtz machines (SHM-1,SHM-2,ESHM-1)

SHM-1 models the distribution of each of the 10 classes of data using a stochastic Helmholtz machine with 64 visible binary (0/1) variables (8×8) and one hidden layer of 16 binary (0/1) variables. The generative and recognition networks are of the form shown in Figures 3.4 and 3.5, and logistic regression is used to implement the conditional relationships. The likelihood bound for a given input pattern is estimated using 20 recognition sweeps. Once the 10 machines have been estimated, classification of a test pattern proceeds by estimating the likelihood bound for each machine. The class corresponding to the machine that gives the highest likelihood bound estimate is output as the prediction.

Before estimating a Helmholtz machine using the wake-sleep algorithm, all of its double precision parameters ($\boldsymbol{\theta}$ and $\boldsymbol{\phi}$) were initialized to uniformly random values on $[-0.01, 0.01)$. A learning rate of 0.01 was used for both phases of learning. The validation procedure used to train each machine was identical to the one used for GM-1, except that instead of obtaining an estimate of the validation set probability as described above, 20 epochs of recognition passes were performed on the validation set to obtain an estimate of the likelihood bound for the validation data.

SHM-2 is similar to SHM-1, except that it uses stochastic Helmholtz machines with a visible layer of 64 binary variables, a middle hidden layer of 16 binary variables, and a top hidden layer of 8 binary variables. The generative and recognition networks are of the form shown in Figure 3.6.

ESHM-1 uses an ensemble of 8 SHM-1 networks to model each class of patterns. Each network in an ensemble is estimated using the above procedure, where a different randomly chosen validation set of $\lfloor T/3 \rfloor$ patterns is set aside for each network. Also, different initial random parameters are chosen for each network. Once 8 networks have been estimated for each of the 10 classes of data, a test pattern is processed by approximating 8 likelihood bounds for each data class. These are averaged together within each class to obtain 10 average likelihood bounds. The final class decision for the test pattern is based on these averages.

3.5.5 The classification and regression tree (CART-1)

This tree-based classifier has previously been run on several classification tasks in DELVE [Rasmussen et al. 1996]. CART-1 uses a binary decision tree to classify the test patterns, where each node in the tree makes a binary decision based on an axis-aligned decision surface in the input space, and each leaf in the tree has a class label. A test pattern is classified by traversing the tree from the root to a leaf, while following the decisions at each node. That is, decision node d_j looks at a particular input variable v_{i_j} and compares it to a threshold t_j. If $v_{i_j} > t_j$, the right child is chosen, and otherwise the left child is chosen. When a leaf is reached, the class of the leaf node is output by the classifier.

The tree is constructed from a training set using 10-fold cross validation. The details of how the tree is produced can be found in [Breiman et al. 1984][1].

3.5.6 The naive Bayes classifier (NBAYESC-1)

The naive Bayes method of modelling can be viewed as a multiple-cause Bayesian network where there aren't any hidden cause variables. That is, we assume that each of the inputs is independent given the class identity. For the binary input case, this model becomes very simple. The naive Bayes model for each class of data is

$$P(\mathbf{v}|\boldsymbol{\theta}) = \prod_{i=1}^{N} \theta_i^{v_i}(1-\theta_i)^{1-v_i}, \tag{3.53}$$

[1] I used Version 1.1 of the CART software provided by California Statistical Software Inc., 961 Yorkshire Ct. Lafayette, California 94549, Tel: +1 415 283 3392.

where $\theta_i \in [0,1]$ is the probability that $v_i = 1$ under the model. For a given class of training data, I use the Bayesian method to obtain a minimum squared-loss estimate of $P(\mathbf{v})$, assuming a uniform prior for $\boldsymbol{\theta}$:

$$P(\mathbf{v}) = \prod_{i=1}^{N} \left[\frac{f_i + 1}{T+2}\right]^{v_i} \left[\frac{T - f_i + 1}{T+2}\right]^{1-v_i}. \tag{3.54}$$

Once one such estimate is obtained for each of the 10 classes of training data, a test pattern is classified by choosing the class that gives the probability to the pattern. Notice that if $f_i = 0$ or T, the probability $P(\mathbf{v})$ is not 0 or 1. This prevents overfitting.

3.5.7 The k-nearest neighbor classifier (KNN-CLASS-1)

This is the only nonparametric classifier studied in this section. The software I used was contributed to DELVE by Michael Revow [Rasmussen et al. 1996]. In order to guess the class of an input pattern $\mathbf{v}$, the k-nearest neighbor classifier considers the classes of the k training patterns that are nearest to $\mathbf{v}$ in Euclidean distance. Let n_j $j = 0, \ldots, J-1$ be the number of such training patterns in class j, so that $\sum_{j=0}^{J-1} n_j = k$. Then, the k-nearest neighbor classifier outputs the most frequent class:

$$j^* = \mathrm{argmax}_j \; n_j. \tag{3.55}$$

If two or more classes have the maximum number of k-nearest neighbor training patterns, then the classifier chooses the class whose training patterns are closest to $\mathbf{v}$ on average in Euclidean distance.

k is chosen using leave-one-out cross validation. If there are T training patterns, T new training sets with $T-1$ patterns each are produced by leaving each pattern out once. k is set to 1, and the misclassification rate on the left out patterns is computed using the k-nearest neighbor classifier. Then, k is increased and this process is repeated until $k = T-1$. The value for k that gives the lowest leave-one-out cross-validation error is used to make predictions for the test patterns.

In order to estimate the *probability* that $\mathbf{v}$ comes from each class, the k-nearest neighbor method uses

$$p_j = n_j/k. \tag{3.56}$$

In this case, the squared difference between the predicted probability vector and the true class identity vector (a vectors of 0's with a single 1) is used as the cross-validation metric to determine k. See the DELVE manual [Rasmussen et al. 1996] for more information.

3.5.8 Results

The performances of the classification methods described above were assessed using the DELVE (data for evaluating learning in valid experiments) system [Rasmussen et al. 1996]. Under this system, the digit classification problem that I am interested in is called a *proto-task*. A particular choice of training set size (*e.g.*, 120 training patterns) is called a *task*. In order to get an accurate measure of the performances of the methods (with error bars), each method was trained and tested at least 4 times using disjoint training set - test set pairs (each of which is called a *task instance*). An original data set consisting of 10,960 patterns (1096 of each class) was partitioned into a training collection of 7680 patterns and a test collection of 3280 patterns. For each of the tasks with training set sizes 120, 240, 480, and 960, the training collection was partitioned into 8 *disjoint* training sets; for the task with training set size 1920, the training collection was partitioned into 4 disjoint training sets. Notice that for the tasks with training set sizes 960 and 1920, all of the training partition cases were used, whereas for the tasks with training set sizes 120, 240, and 480, not all of the training partition cases were used. For each of the tasks with training set sizes 120, 240, 480, and 960, the test collection was partitioned into 8 *disjoint* test sets with 410 patterns each; for the task with training set size 1920, the test collection was partitioned into 4 disjoint test sets with 820 patterns each. This way of partitioning the data eliminates the dependence between each of the 8 experiments performed to assess the performance of each method on each task instance.

Figure 3.7 shows the losses (fraction of patterns misclassified) for each of the tasks (five boxes). Each horizontal bar gives an estimate of the expected loss for a particular method on a particular task. The methods are ordered (from left to right within each box): CART-1, NBAYESC-1, KNN-CLASS-1, MFBN-1, SHM-1, SHM-2, GM-1, ESHM-1, LARC-1 — this is the same ordering as is given top to bottom in the lower-left hand region of the figure. Each vertical bar gives an estimate of the error (one standard deviation) for the corresponding estimate of the expected loss. Numbers in the boxes lying beneath the x-axis are p-values (in percent) for a paired t-test. Choose your favorite method from the list in the lower left-hand corner of the figure and scan from left to right. Whenever you see a number, that means that another method has performed better than your favorite method, *with the given statistical significance*. A low p-value indicates the difference in the misclassification rates is very significant. More precisely, a p-value is an estimate of the probability of obtaining a difference in performance that is equal to or greater than the observed difference, given that we assume the two methods actually perform equally well

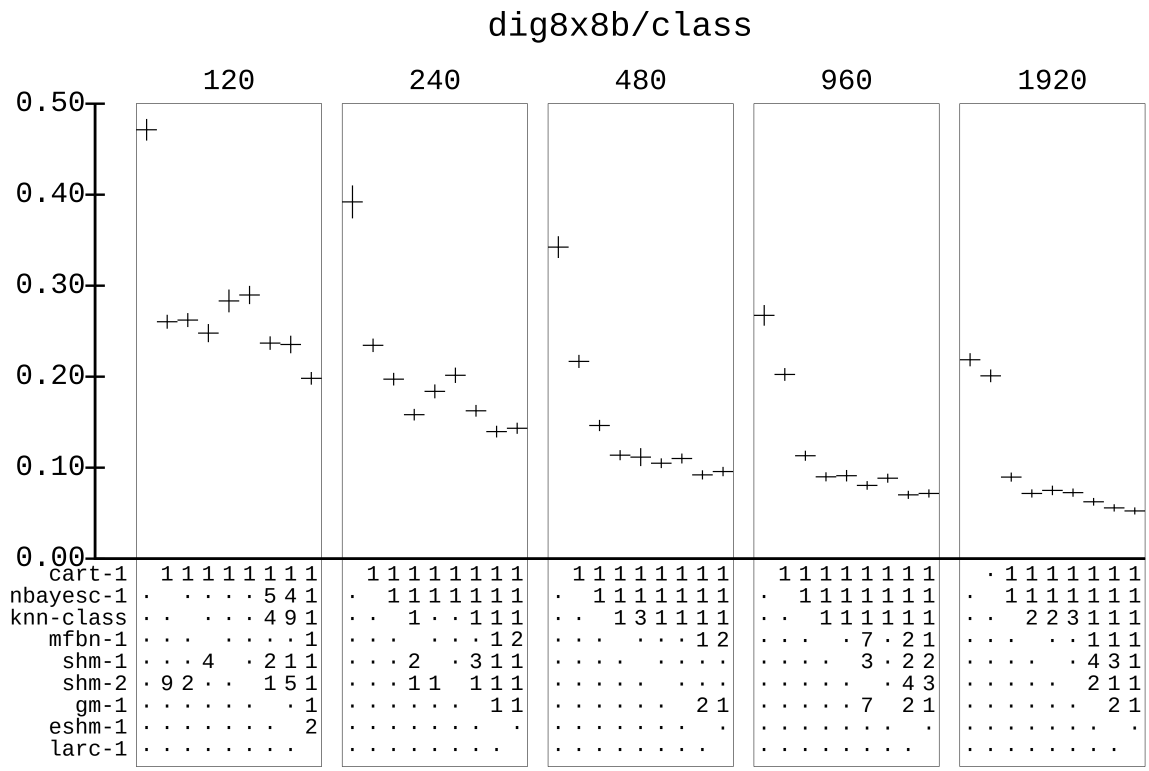

Figure 3.7: Estimates of expected fractions of misclassified patterns for nine methods trained on five different sizes of training sets.

(the null hypothesis).

The ESHM-1 and LARC-1 methods clearly outperform all other methods for all tasks. If we scan the p-values for these two methods from left to right, we see that there is only a single method that performs significantly better than ESHM-1, and that is LARC-1 on the task with the smallest training set size (120). I found that the performance of ELARC-1 (ensemble of logistic autoregressive networks, not shown) was indistinguishable from plain LARC-1 with respect to classification error. In contrast, ESHM-1 performs significantly better than SHM-1. It is of particular interest that LARC-1, which contains no latent variables, performs slightly better than the methods that contain latent variables.

GM-1 performs the best out of all approximate maximum likelihood methods, including MFBN-1, SHM-1, and SHM-2. However, GM-1 required an order of magnitude more training and validation time than SHM-1. For this reason, an ensemble of logistic multiple cause networks was not considered for the Gibbs sampling estimation method. Table 3.1 shows the average time taken to train and test each method for each training set size on a 200MHz MIPS R4400 processor. MFBN-1 also required an order of magnitude more training and validation time than SHM-1, and so an ensemble of mean field Bayesian networks was not

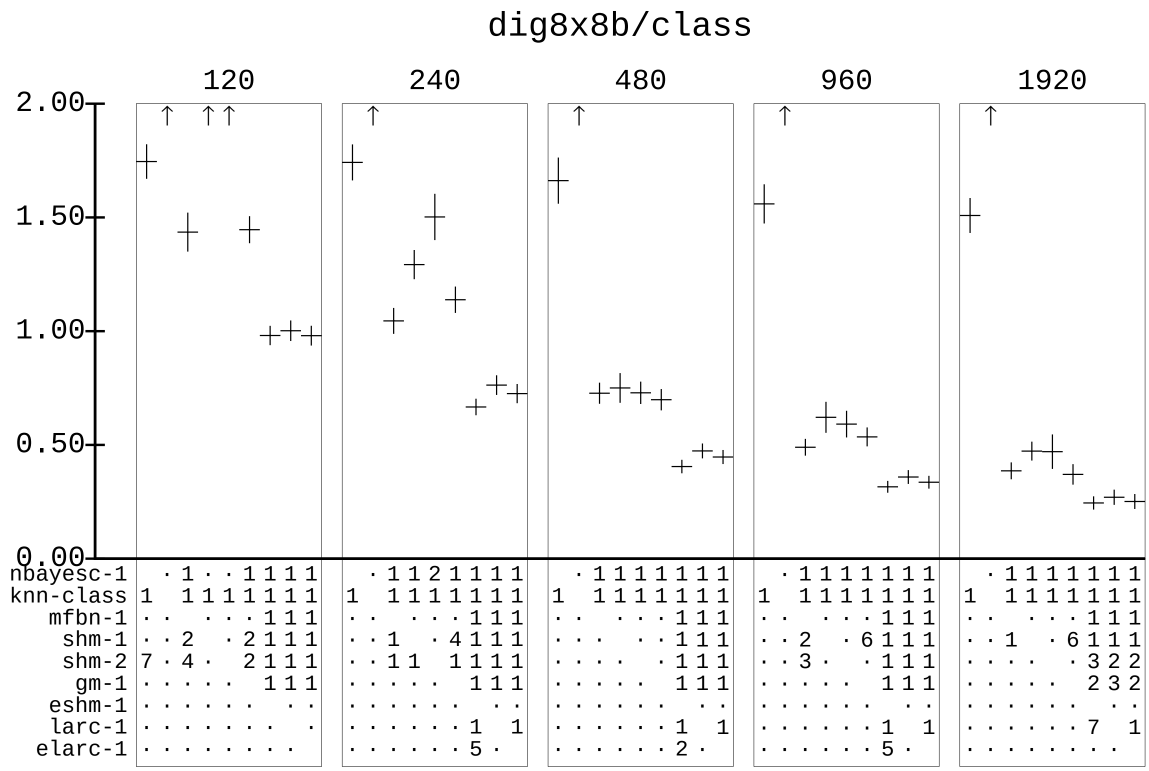

Figure 3.8: Estimates of expected negative log-probability of the true digit identity given by each of eight methods trained on five different sizes of training sets.

Table 3.1: Average time in minutes required to train and test the methods from Figures 3.7 and 3.8 for each of the training set sizes.

	Training set size				
Method	120	240	480	960	1920
CART-1	0.6	1.1	1.9	4.7	5.1
NBAYESC-1	0.0	0.0	0.0	0.0	0.0
KNN-CLASS-1	0.2	1.0	4.6	25.7	192.9
MFBN-1	19.8	64.4	130.5	216.9	344.6
SHM-1	2.3	5.4	11.8	21.8	46.0
SHM-2	3.7	7.8	22.3	41.7	77.6
GM-1	41.7	85.8	176.8	238.0	396.3
ESHM-1	19.4	44.7	95.5	186.7	358.9
LARC-1	0.2	0.5	1.2	3.0	6.5

considered.

Figure 3.8 shows the performance results for soft decisions (the loss is the negative log-probability of the true class under each model). The methods are: NBAYESC-1, KNN-

CLASS-1, MFBN-1, SHM-1, SHM-2, GM-1, ESHM-1, LARC-1, ELARC-1. In this case ELARC-1 performs significantly better than LARC-1. Also, in this case ESHM-1 performs slightly better than LARC-1 and ELARC-1.

3.6 Extracting structure from noisy binary images

The Bayesian networks described so far in this chapter have been supervised, in the sense that they are trained with pattern - class label pairs. Can we come up with algorithms that can organize the training data into meaningful classes, without the help of class labels? This section and the following one give examples of Bayesian network models that exhibit this emergent classification behavior. In this section, I show how a network can learn to recognize vertical and horizontal lines in synthetic images, and even learn to recognize the orientation of those lines. In the following section, I show how a network can learn to extract both categorical and continuous structure simultaneously.

An interesting problem relevant to vision is that of extracting independent horizontal and vertical bars from an image [Foldiak 1990; Saund 1995; Zemel 1993; Dayan and Zemel 1995; Hinton et al. 1995]. Figure 3.9 shows 48 examples of the binary images I am interested in. Each image is produced by randomly choosing between horizontal and vertical orientations with equal probability. Then, each of the 16 possible bars of the chosen orientation is independently instantiated with probability 0.25. Finally, additive noise is introduced by randomly turning on with a probability of 0.25 each pixel that was previously off. So, the production of these images involves three levels of hierarchy: the first and lowest level represents pixel noise, the second represents bars that consist of groups of 16 pixels each, and the third represents the overall orientation of the bars in the image.

3.6.1 Wake-sleep parameter estimation

Using the wake-sleep algorithm, I trained a stochastic binary Helmholtz machine that has 4 top-layer ("meta-cause") variables, 36 middle-layer variables, and 256 bottom-layer image variables. Each conditional distribution is modelled using logistic regression. Learning is performed through a series of iterations, where each iteration consists of one bottom-up wake phase sweep used to adjust the generative network parameters and one top-down sleep phase sweep used to adjust the recognition network parameters. Every 5000 iterations, the recognition network is used to obtain an estimate (with error bars) of the lower bound on the data log-likelihood under the generative network. To do this, 1000 recognition sweeps are performed without learning. During each recognition sweep, binary values for the hidden

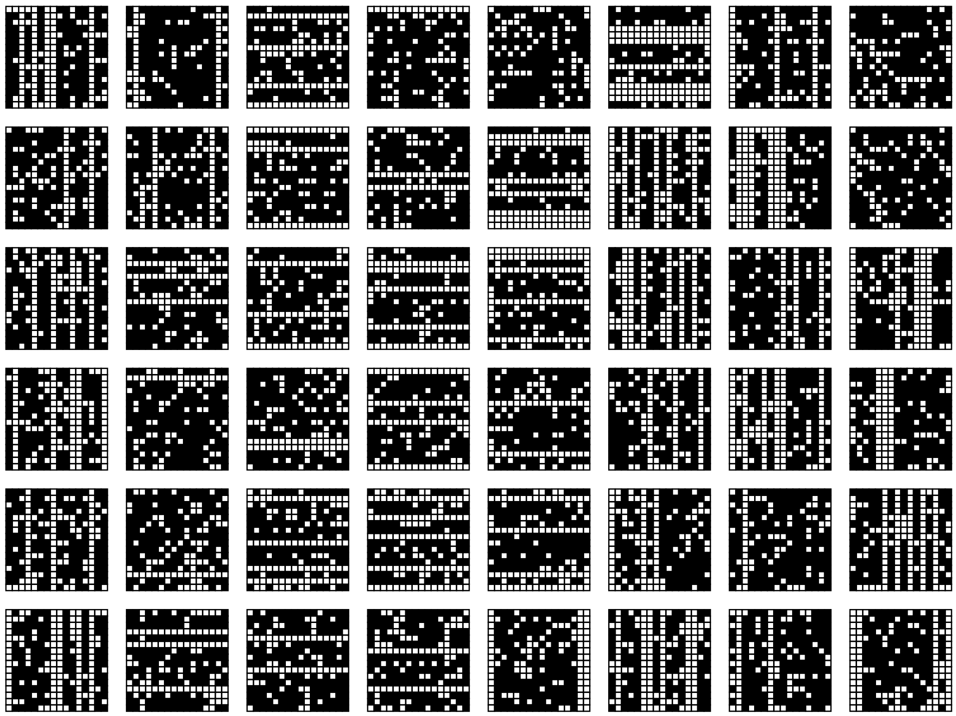

Figure 3.9: Examples of training images whose production involved three levels of hierarchy. First, an orientation (*i.e.*, horizontal or vertical) is randomly chosen with fair odds. Second, each bar of the chosen orientation is randomly instantiated with probability 0.25. Third, additive noise is introduced by randomly turning on with a probability of 0.25 each pixel that was previously off.

variables are obtained for the given training image. The log-likelihood of the values of *all* the variables under the generative network minus the log-likelihood of the hidden variable values under the recognition network gives an unbiased estimate of the log-likelihood bound (3.40). In this way, I obtain 1000 i.i.d. noisy unbiased estimates of the log-likelihood bound. The average of these values gives a less noisy unbiased estimate. Also, the variance of this estimate is estimated by dividing the sample variance by 999.

I am interested in solutions where the generative network can construct the image by adding features, but cannot remove previously instantiated features. If the network parameters are in no way constrained to favor this type of solution, perceptually unattractive solutions are found (see Section 3.6.3). So, I constrain the parameters of the logistic regression model that connects the middle layer to the bottom layer to be positive by setting to zero any negative weights every 20th learning iteration. In order to encourage a solution where each image can be succinctly described by the minimum possible number of causes in the middle layer, I initialize the middle-layer generative biases to -4.0 which favors most middle-layer variables being inactive (value of 0) on average. All other parameters

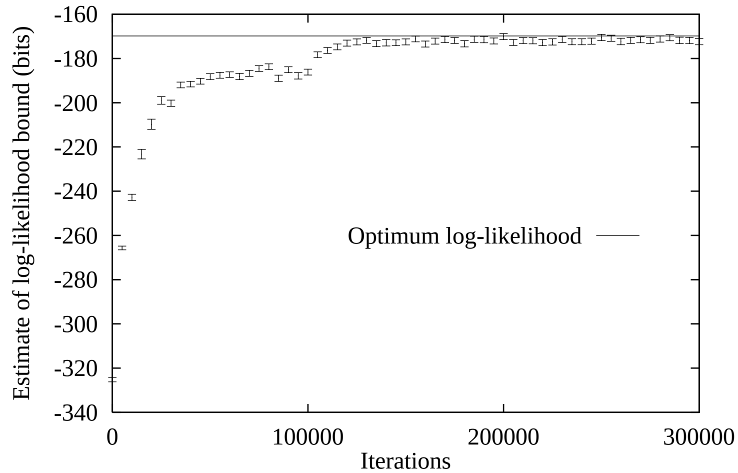

Figure 3.10: Variation of the lower log-likelihood bound with the number of wake-sleep learning iterations for the stochastic Helmholtz machine.

are initialized to 0.0. For the first 100,000 iterations, a learning rate of 0.1 is used for the generative parameters of the model feeding into the bottom layer and for the recognition parameters of the model feeding into the middle layer; the remaining learning rates are set to 0.001. After this, learning is accelerated by setting all learning rates to 0.01.

Figure 3.10 shows the learning curve for the first 300,000 iterations of a simulation consisting of a total of 1,000,000 iterations. Aside from several minor fluctuations, the wake-sleep algorithm maximizes the log-likelihood bound in this case. Eventually, the bound converges to the optimum value (-170 bits) shown by the solid line. This value is computed by estimating the average log-likelihood of the data under the method that was used to produce the data, *i.e.*, the negative entropy of the training data.

By examining the generative parameters after learning, we see that the wake-sleep algorithm has extracted the correct 3-level hierarchical structure. Figure 3.11 shows the parameters for the generative logistic regression models feeding into and out of the middle layer in the network. A black blob indicates a negative parameter and a white blob indicates a positive parameter; the area of each blob is proportional to the magnitude of the parameter (the largest value shown is 7.77 and the smallest value shown is -7.21). There are 36 blocks arranged in a 6×6 grid and each block corresponds to a middle-layer variable. The 4 blobs at the upper-left of a block show the parameters that connect each of the top-layer variables to the corresponding middle-layer variable. The single blob at the upper-right of a block shows the bias for the corresponding middle-layer variable. The 16×16 matrix that forms the bulk of a particular block shows the parameters that connect the corresponding middle-layer variable to the bottom-layer image. These matrices clearly

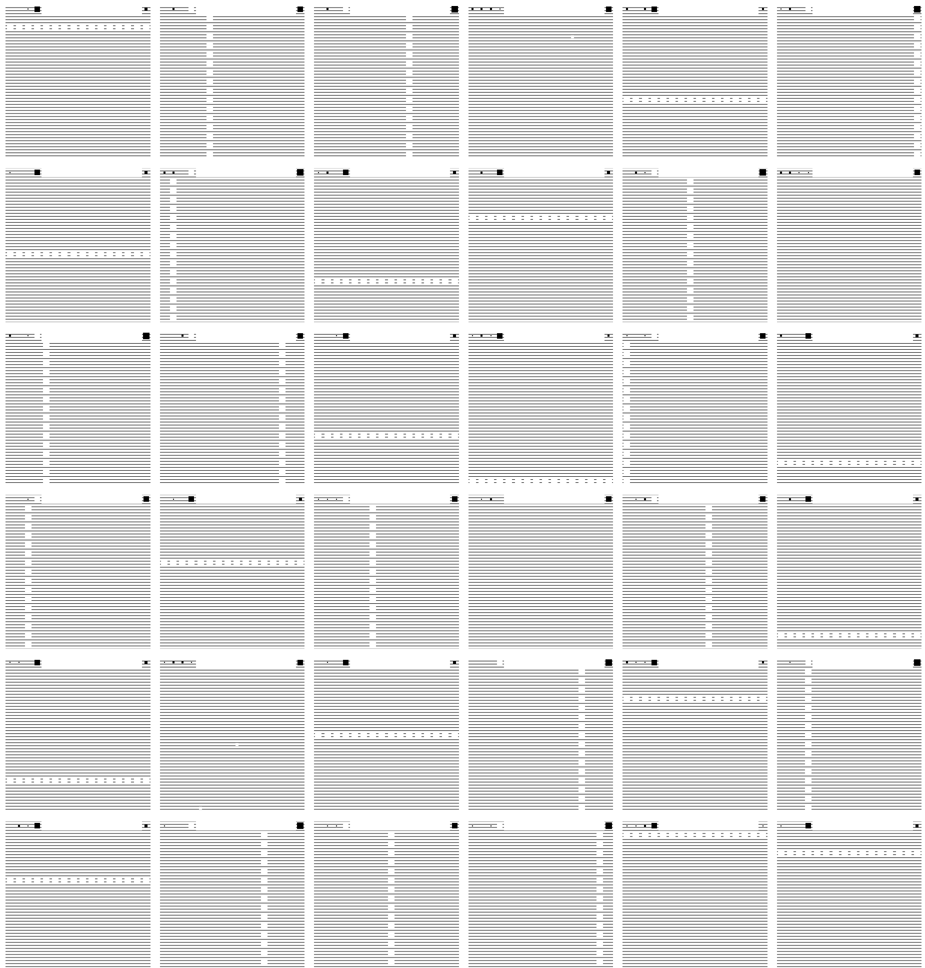

Figure 3.11: Parameters for connections that feed into and out of the middle-layer variables in the generative network. A black blob indicates a negative parameter and a white blob indicates a positive parameter; the area of each blob is proportional to the parameter's magnitude (the largest value shown is 7.77 and the smallest value shown is -7.21).

indicate that 32 of the 36 middle-layer variables are used by the network as "feature variables" to represent the 32 possible bars. These feature variables are controlled mainly by the right-most top-layer "orientation" variable – the parameters connecting all the other top-layer variables to the feature variables are nearly zero. If the orientation variable is off, the probability of each feature variable is determined mainly by its bias. Vertical feature variables have significantly negative biases, causing them to remain off if the orientation neuron is off. Horizontal feature variables have only slightly negative biases, causing them to turn on roughly 25% of the time if the orientation variable is off. The parameters connecting the orientation variable to the vertical feature variables are significantly positive, so that when the orientation variable is on the total input to each vertical feature variable is slightly negative, causing the vertical feature variables to turn on roughly 25% of the time. The parameters connecting the orientation variable to the horizontal feature variables are significantly negative, so that when the orientation variable is on the total input to each horizontal feature variable is significantly negative, causing the horizontal feature variables to remain off. Since the parameters connecting the top-layer variable to the 4 middle-layer nonfeature variables are negative, and since the nonfeature variables have large negative biases, the nonfeature variables are usually inactive. Because the bottom-layer biases (not shown) are only slightly negative, a pixel that is not turned on by a feature variable still has a probability of 0.25 of being turned on. This accounts for the additive noise.

3.6.2 Automatic clean-up of noisy images

Once learned, the recognition network can nonlinearly filter the noise from a given image, detect the underlying bars, and determine the orientation of these bars. To clean up each of the training images shown in Figure 3.9, I apply the learned recognition network to the image and obtain middle-layer variable values which reveal an estimate of which bars are on. The results of this procedure are shown in Figure 3.12 and clearly show that the recognition network is capable of filtering out the noise. Usually, the recognition network correctly identifies which bars were instantiated in the original image. Occasionally, a bar is not successfully detected. In two cases a bar is detected that has an orientation that is the opposite of the dominant orientation; however, usually the recognition network chooses a single orientation. Inspection of the original noisy training images for the two incorrect cases shows that aside from the single-orientation constraint, there is significant evidence that the mistakenly detected bars *should* be on. Further training reduces the chance of misdetection.

3.6.3 Wake-sleep estimation without positive parameter constraints

If all the parameters are initialized to 0.0, the parameters that connect the middle layer to the bottom layer are not constrained to be positive, and all the learning rates are set to 0.01, the estimated generative network does not properly extract the bar structure. Figure 3.13 shows the generative network parameters that connect the middle layer to the bottom layer, after 5,000,000 learning iterations. The black bars indicate that some middle-layer variables are capable of "uninstantiating" bars that may be instantiated by other variables. Although it is imaginable that such a complex scheme is still capable of modelling the training images, the log-likelihood bound for this trained Helmholtz machine is -190 bits – significantly lower

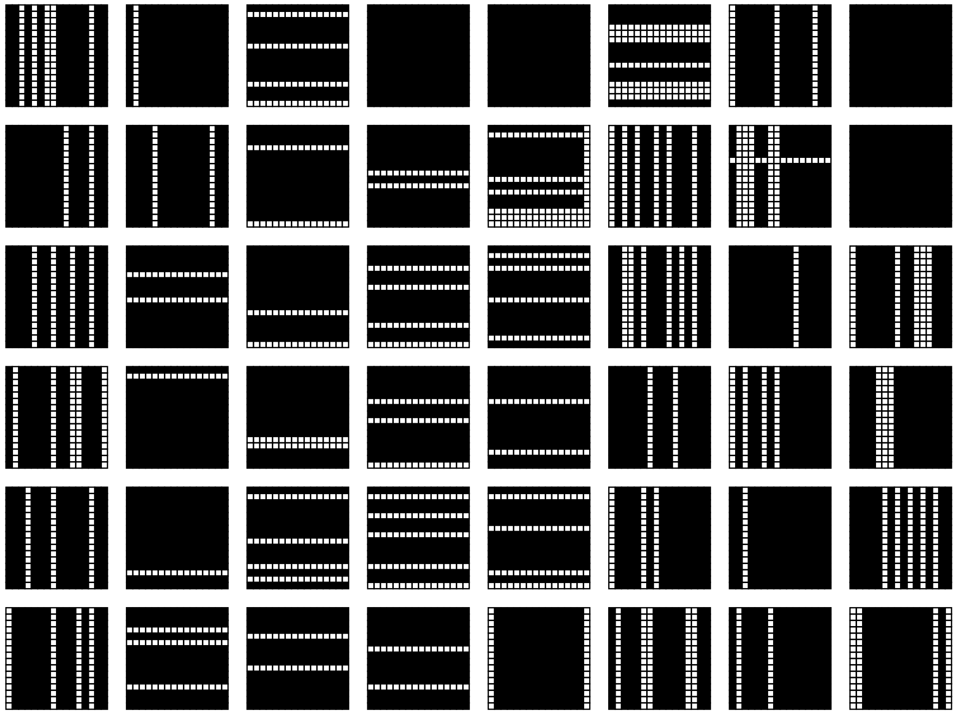

Figure 3.12: Filtered versions of the training examples from Figure 3.9 extracted using the estimated recognition network.

than the optimum value of -170 bits.

3.6.4 How hard *is* the bars problem?

Although this bar extraction problem may seem simple, it must be kept in mind that the network is not given *a priori* topology information – a fixed random rearrangement of the pixels in the training images would not change the learning performance of the network. So, insofar as the network is concerned, the actual training examples look like those shown in Figure 3.14 which were produced by applying a fixed random rearrangement to the pixels in the images from Figure 3.9.

3.7 Simultaneous extraction of continuous and categorical structure

The Bayesian networks presented so far in this chapter have contained discrete variables. However, some hidden variables, such as translation or scaling in images of shapes, are best

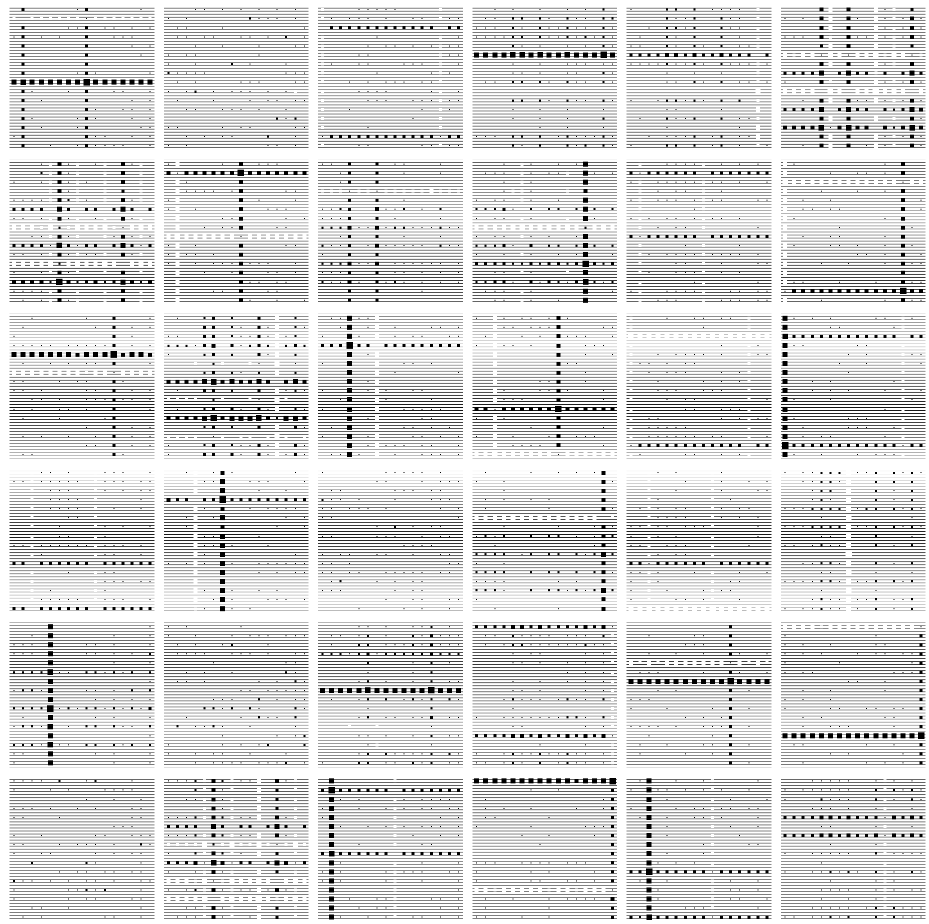

Figure 3.13: Parameters for generative connections that feed out of the middle-layer variables after estimation without special initialization of the weights, without different learning rates between layers, and without positive weight constraints in the generative network.

represented using continuous values. Work done on continuous-valued Bayesian networks has focussed mainly on Gaussian random variables that are linked linearly such that the joint distribution over all variables is also Gaussian [Pearl 1988; Heckerman and Geiger 1995]. Lauritzen *et al.* [1990] have included discrete random variables within the linear Gaussian framework. They consider networks that are singly-connected, so that probability propagation can be used. Most work on continuous-valued Bayesian networks requires that all the conditional distributions represented by the network can be easily derived using information elicited from experts. Hofmann and Tresp [1996] consider estimating continuous Bayesian networks that may be richly connected, but they assume that all variables are observed. As far as nonlinear continuous networks with latent variables are concerned, continuous-valued Boltzmann machines have been developed [Movellan and McClelland

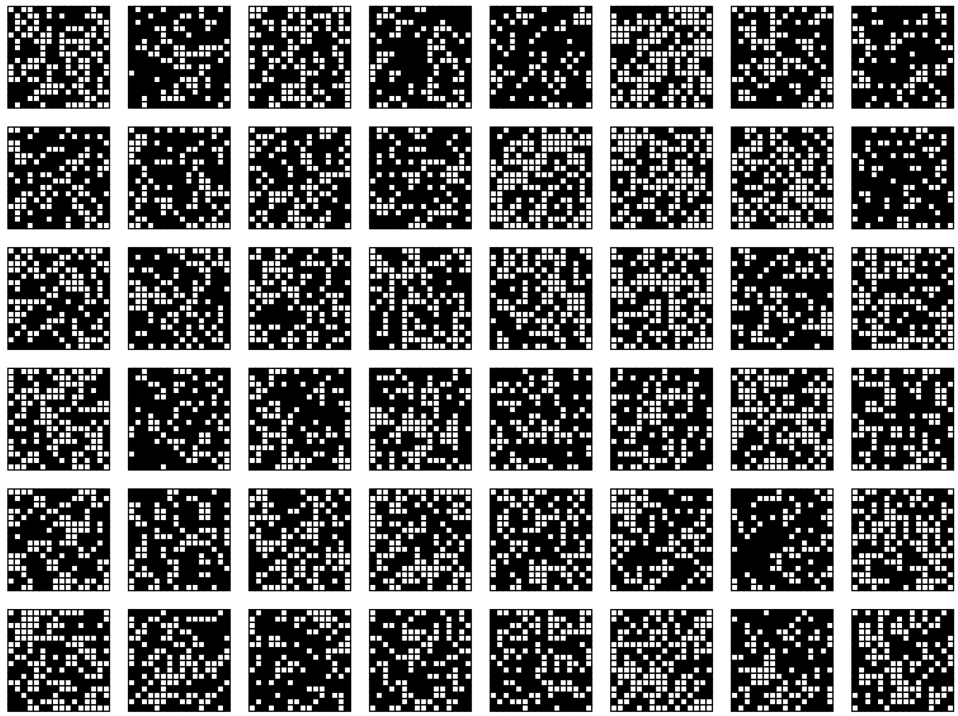

Figure 3.14: Training examples from Figure 3.9 after a fixed random rearrangement of the pixels has been applied. These are indicative of the difficulty of the bars problem in the absence of a topological prior that favors local intensity coherence.

1992], but these suffer from long simulation settling times and the requirement of a "negative phase" during learning. Tibshirani [1992], MacKay [1995] and Bishop *et al.* [1997] consider estimating mappings from a continuous latent variable space to a higher-dimensional input space, effectively using multiple-cause type networks of the form shown in Figure 3.4 on page 63. In this section, I consider a hierarchical Bayesian network with variables that can *adapt* to be continuous or categorical, as needed by the training data [Frey 1997a; Frey 1997b].

3.7.1 An adaptive random variable

The proposed random variable is shown schematically in Figure 3.15a[2]. The parents $\mathbf{a}_i$ effect z_i via a total input,

$$\mu_i \equiv \sum_{j=0}^{i-1}\theta_{ij}z_j, \tag{3.57}$$

[2]Geoffrey Hinton suggested this unit as a way to make factor analysis nonlinear.

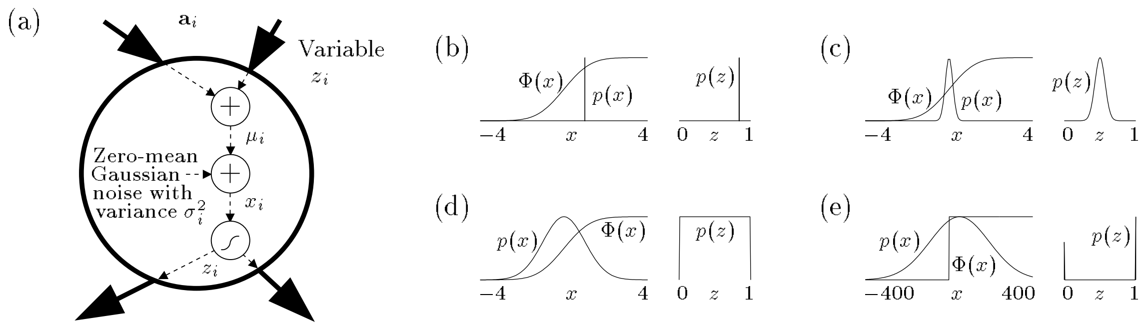

Figure 3.15: (a) schematically shows the dependence of the proposed variable on its parents. (b) to (e) illustrate four quite different modes of behavior: (b) deterministic mode; (c) stochastic linear mode; (d) stochastic nonlinear mode; and (e) stochastic binary mode (note the different horizontal scale). For the sake of graphical clarity, the density functions are normalized to have equal maxima and the subscripts are left off the variables.

where we constrain $\theta_{ij} = 0.0$ if $z_j \notin \mathbf{a}_i$. The probability density over the *presigmoid activity* x_i for variable z_i is

$$p(x_i|\mu_i, \sigma_i^2) \equiv \exp[-(x_i - \mu_i)^2/2\sigma_i^2]/\sqrt{2\pi\sigma_i^2}, \tag{3.58}$$

where σ_i^2 is a parameter associated with variable z_i (formally, part of $\boldsymbol{\theta}$). The value of z_i (its *postsigmoid activity*) is obtained by passing the presigmoid activity through a sigmoidal cumulative Gaussian squashing function:

$$z_i \equiv \Phi(x_i) \equiv \int_{-\infty}^{x} \frac{1}{\sqrt{2\pi}} e^{-\alpha^2/2} d\alpha. \tag{3.59}$$

Networks of these variables can *represent* a broad range of structures, including deterministic multilayer perceptrons [Bishop 1995], binary sigmoidal Bayesian networks [Neal 1992] (see Section 3.4), mixture models, mixture of expert models [Jacobs et al. 1991], hierarchical mixture of expert models [Jordan and Jacobs 1994], and factor analysis models [Everitt 1984]. This versatility is brought about by a range of significantly different modes of behavior available to each variable.

Deterministic mode: If the noise variance σ_i^2 is very small, the postsigmoid activity y_i will be a practically deterministic sigmoidal function of the mean (see Figure 3.15b). This mode is useful for representing deterministic nonlinear mappings such as those found in deterministic multilayer perceptrons and mixture of expert models.

Stochastic linear mode: For a given mean, if the squashing function is approximately linear over the span of the added noise, the postsigmoid distribution will be approximately Gaussian with the mean and standard deviation linearly transformed (see Figure 3.15c).

This mode is useful for representing Gaussian noise effects such as those found in mixture models, the outputs of mixture of expert models, and factor analysis models.

Stochastic nonlinear mode: If the variance of a variable in the stochastic linear mode is increased so that the squashing function is used in its nonlinear region, a variety of distributions are producible that range from skewed Gaussian to uniform to bimodal (see Figure 3.15d).

Stochastic binary mode: This is an extreme case of the stochastic nonlinear mode. If the variance σ_i^2 is very large, then nearly all of the postsigmoid probability mass will lie near the ends of the interval $(0, 1)$ (see Figure 3.15e). *E.g.*, for a standard deviation of 150, less than 1% of the mass lies in $(0.1, 0.9)$. In this mode, the postsigmoid activity z_i appears to be binary with probability of being "on" (*i.e.*, $z_i > 0.5$ or, equivalently, $x_i > 0$):

$$p(i \text{ on}|\mu_i, \sigma_i^2) = \int_0^\infty \frac{\exp[-(x-\mu_i)^2/2\sigma_i^2]}{\sqrt{2\pi\sigma_i^2}} dx = \int_{-\infty}^{\mu_i} \frac{\exp[-x^2/2\sigma_i^2]}{\sqrt{2\pi\sigma_i^2}} dx = \Phi\Big(\frac{\mu_i}{\sigma_i}\Big). \tag{3.60}$$

This sort of stochastic activation is found in binary sigmoidal belief networks [Jaakkola, Saul and Jordan 1996] and in the decision-making components of mixture of expert models and hierarchical mixture of expert models.

3.7.2 Inference using slice sampling

Assuming the variables are labeled in ancestral order, the joint distribution can be written

$$p(\{x_i\}_{i=1}^N) = \prod_{i=1}^N p(x_i|\{x_j\}_{j=0}^{i-1}) \quad \text{or} \quad p(\{z_i\}_{i=1}^N) = \prod_{i=1}^N p(z_i|\{z_j\}_{j=0}^{i-1}), \tag{3.61}$$

where N is the number of variables. $p(x_i|\{x_j\}_{j=0}^{i-1})$ and $p(z_i|\{z_j\}_{j=0}^{i-1})$ are the presigmoid and postsigmoid conditional densities for variable z_i. (Recall that the set of parents is represented by parameter constraints.) As usual, I define $z_0 \equiv 1$ to allow for a constant bias.

Even for small networks of these variables, probabilistic inference can be very difficult. Not only is the inference problem combinatorial, but it involves continuous hidden variables whose distribution when conditioned on visible variables may be multimodal with peaks that are broad in some dimensions but narrow in others. I use a Markov chain Monte carlo procedure, which consists of sweeping a prespecified number of times through the set of hidden variables. A new value is obtained for each hidden variable using slice sampling [Neal 1997] (see Section 2.2.4), based on the distribution for the variable conditioned on all other variables. If an infinite number of slice samples are drawn for each hidden variable before passing on to the next hidden variable, this procedure is equivalent to Gibbs sampling (see Section 2.2.2). In fact, detailed balance still holds if only a fixed number of slice samples are drawn for each variable before passing on to the next variable. In most cases, drawing only one slice sample for each variable before continuing on will be most efficient.

If the parent-child influences cause there to be two very narrow peaks in the conditional distribution $p(z_i|\{z_j\}_{j=0}^{i-1}, \{z_j\}_{j=i+1}^{N})$ for a hidden variable (corresponding to a variable in the stochastic binary mode), the slices will almost always consist of two very short line segments and it will be very difficult for the above procedure to switch from one mode to another. To fix this problem, slice sampling is performed in a new domain, $y_i = \Phi(\{x_i - \mu_i\}/\sigma_i)$. In this domain the parent-child distribution $p(y_i|\{z_j\}_{j=0}^{i-1})$ is uniform on $(0,1)$, so $p(y_i|\{z_j\}_{j=0}^{i-1}, \{z_j\}_{j=i+1}^{N}) = p(y_i|\{z_j\}_{j=i+1}^{N})$. So, I can use the following function for slice sampling:

$$f(y_i) = \exp\big[-\textstyle\sum_{k=i+1}^{N}\big\{x_k - \mu_k^{-i} - \theta_{z_k z_i}\Phi\big(\sigma_i\Phi^{-1}(y_i) + \mu_i\big)\big\}^2/2\sigma_k^2\big], \tag{3.62}$$

where $\mu_k^{-i} = \sum_{j=0, j\neq i}^{i-1} \theta_{z_k z_j} z_j$. Since x_i, y_i and z_i are all deterministically related, sampling from the distribution of y_i will give appropriately distributed values for the other two[3].

3.7.3 Parameter estimation using slice sampling

I use on-line stochastic gradient ascent to perform MLB parameter estimation. This consists of sweeping through the training set and for each training case following the gradient of $\log p(\{x_i\}_{i=1}^{N})$, while sampling hidden unit values as described above. The parameters are changed as follows:

$$\begin{aligned}\Delta\theta_{jk} &\equiv \eta\,\partial \log p(\{x_i\}_{i=1}^{N})/\partial\theta_{jk} = \eta\big(x_j - \textstyle\sum_{l=0}^{j-1}\theta_{jl}y_l\big)y_k/\sigma_j^2,\\ \Delta \log\sigma_j^2 &\equiv \eta\,\partial \log p(\{x_i\})/\partial \log\sigma_j^2 = \eta\big[\big(x_j - \textstyle\sum_{l=0}^{j-1}\theta_{jl}y_l\big)^2/\sigma_j^2 - 1\big]/2,\end{aligned} \tag{3.63}$$

where η is the learning rate.

I designed two experiments meant to elicit the four modes of operation described above. Both experiments were based on a simple network with one hidden layer **h** containing two variables and one visible layer **v** containing two variables. Training data was obtained by carefully selecting model parameters so as to induce various modes of operation and

[3] Both $\Phi()$ and $\Phi^{-1}()$ do not have closed-form expressions, so I use the C-library erf() function to implement $\Phi()$ and table lookup with quadratic interpolation to implement $\Phi^{-1}()$.

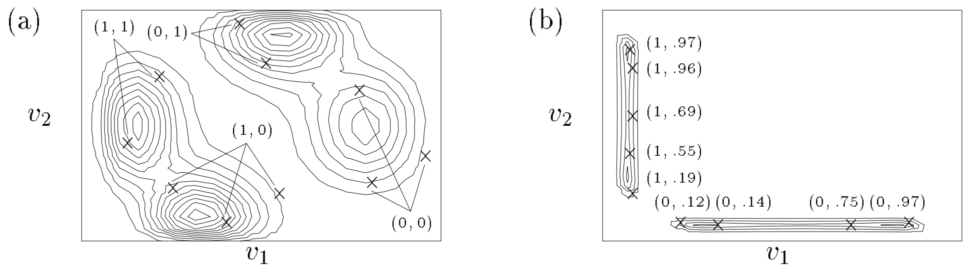

Figure 3.16: For each experiment (a) and (b), contours show the distribution of the 2-dimensional training cases. The inferred mean postsigmoid activity of the two hidden units after learning are shown in braces for several training cases, marked by $\times$.

then generating 10,000 two-dimensional examples. Before training, the log-variances were initialized to 10.0, and the other parameters were initialized to uniformly random values between -0.1 and 0.1. Training consisted of 100 epochs using a learning rate of 0.001 and 20 sweeps of slice sampling to complete each training case. Each task required roughly five minutes on a 200 MHz MIPS R4400 processor.

The distribution of the training cases in visible unit space ($v_1 - v_2$) for the first experiment is shown by the contours in Figure 3.16a. After training the network, I ran the inference algorithm for each of ten representative training cases. The mean postsigmoid activities of the two hidden units are shown beside the cases in Figure 3.16a; clearly, the network has identified four classes that it labels $(0, 0)$, $(0, 1)$, $(1, 0)$, and $(1, 1)$. Based on a 30×30 histogram, the relative entropy between the training set and data generated from the trained network is 0.02 bits. Figure 3.16b shows a similar picture for the second experiment, using different training data. In this case, the network has identified two categories that it labels using the first postsigmoid activity. The second postsigmoid activity indicates how far along the respective "ridge" the data point lies. The relative entropy in this case is 0.04 bits.

The above experiments illustrate how the same network can be used to model two quite different types of data. In contrast, a Gaussian mixture model would require many more components for the second task as compared to the first. Although the methods due to Tibshirani and Bishop *et al.* would nicely model each submanifold in the second task, they would not properly distinguish between categories of data in either task. MacKay's method may be capable of extracting both the submanifolds and the categories, but I am not aware of any results on such a dual problem.

It is not difficult to conceive of models for which naive Markov chain Monte Carlo procedures will become fruitlessly slow. In particular, if two variables are highly correlated, the procedure of sampling one variable at a time will converge extremely slowly. Also, the Markov chain method may be prohibitive for larger networks. One approach to avoiding these problems is to use the Helmholtz machine or variational methods.

Chapter 4

Data Compression

The goal of data compression is to exploit the redundancy in sets of patterns so as to represent individual patterns concisely on average. In this thesis, I focus on lossless data compression, in which the original pattern can be completely recovered from the concise representation. A *source code* maps each input pattern $\mathbf{v}$ to a codeword, such that for each valid codeword there is a unique input pattern. I will consider sources where the patterns are i.i.d. (independent and identically drawn) from a distribution $P_{\mathrm{r}}(\mathbf{v})$.

Shannon's noiseless source coding theorem [Shannon 1948] states that the average codeword length per source pattern cannot be less than the entropy of the source:

$$\mathrm{E}[\ell(\mathbf{v})] \geq \mathcal{H}, \tag{4.1}$$

where $\ell(\mathbf{v})$ is the length in bits of the codeword for input pattern $\mathbf{v}$, and $\mathcal{H}$ is the entropy of the source:

$$\mathcal{H} = -\sum_{\mathbf{v}} P_{\mathrm{r}}(\mathbf{v}) \log_2 P_{\mathrm{r}}(\mathbf{v}). \tag{4.2}$$

Traditional approaches to data compression have focussed on producing source codes whose codeword lengths are nearly optimal, where the optimal length of the codeword for $\mathbf{v}$ is $\log_2 P_{\mathrm{r}}(\mathbf{v})$.

Arithmetic coding [Rissanen and Langdon 1976; Witten, Neal and Cleary 1987] is a practical algorithm for producing near-optimal codewords when the source distribution $P_{\mathrm{r}}(\mathbf{v})$ is known. If $\mathbf{v}$ is binary-valued, $P_{\mathrm{r}}(\mathbf{v})$ can be easily estimated and arithmetic coding can be used to produce near-optimal "fractional bit" codwords. If $\mathbf{v}$ is high-dimensional and the distribution is quite complex (*e.g.*, images of faces), it may be desirable to estimate a more sophisticated flexible probability model $P(\mathbf{v})$. Unfortunately, even if such a model can be estimated so that $P(\mathbf{v}) \approx P_{\mathrm{r}}(\mathbf{v})$, there may not be a way to encode $\mathbf{v}$ using the model. For example, an arithmetic encoder requires a table of the probabilities for all possible inputs. For a 16×16 binary image, this table would have 2^{256} entries! So, not only do we need a model that provides a probability $P(\mathbf{v})$, but we also need a model that somehow decomposes $P(\mathbf{v})$ in a way that allows the encoder to encode the variables one at a time (or in small groups).

Graphical models provide a structured description of $P(\mathbf{v})$, and so they seem like a good place to look for the source models described above. However, it turns out that undirected graphical models do not provide the right type of structure. For example, the Boltzmann machine [Hinton and Sejnowski 1986] (a Markov random field that learns) is poorly suited to data compression, because it does not decompose $P(\mathbf{v})$ in a way that is suitable for efficient piece-wise compression. (A method such as Markov chain Monte Carlo must be used to estimate the partition function, which normalizes the probabilities.) On the other hand, Bayesian networks *do* provide an ideal structure for data compression.

In Section 4.1, I show how Bayesian network source models that do not have latent variables can be used very efficiently to compress data. Then, in Section 4.2, I go on to discuss source models that have many latent variables. Values can be chosen for the latent variables and the entire configuration can be encoded. In this way, a "multi-valued source code" with many codewords for each input pattern is obtained. In many cases, these codewords cannot be mixed together in a tractable way. To remedy this problem, I show how extra information can ride "piggyback" on the choice of codeword and derive the communication rate for this "bits-back" procedure. In Section 4.3, we see that a broad class of approximations to maximum likelihood parameter estimation actually minimizes this communication rate. In Section 4.4, I outline the "bits-back coding" algorithm, which is a practical implementation of the "bits-back" idea. It turns out that by using an arithmetic decoder in the bits-back encoder and an arithmetic encoder in the bits-back decoder, we can achieve a practical communication rate that is nearly optimal. Finally, in Section 4.5, I present compression results for Helmholtz machine source models that are adapted using the wake-sleep algorithm.

4.1 Fast compression with Bayesian networks

Suppose we have at hand a Bayesian network for the binary variables in $\mathbf{v}$, such that $P(\mathbf{v}) \approx P_{\mathrm{r}}(\mathbf{v})$. As discussed in Section 1.2.2, the joint distribution for a Bayesian network can be written

$$P(\mathbf{v}) = \prod_{k=1}^{N} P(v_k|\mathbf{a}_k), \tag{4.3}$$

where N is the number of variables in $\mathbf{v}$, and $\mathbf{a}_k$ are the parents of v_k. This decomposition is very well-suited to arithmetic coding.

In order to encode $\mathbf{v}$, we pick an ancestral order for the variables. I will assume without loss of generality that $v_1, \ldots, v_N$ is such an ordering. Compression begins with v_1, whose observed value is fed into the arithmetic encoder, along with its distribution $\{P(v_1 = 0), P(v_1 = 1)\}$, which is part of the network specification. Next, we compute $\{P(v_2 = 0|\mathbf{a}_2), P(v_2 = 1|\mathbf{a}_2)\}$ using the conditional probability given in the network specification as well as the values of v_2's parents (*i.e.*, either $\{v_1\}$ or $\varnothing$) wich have already been encoded. We feed the observed value of v_2 into the arithmetic encoder, along with its distribution. Encoding continues in this fashion until all network variables have been encoded.

For this procedure to work as described, the Bayesian network must be fully visible. That is, all network variables are part of the input pattern. Suppose there are some latent variables $\mathbf{h}$ that are not part of the input. Then, the Bayesian network models $P(\mathbf{v}, \mathbf{h})$. These variables may be important for representing higher-order structure in the input $\mathbf{v}$, as discussed extensively in Chapter 3. Now, the decomposition in (4.3) cannot be used.

If there aren't many latent variables, we can use a procedure that is similar to the one described above. We pick an ancestral order and proceed as described above, encoding only the observed variables. Whenever we encounter an observed variable that is not dependency-separated from an unobserved variable by the variables that have been encoded so far, the unobserved variable must be integrated out, by summing over its values. The complexity of this encoding procedure is usually exponential in the number of unobserved variables. Sometimes, the graphical structure of the network permits this procedure to be done in a very efficient way. For example, the latent variables in a hidden Markov model [Rabiner 1989] with a fixed state space size can be integrated out in a way so that the encoding complexity is linear in the number of latent variables (number of time steps).

4.2 Communicating extra information through the choice of codeword

In general, when the latent variables in a Bayesian network cannot summed away to compute $P(\mathbf{v})$ in a tractable way, we are left with the option of *picking* values for them. Then, the

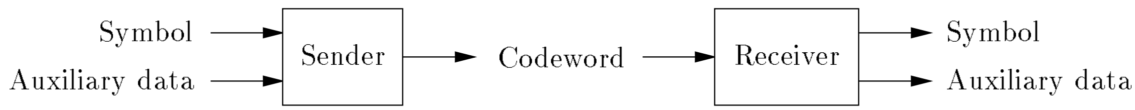

Figure 4.1: A scheme in which auxiliary data is communicated along with the symbol in order to achieve optimal compression when the source code is multi-valued.

entire set of variables $\{\mathbf{v}, \mathbf{h}\}$ can be encoded using the procedure described above, as if all values were observed. This can be viewed as a *multi-valued source code*, where there are many codewords for each input $\mathbf{v}$. The codeword depends on which values are chosen for the latent variables. Often, as with the hidden Markov model, these codewords can be mixed in an efficient way. However, there is an interesting class of multi-valued source codes (*e.g.*, Bayesian networks with latent variables) for which the multiple codewords *cannot* be mixed in a tractable manner.

The approach I take to solve this problem [Frey and Hinton 1996; Frey and Hinton 1997] is motivated by the "bits-back" argument of Hinton and Zemel [1994], which they used to develop a Lyapunov function for machine learning. It turns out that Wallace [1990] devised a similar argument to construct minimum-length integer-length messages for use in minimum-message-length inference. By selecting codewords through the use of extra *auxiliary* data, this information can ride "piggyback" on the codewords for the symbols that we are encoding. Compared to the optimal single-valued source code obtained by mixing together the codewords for an input pattern, the bits communicated in the auxiliary data will make up for the lengths of the suboptimal codewords that are sent. In particular, the communication rate will be *less* than the rate that would be obtained by always picking the shortest codeword. A block diagram for this communication scheme is shown in Figure 4.1. A simple example will help illustrate this procedure.

4.2.1 Example: A simple mixture model

Consider a source that outputs real numbers that are distributed according to a mixture of two Gaussians. These numbers are rounded to some precision to form a set of symbols. The component distributions and the output distribution are shown in Figure 4.2a, where the rounding effect is left out for the sake of graphical simplicity.

The most natural source code to use in this case is one that requires one bit to specify from which Gaussian a given symbol was produced plus however many bits are needed to code the symbol using that Gaussian. However, the identity of the Gaussian that produced a given symbol is often ambiguous. In particular, a number near v_0 could well have come

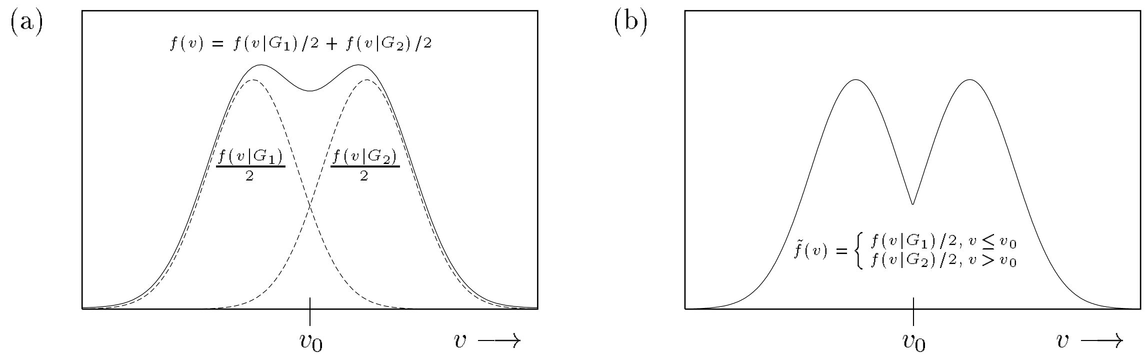

Figure 4.2: The most natural source model may produce multiple codewords for a given symbol. (a) shows a source with a single binary hidden variable which identifies from which Gaussian, G_1 or G_2, the symbol value v is sampled. Values of v near v_0 are likely to have come from either Gaussian. (b) shows the resulting coding density effectively used if we were to always pick the shorter codeword. This density wastes coding space because it is wrongly shaped and has an area significantly less than unity.

from either Gaussian. In these cases the source model maps each symbol to two codewords — one for each Gaussian — producing a multi-valued source code. If we were to always pick the shorter of the two codewords, we would effectively be assuming the symbols were distributed as in Figure 4.2b. However, this distribution is obviously incorrect — it is not even normalized — and will lead to suboptimal compression.

The obvious way around this problem is to use a single-valued code that is based on a summation of the mixture component probabilities. That is, we assign a new codeword to each symbol based on its total probability mass, obtained by summing the contributions from each Gaussian. Although this procedure is obviously computationally feasible for this example, there are more complicated models where it is not (see Section 4.5). In fact, the same rate can be achieved by using the original multi-valued source code and communicating extra information through the *choice* of codeword. This may seem surprising, since for a given symbol both codewords in the multi-valued source code are longer than the codeword in the single-valued source code.

Consider a sender that wishes to encode a rounded value v' that requires 2 bits if encoded using G_1 and requires 3 bits if encoded using G_2 (*i.e.*, v' is twice as likely under G_1 as it is under G_2). Including the single bit required to specify which Gaussian is being used, an optimal source code (where the Gaussian identity is explicit) will thus have codewords with lengths $\ell_1 = 3$ bits and $\ell_2 = 4$ bits. If the sender always picks the shorter codeword, the average codeword length is 3 bits.

Suppose instead that whenever the sender must communicate the particular symbol v', the sender chooses between the two codewords with equal probability. (In general, the ratio of choices will depend on v'.) It appears the average codeword length in this case is $(\ell_1 + \ell_2)/2 = 3.5$ bits, which is higher than that obtained by always choosing the shortest codeword. However, this cost is effectively lowered because the receiver can recover information from the choice of codeword in the following manner. Say the sender has high-entropy auxiliary data available in the form of a queued bit stream with 0 and 1 having equal frequency. When encoding v', the sender uses the next bit in the auxiliary data queue to choose between G_1 and G_2. The sender then produces a codeword that will have an average length of 3.5 bits (it is important to note that this codeword specifies which of G_1 and G_2 is being used).

When decoding, the receiver reads off the bit that says which Gaussian was used and then determines the rounded value v' from the codeword. Given the decoded value, the receiver can run the same encoding algorithm that the sender used, and determine that a choice of equal probability was made between G_1 and G_2. Since the receiver also knows which Gaussian was selected, the receiver can recover the queued auxiliary data bit that was used to make the choice. In this way, on average 1 bit of the auxiliary data is communicated at no extra cost. I refer to these recovered bits as *bits-back*.

If the auxiliary data is useful, the average effective codeword length is reduced by 1 bit due to the savings, giving an effective average length of 2.5 bits — less than the 3 bits required by the shortest codeword. I refer to this method of source coding as *bits-back coding*. It is important to note that the ratio of choices between G_1 and G_2 depends on the symbol being encoded. For example, if the rounded value is far to the right of v_0 in Figure 4.2a, then picking the codewords equally often would be very inefficient, since the codeword under G_1 would be extremely long, making the benefit of the single recovered bit negligible. In this case the sender should pick G_1 much less often and as a result the sender will read off only "part" of a bit from the auxiliary data queue to determine which codeword to use. As we will see below, choosing between the two codewords with equal probability is not optimal in the above example.

4.2.2 The optimal bits-back coding rate

The rate for bits-back coding can be determined by defining a distribution that is used to select codewords for a given input symbol (pattern), $\mathbf{v}$:

$$Q(\mathbf{h}|\mathbf{v}), \tag{4.4}$$

where $\mathbf{h}$ is a binary vector representing the index of the selected codeword for input $\mathbf{v}$. (It is represented as a vector, since it too must be encoded.) Letting $\ell(\mathbf{v}, \mathbf{h})$ be the length of the $\mathbf{h}$th codeword[1] for a specific pattern $\mathbf{v}$, the expected length of the two-part codeword for $\mathbf{v}$ is

$$\mathcal{E}(\mathbf{v}) \equiv \sum_{\mathbf{h}} Q(\mathbf{h}|\mathbf{v})\ell(\mathbf{v},\mathbf{h}). \tag{4.5}$$

The expected bits-back for $\mathbf{v}$ is the information content (entropy) of the distribution used to select codewords:

$$\mathcal{H}(\mathbf{v}) \equiv -\sum_{\mathbf{h}} Q(\mathbf{h}|\mathbf{v}) \log_2 Q(\mathbf{h}|\mathbf{v}). \tag{4.6}$$

The difference between (4.5) and (4.6) gives the communication cost for $\mathbf{v}$:

$$\mathcal{F}(\mathbf{v}) \equiv \mathcal{E}(\mathbf{v}) - \mathcal{H}(\mathbf{v}) = \sum_{\mathbf{h}} Q(\mathbf{h}|\mathbf{v}) \log_2 \frac{Q(\mathbf{h}|\mathbf{v})}{2^{-\ell(\mathbf{v},\mathbf{h})}}. \tag{4.7}$$

The overall rate $\mathcal{F}$ for bits-back coding is given by averaging this cost over the source distribution, $P_{\mathrm{r}}(\mathbf{v})$:

$$\mathcal{F} \equiv \sum_{\mathbf{v}} P_{\mathrm{r}}(\mathbf{v})\mathcal{F}(\mathbf{v}) = \sum_{\mathbf{v}} P_{\mathrm{r}}(\mathbf{v}) \sum_{\mathbf{h}} Q(\mathbf{h}|\mathbf{v}) \log_2 \frac{Q(\mathbf{h}|\mathbf{v})}{2^{-\ell(\mathbf{v},\mathbf{h})}}. \tag{4.8}$$

It is easily proven from (4.8) that for each $\mathbf{v}$ the codeword selection distribution which minimizes the bits-back coding rate is the Boltzmann distribution:

$$Q^*(\mathbf{h}|\mathbf{v}) \equiv 2^{-\ell(\mathbf{v},\mathbf{h})} / \sum_{\mathbf{h}'} 2^{-\ell(\mathbf{v},\mathbf{h}')}. \tag{4.9}$$

I denote by * those quantities determined from the Boltzmann distribution. Note that this distribution depends on the input symbol, $\mathbf{v}$. The optimal rate for a given multi-valued source code is achieved if for each input symbol a codeword is selected using the corresponding Boltzmann distribution. By substituting (4.9) into (4.8), we find that the optimal bits-back coding rate is

$$\mathcal{F}^* = -\sum_{\mathbf{v}} P_{\mathrm{r}}(\mathbf{v}) \log_2 \Big[\sum_{\mathbf{h}} 2^{-\ell(\mathbf{v},\mathbf{h})}\Big]. \tag{4.10}$$

This rate is the same as the rate for a single-valued source code that has codeword lengths which properly reflect the total codeword space associated with each symbol in the multi-valued source code.

[1]The codewords may have fractional lengths produced, say, by arithmetic coding

In the mixture of Gaussians example, where for symbol v' we had $\ell_1 = 3$ bits and $\ell_2 = 4$ bits,

$$
\begin{aligned}
Q^*(G_1|v') &= 2^{-3}/(2^{-3}+2^{-4}) = 2/3,\\
Q^*(G_2|v') &= 2^{-4}/(2^{-3}+2^{-4}) = 1/3,\\
\mathcal{E}^*(v') &= \frac{2}{3}(3\text{ bits}) + \frac{1}{3}(4\text{ bits}) = 3.333\text{ bits},\\
\mathcal{H}^*(v') &= -\frac{2}{3}\log_2\Big(\frac{2}{3}\Big) - \frac{1}{3}\log_2\Big(\frac{1}{3}\Big) = 0.918\text{ bits},\\
\mathcal{F}^*(v') &= 3.333\text{ bits} - 0.918\text{ bits} = 2.415\text{ bits}.
\end{aligned}
\tag{4.11}
$$

This is the minimum $\mathcal{F}(v')$ for the given example. A slightly higher than optimal $\mathcal{F}(v')$ of 2.5 bits was obtained above using $\hat{Q}(G_1|v') = \hat{Q}(G_2|v') = 0.5$.

4.2.3 Suboptimal bits-back coding

For complex source models, the summation in the denominator of (4.9) is usually intractable; in these cases, it is not possible to obtain the exact Boltzmann distribution. When it *is* possible to obtain the exact Boltzmann distribution, the denominator in (4.9) can often be directly used to create a single-valued source code. The advantage of bits-back coding is that when the multi-valued source code is unmixable, an approximation to the Boltzmann distribution can be used. There are a variety of practical algorithms for obtaining such an approximation, including Markov chain Monte Carlo methods [Geman and Geman 1984; Hinton and Sejnowski 1986; Ripley 1987; Potamianos and Goutsias 1993], mean field methods [Chandler 1987; Peterson and Anderson 1987; Zhang 1993; Saul, Jaakkola and Jordan 1996], and inverse model methods [Hinton et al. 1995; Dayan et al. 1995] (see Section 4.5). The rate for an arbitrary codeword selection distribution $Q(\mathbf{h}|\mathbf{v})$ can be compared to the optimal rate given by the Boltzmann distribution:

$$
\mathcal{F} - \mathcal{F}^* = \sum_{\mathbf{v}} P_{\mathrm{r}}(\mathbf{v}) \sum_{\mathbf{h}} Q(\mathbf{h}|\mathbf{v}) \log_2 \frac{Q(\mathbf{h}|\mathbf{v})}{Q^*(\mathbf{h}|\mathbf{v})}. \tag{4.12}
$$

This is the information divergence (a.k.a. relative entropy) between the codeword selection distribution and the Boltzmann distribution, averaged over the source distribution. It is always non-negative and yields the increase in coding rate caused by the approximation to the Boltzmann codeword selection distribution.

A suboptimal codeword selection distribution of particular interest is $Q^{\text{short}}(\mathbf{h}|\mathbf{v})$, which always picks the shortest codeword, $\mathbf{h}^{\mathrm{short}}(\mathbf{v})$. (This is analogous to the two-part codes discussed by Rissanen [1989].) In this case, the rate increase compared to the optimal rate is

$$\mathcal{F}^{\mathrm{short}} - \mathcal{F}^* = \sum_{\mathbf{v}} P_{\mathrm{r}}(\mathbf{v}) \sum_{\mathbf{h}} Q^{\mathrm{short}}(\mathbf{h}|\mathbf{v}) \log_2 \frac{Q^{\mathrm{short}}(\mathbf{h}|\mathbf{v})}{Q^*(\mathbf{h}|\mathbf{v})} = \sum_{\mathbf{v}} P_{\mathrm{r}}(\mathbf{v}) \log_2 \frac{1}{Q^*(\mathbf{h}^{\mathrm{short}}(\mathbf{v})|\mathbf{v})}. \tag{4.13}$$

Bits-back coding makes gains over shortest codeword selection by approximately taking into account the entire codeword space associated with an input symbol, as opposed to just the codeword space associated with the shortest codeword. If several of the shortest codewords have roughly equal lengths, or if there are a large number of codewords with lengths somewhat larger than the shortest, then $Q^*(\mathbf{h}^{\mathrm{short}}(\mathbf{h})|\mathbf{v})$ is significantly less than unity indicating that picking the shortest codeword is far from optimal. Relative to Rissanen's work, bits-back coding provides a tractable way to approximate the stochastic complexity [Rissanen 1989] and furthermore *communicate* at this rate.

4.3 Relationship to maximum likelihood estimation and its approximations

The whole idea of a multi-valued source code may seem absurd. Why waste codeword space by associating multiple codewords with each symbol? An answer to this question must be closely related to the structure of the source model. In addition to the input pattern being encoded, it is often useful and natural to consider extra *latent* variables whose purpose is to capture high-order structure. For example, when modelling grey-scale images, it may help to create a latent variable that measures overall image contrast. The codeword for a particular image will include a binary representation of this contrast value. However, there may be several quite different contrast values that are equally plausible, leading to several different codewords.

A *generative* model of the type described above typically provides a parameterized distribution $P(\mathbf{h}|\boldsymbol{\theta}^{\mathrm{H}})$ that can be used for encoding the set of latent variables $\mathbf{h}$, as well as a distribution $P(\mathbf{v}|\mathbf{h}, \boldsymbol{\theta}^{\mathrm{V}})$ to be used for encoding the input symbol $\mathbf{v}$ for a given setting of the latent variables. Such a codeword will have an optimal length (*e.g.*, obtained using arithmetic coding) given by

$$\ell(\mathbf{v}, \mathbf{h}) \equiv -\log_2 P(\mathbf{h}|\boldsymbol{\theta}^{\mathrm{H}}) - \log_2 P(\mathbf{v}|\mathbf{h}, \boldsymbol{\theta}^{\mathrm{V}}). \tag{4.14}$$

Note that the generative structure implies that $P(\mathbf{v}|\mathbf{h}, \boldsymbol{\theta}^{\mathrm{V}})$ is easy to compute. (Rissanen

[1989] refers to this type of code as a *two-part code*.)

The set of parameters $\boldsymbol{\theta} = \{\boldsymbol{\theta}^{\mathrm{H}}, \boldsymbol{\theta}^{\mathrm{V}}\}$ must be fixed by hand, estimated using a stored data set, or adapted on-line. Estimating these parameters is a difficult task when there are latent variables. The popular technique of maximum likelihood estimation minimizes the following cost:

$$\mathcal{C} = -\sum_{\mathbf{v}} P_{\mathrm{r}}(\mathbf{v}) \log_2 P(\mathbf{v}|\boldsymbol{\theta}) = -\sum_{\mathbf{v}} P_{\mathrm{r}}(\mathbf{v}) \log_2 \Big[\sum_{\mathbf{h}} P(\mathbf{h}|\boldsymbol{\theta}^{\mathrm{H}}) P(\mathbf{v}|\mathbf{h}, \boldsymbol{\theta}^{\mathrm{V}})\Big]. \tag{4.15}$$

Combining (4.14) and (4.15), we find that maximum likelihood estimation minimizes

$$\mathcal{C} = -\sum_{\mathbf{v}} P_{\mathrm{r}}(\mathbf{v}) \log_2 \Big[\sum_{\mathbf{h}} 2^{-\ell(\mathbf{v},\mathbf{h})}\Big], \tag{4.16}$$

which is equal to the optimal bits-back coding rate (4.10). In contrast, maximum likelihood estimation *does not* minimize the rate for an encoder that always picks the shortest codeword.

Often, maximum likelihood estimation is not tractable when the generative model is overly complex. In these cases, it is possible to use various approximations to maximum likelihood estimation. A common approach [Peterson and Anderson 1987; Neal and Hinton 1993; Zhang 1993; Hinton et al. 1995; Dayan et al. 1995; Saul, Jaakkola and Jordan 1996] is to minimize an *upper bound* on $\mathcal{C}$, thus guaranteeing that the cost is lower than a certain value. (This is described in Section 3.3, where it is called *maximum likelihood-bound estimation*.) The logarithmic term in (4.16) is first bounded by introducing an extra distribution $Q(\mathbf{h}|\mathbf{v})$ and using Jensen's inequality:

$$\log_2 \Big[\sum_{\mathbf{h}} 2^{-\ell(\mathbf{v},\mathbf{h})}\Big] = \log_2 \Big[\sum_{\mathbf{h}} Q(\mathbf{h}|\mathbf{v}) \frac{2^{-\ell(\mathbf{v},\mathbf{h})}}{Q(\mathbf{h}|\mathbf{v})}\Big] \geq \sum_{\mathbf{h}} Q(\mathbf{h}|\mathbf{v}) \log_2 \frac{2^{-\ell(\mathbf{v},\mathbf{h})}}{Q(\mathbf{h}|\mathbf{v})}. \tag{4.17}$$

Inserting this bound into (4.16), we get an upper bound on $\mathcal{C}$:

$$\mathcal{C} \leq \sum_{\mathbf{v}} P_{\mathrm{r}}(\mathbf{v}) \sum_{\mathbf{h}} Q(\mathbf{h}|\mathbf{v}) \log_2 \frac{Q(\mathbf{h}|\mathbf{v})}{2^{-\ell(\mathbf{v},\mathbf{h})}}, \tag{4.18}$$

which is equal to the suboptimal bits-back coding rate (4.8). So, these methods — including the algorithms presented in Section 3.4 — minimize the suboptimal bits-back coding rate. As with exact maximum likelihood estimation, these methods *do not* minimize the rate for an encoder that always picks the shortest codeword.

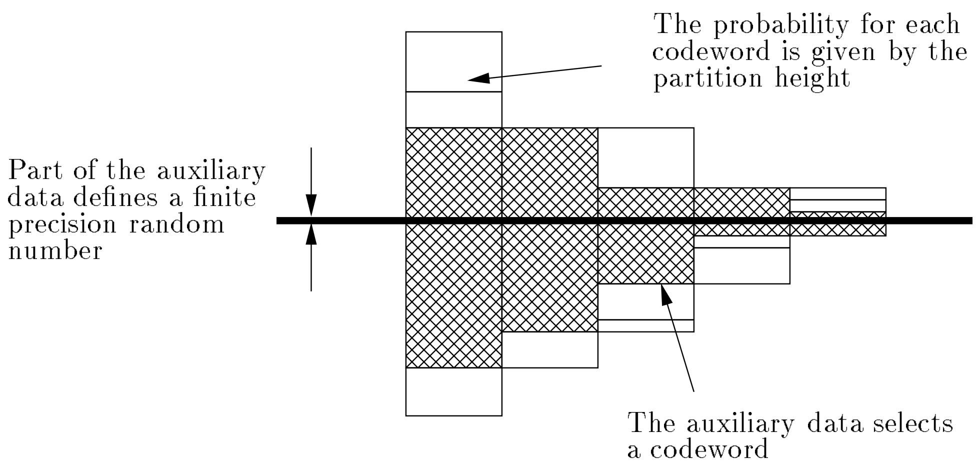

Figure 4.3: Feeding a random number into an arithmetic decoder with appropriate probabilities (shown by the partition heights within a column) selects codewords (shaded partitions), while at the same time conserving information.

4.4 The bits-back coding algorithm

To implement the communication scheme shown in Figure 4.1, we need a general method of recovering the auxiliary data bits from the codeword choices. In the mixture of Gaussians example, we considered a specific input symbol for which there were two codewords. These codewords were selected equally often so that a single bit could be used for bits-back. If the codeword selection distribution is dyadic[2], Huffman *decoding* [Huffman 1952] can be used to pick codewords. In this section, I consider the case of an arbitrary codeword selection distribution. Software that implements the bits-back coding algorithm described in this section can be found at `http://www.cs.utoronto.ca/~frey` and at `ftp://ftp.cs.utoronto.ca/pub/frey/bbc.tar.Z`.

In the case of an arbitrary codeword selection distribution, it is not obvious how random codeword choices can be made without losing auxiliary data information. To address this problem, consider the operation of an arithmetic decoder [Rissanen and Langdon 1976; Witten, Neal and Cleary 1987]. It receives a finite-precision number on $[0, 1)$ and extracts from it a series of decisions according to a table of probabilities. If a collection of uniformly distributed finite-precision numbers on $[0, 1)$ is decoded in parallel, we will obtain a collection of decisions whose distribution exactly matches the table of probabilities. Figure 4.3 shows how an arithmetic decoder can be used to conserve the information in the auxiliary data when making random codeword choices. The probabilities associated with the decisions

[2] *i.e.*, each probability is an integral power of 2.

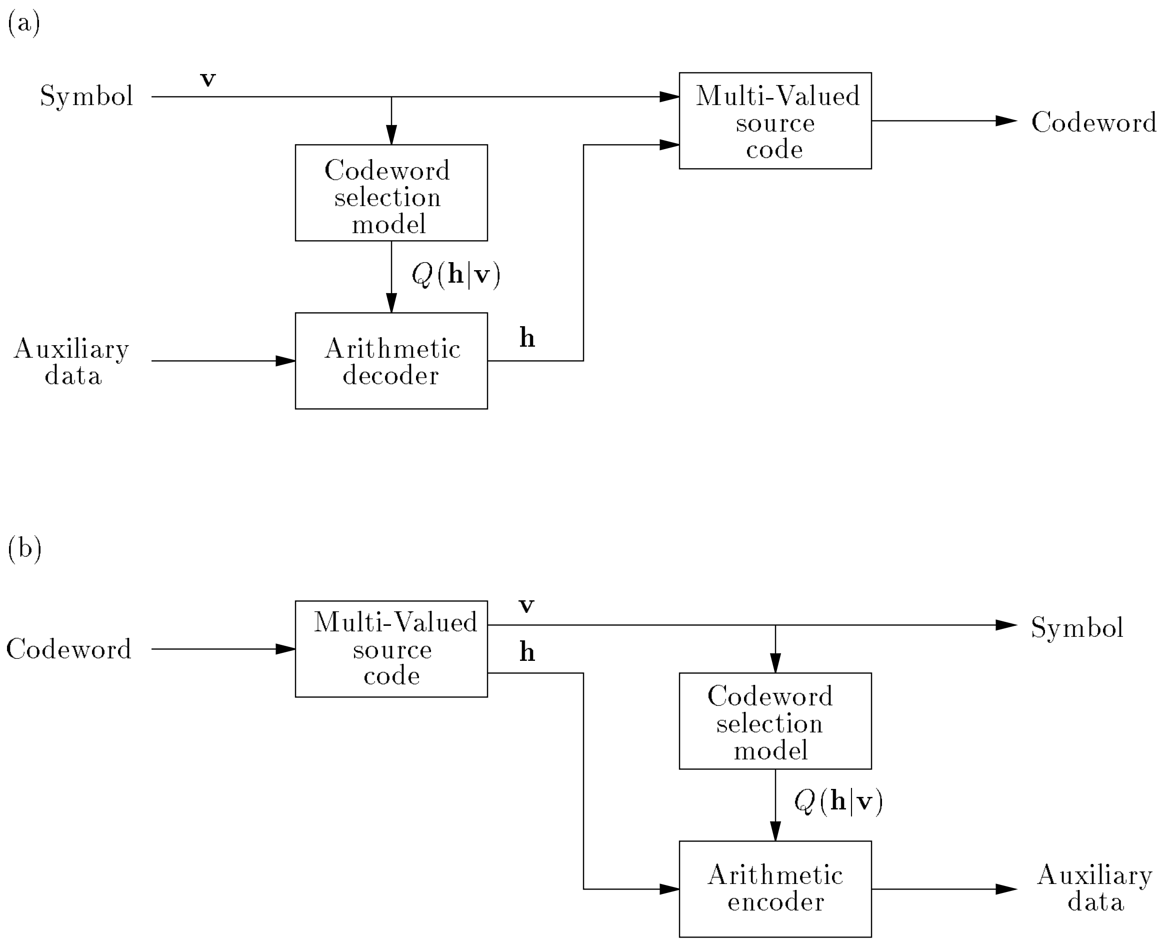

Figure 4.4: The block diagrams for (a) the bits-back encoder and (b) the bits-back decoder.

form the table of the arithmetic decoder, while the auxiliary data defines a random number to be decoded. Each column of the figure corresponds to a single codeword choice and is partitioned into several possible outcomes with the height of each partition proportional to the probability of the corresponding outcome. It is easy to see that if the random number defined by the auxiliary data is uniform, codeword choices will be made according to the codeword selection distribution (as shown for a particular case by the shaded partitions). It is also easy to see that by applying an arithmetic encoder to the sequence of decisions, we can regenerate the random number.

Figures 4.4a and 4.4b show block diagrams for the bits-back encoder and decoder respectively. When the bits-back encoder acquires a symbol $\mathbf{v}$, it uses the codeword selection distribution $Q(\mathbf{h}|\mathbf{v})$ with an arithmetic decoder to choose codeword $\mathbf{h}$, while consuming some auxiliary data bits. The multi-valued source code is then used to produce a codeword of length $\ell(\mathbf{v}, \mathbf{h})$. (For the experiments described in the next section, the multi-valued source

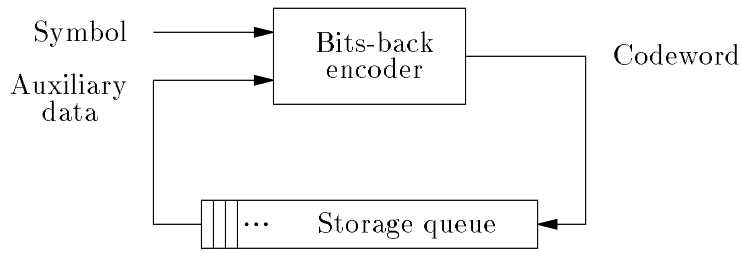

Figure 4.5: The need for extra auxiliary data is eliminated by feeding the codeword bits back into the bits-back encoder as auxiliary data.

code is implemented using an arithmetic encoder in conjunction with a source model distribution $P(\mathbf{h}|\theta)P(\mathbf{v}|\mathbf{h},\theta)$ as described in Section 4.3). When the bits-back decoder receives the codeword, it first decodes $\mathbf{v}$ and $\mathbf{h}$ using the multi-valued source code. It then uses the codeword selection distribution $Q(\mathbf{h}|\mathbf{v})$ with an arithmetic encoder to recover auxiliary data bits back from the codeword choice $\mathbf{h}$.

Insofar as algorithm complexity goes, for an arbitrary codeword selection distribution, the codeword selection procedure described above requires an arithmetic encoder/decoder pair. If codewords are produced using arithmetic coding, the incremental cost of the codeword selection procedure is not overwhelming. In a hardware implementation (*e.g.*, [Feygin 1995]), the codeword selection procedure can run in parallel with codeword production.

4.4.1 The bits-back coding algorithm with feedback

In practice, when encoding a block of symbols, extra auxiliary data is often not readily available. One solution to this problem is to use the binary form of a portion of the block of symbols for auxiliary data. However, so that the bits-back are efficiently utilized, this portion of symbols should first be source coded. Figure 4.5 shows a scheme for using the *same* multi-valued source code for doing just this, when the codewords have integer lengths. In order to encode a block of symbols, some initial *primer* bits (*e.g.*, a few unencoded source symbols) are first placed in the queue. When the next symbol is bits-back encoded, some of the bits in the storage queue are used for auxiliary data. The resulting codeword is fed back into the storage queue so that it can (possibly) be used as auxiliary data later on. Once the entire block of symbols is encoded, the bits-back decoder proceeds to remove the codewords from the storage queue *in reverse order*. Since the decoder has no way of knowing *a priori* how long each codeword is, it is essential that the *encoder* reverse the bits within each codeword before feeding the codeword into the storage queue. The source symbols are decoded in reverse order compared to the order in which they were encoded.

As decoding proceeds, the recovered bits-back are fed into the opposite end of the storage queue and will later be used as codeword bits or primer.

This method is inherently block-oriented, since each block must be decoded in the opposite order in which it was encoded. As a consequence, a block delay is introduced, which is often undesirable. Shorter block lengths will lead to extra overhead due to the primer and also due to framing information (such as a codeword used to indicate the end of the block). However, if the block delay is tolerable, this scheme nicely eliminates the need for extra auxiliary data.

When the multi-valued source code is implemented using arithmetic coding, the above feedback procedure cannot be used as defined. An arithmetic encoder produces a sequence of codeword bits and in general there is no way to break apart this sequence into pieces of integer length such that each piece corresponds to one symbol. This problem is easily solved by dividing the block of symbols into sub-blocks. The arithmetic encoder used to produce codeword bits is halted after each sub-block of symbols is processed. The resulting series of codeword bits is reversed and fed into the storage queue as described above. Practical arithmetic encoders usually waste only a few bits (2 in the implementation described in [Witten, Neal and Cleary 1987]) when encoding is terminated. The sub-block size should be chosen so as to minimize the effect of this wastage. For example, if the optimal bits-back coding rate is 1 bit/symbol, then choosing a sub-block size of 1000 symbols/sub-block will lead to a rate increase of only 0.2%. On the other hand, if the optimal bits-back coding rate is 1000 bits/symbol, arithmetic encoding can be terminated after each symbol (*i.e.*, the sub-block size is 1 symbol/sub-block) and the rate will increase by only 0.2%.

4.4.2 Queue drought in feedback encoders

At first sight, it may appear that queue drought is a serious problem. This can occur if the arithmetic decoder in the bits-back encoder uses up all of the bits in the storage queue and still can't make a codeword choice. In fact, this is usually not a problem because practical arithmetic decoders/encoders [Witten, Neal and Cleary 1987] use a coding value with a restricted size (32 bits in my implementation). Consequently, in my implementation no more than 32 auxiliary data bits will ever be drawn from the storage queue when making a codeword choice. In degenerate cases where the codeword selection distribution places very little mass on one or more short codewords, it is possible for a queue drought to occur when a sequence of very short codewords are chosen that consistently draw a large number of bits each from the storage queue. However, even in such degenerate cases, the sequence of events that leads to a queue drought is highly atypical. I have found that in practice queue drought is not a problem, as long as a reasonable amount of primer (say 20 patterns) is used.

4.5 Experimental results

In this section, I present two sets of results for bits-back data compression. The source models are Helmholtz machines trained using the wake-sleep algorithm (see Section 3.4). The first data set consists of simple patterns of horizontal and vertical bars. The second data set consists of binary images of handwritten digits.

4.5.1 Bits-back coding with a multiple-cause model

In this section, I describe how bits-back coding can be applied to a binary Bayesian network source model, that has one layer of hidden binary variables. Then, I present compression results when the model is fit to images of horizontal and vertical bars using the wake-sleep algorithm described in Section 3.4.3. I compare the compression efficiency of the one-to-many bits-back source coding algorithm with the one-to-one source code obtained using approximate shortest codeword selection, and also with the UNIX `gzip` utility. The multi-valued source code has over 68 billion codewords for each input symbol, and there is no tractable way to mix them, as there is with a hidden Markov model. For a given symbol, most of these codewords are extremely long and therefore play a negligible role in the source code. However, it turns out that the rate for an algorithm that uses a tractable approximation to shortest codeword selection is significantly higher than the suboptimal bits-back coding rate. This indicates that multiple codewords should in some way be accounted for.

It turns out that there isn't an efficient way to convert the multi-valued source code for the sigmoidal Bayesian network into a single-valued source code that achieves a rate that is comparable to the bits-back coding rate. To perform such a conversion, we must compute most of the probability mass corresponding to the codewords for a given data vector. Because of the combinatorial way in which the latent variables $\mathbf{h}$ interact to produce $P(\mathbf{v}|\mathbf{h})$, the marginal probability mass $P(\mathbf{v})$ cannot be computed in a tractable manner. $\mathbf{v}$ could be encoded bit by bit using Gibbs sampling to collect statistics. However, this procedure would require the computationally taxing simulation of a Markov chain for each element in $\mathbf{v}$.

In order to use bits-back coding, we need a codeword selection distribution that is close to $P(\mathbf{h}|\mathbf{v},\boldsymbol{\theta})$. The Helmholtz machine with the wake-sleep learning algorithm provides an estimate of the optimal codeword selection distribution. The learning algorithm jointly estimates the generative network $P(\mathbf{v},\mathbf{h}|\boldsymbol{\theta})$ and a recognition network $Q(\mathbf{h}|\mathbf{v},\boldsymbol{\phi}) \approx P(\mathbf{h}|\mathbf{v},\boldsymbol{\theta})$.

So, an input pattern can be encoded as follows. The sender first uses an ancestral order for the recognition network to compute the probability for the first latent variable (in the ancestral order). This probability and some auxiliary data are then fed into an arithmetic *decoder* which outputs a value for the first latent variable. Given the input pattern and the value for the first latent variable, the sender then computes the probability for the second latent variable, and so on. Once $\mathbf{h}$ has been chosen in this manner, the entire configuration for $\{\mathbf{v}, \mathbf{h}\}$ is arithmetically *encoded* using the method described in Section 4.1.

The receiver decodes the entire configuration for $\{\mathbf{v}, \mathbf{h}\}$ and then computes the probability for the first latent variable using the same ancestral order that was used by the sender. This probability and the value for the variable are fed into an arithmetic *encoder*. While this process is repeated for the remaining latent variables, the arithmetic encoder will output auxiliary data bits.

So that we can compare the performance of bits-back coding with the actual entropy rate of the source, I used a synthetic source to produce 6×6 binary images. The images are iid., and each image is produced by turning on each of the 12 possible horizontal and vertical bars with probability 0.2. (Both types of bars may appear in the same image.) The entropy rate of this source is 8.6 bits/image.

The multiple-cause network had a single hidden layer containing 36 binary variables, in addition to the visible layer containing 36 binary variables. In order to avoid the need for extra auxiliary data, bits-back coding with feedback was used (see Section 4.4.1). The images were grouped into sub-blocks of size 20 images/sub-block and a block size of 200 sub-blocks/block was used. Before each block was encoded, the first sub-block of binary images was used to prime the storage queue. After each block of images was communicated, both the encoding model and the decoding model were adapted using the wake-sleep algorithm with a gradient descent step size of 0.01. The parameters for both the generative network and the recognition network were initialized to 0.0 before any images were processed. I also approximated shortest codeword selection by picking for each image $\mathbf{v}$ the configuration $\mathbf{h}$ that maximized $Q(\mathbf{h}|\mathbf{v}, \phi)$. (The quality of this approximation is discussed below.) Choosing the configuration $\mathbf{h}$ that maximizes $Q(\mathbf{h}|\mathbf{v}, \phi)$ can be done efficiently by considering one latent variable at a time. Figure 4.6 shows the number of codeword bits communicated as a function of the number of blocks encoded for both of these methods. The curves for the uncoded binary image data and the Shannon limit given by the entropy rate of the source are also given. The curve for the UNIX gzip utility with the "-best" option is shown for comparison. (Although the UNIX gzip utility is not really meant for image compression, I include it as a reference point for the reader.) It is evident that if we were to compare the Helmholtz machine with gzip, we would arrive at different conclusions

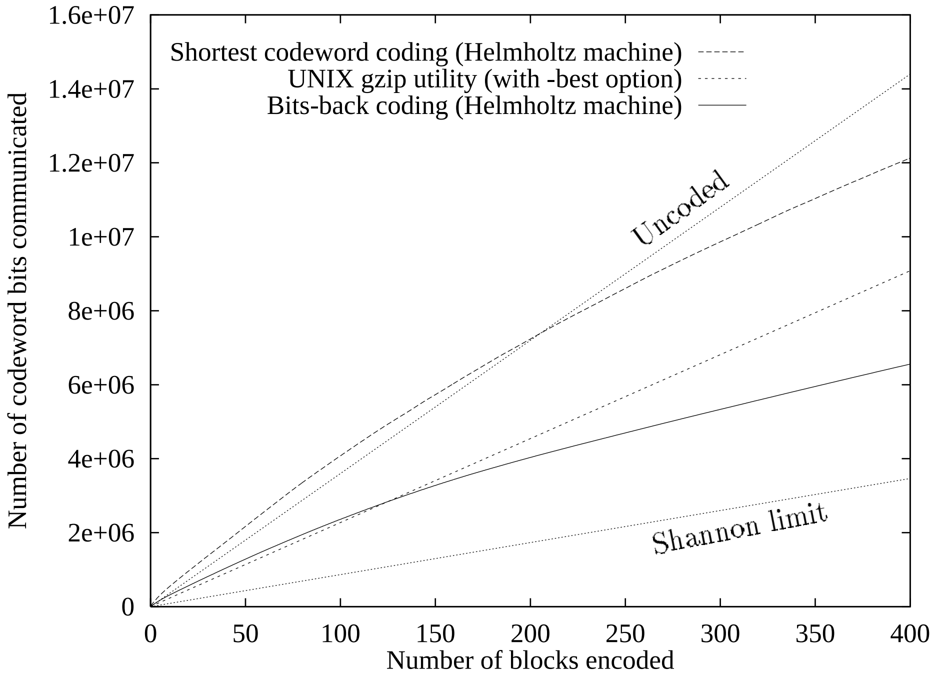

Figure 4.6: Experimental results for a Helmholtz machine with one hidden layer of binary units applied to binary synthetic images.

Table 4.1: Rate comparisons of software-implemented codes for synthetic images.

	Rate (bits/image)
Uncoded binary images	36.0
UNIX gzip utility (with "-best" option)	22.7
Approximate shortest codeword selection (Helmholtz machine)	21.0
Bits-back coding (Helmholtz machine)	12.3
Logistic autoregressive network	10.7
Shannon limit	8.6

depending on whether we used approximate shortest codeword selection or bits-back coding. The bits-back coding curve is clearly superior to the curve for approximate shortest codeword selection.

Table 4.1 gives a comparison of the rates obtained for the next block after 400 blocks of images were processed. The rate for approximate shortest codeword selection is significantly higher than the rate for bits-back coding, indicating that a significant practical savings can be made by using the new algorithm as opposed to shortest codeword selection. However, the communication rate for a logistic autoregressive network that was trained on-line (using

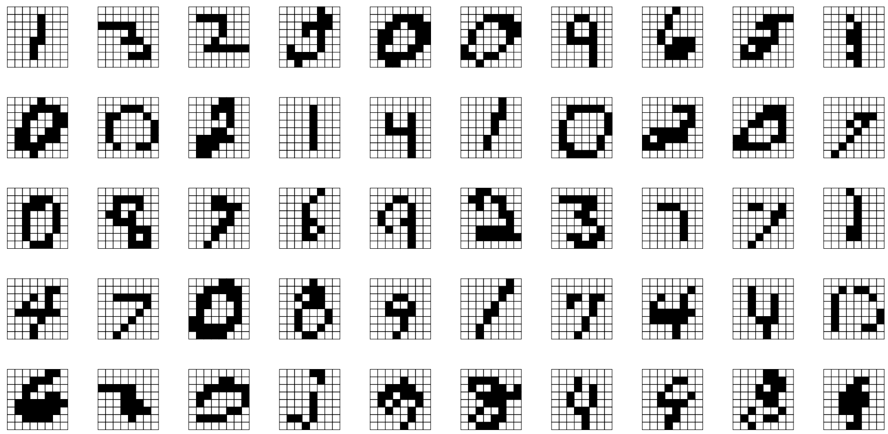

Figure 4.7: Examples of 8×8 binary images of handwritten digits.

a learning rate of 0.01) is also given in Table 4.1, and is significantly lower than the rate for the Helmholtz machine. It appears that although bits-back coding opens the door to new multi-valued source codes, the ones studied in this section are not yet competitive with simpler compression methods.

How close does the approximation to shortest codeword selection come to actually picking the shortest codeword for each data vector? Since there are over 68 billion codewords for each image in the above example, we cannot make a direct comparison by actually searching for the shortest codeword. However, consider the same type of multiple-cause network, except with 9 hidden variables and 9 visible variables, applied to similar synthetic data, except with an image size of 3×3 and a bar probability of 0.1. This network is small enough that an exhaustive codeword search is possible. After processing 400 blocks of 1000 images each, I found that the approximation to shortest codeword selection gave a rate of 5.92 bits/image and exact shortest codeword selection gave a rate of 5.87 bits/image. These two rates are indistinguishable in the first decimal place. I expect that the results for the approximation used for the larger network are also close to the results that would have been obtained if an exhaustive search had been performed.

4.5.2 A Bayesian network that compresses images of handwritten digits

Figure 4.7 shows 50 examples of the binary images that were fed into an adaptive hierarchical Helmholtz machine source model and compressed using bits-back coding. The binary Bayesian network that we use as a source model had three hidden layers of binary variables and one bottom layer of 64 visible variables. From top to bottom, the three layers of causes had 16 variables, 20 variables, and 24 variables, giving a total of 60 latent variables

Table 4.2: Rate comparisons for software-implemented source codes on the binary digit data.

	Rate (bits/image)
Original binary file	64
Shortest codeword selection using the Helmholtz machine	60
`gzip -best`	39
Bits-back coding using the Helmholtz machine	33

(2^{60} codewords for each input pattern). Both the top-down and the bottom-up networks were fully-connected from layer to layer, but had no connections within each layer. The Helmholtz machine was fit to a training set consisting of 100,000 images, using the wake-sleep algorithm.

A comparison of the average rates obtained on the training set using approximate shortest codeword selection and bits-back coding with the estimated binary Bayesian network, as well as the rates obtained by the UNIX `gzip` utility with the `-best` option, are given in Table 4.2. The rate for shortest codeword selection is again significantly higher than the rate for bits-back coding, indicating that a significant practical savings can be made by using bits-back coding.

4.6 Integrating out model parameters using bits-back coding

As noted in Section 4.3, bits-back coding is closely related to statistical inference. In fact, the optimal bits-back coding rate is equivalent to Rissanen's *stochastic complexity* [Rissanen 1989] if we interpret the choice of codeword as a model parameter. Also, if the codewords are constructed by choosing a prior over codeword identities ($P(\mathbf{h}|\boldsymbol{\theta}^{\mathrm{H}})$ in (4.14)), bits-back coding effectively integrates of a discrete set of models.

Both of these relationships lead to an interesting application for bits-back coding. Suppose we are interested in encoding blocks of source symbols and that the source changes from block to block, but not within any single block. Given a model with a continuous parameter vector $\boldsymbol{\theta}$, there is a single block codeword with length $-\log_2 P(\mathcal{D}|\boldsymbol{\theta})$ for each block of source symbols, $\mathcal{D}$. According to the principles of Bayesian analysis, we ought to encode $\mathcal{D}$ by integrating over the entire continuum of models, giving a codeword of length $-\log_2 P(\mathcal{D}) = -\log_2 \int_{\boldsymbol{\theta}} P(\mathcal{D}|\boldsymbol{\theta})P(\boldsymbol{\theta})d\boldsymbol{\theta}$. In practice, this integral is usually impossible to compute and an approximation must be used. One approximation is to use the maximum *a posteriori* (MAP) model (*i.e.*, $\hat{\boldsymbol{\theta}} = \mathrm{argmax}_{\boldsymbol{\theta}} P(\mathcal{D}|\boldsymbol{\theta})P(\boldsymbol{\theta})$), for which the parameters $\hat{\boldsymbol{\theta}}$ are communicated using some (hard to determine) precision.

In fact, bits-back coding can be used to communicate each block of symbols using the entire continuum of models, as long as a good approximation to the posterior distribution, $Q(\boldsymbol{\theta}|\mathcal{D})$, is available. This distribution is used as the *model* selection distribution (in place of the *codeword* selection distribution) and the model parameters are communicated to an *arbitrary* precision. Whereas with the MAP approach, greater precision eventually leads to an *increase* in coding rate, with the bits-back coding approach, greater precision usually leads to a *decrease* in coding rate. Intuitively, this can be seen as an interaction of two processes. First, the extra codeword length caused by greater precision is partly recovered as bits-back. Second, greater precision usually leads to a more accurate approximation to the posterior distribution, and therefore shorter codewords on average. The latter process dominates except in the unusual case when the quantized version of $Q(\boldsymbol{\theta}|\mathcal{D})$ has a lower entropy relative to $P(\boldsymbol{\theta}|\mathcal{D})$ than the unquantized version. I am currently exploring the use of bits-back coding for integrating over continous parameter spaces.

Chapter 5

Channel coding

Our increasingly wired world demands efficient methods for communicating discrete messages over physical channels that introduce errors. Examples of real-world channels include twisted-pair telephone wires, shielded cable-TV wire, fibre-optic cable, deep-space radio, terrestrial radio, and indoor radio. Each of these channels is subject to information-theoretic limitations, physical degradation, and governmental regulation. The prime information-theoretic limitation is Shannon's limit, which gives the maximum average number of information bits that can be communicated per second over a specific channel for a given set of transmitter constraints (*e.g.*, transmission power). Examples of physical degradation include attenuation, thermal noise, self-interference (inter-symbol-interference), multiple-user interference, multiple-path radio reflections, and power limitations in practical circuits. Examples of governmental regulations include transmission power limits, bandwidth usage, and information packet sizes. Together, all of these restrictions and many more define the practical channel coding problem of how to communicate discrete messages reliably.

Despite the multi-faceted nature of the practical channel coding problem, the bottom line is nonetheless quite straightforward. In order to communicate, the transmitter sends a finite-duration real-valued *signalling waveform*. This waveform is determined by a binary information sequence, which we usually assume is uniformly distributed over all possible information sequences. The duration of this waveform may correspond to a relatively short block of information or an infinite-length limiting-case block of information. Once the transmitter has produced a signalling waveform, it is transformed stochastically by the channel and a *received waveform* or *channel output waveform* is obtained at the output of the channel. The receiver then uses the received waveform to make a guess at the information sequence.

Physical channels are usually *band-limited*, meaning that for practical purposes the channel output waveform will not have any frequency components above some limit W Hz. Many channels are also linear (or we assume they are), so that the frequency components of the signalling waveform that are above W Hz will not influence the channel output. Because of this, we need only consider signalling waveforms that are also band-limited to W Hz. Using Nyquist sampling at a rate of $1/\Delta t = 2W$ samples/second, a signalling waveform defined on $[0, N\Delta t]$ can be represented exactly by the discrete-time sequence $\mathbf{a} = \{a_i\}_{i=0}^{N-1}$. The transmission of each sample a_i is called a *channel usage*. Similarly, the channel output waveform can be represented exactly by the discrete-time sequence $\mathbf{y} = \{y_i\}_{i=0}^{N-1}$.

Since the information sequence is effectively random, for multiple trials different signalling sequences will be produced according to some (usually discrete) distribution $p(\mathbf{a})$. The channel output sequence is probabilistically related to this sequence by a channel model $p(\mathbf{y}|\mathbf{a})$.

For a fixed level of additive noise, the transmitter can communicate in an error-free fashion simply by using a very powerful signalling waveform. However, this is an uninteresting and practically expensive solution to the channel coding problem. In practice, a limit is placed on the average transmission power:

$$\int_{\mathbf{a}} p(\mathbf{a}) \left[\frac{1}{N} \textstyle\sum_{i=0}^{N-1} a_i^2\right] d\mathbf{a} \leq P. \tag{5.1}$$

It turns out that the information rate (in bits/channel usage) that can be communicated with arbitrarily low probability of bit error, is bounded from above by the *capacity* C of the channel:

$$C = \max_{\substack{p(\mathbf{a}) \\ \text{subject to (5.1)}}} \frac{1}{N} \int_{\mathbf{a},\mathbf{y}} p(\mathbf{a}) p(\mathbf{y}|\mathbf{a}) \log_2 \frac{p(\mathbf{y}|\mathbf{a})}{p(\mathbf{y})} d\mathbf{a} d\mathbf{y}, \tag{5.2}$$

where the power constraint in (5.1) is enforced during the maximization. This optimal information rate was introduced by Shannon [1948], and is just the mutual information between the channel input sequence and the channel output sequence. (As a practical note, to lower the bit error rate or to use an information rate that is closer to C, we must generally use longer signalling waveforms.)

The channel coding design game essentially consists of devising *encoders* (ways to map information sequences to signalling sequences) and *decoders* (ways to guess at what the information sequence is for a given received sequence). In this thesis, I am mainly interested in conveying to the reader the insight and breadth of application offered by describing channel coding problems using Bayesian networks and using the probabilistic inference algorithms presented in Chapter 2 to perform decoding. For this reason, I begin this chapter by distilling out the essence of the channel coding problem and presenting a simple prototypical problem that will be the focus for the remainder of the chapter. In the prototypical problem, the transmitter sends a discrete-time binary sequence of $+1$'s and -1's, and each of these values is corrupted by additive Gaussian noise. So, the encoder maps each information sequence to a binary signalling sequence, and given a received noisy sequence, the decoder makes a guess at the binary information sequence. It turns out that the solution to this problem has far-reaching consequences in multi-level (nonbinary) coding [Imai and Hirakawa 1977], mainly due to recent proofs by Wachsmann and Huber [1995] and Forney [1997].

In Section 5.2, I show how Bayesian networks and probability propagation can be used to describe and decode Hamming codes, convolutional codes, turbo-codes, serially-concatenated convolutional codes, and low-density parity-check codes. In Section 5.3, I introduce "interleaved trellis-constraint codes", which are a trellis-based generalization of all of the above codes. In Section 5.4, I present a method for speeding up iterative decoders that are implemented on serial machines.

5.1 Simplifying the playing field

The real-valued signalling sequences described above are the price to pay for an efficient description of digital communication in the real world, where signal amplitudes are usually real-valued. The channel coding problem would be much simpler to pose and implement if (1) signal levels were discrete, (2) the channel model was simple, and (3) the mapping from information sequences to channel inputs was assumed to be of a relatively simple form. While this approach can simplify the problem, it can also lead to a communication rate that is far below the general capacity given in (5.2). In this section, I simplify the coding problem in the ways described above, while attempting to argue that if done properly, the simplification will lead to a communication rate that is practically very close to capacity.

5.1.1 Additive white Gaussian noise (AWGN)

A channel model that is simple and works well in practice is the AWGN channel. Additive white Gaussian noise with single-sided spectral density N_0 is added to the signalling waveform to obtain the channel output waveform. Assuming the channel is bandlimited to W Hz (as described above), the decoder can apply a low-pass filter with bandwidth W Hz and sample the noisy waveform at the Nyquist rate to get a discrete-time sequence $\{y_i\}_{i=0}^{N-1}$. It turns out that an AWGN channel simply adds independent Gaussian noise to each input value a_i, where the variance of the noise is related to N_0 by $\sigma^2 = N_0/2$:

$$p(\mathbf{y}|\mathbf{a}) = \prod_{i=0}^{N-1} p(y_i|\mathbf{a}) = \prod_{i=0}^{N-1} p(y_i|a_i)$$

$$p(y_i|a_i) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-(y_i-a_i)^2/2\sigma^2}, \quad i = 0, 1, \ldots, N-1. \tag{5.3}$$

If the decoder applies a low-pass filter with a higher bandwidth, then frequency components of the AWGN that are above W Hz will increase the effective noise on the sequence $\{y_i\}_{i=0}^{N-1}$.

The AWGN channel leads to an appealing formulation of maximum likelihood (ML) signal detection. The log-probability density of the received sequence given the signalling sequence is

$$\log p(\mathbf{y}|\mathbf{a}) = \log \prod_{i=0}^{N-1} p(y_i|a_i) = -\frac{1}{2\sigma^2} \sum_{i=0}^{N-1} (y_i - a_i)^2 - N \log \sqrt{2\pi\sigma^2}. \tag{5.4}$$

So, ML signal detection for the AWGN channel consists of finding the allowed signalling sequence $\mathbf{a}$ that is closest to $\mathbf{y}$ in *Euclidean distance*.

5.1.2 Capacity of an AWGN channel

For the AWGN channel, each channel output y_i depends only on a_i, and not any a_j, $j \neq i$. Consequently, the signalling distribution $p(\mathbf{a})$ that will give the highest mutual information is of product form:

$$p(\mathbf{a}) = \prod_{i=0}^{N-1} p(a_i). \tag{5.5}$$

(This distribution allows us to stuff as much information into each a_i as possible.) In this case, the capacity in (5.2) simplifies to

$$C = \max_{\substack{p(a_i) \\ \mathrm{VAR}[a_i] \leq P}} \int_{a_i, y_i} p(a_i) p(y_i|a_i) \log_2 \frac{p(y_i|a_i)}{p(y_i)} da_i dy_i \tag{5.6}$$

bits per channel usage. Note that for a product-form signalling distribution, the power limit in (5.1) becomes $\mathrm{VAR}[a_i] = \int_{a_i} p(a_i) a_i^2 da_i \leq P$.

It turns out that the maximum in (5.6) is obtained by a Gaussian signalling distribution with variance P (see [Cover and Thomas 1991]), and the capacity is

$$C = \frac{1}{2}\log_2\big(1 + \frac{P}{\sigma^2}\big). \tag{5.7}$$

For example, if $P = 3\sigma^2$, then $C = 1$ bit/channel usage. For reasonable power levels, it is not possible to deterministically map C bits of information to a value a_i that will have a Gaussian distribution (or one that is even close to Gaussian). For example, try mapping 1 bit of information to a variable whose distribution close to Gaussian!

The optimality of a Gaussian signalling distribution leads to a new type of coding concept called *shaping*. A signalling technique has good shape if the marginal signalling distributions are nearly Gaussian. If the signalling shape is poor, then the capacity given in (5.7) cannot be achieved no matter how good a code is used. For example, if binary signalling is used ($a_i \in \{-\sqrt{P}, +\sqrt{P}\}$), then the channel capacity cannot be achieved, as shown in Section 5.1.6 and Figure 5.2.

The interplay between shaping and coding is very important. As another example, here is a method that has an excellent signalling shape, but uses a poor code. We first construct a table that maps each information vector $\mathbf{u}$ to a real value $c_{\mathbf{u}}$ in a way so that a uniform distribution over information vectors induces a nearly Gaussian distribution over $c_{\mathbf{u}}$. For a given information vector $\mathbf{u}$, the transmitter simply sends a constant waveform, $a_i = c_{\mathbf{u}}$, $i = 0, \ldots, N-1$. Using this method, each marginal distribution $p(a_i)$ can be made to be as close to Gaussian as desired, by increasing N and refining the map from $\mathbf{u}$ to $c_{\mathbf{u}}$. However, because the waveform is constant there is no way to introduce a good code. A fruitful structure that leads to a nice mix between coding and shaping is the *signal constellation.*

5.1.3 Signal constellations

Since the information sequence is discrete and the signalling sequence is determined from the information sequence, the allowable set of signalling sequences is also discrete. How should we specify the set of allowed signalling sequences? One way is to require that the signalling variable at each time step be a member of a fixed signal set. Figure 5.1a shows the signalling points for two signalling variables a_0 and a_1, where each variable can take on one of eight values. Even if a good code is used with these signalling points, the marginal signalling distributions are quite far from Gaussian and so the rate will be below capacity. Instead, consider breaking the signalling sequence into a series of groups (*i.e.*, subspaces) containing n values each. A discrete set of values (called a *constellation*) is then judiciously chosen within each n-dimensional subspace in a way that leads to marginal signalling distributions that are close to Gaussian.

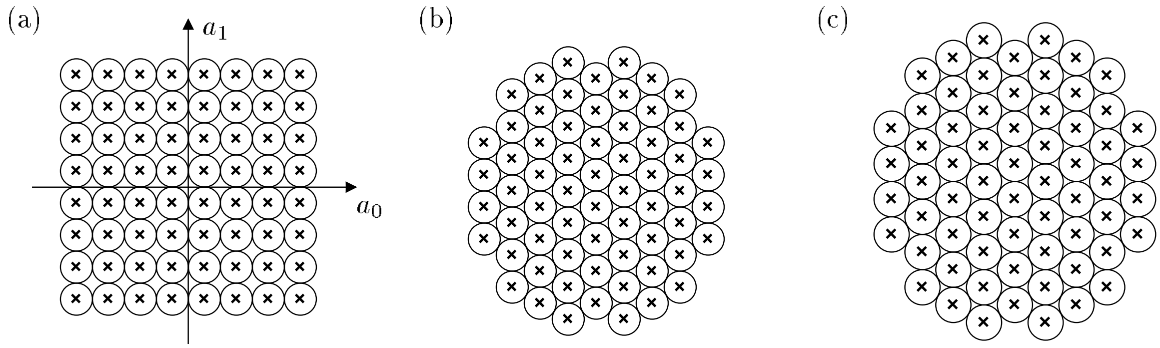

Figure 5.1: Signal constellations can be used to increase the Euclidean distance between signalling points (indicated by crosses). (a) A naive constellation for an $n = 2$ signalling set with 64 points. (b) The same 64 points can be rearranged in order to reduce the transmission power. (c) The constellation from (b) scaled up so that its transmission power is the same as the power for (a) — notice that the nearest-neighbor distance has increased.

Another way to understand the benefit of using signal constellations is through a sphere-packing argument. Consider the 2-dimensional constellation shown in Figure 5.1a that corresponds to the naive approach described above. For a fixed noise level in an AWGN channel, detection error falls off with distance between nearest-neighbor signals. Imagine centering a 2-dimensional sphere on each signal point as shown. Now, by trying to pack the spheres as tightly as possible, we obtain the constellation shown in Figure 5.1b. The nearest-neighbor distance has not changed, but the transmission power has decreased (since the sum of squared distances to signalling points is lower). In order to use the same power as the naive approach uses, we can now increase the Euclidean distance between nearest-neighbors as shown in Figure 5.1c. This will increase the noise-tolerance of the system, and so increase the communication rate relative to the naive approach. For higher-dimensional constellations, this sphere-packing gain becomes more valuable. (This simple example ignores the increase in the number of nearest neighbors from 4 to 6. See Lee and Messerschmitt [1994] for more details.)

5.1.4 Linear binary codes are all we need!

Although the design of optimal high-dimensional constellations is straightforward in theory, it is very difficult to implement practical encoders and decoders that use these constellations. Consequently, we must approximate optimal constellations by practical ones. Ways of doing this include trellis codes (a.k.a. coset codes) [Ungerboeck 1982; Calderbank and Sloane 1987; Forney 1988], which your telephone modem probably uses. Alternatively, Wachsmann and Huber [1995] and Forney [1997] have shown that by using a technique called *multilevel coding* [Imai and Hirakawa 1977], we can achieve the capacity in (5.2) by combining several relatively simple *linear binary codes*. That is, optimal constellations can be well approximated if we can design appropriate linear binary codes. I refer to these new proofs to justify my focus on linear binary codes in this thesis.

A *binary code* maps each binary information vector $\mathbf{u}$ of length K to a binary codeword vector $\mathbf{x}$ of length N. The *rate* R of such a binary code is defined as

$$R = K/N. \tag{5.8}$$

I will sometimes highlight the mapping by writing the codeword for $\mathbf{u}$ as $\mathbf{x}(\mathbf{u})$. A binary code is *linear* if for any $\mathbf{u}_1$ and $\mathbf{u}_2$, $\mathbf{x}(\mathbf{u}_1 \oplus \mathbf{u}_2) = \mathbf{x}(\mathbf{u}_1) \oplus \mathbf{x}(\mathbf{u}_2)$, where, "$\oplus$" indicates component-wise modulo 2 addition ($0 \oplus 0 = 0$, $0 \oplus 1 = 1$, $1 \oplus 0 = 1$, and $1 \oplus 1 = 0$). Note that this form of linearity is highly nonlinear in the sense of continuous algebra (where $1 + 1 = 2$). In general, linear codes are easier to analyze than nonlinear ones.

Each bit in the codeword can be transmitted using *binary signalling*, also called *binary antipodal signalling*. (If the binary signal is modulated by a carrier so that it is a passband signal, it is sometimes called *binary phase-shift keying* (BPSK).) For $x_i = 1$ we transmit $a_i = \sqrt{P}$ and for $x_i = 0$ we transmit $a_i = -\sqrt{P}$. In this way, the average transmitted power is P. For an AWGN channel, we can write the probability density of channel output y_i directly in terms of x_i (bypassing a_i):

$$p(y_i|x_i) = \begin{cases} \frac{1}{\sqrt{2\pi\sigma^2}} e^{-(y_i - \sqrt{P})^2/2\sigma^2} & \text{if } x_i = 1 \\ \frac{1}{\sqrt{2\pi\sigma^2}} e^{-(y_i + \sqrt{P})^2/2\sigma^2} & \text{if } x_i = 0. \end{cases} \tag{5.9}$$

A simple linear binary code is the repetition code. Each information bit is transmitted m times, so that $R = K/mK = 1/m$. Using (5.9), the probability density of channel outputs $y_0, \ldots, y_{m-1}$ given x_0 is

$$p(y_0, \ldots, y_{m-1}|x_0) = \prod_{i=0}^{m-1} p(y_i|x_0) \propto \begin{cases} e^{-(\frac{1}{m}\sum_{i=0}^{m-1} y_i - \sqrt{P})^2/2(\sigma^2/m)} & \text{if } x_0 = 1 \\ e^{-(\frac{1}{m}\sum_{i=0}^{m-1} y_i + \sqrt{P})^2/2(\sigma^2/m)} & \text{if } x_0 = 0, \end{cases} \tag{5.10}$$

where the constant of proportionality does not depend on x_0. By basing the decoding decision on $\frac{1}{m}\sum_{i=0}^{m-1} y_i$, the receiver effectively reduces the noise variance by a factor of $1/m$. It turns out that this is a very poor code, because the suppression of noise comes at too high a cost in terms of decreasing the code rate.

5.1.5 Bit error rate (BER) and signal-to-noise ratio (E_b/N_0)

The error-correcting performance of a coding system is often given as a bit error rate (BER), which is the average probability that an information bit will be guessed incorrectly by the decoder. In order to compare the BERs of different coding schemes, we need a relatively robust measure of the noise level that each system is being exposed to. Simply stating the noise variance for an AWGN channel is not sufficient, since one system may be transmitting at a much higher power than another. Also, as shown above, performance can be improved in a trivial fashion simply by repeating signals. A reasonably robust measure of the noise level is

$$E_b/N_0 = \frac{P}{N_0 R} = \frac{P}{2\sigma^2 R}, \tag{5.11}$$

where P is the transmitter power, N_0 is the single-sided spectral density of the AWGN, σ^2 is the AWGN variance, and R is the rate of the code. E_b/N_0 is the ratio between the power that is transmitted per information bit, and the AWGN power. It is usually given in units of decibels (dB),

$$10 \log_{10} E_b/N_0. \tag{5.12}$$

Notice that although dividing by R in (5.11) does cancel the effect of the improvement obtained trivially by repeating signals, it does not take into account the increased bandwidth needed for lower rate codes. In fact, in the next section we see that the minimum E_b/N_0 needed for error-free communication depends on the rate. So, when comparing one coding system to another that uses a lower rate, we must keep in mind that there is usually some way to modify the former system so as to lower its rate and at the same time lower the E_b/N_0 it needs to achieve error-free communication at that rate.

5.1.6 Capacity of an AWGN channel with $+1/-1$ signalling

Engineering bandwidth restrictions aside, what are the communication limits for an AWGN channel when we use $+1/-1$ signalling (*i.e.*, binary antipodal signalling with $P = 1$)? (Without loss of generality, we will assume that $P = 1$.) The answer to this question depends on whether we are willing to tolerate a certain non-vanishing BER. Before considering the non-vanishing BER case in the next section, I will address the simpler case of a vanishing BER. More specifically, what is the minimum E_b/N_0 needed to communicate error-free using a rate R code on an AWGN channel with $+1/-1$ signalling?

The mutual information between the channel input a_i and the channel output y_i at time step i gives the number of bits that can be communicated per channel usage on average. For an AWGN channel with $+1/-1$ signalling, the mutual information as a function of the noise variance is

$$\begin{aligned} M(\sigma^2) &= \sum_{a_i \in \{-1,+1\}} \int_{y_i} p(a_i, y_i) \log_2 \frac{p(a_i, y_i)}{p(a_i)p(y_i)} dy_i \\ &= \sum_{a_i \in \{-1,+1\}} \int_{y_i} p(a_i, y_i) \log_2 p(y_i|a_i) da_i dy_i - \int_{y_i} p(y_i) \log_2 p(y_i) dy_i \end{aligned} \tag{5.13}$$

The first term is the entropy of y_i given a_i, which is just the entropy of a Gaussian distribution, $0.5 \log_2(2\pi\sigma^2 e)$. Since $p(y_i)$ is a mixture of two Gaussians, the second term is

$$\int_{y_i} \Big[\frac{e^{-(y_i-1)^2/2\sigma^2}}{2\sqrt{2\pi\sigma^2}} + \frac{e^{-(y_i+1)^2/2\sigma^2}}{2\sqrt{2\pi\sigma^2}} \Big] \log_2 \Big[\frac{e^{-(y_i-1)^2/2\sigma^2}}{2\sqrt{2\pi\sigma^2}} + \frac{e^{-(y_i+1)^2/2\sigma^2}}{2\sqrt{2\pi\sigma^2}} \Big] dy_i, \tag{5.14}$$

which can be approximated quite well using a Monte Carlo method. In this fashion, it is possible to obtain a good estimate of $M(\sigma^2)$.

To communicate error-free, the rate of the code must be less than the mutual information between the channel input and the channel output: $R < M(\sigma^2)$ [Shannon 1948]. Inserting $\sigma^2 = 1/(2RE_b/N_0)$ (see (5.11)) into this inequality, we get $R < M(\frac{1}{2RE_b/N_0})$. After rearrangement, we have

$$E_b/N_0 > \frac{1}{2RM^{-1}(R)}. \tag{5.15}$$

This bound (based on an interpolated inverse of a Monte Carlo estimate of $M(\sigma^2)$) is shown in Figure 5.2a, along with the minimum E_b/N_0 required by optimal (Gaussian) signalling (see Section 5.1.2).

For example, an $R = 1/2$ code requires $E_b/N_0 > 0.2$ dB. To communicate error-free without coding ($R = 1$), an infinite E_b/N_0 is needed.

A standard result from information theory is that regardless of rate, an E_b/N_0 of at *least* $\log_e(2)$ (-1.5917 dB) is required for error-free communication [Cover and Thomas 1991]. This limit is apparent from the convergence of the curve as $R \to 0$.

5.1.7 Achievable BER for an AWGN channel with +1/-1 signalling

If we are willing to tolerate a certain non-vanishing BER while using a rate R code, it turns out we can use a lower E_b/N_0 than described in the previous section. One way to pose

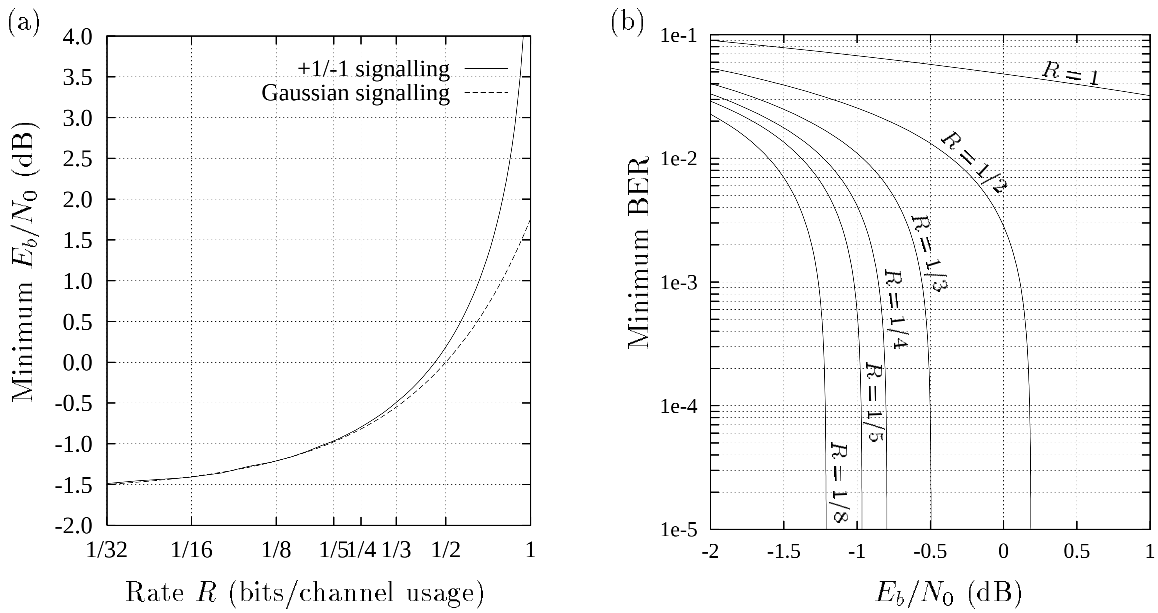

Figure 5.2: (a) The minimum E_b/N_0 needed for error-free communication with a rate R code, over an AWGN channel using $+1/-1$ signalling and optimal (Gaussian) signalling. (b) The minimum achievable BER as a function of E_b/N_0 for several different code rates using $+1/-1$ signalling.

the problem for this scenario is: For a code with rate R and a specified BER, what is the minimum required E_b/N_0? We can think of this as a two-stage problem. First, we find a shorter representation for the information vector. This representation will obviously be lossy, since a uniformly random vector of information bits cannot be represented losslessly on average by a shorter binary vector. Second, we use a new code to communicate this shorter representation error-free over the channel with the largest tolerable noise variance. Since the representation is shorter than the information vector, the new code rate R' will be lower than the old one: $R' < R$. So, the tolerable noise variance for error-free communication of the representation will be higher than the tolerable noise variance for error-free communication of the information vector.

We would like to use a representation that is as short as possible, so that R' will be as low as possible and the tolerable noise variance will be as large as possible. However, shorter representations are also more lossy and will lead to higher BER's. What is the minimum ratio between the length of the representation and the length of the information vector, such that the error rate does not rise above the specified BER? It turns out that the minimum ratio is just the mutual information between a uniformly random bit and its noisy duplicate, where the probability that the value of the duplicate is flipped is BER. (This can be viewed as a result of rate-distortion theory applied to a Bernoulli source [Cover and

Thomas 1991].) This mutual information is

$$1 + \mathrm{BER}\log_2(\mathrm{BER}) + (1 - \mathrm{BER})\log_2(1 - \mathrm{BER}). \tag{5.16}$$

The new code rate is

$$R' = R\big[1 + \mathrm{BER}\log_2(\mathrm{BER}) + (1 - \mathrm{BER})\log_2(1 - \mathrm{BER})\big]. \tag{5.17}$$

For a specified R and BER, we can compute R', determine the maximum tolerable noise variance $\sigma^2 = M^{-1}(R')$, and compute the minimum $E_b/N_0 = 1/(2\sigma^2 R)$ (note that to compute E_b/N_0, we use the original R, not R'). Figure 5.2b shows the achievable BER as a function of E_b/N_0 for several different rates. For each rate, the value for E_b/N_0 at which the BER converges to zero is the same as the value shown in Figure 5.2a. These achievable BER curves are used as guides for ascertaining the performances of codes and decoders later in this chapter.

5.2 Bayesian networks for channel coding

A critical component of a channel coding system is the decoder. Even if the code gives excellent performance when optimal decoding is used, if there is no way to implement a practical decoder that gives similar performance, it is not clear that the code is of any use. Channel decoders can be broken into two classes: algebraic and probabilistic. Algebraic decoders for binary codes usually quantize the channel output to two or three levels. The received vector $\mathbf{y}$ is interpreted as a copy of the binary codeword vector $\mathbf{x}$, with some of the bits flipped. Alternatively, received values that are highly ambiguous (*e.g.*, the value 0.1 when $+1/-1$ signalling is used) are considered as *erasures* — *i.e.*, the corresponding bit in $\mathbf{y}$ is assumed to be unknown. In both cases, decoding is a matter of using linear algebra (in a finite field) to find the binary codeword vector $\mathbf{x}$ that is closest to $\mathbf{y}$ in Hamming distance (dimensions that are erased are ignored). There are many techniques for algebraic decoding [Lin and Costello 1983; Blahut 1990; Wicker 1995] and algebraic decoders usually take advantage of special structure that is built into the code to make decoding easier. However, it is obvious that by using such a coarsely quantized form of the channel output, these decoders are suboptimal (*e.g.*, the value 0.1 from above does provide *some* evidence that a signal value of $+1$ was sent).

Probabilistic decoders are designed to make as much use as is practically possible of the real-valued unquantized channel output. The goal of probabilistic decoding is either maximum likelihood (ML) information sequence detection, or maximum *a posteriori* (MAP) information bit detection:

$$\mathbf{u}^{\mathrm{ML}} = \underset{\mathbf{u}}{\operatorname{argmax}} \; p(\mathbf{y}|\mathbf{u}),$$

$$u_k^{\mathrm{MAP}} = \underset{u_k}{\operatorname{argmax}} \; p(u_k|\mathbf{y}) \;\; 0 \leq k \leq K-1. \tag{5.18}$$

Obviously, ML sequence detection minimizes the word error rate (we usually assume that all words are equally likely *a priori*), while MAP bit detection minimizes the BER. So, by definition, optimal probabilistic decoders are superior to optimal algebraic decoders. However, can we implement useful probabilistic decoders? The success of algebraic decoders is due to the way they take advantage of the algebraic structure of a code. Is there an analogous structure that probabilistic decoders can use? In this section, I show how Bayesian networks can be used to describe probabilistic structure for channel codes and how the inference algorithms that make use of this structure can be used for probabilistic decoding.

5.2.1 Hamming codes

Hamming codes are an extension of the notion of adding a single parity-check bit to a vector of information bits. Instead of adding a single bit, multiple bits are added and each of these parity-check bits depends on a different subset of the information bits. Hamming developed these codes with a special algebraic structure in mind. Consequently, they are really meant for binary channels where the noise consists of randomly flipping bits. However, Hamming codes are short and easy to describe, so they make a nice toy example for the purpose of illustrating probabilistic decoding.

An (N, K) Hamming code takes a binary information vector of length K and produces a binary codeword of length N. For an integer $m \geq 2$, N and K must satisfy $N = 2^m - 1$ and $K = 2^m - m - 1$. The Bayesian network for a $K = 4$, $N = 7$ rate 4/7 Hamming code is shown in Figure 5.3a. The algebraic structure of this code can be cast in the form of the conditional probabilities that specify the Bayesian network. Assuming the information bits are uniformly random, we have $P(u_k = 1) = P(u_k = 0) = 0.5$, $k = 0, 1, 2, 3$. Codeword bits 0 to 3 are direct copies of the information bits: $P(x_k|u_k) = \delta(x_k, u_k)$, $k = 0, 1, 2, 3$. Codeword bits 4 to 6 are parity-check bits:

$$P(x_4|u_0, u_1, u_2) = \delta(x_4, u_0 \oplus u_1 \oplus u_2),$$

$$P(x_5|u_0, u_1, u_3) = \delta(x_5, u_0 \oplus u_1 \oplus u_3),$$

$$P(x_6|u_1, u_2, u_3) = \delta(x_6, u_1 \oplus u_2 \oplus u_3). \tag{5.19}$$

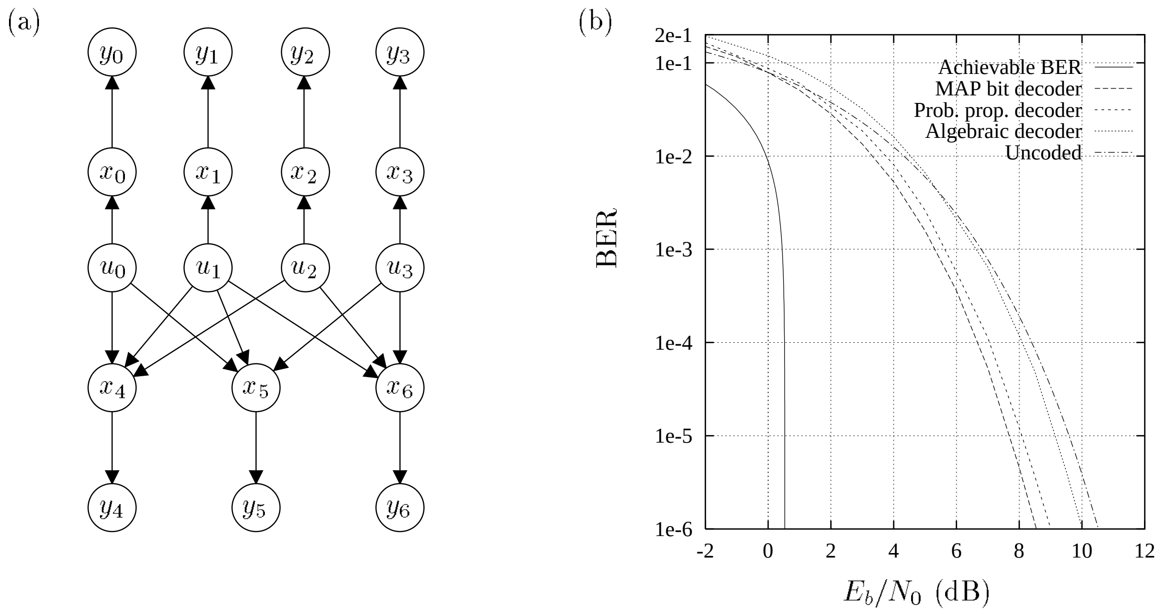

Figure 5.3: (a) The Bayesian network for a $K = 4$, $N = 7$ Hamming code. (b) BER performance for the maximum likelihood decoder, the iterative probability propagation decoder, and an algebraic decoder. (The key lists the curves in the order in which they appear from left to right at BER = 1e-6.)

Assuming binary antipodal signalling with power P over an AWGN channel, the conditional channel probabilities $p(y_i|x_i)$, $i = 0, 1, 2, 3, 4, 5, 6$ are given by (5.9), where σ^2 is related to E_b/N_0 by (5.11).

This code is small enough that we can compute the MAP bit values in (5.18) exactly using Bayes rule. The BER-E_b/N_0 curve for MAP bit decoding and the achievable BER (see Section 5.1.7) at rate 4/7 are shown in Figure 5.3b. Although there is an 8 dB gap between these curves at a BER of 10^{-6}, the MAP decoder gives a significant improvement of 2 dB over uncoded transmission (whose corresponding curve is also shown).

By making hard decisions for the channel outputs (calling a value below 0.0 a "0" and calling a value above 0.0 a "1"), an algebraic decoder can be used. This decoder applies a *parity-check matrix* to the received binary word in order to try to locate any errors. (See Lin and Costello [1983] for details.) In this fashion, it can correct up to one bit error per codeword. The curve for algebraic decoding is also shown in Figure 5.3b. Algebraic decoding gives an improvement of only 0.5 dB over uncoded transmission at a BER of 10^{-6}. Although this may seem surprising, keep in mind that the receiver for the uncoded transmission is allowed to *average* the channel output to reduce the effective noise (see Section 5.1.4) 7/4 times longer than the receiver for the algebraic decoder.

One way to approximate the probabilities $P(u_k|\mathbf{y})$ used for MAP bit decoding is to apply the probability propagation inference algorithm (Section 2.1) to the Bayesian network shown in Figure 5.3a. Probability propagation is only approximate in this case because the network is multiply-connected or "loopy" (*e.g.*, u_0-x_4-u_1-x_5-u_0). Once a channel output vector $\mathbf{y}$ is observed, propagation begins by sending a message from y_k to x_k for $k = 0, 1, 2, 3, 4, 5, 6$. Then, a message is sent from x_k to u_k for $k = 0, 1, 2, 3$. An *iteration* now begins by sending messages from the information variables u_0, u_1, u_2, u_3 to the parity-check variables x_4, x_5, x_6 in parallel. The iteration finishes by sending messages from the parity-check variables back to the information variables in parallel. Each time an iteration is completed, new estimates of $P(u_k|\mathbf{y})$ for $k = 0, 1, 2, 3$ are obtained. The curve for probability propagation decoding using 5 iterations is shown in Figure 5.3b. It is quite close to the MAP decoder, and significantly superior to the algebraic decoder. The interactive software package BNC (Bayesian Networks for Coding) that was used to obtain these results is described in Appendix B.

For this simple Hamming code, the complexities of the probability propagation decoder and the MAP decoder are comparable. However, the similarity in performance between these two decoders raises the question: "Can probability propagation decoders give performances comparable to MAP decoding in cases where MAP decoding is computationally intractable?" Before exploring a variety of systems where probability propagation in multiply-connected networks gives surprisingly good results, I will review convolutional codes, whose Bayesian networks are essentially singly-connected chains. For these networks, the probability propagation algorithm is exact and it reduces to the well-known forward-backward algorithm [Baum and Petrie 1966] (a.k.a. BCJR algorithm [Bahl et al. 1974]).

5.2.2 Convolutional codes

Convolutional codes are produced by driving a finite state machine with information bits. The outputs of the finite state machine (which may include copies of the inputs) are then used as codeword bits. A code for which each information bit appears as a codeword bit is called *systematic*. Typically, linear convolutional codes are used, and any code in this class can be represented by a *linear feedback shift register* (LFSR). An example of a systematic code of this type with a memory of 7 bits is shown in Figure 5.4a. Each box represents a 1-bit memory element and D is a delay operator: $D^n u_k = u_{k-n}$. In this example, there is no feedback from the shift register to its input; a convolutional code of this type is called *nonrecursive*. An output is produced by adding (modulo 2) values "tapped" from the memory chain. The output taps for this rate 1/2 systematic nonrecursive convolutional code were chosen to maximize the minimum distance $d_{\min}$ between codewords [Lin and Costello 1983]. For this code, $d_{\min} = 7$, meaning that the codeword vectors for *any* two information vectors will differ in at least 7 places. Using the delay operator, this code can be described by the following two equations:

$$x_{2k} = u_k, \quad x_{2k+1} = G(D)u_k = (1 + D + D^3 + D^5 + D^6 + D^7)u_k, \tag{5.20}$$

where $G(D)$ is called the *generator polynomial*. This polynomial is often expressed in octal form by letting the coefficient of D^0 be the least significant bit and the coefficient of D^7 be the most significant bit. In this case the octal representation is 353_8.

Since $d_{\min}$ plays the central role in determining the error-correcting capabilities of a code at high signal-to-noise ratio E_b/N_0, we would like to use codes that have large $d_{\min}$. One way to obtain a greater $d_{\min}$ for convolutional codes is to use a larger memory. However, it turns out that decoding complexity increases exponentially with the size of the memory. In fact, it is possible to increase the minimum distance of any systematic nonrecursive convolutional code without using more memory. Figure 5.4b shows a rate 1/2 nonsystematic nonrecursive convolutional code that has $d_{\min} = 10$. (The two sets of output taps that maximize $d_{\min}$ were found using a method described in [Lin and Costello 1983].) This code can be described as follows:

$$\begin{aligned} x_{2k} &= G_1(D)u_k = (1 + D + D^2 + D^5 + D^7)u_k, \\ x_{2k+1} &= G_2(D)u_k = (1 + D^3 + D^4 + D^5 + D^6 + D^7)u_k. \end{aligned} \tag{5.21}$$

For a nonsystematic convolutional code, there are two generator polynomials corresponding to the two sets of output taps. For this code, the octal representation is $(247_8, 371_8)$.

Although the performance of the nonsystematic code described above is better than the systematic one at high E_b/N_0, it is the other way around for values of E_b/N_0 near the Shannon limit. Berrou and Glavieux [1996] have argued that a nice compromise between these codes is a systematic recursive convolutional code. The code in Figure 5.4b can be converted to a systematic code by taking one set of the output taps (either one will do) and using them as *feedback* to the input of the shift register, making a LFSR. If we do this with the upper set of taps, we obtain the rate 1/2 systematic recursive convolutional code shown in Figure 5.4c. This code can be described by the following two equations:

$$\begin{aligned} x_{2k} &= u_k, \\ x_{2k+1} &= G(D)u_k = \frac{1 + D^3 + D^4 + D^5 + D^6 + D^7}{1 + D + D^2 + D^5 + D^7}u_k. \end{aligned} \tag{5.22}$$

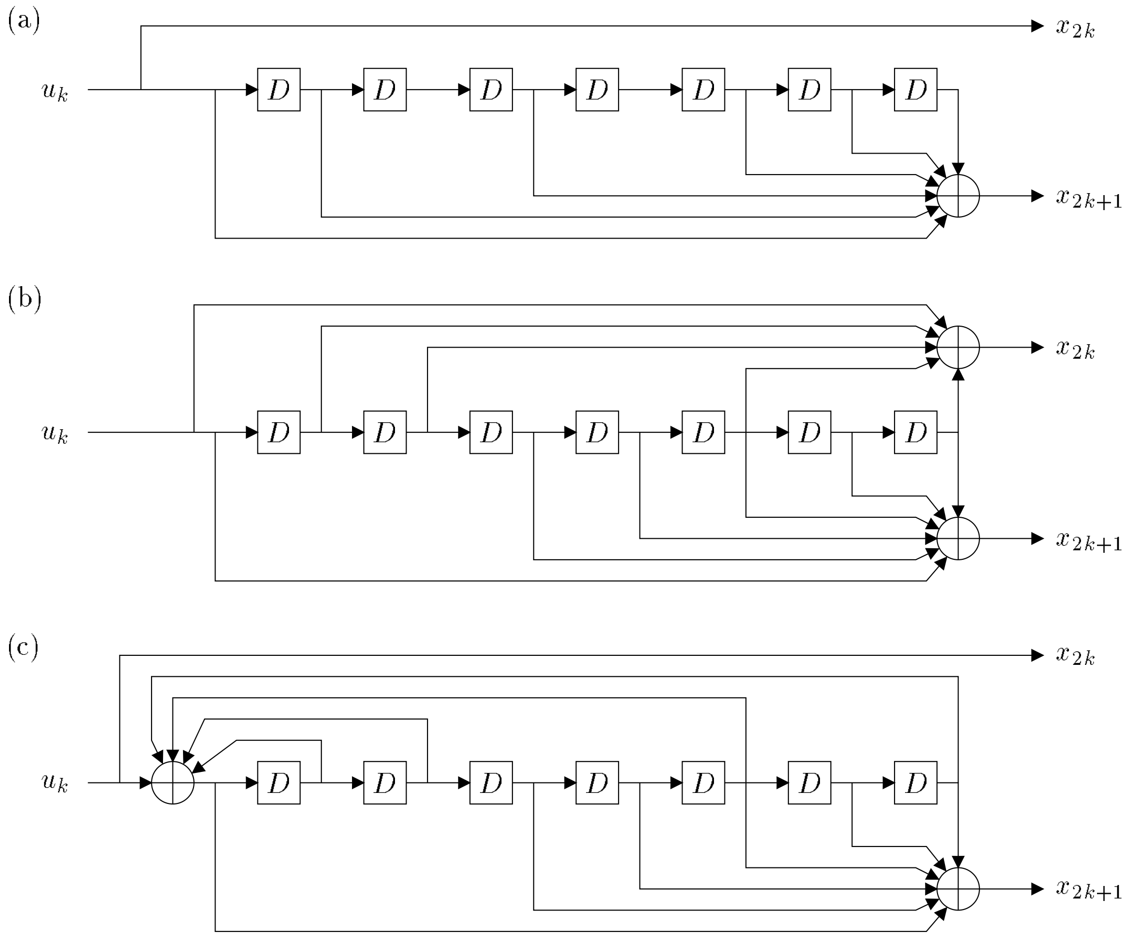

Figure 5.4: The linear feedback shift register (LFSR) configurations for rate 1/2 convolutional codes with maximum $d_{\min}$. (a) A systematic nonrecursive convolutional code ($d_{\min} = 7$). (b) A nonsystematic nonrecursive convolutional code ($d_{\min} = 10$). (c) A systematic recursive convolutional code ($d_{\min} = 10$).

The second equation is to be interpreted as

$$(1 + D + D^2 + D^5 + D^7)x_{2k+1} = (1 + D^3 + D^4 + D^5 + D^6 + D^7)u_k, \tag{5.23}$$

which can be derived from the figure[1]. The former expression allows us to retain the G(D) notation, which in this case is $247_8/371_8$. From the point of view of linear algebra, we have obtained this new code simply by dividing $G_1(D)$ and $G_2(D)$ from above by $G_1(D)$.

[1] In fact, this representation is algebraically consistent. We can, for example, multiply the numerator and the denominator in (5.22) by a polynomial in D without changing the set of output sequences that the LFSR can produce. See [Wicker 1995] for a textbook treatment.

It can be shown that this operation does not change the algebraic structure of the code. For example, the new code has $d_{\min} = 10$ as before. However, as we saw in the previous section, there is more to channel coding than algebraic structure. It turns out that this systematic recursive code performs better than the above nonsystematic nonrecursive code at low E_b/N_0.

5.2.3 Decoding convolutional codes by probability propagation

Bayesian networks for nonsystematic and systematic convolutional codes are shown in Figures 5.5a and 5.5d. In the former case, both codeword bits at stage k depend on the encoder state as well as the information bit, whereas in the latter case, one codeword bit is simply a direct copy of the information bit. Notice that because of the dependency of at least one codeword bit at stage k on the encoder state *and* the information bit, these networks are not singly-connected. However, they can be converted to singly-connected networks in the following way. By *duplicating* the information bits, we obtain the networks shown in Figures 5.5b and 5.5e (see Section 2.1.4). By *grouping* each state variable with one of these duplicates as shown by dashed loops, we obtain the singly-connected networks shown in Figures 5.5c and 5.5f (see Section 2.1.4).

In the new networks, each state variable actually contains a copy of the current information bit. We can interpret each state variable as a binary number whose least significant bit (LSB) is a copy of the current information bit and whose most significant bit (MSB) is the oldest value in the LFSR (*i.e.*, the value in the memory element that appears on the far right in the LFSRs shown in Figure 5.4). Let $s_k/2$ be the binary number obtained by cutting off the LSB of s_k, and let $s_k\%2$ be the value of the LSB of s_k. Let $f(s_{k-1})$ be the binary number obtained by cutting off the MSB of s_{k-1} and replacing the LSB of s_{k-1} with the value of the LFSR feedback bit obtained by adding (modulo 2) the values of the bits in $s_{k-1}/2$ that correspond to the LFSR feedback taps. So, $f(s_{k-1})$ is the value of the new state at stage k, *excluding* information bit u_k. Finally, let $g(s_k)$ be the bit obtained by adding (modulo 2) the values of the bits in s_k that correspond to the LFSR output taps. If there are two sets of taps, then there will be two output functions $g_1(s_k)$ and $g_2(s_k)$.

Now, we can specify the conditional probabilities for the convolutional code Bayesian networks. For the sake of brevity, I will consider only the systematic code shown in Figure 5.5f. Assuming the information bits are uniformly random, we have $P(u_k = 1) = P(u_k = 0) = 0.5$, $k = 0, \ldots, K-1$. The state transition probabilities are

$$P(s_k|s_{k-1}, u_k) = \delta(s_k/2, f(s_{k-1}))\delta(s_k\%2, u_k), \quad k = 0, \ldots, K-1, \tag{5.24}$$

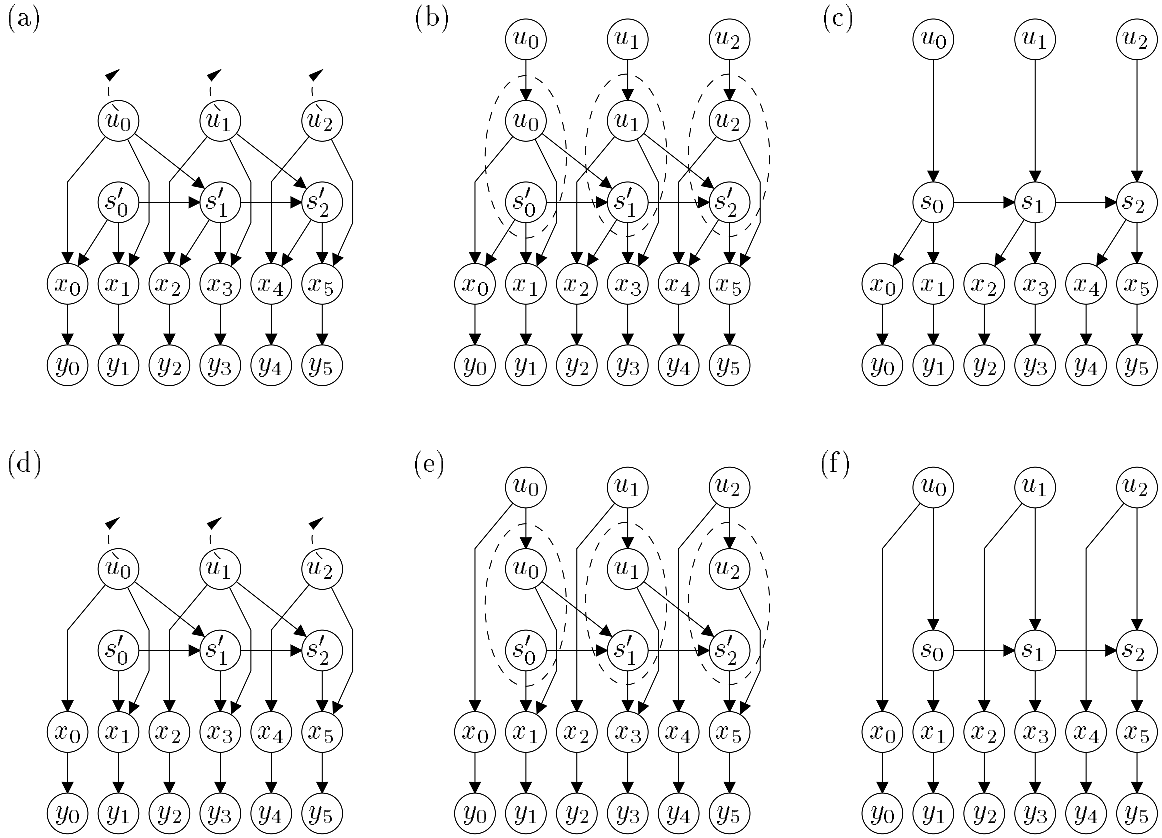

Figure 5.5: The multiply-connected Bayesian network (a) for a nonsystematic convolutional code can be converted to a singly-connected network by duplicating the information variables (b) and then grouping together information variables and state variables (c). The multiply-connected network for a systematic convolutional code can be converted to a singly connected one (d) — (f).

where we assume $s_{-1} = 0$ to initialize the chain. The codeword bit probabilities are

$$P(x_{2k}|u_k) = \delta(x_{2k}, u_k), \quad P(x_{2k+1}|s_k) = \delta(x_{2k+1}, g(s_k)), \quad k = 0, \dots, K-1. \tag{5.25}$$

Assuming binary antipodal signalling with power P over an AWGN channel, the conditional channel probabilities $p(y_i|x_i)$, $i = 0, \dots, 2K-1$ are given by (5.9), where σ^2 is related to E_b/N_0 by (5.11).

Using the singly-connected Bayesian networks for convolutional codes, probability propagation can be used to compute the *a posteriori* bit probabilities $P(u_k|\mathbf{y})$ exactly. The MAP values u_k^{MAP} can be obtained by applying a threshold of 0.5 to these probabilities. Although the probability messages can be passed in any order, the *forward-backward message-passing schedule* gives the lowest number of total messages passed, and so it is most appropriate for decoding on a serial machine. Figure 5.6 shows how messages are passed according to

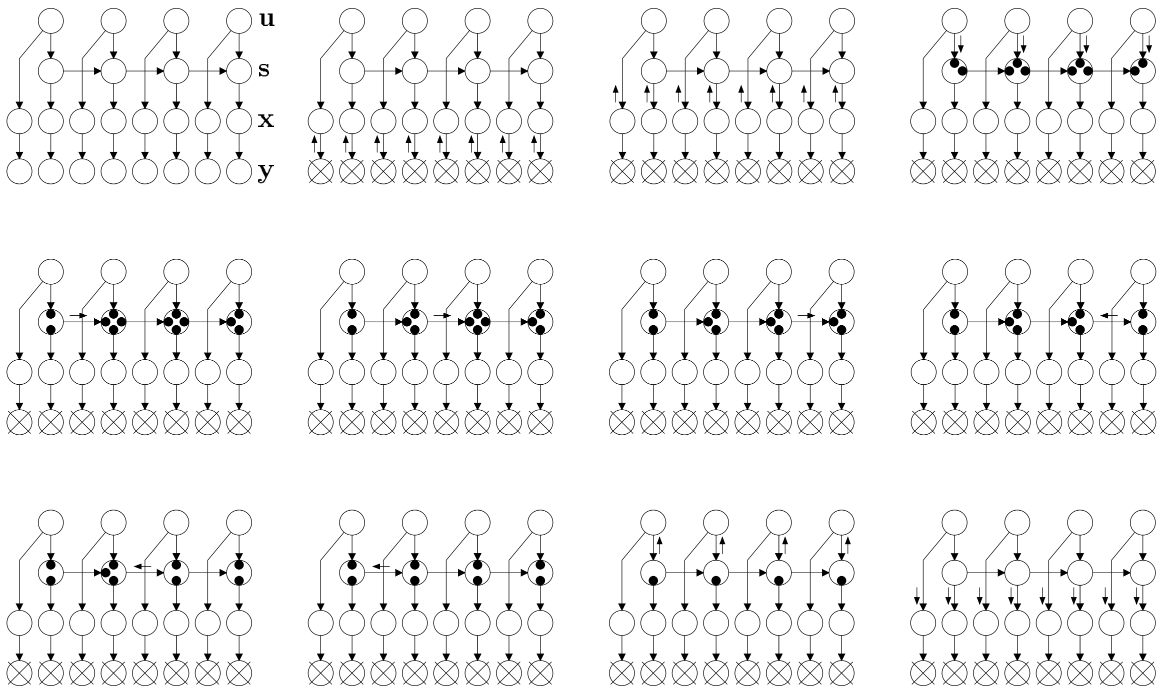

Figure 5.6: The computation of $P(u_k|\mathbf{y})$, $P(s_k|\mathbf{y})$, and $P(x_i|\mathbf{y})$ by probability propagation using the *forward-backward message-passing schedule*, which minimizes the total number of messages passed. Arrows represent messages in transit, whereas solid dots represent messages waiting to be sent.

this schedule in the Bayesian network for a simple systematic convolutional code. First, probability messages are propagated from the observed channel output variables (crossed vertices) to the "backbone" of the chain (the state variables). Then, the messages are buffered as shown. (See Section 2.1.3 for an explanation of buffered messages in probability propagation.) Pictorially, when a message arrives at a vertex on an edge, but is buffered and not propagated on to the other neighbors, I draw a small dot adjacent to each of the other edges. Each of these dots can be turned into an arrow (indicating a message is being passed) at any time. Next messages are passed forward along the chain, and then backward along the chain. Finally, messages are propagated to the information bits and to the codeword bits. (It is not necessary to propagate probabilities to the observed variables, since $P(y_i|\mathbf{y})$ is trivial to compute.) Notice that this algorithm computes $P(u_k|\mathbf{y})$, $P(s_k|\mathbf{y})$, *and* $P(x_i|\mathbf{y})$. If all we need are the information bit probabilities $P(u_k|\mathbf{y})$, then it is not necessary to propagate the last set of messages shown in the figure.

Figure 5.7 shows the performances of the three convolutional codes described above. The systematic nonrecursive convolutional code has a BER that is significantly higher than the BERs for the other two codes at reasonably high E_b/N_0. The nonsystematic nonrecursive convolutional code and the systematic recursive convolutional codes have similar BERs, except for low E_b/N_0, where the systematic code has a significantly lower BER. The software

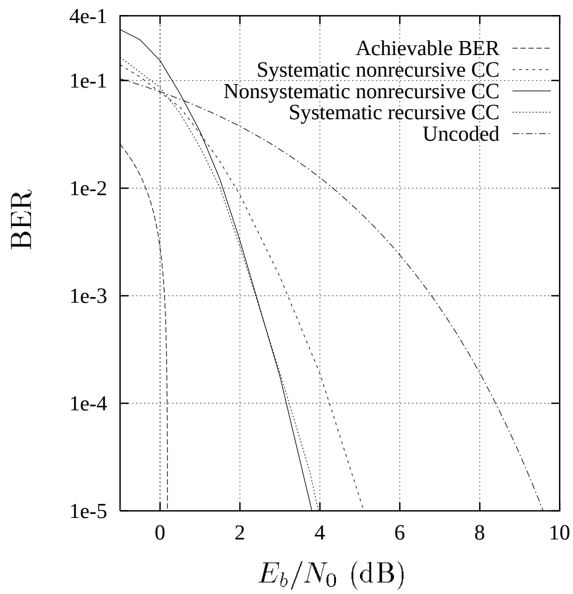

Figure 5.7: Performances of 7-bit memory LFSR convolutional codes with maximum $d_{\min}$.

package BNC was used to obtain these results.

5.2.4 Turbo-codes: parallel concatenated convolutional codes

Although the convolutional codes and decoder described above give roughly a 5.7 dB improvement over uncoded transmission at a BER of 10^{-5}, they are still roughly 3.7 dB from Shannon's limit at this BER. Up until the last few years, a serially-concatenated Reed-Solomon convolutional code [Lin and Costello 1983] was considered to be the state of the art. At a BER of 10^{-5}, this system is roughly 2.3 dB from Shannon's limit. However, in 1993, Berrou, Glavieux, and Thitimajshima introduced the *turbo-code* and the practical iterative *turbo-decoding* algorithm. Their system was roughly 0.5 dB from Shannon's limit at a BER of 10^{-5}. Also, these binary codes have been successfully combined with multi-level coding to obtain bandwidth-efficient coding within 0.7 dB of Shannon's limit [Wachsmann and Huber 1995].

The original presentation of turbo-codes lacked a principled framework. For example, it was not at all clear how decoding should procede when there were three or more constituent convolutional codes instead of two [Divsalar and Pollara 1995]. However, it turns out that the turbo-code can be concisely described as a multiply-connected Bayesian network, and that the turbo-decoding algorithm is just probability propagation in this network [Frey and Kschischang 1996; Kschischang and Frey 1997; MacKay, McEliece and Cheng 1997]. This general graphical model framework makes it easier to describe new codes and their

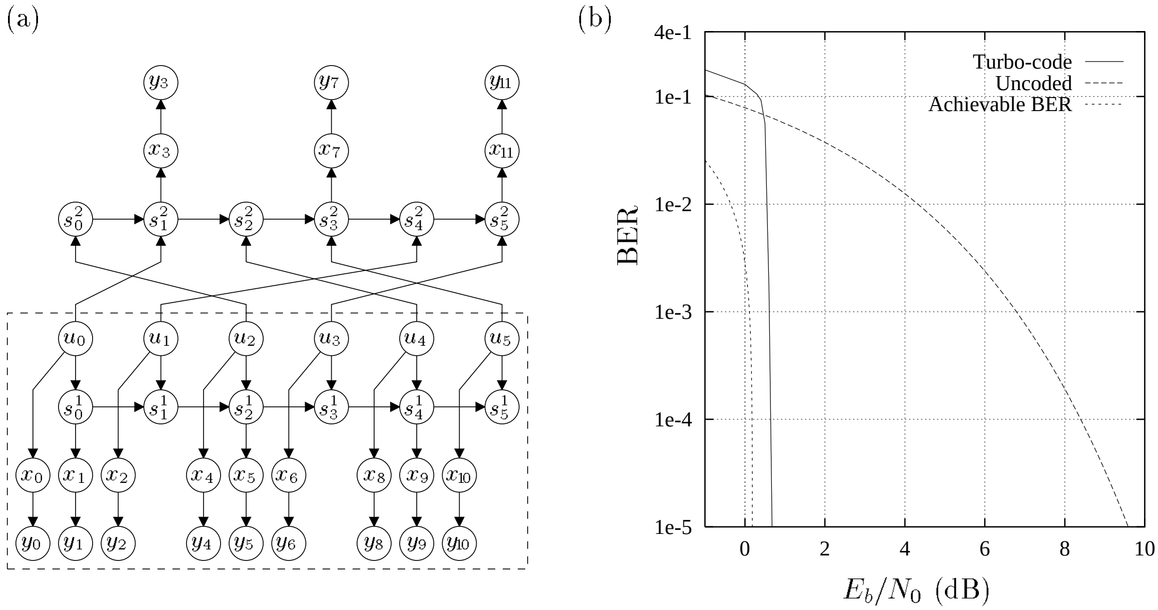

Figure 5.8: (a) The Bayesian network for a $K = 6$, $N = 12$ rate 1/2 turbo-code. (b) The performance of a $K = 65,536$ rate 1/2 turbo-code using 18 iterations of turbo-decoding.

corresponding iterative decoding algorithms. For example, decoding a turbo-code that has three constituent convolutional codes is just a matter of propagating probabilities in the corresponding Bayesian network.

Figure 5.8a shows the Bayesian network for a rate 1/2 turbo-code. For a given information vector, the codeword consists of the concatenation of two constituent convolutional codewords, each of which is based based on a different permutation in the order of the information bits. The subnetwork indicated by a dashed loop is essentially the same as the network for the systematic convolutional code described above. The only difference is that every second LFSR output is left off, for a reason given below. The information bits are also fed into the upper convolutional encoder, but in permuted order. Every second LFSR output of the upper code is also left off. By leaving off every second LFSR output in both constituent codes, the total number of codeword bits is twice the number of information bits, so the rate is 1/2.

Once the channel output $\mathbf{y}$ for an encoded information vector is observed, probability propagation can be used to approximate $P(u_k|\mathbf{y})$ and perform approximate MAP bit decoding. Figure 5.8b shows the performance of the probability propagation decoder for a $K = 65,536$ rate 1/2 turbo-code with a randomly drawn permuter. The scripts used with the BNC software package to obtain these results are given in Appendix B.3.

Each (identical) constituent convolutional code uses a 4-bit LFSR with polynomials $(21/37)_8$. Although at low E_b/N_0 the turbo-code gives a BER that is significantly higher

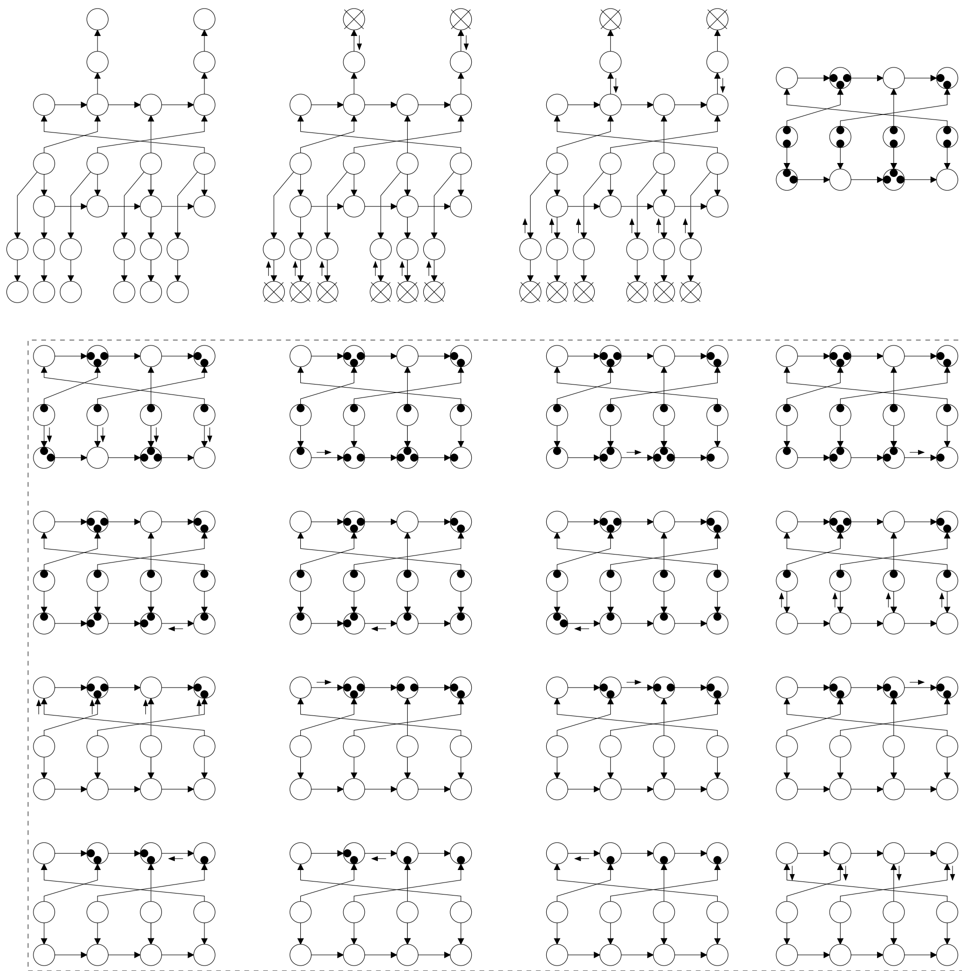

Figure 5.9: The message-passing schedule corresponding to the standard turbo-decoding algorithm.

than the BER for uncoded transmission, the turbo-code curve drops below a BER of 10^{-5} at less than 0.5 dB from Shannon's limit. Berrou and Glavieux suggest that for very low BER performance (say 10^{-10}), the permuter should be designed to maximize $d_{\min}$ [Berrou and Glavieux 1996]. I have found that for BERs at or above 10^{-5}, a randomly drawn permuter typically works fine.

Since the turbo-code network is multiply-connected, we must specify a message-passing schedule in order to decode by probability propagation. That is, the order in which messages are passed can affect the final result as well as the rate of convergence to a good decoding solution. Since the network is multiply-connected, we must also specify when to stop passing messages, since otherwise they would propagate indefinitely. Figure 5.9 shows how messages are passed up to the end of the first *iteration* of turbo-decoding. First, messages are passed from the channel output variables (crossed vertices) to the state variables of both constituent codes. Assuming we are only interested in estimating $P(u_k|\mathbf{y})$, we can now ignore the channel output variables and the codeword variables. The simplified network with buffered messages waiting to be sent is shown in the upper-right picture in Figure 5.9

Next, messages are passed from the information variables to the state variables of one of the constituent codes. This chain is processed in the forward-backward manner and then messages are propagated to the information variables. Messages are then passed to the state variables of the other constituent code. These messages are called "extrinsic information" in [Berrou and Glavieux 1996]. Once the second chain has been processed in the forward-backward manner, messages are propagated back to the information variables, as shown in the lower-right picture in Figure 5.9. This completes the first *iteration* of turbo-decoding. Messages are then propagated from the information variables back to the first constituent code chain, and so on. The series of 16 pictures outlined by a dashed rectangle in Figure 5.9 shows how messages are passed during one complete iteration of turbo-decoding. (Note that after the first iteration, there aren't any buffered messages in the first picture within the dashed rectangle. The buffered messages in this picture are due to the initial observations.)

5.2.5 Serially-concatenated convolutional codes, low-density parity-check codes, and product codes

It turns out that many of the iterative decoding algorithms for a variety of codes can be viewed as probability propagation in the corresponding Bayesian networks for the codes [Frey and Kschischang 1996]. Figure 5.10a shows the Bayesian network for a serially-concatenated convolutional code [Benedetto and Montorsi 1996b]. The information bits are first encoded using the upper convolutional code. The generated codeword bits $\mathbf{x}^1$ are then permuted and fed into a second convolutional encoder, whose output bits $\mathbf{x}^2$ are transmitted over the channel. The iterative decoding algorithm introduced in [Benedetto and Montorsi 1996a] was presented without reference to any of the literature on probability propagation. However, their iterative decoding algorithm is in fact probability propagation in the corresponding Bayesian network. After observing the channel output $\mathbf{y}$, the decoder propagates messages from $\mathbf{y}$ to the lower chain. Then, messages are propagated forward and backward along the lower chain before being passed to the upper chain. The upper chain is processed and then messages are passed back to the lower chain, and so on.

The theoretical ML-decoding upper bounds on BER-E_b/N_0 for serially-concatenated convolutional codes are superior to those for turbo-codes [Benedetto et al. 1997]. However, it is not clear that these theoretical bounds are of any practical value. First of all, the bounds are based on the average performance over all possible permuters. Suppose that on average 1 in every 1000 permuters gives a very poor code that when ML-decoded gives a BER of 0.1.

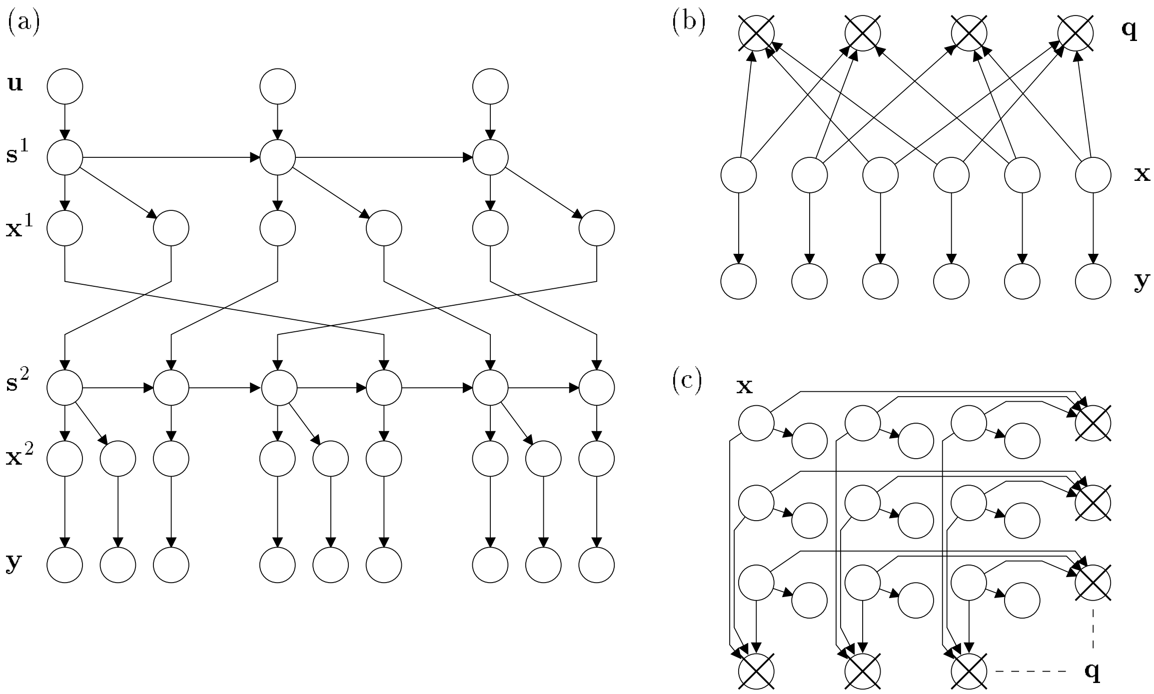

Figure 5.10: The Bayesian networks for (a) a $K = 3$, $N = 9$ rate 1/3 serially-concatenated convolutional code; (b) a(nonsystematic) $K = 2$, $N = 6$ rate 1/3 low-density parity-check code; and (c) a (nonsystematic) $K = 3$, $N = 9$ rate 1/3 product code.

Further, suppose that the other permuters give codes that when ML-decoded give BERs of 10^{-10}. If we randomly pick a permuter, we are very likely to get a code that gives a BER of 10^{-10}. However, the average performance over all permuters is $0.001 \cdot 0.1 + 0.999 \cdot 10^{-10} \approx 10^{-4}$. In this way, the average performance over permuters can be misleading. Second of all, since ML decoding is intractable, in practice we must use a suboptimal decoder, such as probability propagation. Even if the ML-curve for one code is superior to that of another code, the performance of the practical iterative decoder may be inferior.

It is suggested in [Benedetto et al. 1997] that for short block lengths (say, $K = 200$) serially-concatenated convolutional codes give better performance than turbo-codes, when iterative decoding is used. However, for short block lengths, it is not at all clear that either of these codes performs better than sequential decoding [Lin and Costello 1983] with a convolutional code with large memory.

Figure 5.10b shows the Bayesian network for a low-density parity-check code [Gallager 1963; Tanner 1981; MacKay and Neal 1996]. These codes were largely forgotten in the channel coding community for roughly 35 years, probably due to the computationally intensive encoder and decoder that Gallager proposed. However, it turns out that they have excellent theoretical performance [MacKay 1997] and that the iterative decoder proposed by Gallager is in fact equivalent to probability propagation in the network shown above. In these codes, each parity-check vertex q_i requires that the codeword bits $\{x_j\}_{j \in Q_i}$ to which q_i is connected have even parity:

$$P(q_i|\{x_j\}_{j\in Q_i}) = \delta\big(q_i, \bigoplus_{j\in Q_i} x_j\big), \tag{5.26}$$

where $\mathbf{q}$ is clamped (observed) to $\mathbf{0}$ to ensure even parity. The term "low-density" refers to the fact that each parity-check variable is connected to very few codeword bits (a vanishing fraction, as $N \to \infty$). (Notice that since this network is parity-check oriented and does not show how an information vector is mapped to a codeword, it appears an encoder must use a pre-derived generator matrix and encode the K information bits in $\mathcal{O}(K^2)$ time.) The iterative decoder passes messages between the parity-check variables and the codeword bit variables. Due to the simplicity of the codeword constraints (parity-checks), the decoder is simpler than the iterative decoder for turbo-codes. However, it appears they do not come as close to Shannon's limit as do turbo-codes for rates of 1/3 and 1/2 [MacKay and Neal 1996].

Figure 5.10c shows the Bayesian network for a product code. In this network, each variable q_i is a generalized parity-check variable — for example, q_i may require that $\{x_j\}_{j\in Q_i}$ be a codeword in a convolutional code. Recently proposed iterative decoders for product codes [Lodge et al. 1993; Hagenauer, Offer and Papke 1996] can be viewed as probability propagation in the corresponding networks. As with the low-density parity-check code, the decoder iteratively passes messages between the generalized parity-check variables and the codeword bit variables.

5.3 Interleaved trellis-constraint codes (ITC codes)

In the previous section, I presented the Bayesian networks for a variety of codes whose iterative decoding algorithms can be viewed as probability propagation in corresponding Bayesian networks. Can we use this perspective to propose new codes and derive new iterative decoders? Partly, the answer is "yes". However, we cannot expect to obtain good results simply by tossing the ingredients of a Bayesian network into a bag and shaking. First of all, we want the resulting code to give excellent performance if ML decoding is used. Second of all, we want the resulting code to give good results when decoded by probability propagation, which is only an approximation to maximum likelihood decoding. Keeping these issues in mind, a wise approach to proposing new code networks is to incrementally generalize previous work. In this section, I present a code that can be viewed as a trellis-based generalization of turbo-codes, serially-concatenated convolutional codes, low-density parity-check codes, and product codes.

5.3.1 Constraint codes

A binary (N, K) code is a set of 2^K codewords, that is a subset of a (usually much larger) set of 2^N binary vectors of length N. So, one way to view a code is as the set of N-vectors that satisfy a set of constraints. I will refer to a code that is described in this way as a *constraint code*. For example, any (N, K) linear binary code can be described by a set of $N - K$ linearly independent parity-check equations. A more complex example is an (N, K) binary convolutional code whose codewords are derived from the 2^K allowed configurations of the Markov chain that describes the code (*e.g.*, see Figure 5.6). This view of codes is similar to the systems approach of Wiberg [1996].

We can construct a Bayesian network that describes the parity-check equations for a code by creating one vertex q_i for each parity-check equation i, and one vertex x_j for each codeword bit. The parents of parity-check variable q_i are the codeword variables $\{x_j\}_{j \in Q_i}$ which appear in equation i. The conditional probability for parity-check variable i is

$$P(q_i|\{x_j\}_{j \in Q_i}) = \delta\big(q_i, \bigoplus_{j \in Q_i} x_j\big). \tag{5.27}$$

Finally, clamping $\mathbf{q} = \mathbf{0}$ defines the allowed configurations of the graphical model. If there are N codeword bits and $N - K$ parity-check variables whose parity-check equations are linearly independent, then the number of allowed configurations is 2^K. A code that is described in this way can be iteratively decoded by propagating probabilities back and forth between the set of parity-check vertices and the set of codeword bit vertices. (The Bayesian networks and iterative decoders for low-density parity-check codes and product codes fit into this framework — see Section 5.2.5.)

For example, Figure 5.11a shows the parity-check network for the simple turbo-code shown in Figure 5.8a. The channel output variables have been left out for the sake of graphical simplicity. As usual, the parity-check variables are clamped to 0. The 6 systematic codeword bits are in the center row of unclamped vertices. The two sets of 3 nonsystematic codeword bits are in the upper and lower rows of unclamped vertices. Notice that each parity-check vertex checks a single nonsystematic bit as well as all systematic bits to its left (up to the permutation). One way to decode this network is to propagate probabilities from the systematic bits to the upper row of parity-check bits, back down all the way to the lower row of parity-check bits, and so on. Notice that each systematic-parity-systematic sweep of propagation is not exact, since both the upper and lower subnetworks (obtained

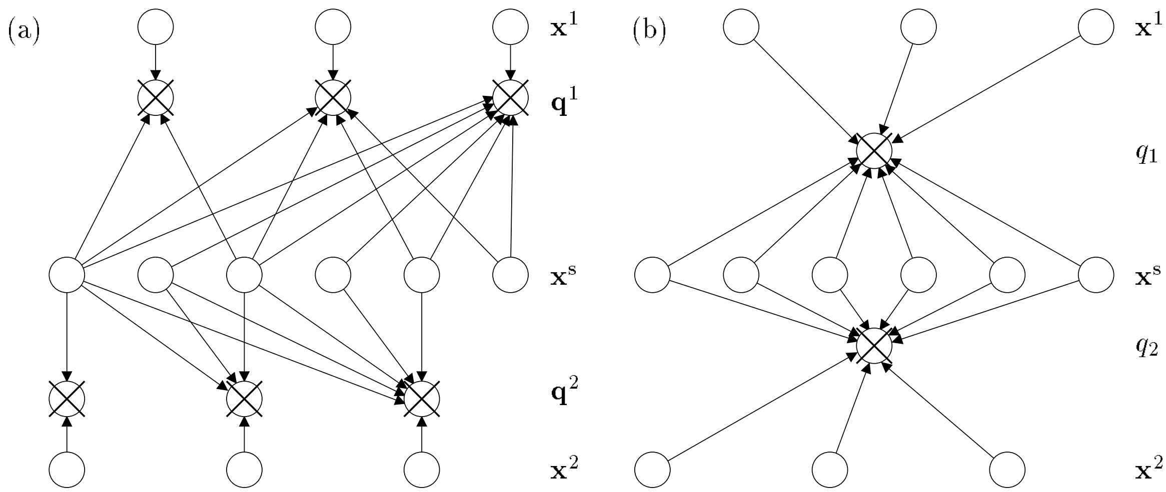

Figure 5.11: (a) The constraint network for the turbo-code shown in Figure 5.8a. (b) Each of the upper and lower subnetworks in (a) can be made singly-connected by grouping parity-check vertices.

by a horizontal cut across the systematic bits) are multiply-connected.

5.3.2 A code by any other network would not decode the same

Since any linear binary code can be described by a set of parity-check equations, it may seem that a fruitful approach to getting closer to capacity is to simply find a good code (*e.g.*, a random linear code), write down its parity-check equations, construct the corresponding Bayesian network, and then decode it using probability propagation. However, in general the parity-check network will be multiply-connected. Since probability propagation is only approximate in such networks, the performance of the decoder will depend heavily on which set of linearly independent equations is used. Operations such as grouping parity-check variables together (creating generalized check variables that have more than 1 degree of constraint), will also heavily influence the decoder's performance.

For example, after grouping each set of 6 parity-check vertices into one check vertex with 6 degrees of constraint, we obtain the network shown in Figure 5.11b. In this case, both the upper and lower subnetworks are singly-connected, so each systematic-parity-systematic sweep of propagation is exact. Obviously, iterative decoding in this network will give different results than iterative decoding in the original network. Notice that by grouping several parity-check variables, we obtained a check vertex with greater complexity than a single parity-check. In general, this will lead to a check vertex for which exact propagation is intractable. However, with judicious design, even a very high-order check vertex can still be processed in a tractable way. In the above example, each check vertex can be processed using the forward-backward algorithm.

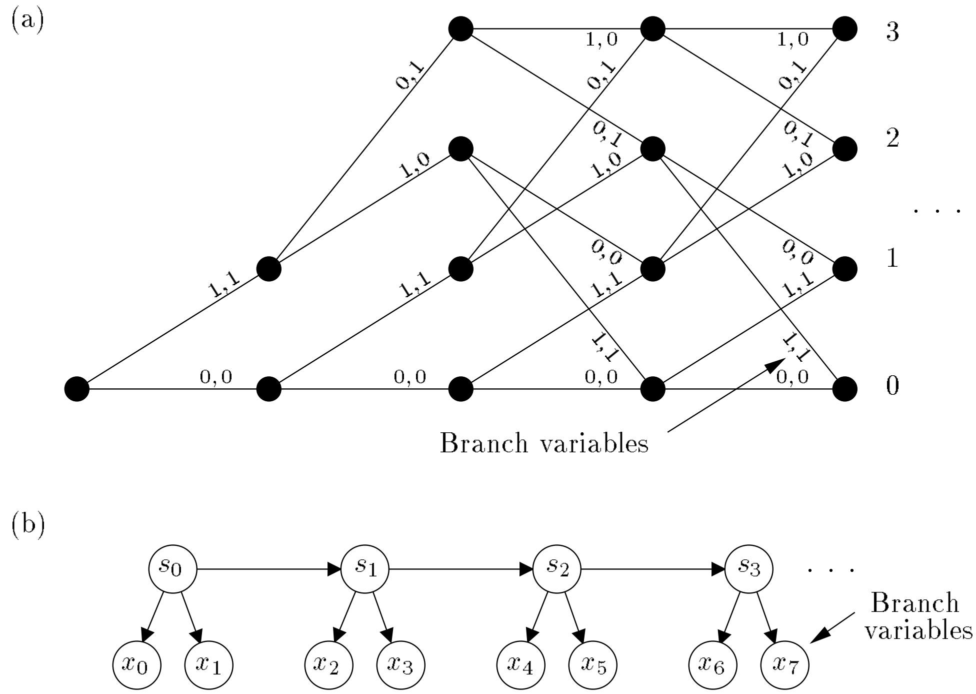

Figure 5.12: (a) shows the trellis for a simple rate 1/2 systematic recursive convolutional code. Each branch indicates an allowed state transition, and the corresponding pair of output bits are written beside the branch. (b) shows the corresponding Bayesian network, which requires one extra bit of state so that the outputs can be determined directly from the state variables.

In order to obtain a good coding *system*, we need to simultaneously find a good code and a corresponding Bayesian network that gives good performance when decoded by probability propagation.

5.3.3 Interleaved trellis-constraint codes

The term *trellis* was introduced by Forney [Forney 1973] and refers to a diagram that explicitly shows the values of a discrete state variable at each time step and the allowed state transitions. A trellis is more general than a LFSR, since in a trellis the state transitions and even the number of states may vary with time. (Also, a trellis can represent a nonlinear code.) Figure 5.12a shows the trellis for the first 4 time steps of a rate 1/2 systematic recursive convolutional code with LFSR polynomials $(5/7)_8$. *I.e.*, $x_{2k} = u_k$ and $x_{2k+1} = (1 + D^2)u_k/(1 + D + D^2)$. The levels of the state variable (black discs) corresponds to the memory of the LFSR, and in this case there are 2 bits of memory. Each branch in the trellis indicates an allowed state transition, and the corresponding branch variable values (in this case the LFSR outputs x_{2k}, x_{2k+1}) are written beside each branch. Figure 5.12b shows the corresponding Bayesian network. In the Bayesian network, the branch variables are functions of the state alone, and so each state variable must have 8 levels instead of the

4 levels used in the trellis.

An *interleaved trellis-constraint code* (ITC code) is a constraint code whose allowed configurations are defined by the interleaved interactions between the branch variables of two or more trellises. Because of the permuters, the branch variable interactions can lead to an ITC code whose equivalent single trellis is very complex, even if the constituent trellises in the ITC code are simple (*e.g.*, 16 states in the experiments below). The permuters (interleavers) may be structured or random, and there is no restriction on which branch variables are allowed to interact. For example, the systematic bits, the nonsystematic bits, or a mixture of both may interact with the other trellises. Also, there is no restriction on *which* variables are used as codeword symbols. For example, in an ITC code with two trellises, the codeword bits may be the nonsystematic bits of one trellis, the nonsystematic bits of both trellises, the systematic bits of one trellis and the nonsystematic bits of the other trellis, *etc.*

Figure 5.13a shows the Bayesian network for a general ITC code with n_t trellises and a vector of constraint satisfaction indicator variables $\mathbf{c}$. (Each double-track arrow represents parallel directed edges corresponding to the branch variables that participate in the constraints.) Let n_i be the number of branch variables participating in the ith constraint, and let the corresponding branch variables be $x_{i,1}, \ldots, x_{i,n_i}$. (*E.g.*, if each trellis contributes one branch variable to each constraint, we have $n_i = n_\mathrm{t}$ for all i, and $x_{i,j}$ is the branch variable that trellis j contributes to constraint i.) Let c_i be the constraint satisfaction indicator for the ith constraint. That is, $c_i = 1$ if and only if $\{x_{i,j}\}_{j=1}^{n_i}$ is a valid configuration for constraint i; otherwise, $c_i = 0$. For example, we may require that the labels participating in constraint i have even parity, be equal, or form the codeword of a short code. For the equality-constraint interaction, the conditional probability for constraint indicator i is

$$P(c_i|\{x_{i,j}\}_{j=1}^{n_i}) = \delta\big(c_i, \textstyle\prod_{j=1}^{n_i} x_{i,j} + \prod_{j=1}^{n_i}[1 - x_{i,j}]\big), \tag{5.28}$$

where $x_{i,j} \in \{0,1\}$. Notice that the conditional distribution for the constraint vector variable $\mathbf{c}$ factors:

$$P(\mathbf{c}|\{\{x_{i,j}\}_{j=1}^{n_i}\}_{i=1}^{n_\mathrm{c}}) = \prod_{i=1}^{n_\mathrm{c}} P(c_i|\{x_{i,j}\}_{j=1}^{n_i}). \tag{5.29}$$

(This could of course be shown graphically in the Bayesian network, but the figure would become much too cluttered.) The constraints are enforced by clamping $\mathbf{c} = \mathbf{1}$.

Viewed as a generalization of turbo-codes and serially-concatenated convolutional codes, ITC codes retain the graphical structure of two or more long chains that interact through

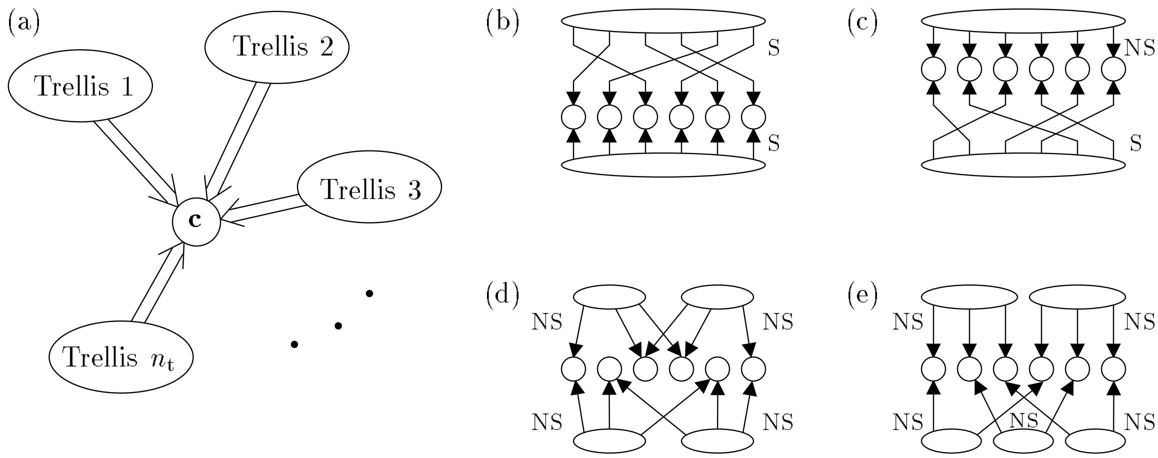

Figure 5.13: (a) The Bayesian network for a general interleaved trellis-constraint code (ITC code). The networks for ITC codes corresponding to turbo-codes, serially-concatenated codes, low-density parity-check codes, and product codes are shown in (b) — (e). "S" = systematic coupling; "NS" = nonsystematic coupling.

a permuter. As with other iterative decoders, I have found that the decoding complexity of the probability propagation decoder for ITC codes scales linearly with block length. However, the encoding complexity for an ITC code is not guaranteed to be linear, as it is for turbo-codes and serially-concatenated convolutional codes. Later in this section, I give an example of an ITC code whose BER-E_b/N_0 performance is competitive with a turbo-code's, but whose encoding time is superlinear (possibly quadratic) in the block length. However, it should be kept in mind that the encoder can use binary operations, whereas iterative decoders use floating point or fixed (integer) point operations. So, it is often the complexity of the iterative decoder that is most important for practical block lengths (*e.g.*, in broadcast applications, where an expensive encoder can be used, but the decoder must be highly affordable).

5.3.4 ITC codes with equality constraints

The equality constraint is the most severe constraint. If there are n_i branch variables participating in constraint i, then an equality constraint has $n_i - 1$ degrees of constraint. (I exclude constraints with n_i degrees of constraint from consideration, since they do not actually couple the trellises.)

Figures 5.13b to 5.13e show the ITC codes corresponding to a simple turbo-code, a serially-concatenated convolutional code, a low-density parity-check code, and a product code. Each elongated ellipse corresponds to a constraint trellis, and each horizontal row of vertices corresponds to the constraint vertices. Each group of edges leaving a trellis is labeled "NS" (nonsystematic) if the corresponding set of branch variables is constrained by the trellis. Each group of edges leaving a trellis is labeled "S" (systematic) if the corresponding set of branch variables is not constrained by the trellis (*i.e.*, the set of branch variables is a subset of a possible set of systematic branch variables). In all four cases, the constraints are equality constraints. The ITC code corresponding to a turbo-code consists of two or more trellises that have equal (up to an interleaving) systematic bits. The ITC code corresponding to a serially-concatenated convolutional code consists of two trellises, where the systematic bits of one trellis are equal to the permuted nonsystematic bits of another trellis. The ITC code corresponding to a low-density parity-check code consists of a large number of simple parity-check trellises, where each constraint ensures that one nonsystematic branch variable from each of a very small number of trellises are equal (two or three trellises are used in [MacKay and Neal 1996]). Interestingly, the standard iterative decoders for low-density parity-check codes [Gallager 1963; MacKay and Neal 1996] process the soft decisions for each parity-check equation by applying the forward-backward algorithm to a parity-check trellis. The ITC code corresponding to a product code consists of one parity-check trellis for each row and column of a rectangular arrangement of the constraints, where each constraint ensures that the nonsystematic branch variables from the "row trellis" and "column trellis" are equal.

The networks discussed above do not show which variables are used as codeword symbols. Although the graphical structure of an ITC code may be well-suited to decoding by probability propagation, the quality of the code will depend on which symbols are used as codeword symbols, among other things. For example, if the nonsystematic bits for only one of the turbo-code trellises are sent, the double-trellis ITC code degenerates into a single-trellis "ITC code" that is equivalent to a convolutional code.

5.3.5 Nonsystematic ITC codes

In this section, I present a new type of ITC code that fills a gap in the spectrum of ITC codes shown in Figure 5.13, and give results that show this new ITC code is competitive with turbo-codes and low-density parity-check codes. The ITC codes in Figure 5.13 vary in both their number of constituent trellises and in which type of branch variables (systematic or nonsystematic) they couple. ITC codes of the type shown in Figures 5.13b and 5.13c have a small number of very long trellises, whereas ITC codes of the type shown in Figures 5.13d and 5.13e have a large number of relatively short trellises. In contrast, ITC codes of the type shown in Figure 5.13b employ a systematic-systematic coupling, whereas ITC codes of the type shown in Figure 5.13c employ a systematic-*nonsystematic* coupling. Figure 5.14a shows a new type of ITC code that has very long trellises and employs a *nonsystematic-nonsystematic* coupling. I will refer to this type of ITC code as a *nonsystematic ITC*

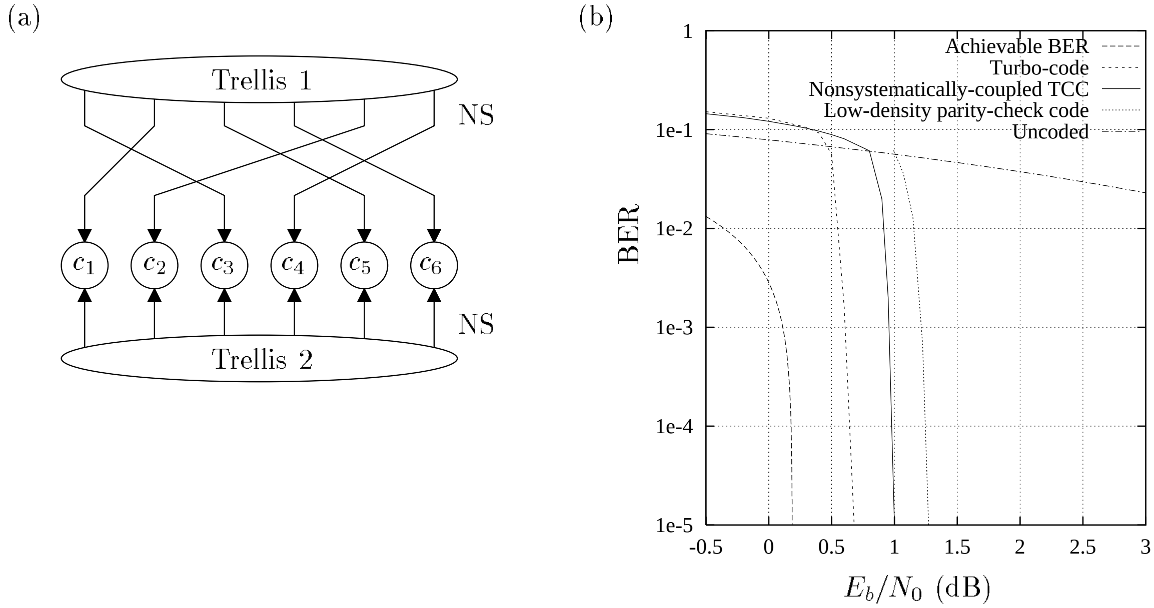

Figure 5.14: (a) The Bayesian network for a new ITC code, called a *nonsystematic ITC code*. (b) The performance of a $K = 65,536$, $N = 131,072$ rate 1/2 nonsystematic ITC code compared to a standard turbo-code and a low-density parity-check code.

code, in order to emphasize how it differs from the ITC codes for turbo-codes and serially-concatenated convolutional codes.

Consider an ITC code of this type, where the nonsystematic branch variables (which are constrained to be the equal) are transmitted as codeword bits. Let N be the number of codeword bits and let R_j be the rate of trellis j (*i.e.*, trellis j imposes $N(1-R_j)$ degrees of constraint on the codeword). Assuming that the constraints for all n_t trellises are linearly independent, the degrees of freedom left over after all trellises are coupled is $K = N - \sum_{j=1}^{n_\mathrm{t}} N(1-R_j)$. So, the overall rate of the code is

$$R \geq 1 - \sum_{j=1}^{n_\mathrm{t}} (1 - R_j). \tag{5.30}$$

For each constraint that is linearly *dependent* on the other constraints, R is *increased* by $1/N$.

Figure 5.14b shows the performance of a $K = 65,536$, $N = 131,072$ rate 1/2 nonsystematic ITC code, with two trellises ($N_\mathrm{t} = 2$). Each trellis was obtained by puncturing every fifth nonsystematic bit of a rate 4/5 nonsystematic convolutional code with maximum d_{min}. (The generators for this code were obtained from [Daut, Modestino and Wismer 1982] and are $(32, 4, 22, 15, 17)_8$.) After puncturing, each convolutional code had rate 3/4, so that the overall rate of the ITC code was 1/2. The BNC software package was used to obtain these results. Although this nonsystematic ITC code does not perform as well as a turbo-code with the same K and N, it does perform significantly better than the best rate 1/2 low-density parity-check code published to date [MacKay and Neal 1996] with $K = 32,671$ and $N = 65,389$. (I have observed that for long block lengths ($N > 50,000$), the only significant effect that increasing the block length has is to steepen the slope of the BER-E_b/N_0 curve to the *right* of the point of high curvature.)

5.4 Speeding up iterative decoding by early-detection

The excellent bit error rate performance of iterative probability propagation decoders is achieved at the expense of a computationally burdensome decoding procedure. In this section, I present a method called *early-detection* that can be used to reduce the computational complexity of a variety of iterative decoders. Using a confidence criterion, some information symbols, state variables and codeword symbols are detected early on in the iterative decoding procedure. In this way, the complexity of further processing is reduced with a controllable increase in BER. I present an easily implemented instance of this algorithm, called *trellis splicing*, that can be used with turbo-decoding. For a simulated system of this type, I obtain a reduction in computational complexity of up to a factor of four, relative to a turbo-decoder that performs the fewest iterations needed to achieve the same BER.

5.4.1 Early-detection

One way to view early-detection is as a refinement of a block-oriented stopping criterion used to terminate the iterative process in iterative decoders. For example, Hagenauer *et al.* [Hagenauer, Offer and Papke 1996] proposed monitoring the relative entropy between the set of soft information bit decisions for the current iteration and the previous iteration. When the change in this relative entropy falls below some threshold, the iterative decoding process is terminated. The basic idea is that iterative decoding should stop when the decoder's soft decisions are stable. Block-oriented stopping criteria lead to iterative decoders that are more efficient than fixed-complexity iterative decoders, since the stopping criteria effectively allows the decoder to spend more iterations on "tough" blocks, and fewer iterations on "easy" blocks.

Taking this reasoning one step further, I believe that in some cases, *parts of the codeword* may be more easily decoded than other parts. Although different parts of a codeword are usually inter-dependent, for particular noise patterns the coupling between parts may be weak. In these cases, it makes sense that the decoder should spend more computations

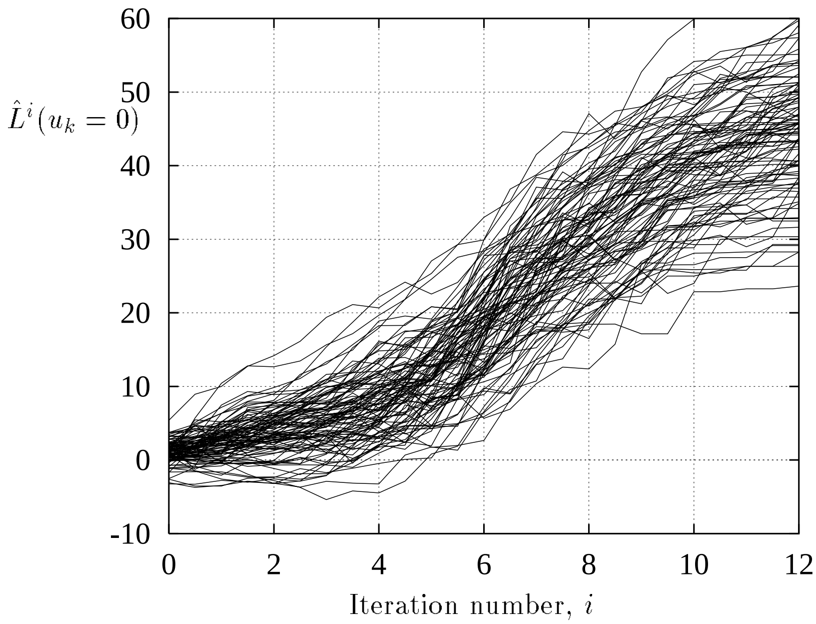

Figure 5.15: A plot of the log-odds ratio versus iteration number, for the correct value of each information bit in a randomly selected set of 100 bits *within the same block* of 10 000 bits.

on "tough" parts, and fewer computations on "easy" parts. During decoding, those parts that are deemed to be successfully decoded are clamped. Decoding computations are then focussed on the remaining parts.

For example, Figure 5.15 shows how the soft decisions for a randomly selected subset of information bits $\{u_k\}$ *within the same block* evolve during iterative decoding of a turbo-code. The all-zero codeword was transmitted, so a large positive value of $\hat{L}^i(u_k = 0)$ indicates that the decoder is quite confident of the value for u_k, *and* that this value is correct. Large negative values (none shown) of $\hat{L}^i(u_k = 0)$ indicate that the decoder is quite confident of the value for u_k, and that this value is *wrong*.

These curves were produced by simulating the transmission of a single binary block over a 0.2 dB AWGN channel, for a rate 1/3 unpunctured turbo-code that had 10 000 information bits, identical constituent encoders $(21, 37)_{\text{octal}}$, and a randomly drawn permuter. Clearly, the decoder is correctly confident of many information bits long before it has sorted out the values of other information bits. By detecting some of the well-determined bits early, computations can be refocussed on decoding the less well-determined bits.

The notion of revisiting a decoding operation after "pinning" some of the variables has been used before to *improve BER performance*. The most common application is for decoding the serial concatenation of a Reed-Solomon outer code with a convolutional inner code. For practical purposes, the Reed-Solomon decoder either outputs an error-free codeword segment or flags the segment as a decoding failure. After the convolutional code has been decoded and its output decoded by the Reed-Solomon decoder, the codeword segments that are practically known to be error-free can be fed back to the convolutional decoder and used to pin certain trellis states for a second round of improved decoding. By using this approach, substantial coding gains have been reported by Lee [Lee 1977], Collins [Collins 1993], and Hagenauer *et al.* [Hagenauer, Offer and Papke 1993].

The present application of "pinning", called *early-detection*, is meant to decrease the *computational complexity* of decoding, but not improve BER performance or improve coding gain. For example, turbo-codes do not have component decoders that can flag decoding failures, so there is no way to be practically certain that an early-detected variable is correct. When applied to some types of iterative decoders such as turbo-decoders, early-detection actually worsens the BER performance. However, if the main concern in a system is the computational complexity of the decoder, early-detection can be used to reduce the complexity of an iterative decoder in a way that leads to a smaller increase in BER compared to other techniques, such as performing fewer decoding iterations.

5.4.2 Early-detection criteria

As discussed in the next section, the computation time of an iteration decreases with the number of early-detected variables. So, in order to obtain the greatest speed-up, the decoder should early-detect as many variables as possible. However, an overly aggressive early-detection criterion will lead to a high rate of *erroneous* decisions, spoiling the BER performance. In addition to this constraint, the early-detection criterion should be relatively simple, so that the overhead of ascertaining which variables ought to be early-detected does not overshadow the reduction in the computational complexity of subsequent iterative decoding. In this section, I explore criteria that use the soft decision reliabilities in order to ascertain whether or not an early-detection should occur.

The soft decisions used for iterative decoding can be represented as log-odds ratios that approximate the true *a posteriori* log-odds ratios. The log-odds ratio for an information symbol, state variable, or codeword symbol z at iteration i given the channel output $\mathbf{y}$ is

$$\hat{L}^i(z = z') = \log \frac{\hat{P}^i(z = z'|\mathbf{y})}{\hat{P}^i(z \neq z'|\mathbf{y})}, \tag{5.31}$$

where $\hat{P}^i(z|\mathbf{y})$ is the approximation to the *a posteriori* distribution $P(z|\mathbf{y})$ produced at iteration i. I will let i be fractional when the meaning is clear. For example, in a turbo-

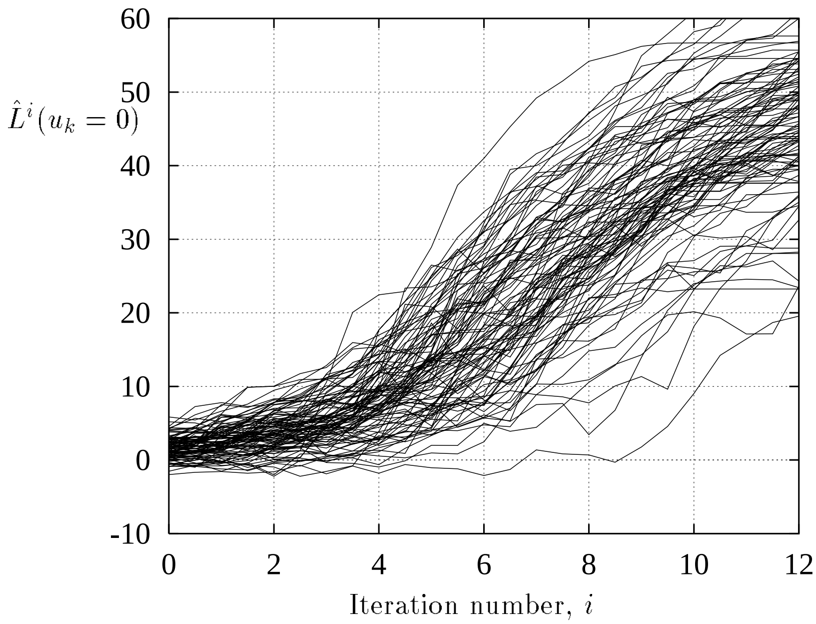

Figure 5.16: A plot of the log-odds ratio versus iteration number, for the correct value of a randomly positioned information bit in each of 100 decoded turbo-code blocks.

decoder with two constituent codes, $i = 0.5$ refers to quantities produced by processing the first constituent code for the first time.

In order to determine an appropriate early-detection criterion, I simulated the transmission of 100 blocks from a rate 1/3 unpunctured turbo-code that had 10 000 information bits, identical constituent encoders $(21, 37)_{\mathrm{octal}}$, and a randomly drawn permuter. I used binary signalling over an additive white Gaussian noise (AWGN) channel with $E_b/N_0 = 0.2$ dB. To speed up decoding, our forward-backward algorithm was implemented using a linear interpolation approximation to the function $\log(1 + \exp(\cdot))$. Also, our decoder did not weight the "extrinsic information" by the reliability variances as was originally suggested by Berrou *et al.* [Berrou and Glavieux 1996]. (I found that this weighting operation is not necessary at BER greater than 10^{-6}.) Figure 5.16 shows a plot of the log-odds ratio versus iteration number for the correct value of a randomly positioned information bit in each of the 100 blocks. In contrast to Figure 5.15, this figure shows the diversity of log-odds ratio convergence rates between blocks.

It appears from Figure 5.16 that the only simple criterion that a decoder can use without introducing too many early-detection errors is a simple threshold. Higher order criteria, such as the change in $\hat{L}(u_k)$, would produce too many erroneous early-detections. Although the relative entropy from one iteration to the next was successfully used in [Hagenauer, Offer

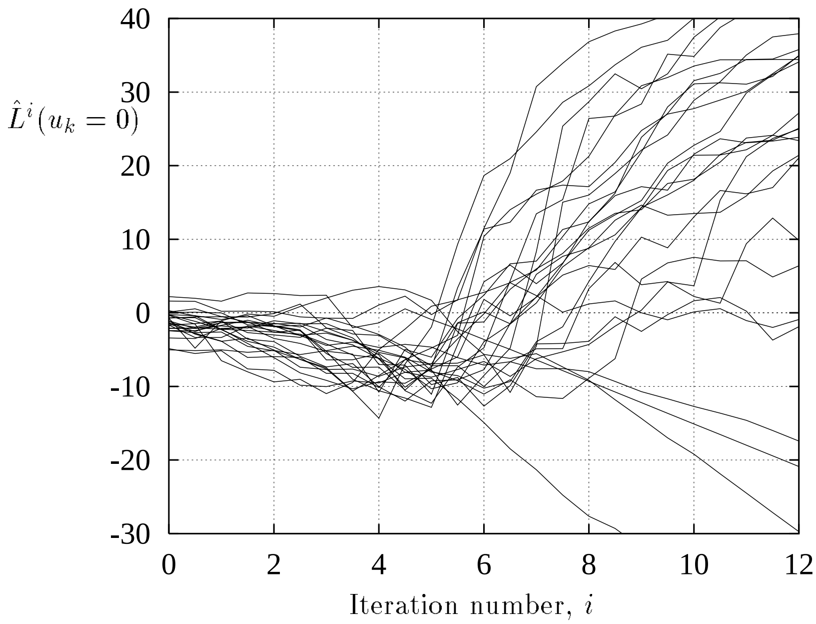

Figure 5.17: A plot of the log-odds ratio versus iteration number, for the correct value of 25 information bits (from different blocks) for which the log-odds ratio dropped below -10.0 during decoding.

and Papke 1996] as a block-oriented termination criterion, the same rule would not work at the more refined symbol-oriented level of early-detection. The decoder remains undecided on some variables for many iterations (up to $i = 8.5$ for one curve in Figure 5.16), and consequently $\hat{L}_{(u_k)}$ does not change much for those variables. However, eventually the decoder finds a consistent codeword segment and then the log-odds ratios for the related information bits change drastically.

For the turbo-code system described above, Figure 5.17 shows 25 randomly selected cases (each from a different block) for which the log-odds ratios drop *below* -10.0 during decoding. These traces show that the decoder can become incorrectly confident of the value of an information bit, but then with further iterations become correctly confident. By using a threshold of 15.0 for early-detection, all of the bits that the decoder correctly decodes as $i \to \infty$ can be detected early and correctly (the four curves that are between -5.0 and 10.0 at $i = 12$ eventually rise above 15.0). On the other hand, if the change in the log-odds ratio is used for early-detection, many of the bits that the decoder correctly decodes as $i \to \infty$ would be incorrectly detected early at the values for i where the curves stop falling and begin to rise. That is, the change in $\hat{L}^i(u_k)$ is close to zero at the iteration where the decoder begins to *correct* the bit. Higher order criteria may actually help in this case (by detecting that a curve is turning around), but it appears the data is too noisy for this approach to be successful. Also, higher-order criteria increase the computational overhead of early-detection.

5.4.3 Reduction in decoding time due to early-detection

The Bayesian networks for a variety of codes are shown in the first column of pictures in Figure 5.18. (The channel output variables are not shown — their likelihoods are to be included as "bias" effects on the state variables, codeword bits, and information bits (where applicable) during decoding.)

Let $|P(z_i|\mathbf{a}_i)|$ be the number of configurations of a discrete variable z_i and its discrete parents $\mathbf{a}_i$ for which $P(z_i|\mathbf{a}_i) \neq 0$, and let $|\mathbf{a}_i|$ be the number of parents for z_i. (If z_i has no parents, let $|\mathbf{a}_i| = 1$.) In general, the time needed for an iteration of iterative decoding scales as

$$\chi = \sum_{i=1}^{N} |P(z_i|\mathbf{a}_i)||\mathbf{a}_i|^2. \tag{5.32}$$

For example, if the constituent convolutional code for a turbo-code has memory ν, then $|P(s_i^1|s_{i-1}^1, u_i)| = 2^{\nu+1}$ and the state variable s_i^1 contributes a complexity of $|P(s_i^1|s_{i-1}^1, u_i)| \cdot 2^2 = 4 \cdot 2^{\nu+1}$. A notable exception to the above formula is the time needed to process one set of parents for a parity check in a low-density parity-check code. In this case, $|P(q_i|\{x_j\}_{j\in Q_i})|$ is exponential in the number of parents (see Section 5.2.5). However, the time needed to process each such parity-check vertex q_i is *linear* in the number of parents.

An early-detection can reduce the computational complexity given in (5.32) both directly and indirectly. The first three pictures in Figure 5.18a show how the early-detection of information bit u_2 directly simplifies the Bayesian network, thereby decreasing χ. The modified sum in (5.32) no longer includes the term $|P(u_2)| \cdot 1$ $(= 2)$ for u_2, and in each of the terms for the children s_2^1 and s_3^2 of u_2, the number of configurations is reduced by a *factor* of 2 and the number of parents is decreased by 1. In the case of the turbo-code, the former reduction decreases the complexity contributed by s_i^1 from $4 \cdot 2^{\nu+1}$ to 2^ν.

The indirect effect of detecting u_2 early is shown by the fourth picture in Figure 5.18a. Since the objective of the decoder is to make decisions for the information bits, the two states s_2^1 and s_3^2 can actually be removed from the network. Suppose $\hat{u}_2$ is the early-detected

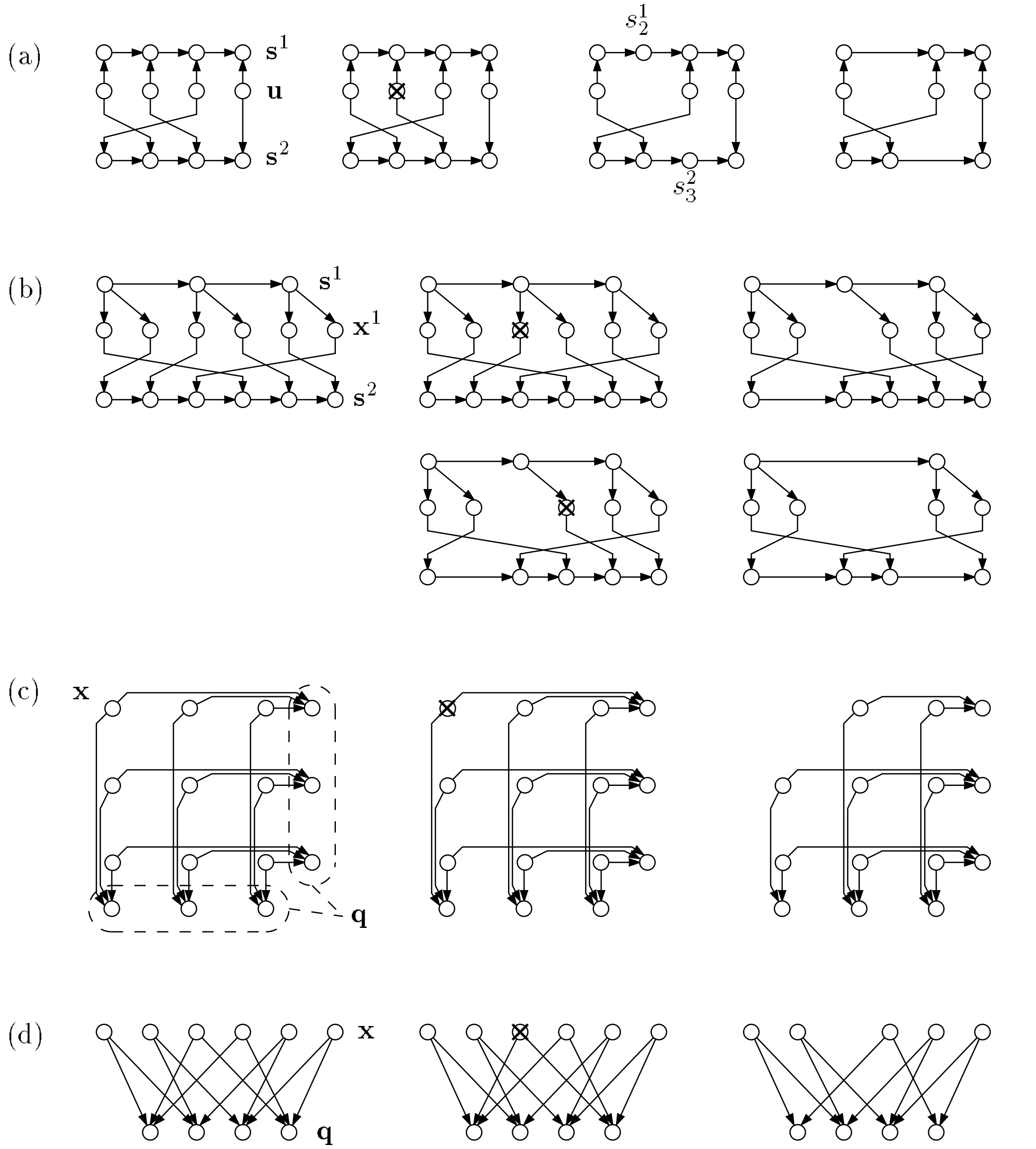

Figure 5.18: Early-detection effectively modifies the Bayesian network for (a) a turbo-code, (b) a serially-concatenated convolutional code, (c) a product code, (d) a low-density parity-check code.

value of u_2. Then, the new conditional distributions for s_3^1, after s_2^1 has been removed, is

$$\begin{aligned} P'(s_3^1|s_1^1, u_3) &= P(s_3^1|s_1^1, u_3, \hat{u}_2, y_2^1) \\ &= N \sum_{s_2^1} P(s_3^1|s_2^1, u_3) P(s_2^1|s_1^1, \hat{u}_2) p(y_2^1|s_2^1), \end{aligned} \tag{5.33}$$

where N is a normalization operator, which ensures that $\sum_{s_3^1} P'(s_3^1|s_1^1, u_3) = 1 \quad \forall s_1^1, u_3$. Notice that each of the terms in this sum includes a channel likelihood. The computation of these new conditional probabilities is actually performed as a normal part of iterative decoding. So, in practice all that is needed is a small integer lookup table to relate the configurations of s_1^1 and u_3 to the proper values of s_3^1.

Figs. 5.18b to 5.18d show how the networks for other compound codes are simplified by detecting variables. In the serially-concatenated convolutional code, detecting information bits (not shown) early leads to relatively little reduction in χ. Instead, the intermediate codeword bits can be early-detected to obtain a significant reduction in the complexity of decoding. Notice that only one trellis is simplified by a single early-detection. Each section of the upper trellis requires that both its outputs be early-detected, as shown by the lower two pictures in Figure 5.18b. For the product code and the low-density parity-check code, detecting codeword bits early simplifies the relevant constituent parity check equations.

5.4.4 Early-Detection for Turbo-Codes: Trellis Splicing

In this section, I illustrate how early-detection applied to turbo-codes can be used to reduce the overall decoding complexity. For turbo-codes, the Bayesian network consists of two or more chains that are processed using a special case of the probability propagation algorithm, called the forward-backward (a.k.a. "BCJR") algorithm [Baum and Petrie 1966; Bahl et al. 1974]. This algorithm computes the *a posteriori* information bit probabilities using the channel output and *a priori* information bit probabilities. The forward-backward algorithm can be viewed simply as a combination of probabilistic "flows" [McEliece 1996] computed in the forward direction and in the backward direction. Alternatively, a soft-output Viterbi algorithm (SOVA) [Hagenauer, Offer and Papke 1996] can be used. Here, I consider early-detection for information symbols only. As discussed earlier, early-detection of a single information symbol reduces the complexity of both constituent codes.

Consider the simple two-state trellis shown in Figure 5.19a. Let u_k be the random variable for the information bit in the kth section of the trellis, and let s_k be the random variable for the state at the beginning of the kth section of the trellis. The edge in the kth section of the trellis that leaves state $s_k \in \{0,1\}$ in response to information bit $u_k \in \{0,1\}$ has an associated branch metric, $\gamma_k^{u_k}(s_k)$. These metrics are determined from the received signals and the *a priori* probabilities regarding the transmitted information bit values. (In a systematic code, the likelihoods for the noisy received information bits can be included in the *a priori* probabilities.) If $p(y_k|u_k, s_k)$ is the likelihood function for the kth received signal and $P(u_k)$ is the *a priori* probability for information bit u_k, then $\gamma_k^{u_k}(s_k) = P(u_k)p(y_k|u_k, s_k)$. The forward pass consists of computing the flows from these metrics in the forward direction. This results in a flow value $\alpha_k(s_k)$ for each state s_k at each section $k, k = 0 \ldots K-1$, computed as $\alpha_{k+1}(0) = \gamma_k^0(0)\alpha_k(0) + \gamma_k^1(1)\alpha_k(1)$, and $\alpha_{k+1}(1) =$

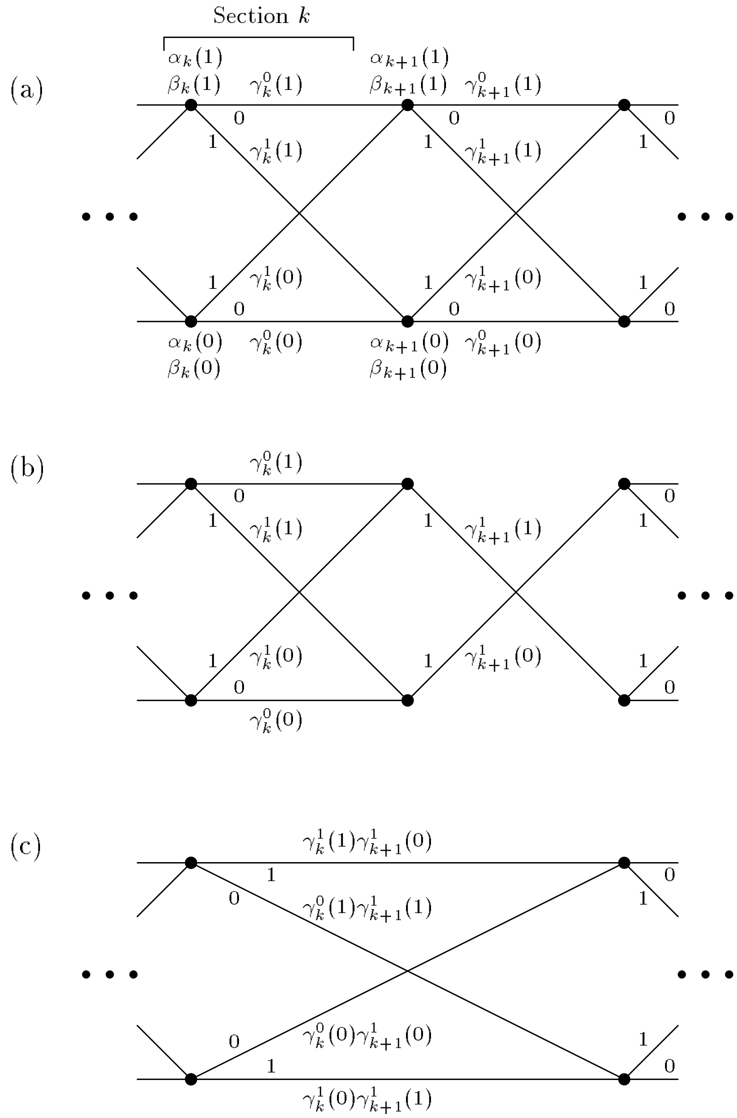

Figure 5.19: Trellis splicing. (a) shows a two-state trellis with edges accompanied by information bit labels and metrics and with nodes accompanied by flows. (b) and (c): If we know that information bit $k+1$ has a value of 1, we can cut the corresponding section out of the trellis and splice the trellis back together, introducing new information bit labels and new metrics for the connecting edges.

$\gamma_k^1(0)\alpha_k(0) + \gamma_k^0(1)\alpha_k(1)$. The backward pass simply consists of a flow computation in the reverse direction in order to obtain a flow value $\beta_k(s_k)$ for each state at each section: $\beta_k(0) = \gamma_k^0(0)\beta_{k+1}(0) + \gamma_k^1(0)\beta_{k+1}(1)$, and $\beta_k(1) = \gamma_k^1(1)\beta_{k+1}(0) + \gamma_k^0(1)\beta_{k+1}(1)$. These two types of flow are combined to obtain the *a posteriori* log-odds ratio that each information bit is 1 versus 0, given the received signal sequence $\mathbf{y}$:

$$\log \frac{P(u_k = 1|\mathbf{y})}{P(u_k = 0|\mathbf{y})} = \log \frac{\alpha_k(0)\gamma_k^1(0)\beta_{k+1}(1) + \alpha_k(1)\gamma_k^1(1)\beta_{k+1}(0)}{\alpha_k(0)\gamma_k^0(0)\beta_{k+1}(0) + \alpha_k(1)\gamma_k^0(1)\beta_{k+1}(1)}. \tag{5.34}$$

The computational cost of each section in the forward-backward algorithm thus consists of the time spent computing the α's and β's for each state, as well as the time spent computing the *a posteriori* log-odds ratios. Although there are various useful techniques and approximations for decreasing this cost [Hagenauer, Offer and Papke 1996; Benedetto et al. 1996], such as the SOVA [Hagenauer, Offer and Papke 1996], I will define it as a basic computational unit, and refer to it as a trellis *section operation.*

Suppose that according to some early-detection criterion, we decide that the value of information bit u_{k+1} is 1. (Here, I will consider early-detection for information bits only.) As a consequence, the trellis simplifies to the one shown in Figure 5.19b. The trellis can be simplified further by multiplying out the path metrics, giving the trellis shown in Figure 5.19c. Note that not only have the path metrics changed, but also the transitions now correspond to different information bit values. In general, portions of the trellis corresponding to early-detected information bits can be cut away, and the remaining segments spliced together with new path metrics and new information bit edge labels. If the values of b information bits are known, the spliced trellis will be b sections shorter, leading to a computational savings of b section operations for each future forward-backward sweep.

In order to implement trellis splicing, an integer array must be used to determine the state transitions, $(s_k, u_k) \longrightarrow s_{k+1}$. Whereas in the original trellis this mapping is very regular, after trellis splicing it is usually not. (*E.g.*, the information bits associated with the outgoing edges of the kth state in Figure 5.19c have *opposite* values compared to those in Figure 5.19a.) The use of this array slightly increases the computational complexity of each section operation. Also, the array must be modified each time a section is cut away. However, both of these computational costs are insignificant compared to the cost of the basic section operation. In the implementation of trellis splicing used for the experiments presented in Section 5.4.5, I found that the percentage of cpu time spent on trellis splicing was less than 6%. The integer array also requires extra memory. However, the total memory used actually *decreases* during decoding while using trellis splicing. When a single section is cut away, the memory liberated by the elimination of γs, αs and βs more than makes up for the extra integer array memory introduced. Moreover, if sections adjacent to the first are cut away, the transition array is simply modified, so that the memory associated with the γs, αs and βs of the adjacent sections is completely recovered.

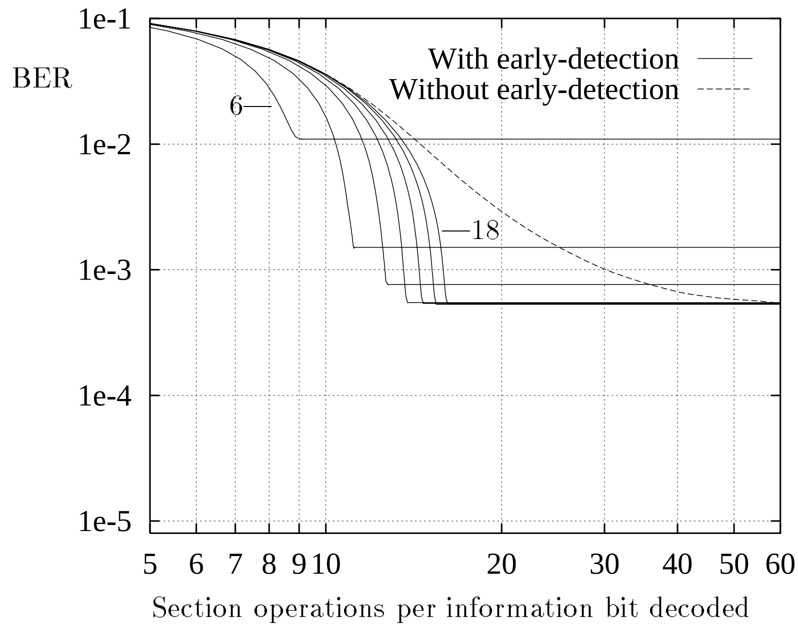

Figure 5.20: BER performance of turbo-decoding with and without early-detection, for $E_b/N_0 = 0.1$ dB, for thresholds of 6, 8, 10, 12, 14, 16, 18.

5.4.5 Experimental results

I have simulated trellis splicing results at $E_b/N_0 = 0.1, 0.2$ and 0.3 dB, for the turbo-decoding system described in Section 5.4.2. At the end of each half-iteration of turbo-decoding, the log-odds ratio of each information bit was compared with a threshold in order to decide whether or not the bit should be early-detected. In order to average out the effects of block failure modes (*i.e.*, failure modes where a large fraction of the information block is incorrectly decoded), I simulated the transmission of 20 000 information blocks for each value of the threshold. The resulting number of errors and number of section operations were then averaged over block transmissions. Figs. 5.20, 5.21 and 5.22 show plots of BER versus average number of section operations per information bit decoded, for a variety of thresholds. The curves for turbo-decoding *without* early-detection are also shown. For these simulations, a fixed number of decoding iterations were performed for each block.

For a given BER, the computational complexity of decoding can be reduced the most compared to standard turbo-decoding by using the threshold that corresponds to the curve in each figure that bottoms out at the prespecified BER. Thus, the locus of points corresponding to the knees of the curves gives the optimal achievable BER-complexity performance. Using these curves, we can answer the question, "At a specified E_b/N_0 and for different BER, what is the computational gain obtained by using early detection compared to using fewer decoding iterations without early detection?" The locus of points described

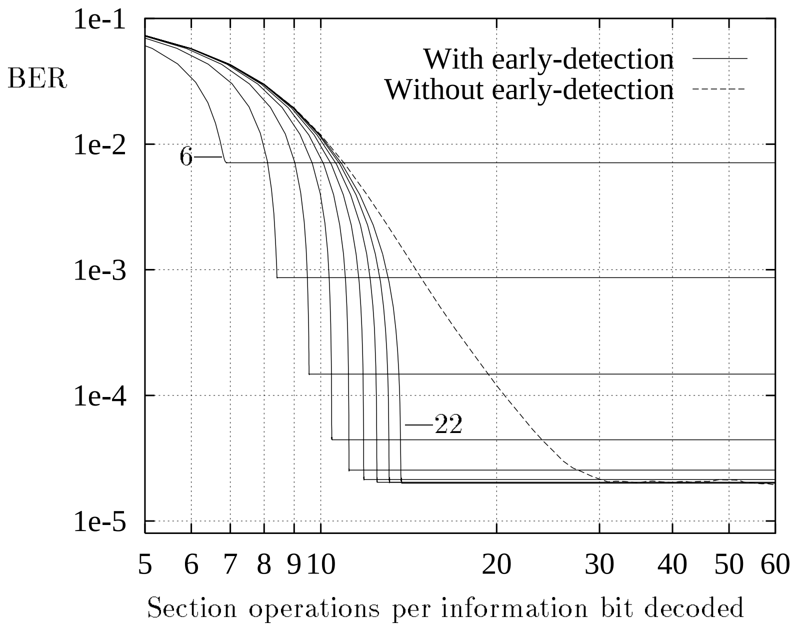

Figure 5.21: BER performance of turbo-decoding with and without early-detection, for $E_b/N_0 = 0.2$ dB, for thresholds of 6, 8, 10, 12, 14, 16, 18, 20, 22.

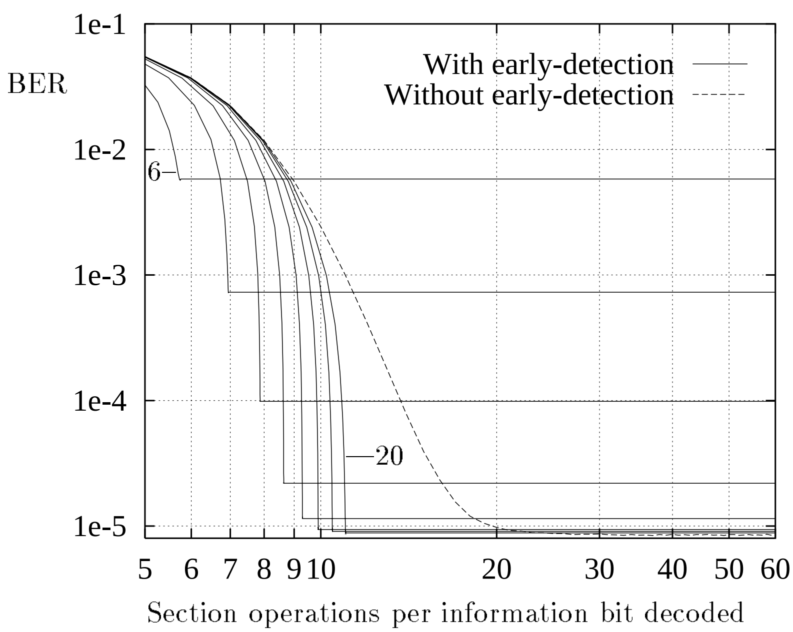

Figure 5.22: BER performance of turbo-decoding with and without early-detection, for $E_b/N_0 = 0.3$ dB, for thresholds of 6, 8, 10, 12, 14, 16, 18, 20.

above is interpolated in Figure 5.23 which shows the computational gain as a function of BER for the different values of E_b/N_0. For all three values of E_b/N_0, the greatest computational gain is obtained near the minimal BER.

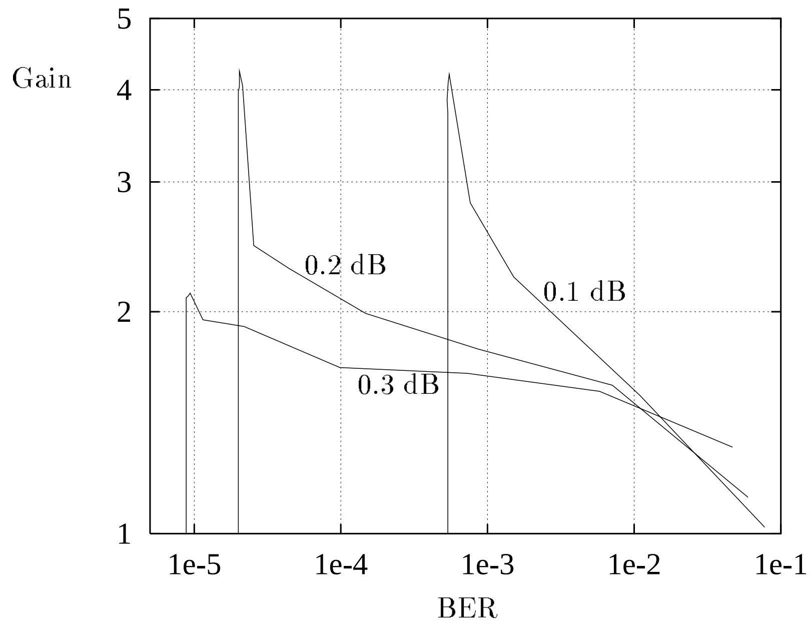

Figure 5.23: The computational gain for turbo-decoding with early-detection compared to turbo-decoding without early-detection as a function of BER, for $E_b/N_0 = 0.1$, 0.2, and 0.3 dB.

5.5 Parallel iterative decoding

The decoding algorithm for low-density parity-check codes proposed by Gallager [1963] and later by MacKay and Neal [1996] is inherently a parallel algorithm. As described in Section 5.2.5, probability propagation in the Bayesian network for a low-density parity-check code consists of passing *sets* of messages back and forth between the codeword bits and the clamped parity-check variables. It turns out that the standard decoders for turbo-codes and serially-concatenated convolutional codes are inherently *serial*. In this section, I consider a parallel message-passing schedule for turbo-decoding.

5.5.1 Concurrent turbo-decoding

If each chain in a turbo-code is viewed as a single unrefined vertex (*e.g.*, Figure 5.11b), then turbo-decoding can also be viewed as a "parallel" algorithm[2]. However, if each chain in a turbo-code is viewed at a refined level (*e.g.*, Figure 5.8), then the standard turbo-decoding algorithm is inherently serial. That is, when messages are passed as shown in Figure 5.9, most of the computations are used to compute messages that cannot be propagated in parallel.

[2]When there is more than one chain in a turbo-code, messages may be passed between chains in either a serial or parallel manner.

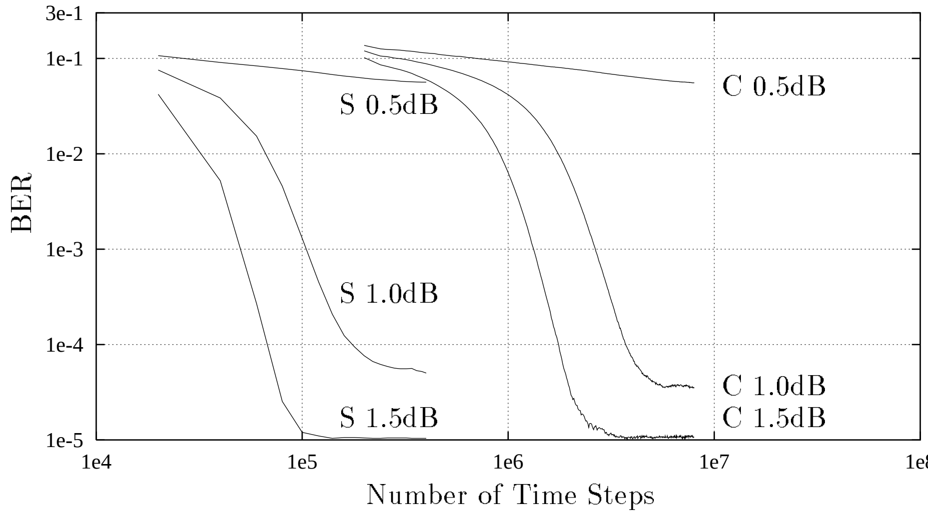

Figure 5.24: Performance of standard (S) and concurrent (C) turbo-decoding when implemented on a serial computer, for 3 values of E_b/N_0.

Here, I consider *concurrent turbo-decoding* in which messages are passed in a parallel fashion. One time step of concurrent turbo-decoding consists of simultaneously passing messages in both directions on all graph edges in the Bayesian network for the code. (Although "concurrent" is not quite the right term for such a parallel algorithm, the term "parallel" is used in the other name for turbo-codes, "parallel concatenated convolutional codes".) Notice that concurrent turbo-decoding is not just a parallel implementation of standard turbo-decoding. It is a different algorithm which may have different properties.

A naive approach to a hardware implementation of concurrent turbo-decoding would be to build one simple processor for each vertex in the Bayesian network for a code. Of course, for reasonably long block lengths, a prohibitively large number of these processors would be needed for a fully parallel VLSI implementation of concurrent turbo-decoding. In the following section, I empirically compare the time complexity of standard decoding with concurrent decoding, while ignoring practical implementation issues such as wiring complexity. In practice, a more space-efficient implementation (*e.g.*, time-shared processors) would be used at some detriment to the computational efficiency.

5.5.2 Results

The code used for the simulations was a rate 1/2 $K = 5,000$, $N = 10,000$ turbo-code with two constituent convolutional codes, each with generator polynomials $(21/37)_{\text{octal}}$. The constituent chains were connected by a randomly selected permuter. Every second output of each constituent chain was punctured to get a rate of 1/2. For each of three values of

E_b/N_0, the transmission of 10^7 information bits was simulated, and the results are shown in Figures 5.24 and 5.25. Interestingly, for a given E_b/N_0 it appears that both algorithms converge to the same BER.

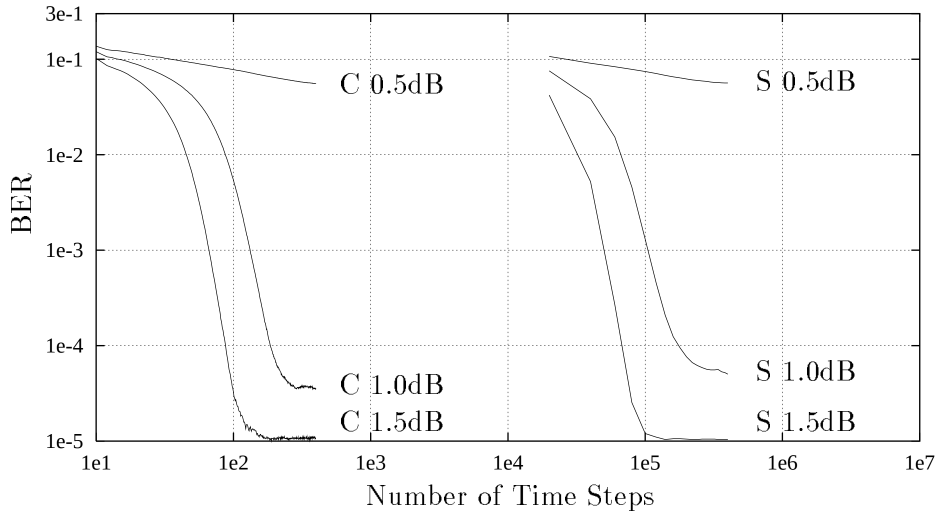

Figure 5.25: Performance of standard (S) and concurrent (C) turbo-decoding when implemented on a parallel computer, for 3 values of E_b/N_0.

Figure 5.24 shows the BER versus the number of messages passed in the constituent chains, for standard and concurrent turbo-decoding. (The computation of the messages passed in the constituent chain dominates the decoding time). The number of messages passed gives a good indication of decoding complexity on a serial computer. Not suprisingly, it is apparent that the standard algorithm is better suited to serial implementation. It is interesting that for a given BER, the concurrent decoding algorithm is roughly $3,000$ times *slower* than the standard algorithm when implemented on a serial computer.

Figure 5.25 shows the BER versus the number of time steps for the case where $10,000$ processors are available for concurrent turbo-decoding. We assume that pipelining is *not* used for standard turbo-decoding. For a given BER, the concurrent decoding algorithm is roughly 850 times *faster* than the standard algorithm when implemented on a parallel computer. If one processor is used for each half iteration of 5 iterations of standard turbo-decoding in a pipeline fashion, standard decoding can be sped up by a factor of only 10 (extra pipeline stages do not improve the BER). Concurrent turbo-decoding is still 85 times faster.

Chapter 6

Conclusions and Future Research

My goal in this thesis has been to present to the reader a unified graphical model framework for describing problems and developing inference algorithms in the areas of pattern classification, data compression, and channel coding. The previous three chapters have shown how Bayesian networks and various inference algorithms can be applied to problems in these areas. A major theme of this work is that probabilistic structure can be exploited to come up with efficient algorithms. I conclude by highlighting the contributions made in this thesis and the importance of these contributions.

6.1 A statistically valid comparison of Bayesian network pattern classifiers

Recent research papers on new learning methods for parameterized Bayesian networks have suggested that the new algorithms could produce good density estimators. These methods include parameter estimation by Markov chain Monte Carlo, wake-sleep learning in the Helmholtz machine, and variational estimation. One outcome of my research (Section 3.5) is a statistically valid comparison of the performance of these methods and other standard algorithms using the DELVE (data for evaluating learning in valid experiments) test system. The Bayesian network classifiers perform very well compared to other standard methods, such as the k-nearest neighbor classifier. One surprise is that the simple logistic autoregressive network (which does not have any latent variables) performs very well.

6.2 Wake-sleep learning in the Helmholtz machine

The wake-sleep algorithm is only an approximate form of the generalized expectation maximization algorithm. As such, experimental confirmation that the wake-sleep algorithm works at all is important. In Section 3.6, I presented original results showing that the wake-sleep algorithm can extract high-order structure from noisy images that were produced by a hierarchical graphics model. After estimation, the Helmholtz machine can be used to non-linearly filter a test image and recover its hierarchical description.

6.3 Multi-valued source codes

Aside from pattern classification, another use for parameterized Bayesian networks is data compression. Bayesian networks that have many "unmixable" latent variables naturally lead to multi-valued source codes in which there are a very large number of codewords for each input pattern. Previous research on source coding has focussed on single-valued source codes, since at first glance it seems that a multi-valued code must irrevocably waste codeword space. Another outcome of my research (Chapter 4) is a practical bits-back encoder that is often able to use multi-valued source codes to communicate in a highly efficient manner, even when the codewords are unmixable. The compression rate for bits-back coding is identical to the cost function for the generalized expectation maximization algorithm. It follows that parameterized Bayesian networks that are estimated using the generalized expectation maximization algorithm (or an approximation to it, such as the wake-sleep algorithm) are well-suited to bits-back coding.

The results I obtained for bits-back coding indicate that for the source models I explored, the method is currently not a strong contender in the data compression practice. The main reason for this is that the underlying source models were not good enough. However, bits-back coding does provide an extra degree of source model design freedom, and so it leaves open a door for further research into models that produce multi-valued source codes.

6.4 Integrating out model parameters using bits-back coding

The work on bits-back coding described above leads to a practical coding method for integrating out continousou model parameters. Suppose we are interested in encoding blocks of patterns and that the source changes from block to block, but not within any single block. Given a block of patterns, a parameterized model can be estimated. Then, the parameters can be quantized to some precision and communicated. The block of patterns is then encoded using the quantized parameters. According to the principles of Bayesian analysis, we ought to integrate out the model parameters and use an ensemble of models to communicate the block of patterns. It turns out that bits-back coding can be used to communicate each block of patterns using the ensemble of models, as long as a good approximation to the posterior distribution over the parameters given the block of patterns is available. In this case, the ensemble of models corresponds to the multi-valued source code. I am currently exploring the use of bits-back coding for integrating over continous parameter spaces.

6.5 A graphical model framework for iterative channel decoding

A recent class of iterative error-correcting decoding algorithms are showing great promise in closing the gap between practical channel coding and Shannon's 50-year-old theoretical limit. In Chapter 5, I showed that this class of algorithms can be viewed as probability propagation in Bayesian networks that describe different types of error-correcting code. This overarching framework is important, since prior to this work, iterative decoders have been proposed in an *ad. hoc.* way with only a limited exposition of the similarities between the algorithms. This framework also lead to a contribution in the area of reduced-complexity iterative decoding, both for serial implementations and for parallel implementations.

6.6 Interleaved trellis-constraint codes (ITC codes)

In Section 5.3, I proposed a general class of "interleaved trellis-constraint codes", which have a graphical structure consisting of two or more finite-state chains that interact through permuted state transition labels. This class of codes includes turbo-codes, serially-concatenated convolutional codes, low-density parity-check codes, and product codes, all of which have recently been shown to give good performance. The generalization shows that this spectrum of codes has several gaps, one of which I refer to as a "nonsystematic ITC code". An instance of this new code performs nearly as well as a standard turbo-code, and significantly better than the best known low-density parity-check code with the same communication rate. I believe the Bayesian network and probability propagation frameworks cement a broad foundation for understanding a variety of new decoders and for developing new decoding algorithms.

Appendix A

Proofs

A.1 Probability propagation in Bayesian Networks

In order to prove that the probability propagation algorithm described in Section 2.1.3 computes $P(z_i|\mathbf{v})$ $i = 1, \ldots, |\mathbf{z}|$ exactly, I will show that the propagation rules implement a locally consistent form of probabilistic inference. After some arguments regarding the unimportance of the order in which messages are passed, the global validity of the algorithm will follow by induction. The definition of dependency-separation and the rules for determining dependency-separation (Section 1.2.4) are used extensively in the following derivations.

Recall that in probability propagation, there are two types of messages. π-*messages* are passed from parents to children in the direction of the edges (use the first consonant of "π" and "parent" as a mnemonic here). λ-*messages* are passed from children to parents in the opposite direction of the edges. Both types of vector have lengths that are equal to the number of values the *parent variable* can take on. Consider the network fragment shown in Figure 2.1c, where $\mathbf{x}$ is the set of parents of y, and $\mathbf{z}$ is the set of children of y. I *define* the incoming λ-messages as follows:

$$\lambda_y^{Z_j Y} = \beta_j P(\mathbf{v}^{Z_j - Y}|y), \quad y = 1, \ldots, |y|, \tag{A.1}$$

where $\mathbf{v}^{Z_j - Y} \subseteq \mathbf{v}$ is the set of observations that are connected directly or indirectly to z_j through paths that *do not* go through y. Associated with each child z_j, there is a constant β_j that does not depend on the value of y. That is, each message need only be *proportional* to the probability vector.

I define the incoming π-messages as follows:

$$\pi_{x_i}^{X_iY} = \rho_i P(x_i|\mathbf{v}^{X_i-Y}), \quad x_i = 1, \ldots, |x_i|, \tag{A.2}$$

where $\mathbf{v}^{X_i-Y} \subseteq \mathbf{v}$ is the set of observations that are connected directly or indirectly to x_i through paths that *do not* go through y. Associated with each parent x_i, there is a constant ρ_i that does not depend on the value of x_i. Notice that regarding the unobserved variables in the network, π-messages are probababilities, whereas λ-messages are likelihoods.

A.1.1 Computing $P(y|\mathbf{v})$ from the incoming messages

Consider the fusion formula (2.11) that is used to compute $P(y|\mathbf{v})$ for an unobserved variable y. (If y is observed, the computation of $P(y|\mathbf{v})$ is trivial.) Substituting the above definitions and the definition for $P_{\mathbf{x}y}^{\mathbf{X}Y}$ (2.1) into the final fusion formula (2.11) (and renaming the function computed by the fusion formula $F(y)$), we get

$$F(y) = \left[\alpha \textstyle\prod_{k=1}^{|\mathbf{z}|} \beta_k \prod_{i=1}^{|\mathbf{x}|} \rho_i\right] \left[\prod_{k=1}^{|\mathbf{z}|} P(\mathbf{v}^{Z_k-Y}|y)\right] \left[\displaystyle\sum_{\mathbf{x}} P(y|\mathbf{x}) \textstyle\prod_{i=1}^{|\mathbf{x}|} P(x_i|\mathbf{v}^{X_i-Y})\right]. \tag{A.3}$$

α is an arbitrary constant used to normalize $F(y)$ later on, so the first term in braces can be replaced by a new constant α', which will be computed to normalize $F(y)$.

Since the network is singly-connected, the observations connected to the children of y by paths that do not go through y are dependency-separated from each other by y (condition 2 in Section 1.2.4), and thus $\prod_{k=1}^{|\mathbf{z}|} P(\mathbf{v}^{Z_k-Y}|y) = P(\mathbf{v}^{\mathbf{Z}-Y}|y)$. Since the network is singly-connected, the parents of y are connected to each other only through y. Consequently, the parents of y are dependency-separated from each other by the observations that are connected to the parents of y by paths that do not go through y (condition 3 in Section 1.2.4). Thus, $\prod_{i=1}^{|\mathbf{x}|} P(x_i|\mathbf{v}^{X_i-Y}) = P(\mathbf{x}|\mathbf{v}^{\mathbf{X}-Y})$. The parents of y dependency-separate y from $\mathbf{v}^{\mathbf{X}-Y}$ (condition 1), so $P(y|\mathbf{x})P(\mathbf{x}|\mathbf{v}^{\mathbf{X}-Y}) = P(y, \mathbf{x}|\mathbf{v}^{\mathbf{X}-Y})$. Making these substitutions, we get

$$F(y) = \alpha' P(\mathbf{v}^{\mathbf{Z}-Y}|y) \sum_{\mathbf{x}} P(y, \mathbf{x}|\mathbf{v}^{\mathbf{X}-Y}) = \alpha' P(\mathbf{v}^{\mathbf{Z}-Y}|y) P(y|\mathbf{v}^{\mathbf{X}-Y}). \tag{A.4}$$

However, $\mathbf{v}^{\mathbf{Z}-Y}$ is dependency-separated from $\mathbf{v}^{\mathbf{X}-Y}$ by y (condition 1), and as a result we have $P(\mathbf{v}^{\mathbf{Z}-Y}|y) = P(\mathbf{v}^{\mathbf{Z}-Y}|y, \mathbf{v}^{\mathbf{X}-Y})$, and

$$F(y) = \alpha' P(\mathbf{v}^{\mathbf{Z}-Y}|y, \mathbf{v}^{\mathbf{X}-Y}) P(y|\mathbf{v}^{\mathbf{X}-Y}) = \alpha' P(\mathbf{v}^{\mathbf{Z}-Y}, y|\mathbf{v}^{\mathbf{X}-Y}). \tag{A.5}$$

After computing the α' that normalizes $F(y)$ with respect to y, we get

$$F(y) = P(y|\mathbf{v}^{\mathbf{Z}-Y}, \mathbf{v}^{\mathbf{X}-Y}) = P(y|\mathbf{v}), \tag{A.6}$$

which justifies the final fusion equation (2.11).

A.1.2 Outgoing π-messages

1. *y observed*: If y has the observed value y°, then $P(y|\mathbf{v}^{Y-Z_j}) = \delta(y, y^{\circ})$, since $\{y^{\circ}\} \subseteq \mathbf{v}^{Y-Z_j}$. From (2.7), it follows that

$$\pi_y^{YZ_j} = \delta(y, y^{\circ}) = P(y|\mathbf{v}^{Y-Z_j}). \tag{A.7}$$

2. *y unobserved*: For unobserved y, the formula for for an outgoing *pi*-message (2.5) after substituting the definitions for the incoming messages, is

$$\pi_y^{YZ_j} = c_1\Big[\prod_{\substack{k=1\\k\neq j}}^{|\mathbf{z}|} P(\mathbf{v}^{Z_j-Y}|y)\Big]\Big[\sum_{\mathbf{x}} P(y|\mathbf{x})\prod_{i=1}^{|\mathbf{x}|} P(x_i|\mathbf{v}^{X_i-Y})\Big], \tag{A.8}$$

where the product of the constants has been replaced by c_1. According to the same type of arguments as presented in the previous section, the first term equals $P(\{\mathbf{v}^{Z_j-Y}\}_{k=1,k\neq j}^{|\mathbf{z}|}|y)$, and the second term equals $P(y|\mathbf{v}^{\mathbf{X}-Y})$, so we get

$$\pi_y^{YZ_j} = c_1 P(\{\mathbf{v}^{Z_j-Y}\}_{k=1,k\neq j}^{|\mathbf{z}|}|y) P(y|\mathbf{v}^{\mathbf{X}-Y}). \tag{A.9}$$

The observations $\{\mathbf{v}^{Z_j-Y}\}_{k=1,k\neq j}^{|\mathbf{z}|}$ are dependency-separated from $\mathbf{v}^{\mathbf{X}-Y}$ by y (condition 1), and so $P(\{\mathbf{v}^{Z_j-Y}\}_{k=1,k\neq j}^{|\mathbf{z}|}|y) = P(\{\mathbf{v}^{Z_j-Y}\}_{k=1,k\neq j}^{|\mathbf{z}|}|y, \mathbf{v}^{\mathbf{X}-Y})$, and

$$\begin{aligned}
\pi_y^{YZ_j} &= c_1 P(\{\mathbf{v}^{Z_j-Y}\}_{k=1,k\neq j}^{|\mathbf{z}|}|y, \mathbf{v}^{\mathbf{X}-Y}) P(y|\mathbf{v}^{\mathbf{X}-Y}) = c_1 P(\{\mathbf{v}^{Z_j-Y}\}_{k=1,k\neq j}^{|\mathbf{z}|}, y|\mathbf{v}^{\mathbf{X}-Y}) \\
&= c_1 P(\{\mathbf{v}^{Z_j-Y}\}_{k=1,k\neq j}^{|\mathbf{z}|}|\mathbf{v}^{\mathbf{X}-Y}) P(y|\{\mathbf{v}^{Z_j-Y}\}_{k=1,k\neq j}^{|\mathbf{z}|}, \mathbf{v}^{\mathbf{X}-Y}) \\
&= c_2 P(y|\{\mathbf{v}^{Z_j-Y}\}_{k=1,k\neq j}^{|\mathbf{z}|}, \mathbf{v}^{\mathbf{X}-Y}).
\end{aligned} \tag{A.10}$$

Noting that $\mathbf{v}^{Y-Z_j} = \{\mathbf{v}^{Z_j-Y}\}_{k=1,k\neq j}^{|\mathbf{z}|} \cup \mathbf{v}^{\mathbf{X}-Y}$, we get

$$\pi_y^{YZ_j} = c_2 P(y|\mathbf{v}^{Y-Z_j}). \tag{A.11}$$

(A.7) and (A.11) show that probability propagation is locally consistent in the outgoing π-messages; *i.e.*, the outgoing π-messages are proportional to the right probability vectors.

A.1.3 Outgoing λ-messages

1. *y observed*: After substituting the definitions for the incoming messages into the formula (2.10) for computing outgoing λ-messages when y is obseved to have the value $y^{\mathbf{v}}$, we get

$$\lambda_{x_i}^{YX_i} = c_1 \sum_{\mathbf{x}':x_i'=x_i} P(y^{\mathrm{o}}|\mathbf{x}') \prod_{\substack{k=1\\k\neq i}}^{|\mathbf{x}|} P(x_k'|\mathbf{v}^{X_k-Y}), \tag{A.12}$$

where the product of the constants has been replaced by c_1. Using the same type of arguments as were used in Section A.1.1, it can be shown that the summand equals $P(y^{\mathrm{o}}, \{x_k'\}_{k=1,k\neq i}^{|\mathbf{x}|}|x_i, \{\mathbf{v}^{X_k-Y}\}_{k=1,k\neq i}^{|\mathbf{x}|})$. After summing over $\mathbf{x}'$ we get

$$\lambda_{x_i}^{YX_i} = c_1 P(y^{\mathrm{o}}|x_i, \{\mathbf{v}^{X_k-Y}\}_{k=1,k\neq i}^{|\mathbf{x}|}). \tag{A.13}$$

Notice that this formula does not include the observations $\mathbf{v}^{\mathbf{Z}-Y}$ connected to y's children. This makes sense, since if y is observed, the likelihood of $\mathbf{v}^{\mathbf{Z}-Y}$ does not depend on x_i. (It would if y was not observed.) However, I now include the likelihood of $\mathbf{v}^{\mathbf{Z}-Y}$ for the sake of notational clarity later on. Since $P(\mathbf{v}^{\mathbf{Z}-Y}|y^{\mathrm{o}})$ is just a constant (with respect to x_i), we can write

$$\lambda_{x_i}^{YX_i} = c_2 P(\mathbf{v}^{\mathbf{Z}-Y}|y^{\mathrm{o}}) P(y^{\mathrm{o}}|x_i, \{\mathbf{v}^{X_k-Y}\}_{k=1,k\neq i}^{|\mathbf{x}|}). \tag{A.14}$$

Since y dependency-separates $\mathbf{v}^{\mathbf{Z}-Y}$ from x_i and $\{\mathbf{v}^{X_k-Y}\}_{k=1,k\neq i}^{|\mathbf{x}|}$ (condition 1), we have $P(\mathbf{v}^{\mathbf{Z}-Y}|y^{\mathrm{o}}) = P(\mathbf{v}^{\mathbf{Z}-Y}|y^{\mathrm{o}}, x_i, \{\mathbf{v}^{X_k-Y}\}_{k=1,k\neq i}^{|\mathbf{x}|})$, and so

$$\begin{aligned} \lambda_{x_i}^{YX_i} &= c_2 P(\mathbf{v}^{\mathbf{Z}-Y}|y^{\mathrm{o}}, x_i, \{\mathbf{v}^{X_k-Y}\}_{k=1,k\neq i}^{|\mathbf{x}|}) P(y^{\mathrm{o}}|x_i, \{\mathbf{v}^{X_k-Y}\}_{k=1,k\neq i}^{|\mathbf{x}|}) \\ &= c_2 P(\mathbf{v}^{\mathbf{Z}-Y}, y^{\mathrm{o}}|x_i, \{\mathbf{v}^{X_k-Y}\}_{k=1,k\neq i}^{|\mathbf{x}|}). \end{aligned} \tag{A.15}$$

Of course, x_i and $\{\mathbf{v}^{X_k-Y}\}_{k=1,k\neq i}^{|\mathbf{x}|}$ are dependency-separated (by nothing) (condition 3) so that $P(\{\mathbf{v}^{X_k-Y}\}_{k=1,k\neq i}^{|\mathbf{x}|}|x_i) = P(\{\mathbf{v}^{X_k-Y}\}_{k=1,k\neq i}^{|\mathbf{x}|})$ and

$$\begin{aligned} \lambda_{x_i}^{YX_i} &= c_2 \frac{P(\mathbf{v}^{\mathbf{Z}-Y}, y^{\mathrm{o}}, \{\mathbf{v}^{X_k-Y}\}_{k=1,k\neq i}^{|\mathbf{x}|}|x_i)}{P(\{\mathbf{v}^{X_k-Y}\}_{k=1,k\neq i}^{|\mathbf{x}|}|x_i)} = c_2 \frac{P(\mathbf{v}^{\mathbf{Z}-Y}, y^{\mathrm{o}}, \{\mathbf{v}^{X_k-Y}\}_{k=1,k\neq i}^{|\mathbf{x}|}|x_i)}{P(\{\mathbf{v}^{X_k-Y}\}_{k=1,k\neq i}^{|\mathbf{x}|})} \\ &= c_3 P(\mathbf{v}^{\mathbf{Z}-Y}, y^{\mathrm{o}}, \{\mathbf{v}^{X_k-Y}\}_{k=1,k\neq i}^{|\mathbf{x}|}|x_i). \end{aligned} \tag{A.16}$$

Noting that $\mathbf{v}^{Y-X_i} = \mathbf{v}^{\mathbf{Z}-Y} \cup \{y^{\mathrm{o}}\} \cup \{\mathbf{v}^{X_k-Y}\}_{k=1,k\neq i}^{|\mathbf{x}|}$, we get

$$\lambda_{x_i}^{YX_i} = c_3 P(\mathbf{v}^{Y-X_i}|x_i). \tag{A.17}$$

2. *y unobserved*: After substituting the definitions for the incoming messages into the formula (2.8) for computing outgoing λ-messages when y is unobseved, we get

$$\lambda_{x_i}^{YX_i} = c_1 \sum_y \Big[\prod_{j=1}^{|\mathbf{z}|} P(\mathbf{v}^{Z_j - Y}|y)\Big] \Big[\sum_{\mathbf{x}':x'_i = x_i} P(y|\mathbf{x}') \prod_{\substack{k=1\\k\neq i}}^{|\mathbf{x}|} P(x'_k|\mathbf{v}^{X_k - Y})\Big], \tag{A.18}$$

where the product of the constants has been replaced by c_1. According to the same type of arguments as presented in Section A.1.1, the first term in braces equals $P(\mathbf{v}^{\mathbf{Z}-Y}|y)$. The summand of the inner sum equals $P(y, \{x'_k\}_{k=1,k\neq i}^{|\mathbf{x}|}|x_i, \{\mathbf{v}^{X_k-Y}\}_{k=1,k\neq i}^{|\mathbf{x}|})$, but after summing over $\mathbf{x}'$ the second term in braces equals $P(y|x_i, \{\mathbf{v}^{X_k-Y}\}_{k=1,k\neq i}^{|\mathbf{x}|})$, and thus

$$\lambda_{x_i}^{YX_i} = c_1 \sum_y P(\mathbf{v}^{\mathbf{Z}-Y}|y) P(y|x_i, \{\mathbf{v}^{X_k-Y}\}_{k=1,k\neq i}^{|\mathbf{x}|}). \tag{A.19}$$

Since y dependency-separates $\mathbf{v}^{\mathbf{Z}-Y}$ from x_i and $\{\mathbf{v}^{X_k-Y}\}_{k=1,k\neq i}^{|\mathbf{x}|}$ (condition 1), we have $P(\mathbf{v}^{\mathbf{Z}-Y}|y) = P(\mathbf{v}^{\mathbf{Z}-Y}|y, x_i, \{\mathbf{v}^{X_k-Y}\}_{k=1,k\neq i}^{|\mathbf{x}|})$, and so

$$\begin{aligned}
\lambda_{x_i}^{YX_i} &= c_1 \sum_y P(\mathbf{v}^{\mathbf{Z}-Y}|y, x_i, \{\mathbf{v}^{X_k-Y}\}_{k=1,k\neq i}^{|\mathbf{x}|}) P(y|x_i, \{\mathbf{v}^{X_k-Y}\}_{k=1,k\neq i}^{|\mathbf{x}|}) \\
&= c_1 \sum_y P(\mathbf{v}^{\mathbf{Z}-Y}, y|x_i, \{\mathbf{v}^{X_k-Y}\}_{k=1,k\neq i}^{|\mathbf{x}|}) \\
&= c_1 P(\mathbf{v}^{\mathbf{Z}-Y}|x_i, \{\mathbf{v}^{X_k-Y}\}_{k=1,k\neq i}^{|\mathbf{x}|}).
\end{aligned} \tag{A.20}$$

Of course, x_i and $\{\mathbf{v}^{X_k-Y}\}_{k=1,k\neq i}^{|\mathbf{x}|}$ are dependency-separated (by nothing) (condition 3) so that $P(\{\mathbf{v}^{X_k-Y}\}_{k=1,k\neq i}^{|\mathbf{x}|}|x_i) = P(\{\mathbf{v}^{X_k-Y}\}_{k=1,k\neq i}^{|\mathbf{x}|})$ and

$$\begin{aligned}
\lambda_{x_i}^{YX_i} &= c_1 \frac{P(\mathbf{v}^{\mathbf{Z}-Y}, \{\mathbf{v}^{X_k-Y}\}_{k=1,k\neq i}^{|\mathbf{x}|}|x_i)}{P(\{\mathbf{v}^{X_k-Y}\}_{k=1,k\neq i}^{|\mathbf{x}|}|x_i)} = c_1 \frac{P(\mathbf{v}^{\mathbf{Z}-Y}, \{\mathbf{v}^{X_k-Y}\}_{k=1,k\neq i}^{|\mathbf{x}|}|x_i)}{P(\{\mathbf{v}^{X_k-Y}\}_{k=1,k\neq i}^{|\mathbf{x}|})} \\
&= c_2 P(\mathbf{v}^{\mathbf{Z}-Y}, \{\mathbf{v}^{X_k-Y}\}_{k=1,k\neq i}^{|\mathbf{x}|}|x_i).
\end{aligned} \tag{A.21}$$

Noting that $\mathbf{v}^{Y-X_i} = \mathbf{v}^{\mathbf{Z}-Y} \cup \{\mathbf{v}^{X_k-Y}\}_{k=1,k\neq i}^{|\mathbf{x}|}$, we get

$$\lambda_{x_i}^{YX_i} = c_2 P(\mathbf{v}^{Y-X_i}|x_i). \tag{A.22}$$

(A.17) and (A.22) show that probability propagation is locally consistent in the outgoing λ-messages. That is, the outgoing λ-messages are proportional to the right likelihood vectors.

A.1.4 Global consistency

Sections A.1.1, A.1.2 and A.1.3 show that if the incoming messages to vertex y are proportional to the appropriate likelihood vectors (A.1) and probability vectors (A.2), then the propagation equations compute $P(y|\mathbf{v})$ as well as outgoing messages that are proportional to the right probability vectors and likelihood vectors. In this sense, probability propagation is locally consistent. In this section, I show that if the propagation rules described in Section 2.1.3 are followed until there are no more buffered messages, then each vertex will have available all incoming messages as defined in (A.1) and (A.2).

First, note the the message passing formulas accumulate the effects of observations. That is, if a message is passed from z_i to z_j in response to the observation of z_1, then when a message is passed from z_i to z_j in response to the observation of z_2, the latter message will include the effects of z_1 *and* z_2. Second, note that the rules for probability propagation ensure that once propagation is complete, the final message passed from z_i to z_j is computed from the final messages passed to z_i from all *other* neighbors of z_i. Combining the above two comments, it follows that the final message passed from z_i to z_j will contain the effects of all observations $\mathbf{o}^{Z_i - Z_j}$ connected both directly and indirectly to z_i by paths that do not go through z_j. (Notice that network initialization is required in order to propagate the effects of null observations.) In other words, once propagation is complete each vertex has available the incoming messages as defined in (A.1) and (A.2).

A.2 Grouping variables in Bayesian networks

As described in Section 2.1.4, two variables z_j and z_k may be grouped into a single vertex, as long as z_j is not an indirect descendent of z_k and *vice versa*. Here, I show that this grouping operation preserves the representational capacity of the network. That is, the new network can represent at least all those distributions that the old network could represent.

Grouping introduces new conditional probabilities for the variables that are grouped and for variables whose set of parents includes one or both of the variables that are grouped. The new joint distribution is

$$P'(\mathbf{z}) = P'(z_j, z_k|\mathbf{a}_j, \mathbf{a}_k) \prod_{\substack{i:i\neq j, i\neq k\\ z_j \notin \mathbf{a}_i, z_k \notin \mathbf{a}_i}} P'(z_i|\mathbf{a}_i) \prod_{\substack{i:i\neq j, i\neq k\\ z_j\in\mathbf{a}_i \text{ and/or } z_k\in\mathbf{a}_i}} P'(z_i|\mathbf{a}_i, z_j, z_k). \tag{A.23}$$

Now, set

$$P'(z_j, z_k|\mathbf{a}_j, \mathbf{a}_k) = P(z_j|\mathbf{a}_j)P(z_k|\mathbf{a}_k). \tag{A.24}$$

For all i such that $i \neq j$, $i \neq k$, $z_j \notin \mathbf{a}_i$, and $z_k \notin \mathbf{a}_i$, set

$$P'(z_i|\mathbf{a}_i) = P(z_i|\mathbf{a}_i). \tag{A.25}$$

For all i such that $i \neq j$, $i \neq k$, and also such that $z_j \in \mathbf{a}_i$ and/or $z_k \in \mathbf{a}_i$, set

$$P'(z_i|\mathbf{a}_i, z_j, z_k) = P(z_i|\mathbf{a}_i). \tag{A.26}$$

Substituting these into (A.23), we see that

$$\begin{aligned} P'(\mathbf{z}) &= P(z_j|\mathbf{a}_j)P(z_k|\mathbf{a}_k) \prod_{\substack{i:i\neq j, i\neq k \\ z_j\notin\mathbf{a}_i, z_k\notin\mathbf{a}_i}} P(z_i|\mathbf{a}_i) \prod_{\substack{i:i\neq j, i\neq k \\ z_j\in\mathbf{a}_i \text{ and/or } z_k\in\mathbf{a}_i}} P(z_i|\mathbf{a}_i) \\ &= \prod_i P(z_i|\mathbf{a}_i) = P(\mathbf{z}). \end{aligned} \tag{A.27}$$

In this way, the joint distribution of the old network can be represented by the new network.

A.3 Proof of condition for inference by ancestral simulation

Here, I show that if the parents of the visible variables in a Bayesian network are dependency-separated from the hidden variables of interest $\mathbf{h}^{\mathrm{I}}$ by the visible variables $\mathbf{v}$, then ancestral simulation can be used to obtain a sample from $P(\mathbf{h}^{\mathrm{I}}|\mathbf{v})$. If the condition holds, then every path connecting each variable in $\mathbf{h}^{\mathrm{I}}$ to the parents of each visible variable is blocked. This means that *disconnecting* each visible variable from its parents will not change the distribution $P(\mathbf{h}^{\mathrm{I}}|\mathbf{v})$. Since each visible variable will then have no parents, its value can be included as fixed constant in the conditional probability functions for its children. We are then left with a new Bayesian network that describes a distribution $P'(\mathbf{h})$ over the variables $\mathbf{h}$ that were not observed in the original network. Although in general $P'(\mathbf{h}) \neq P(\mathbf{h}|\mathbf{v})$, as shown above we have $P'(\mathbf{h}^{\mathrm{I}}) = P(\mathbf{h}^{\mathrm{I}}|\mathbf{v})$. So, we may simply use ancestral simulation in the new network to obtain samples from $P(\mathbf{h}^{\mathrm{I}}|\mathbf{v})$. Notice that ancestral simulation in the new network is equivalent to ancestral simulation for the unobserved variables in the original network.

A.4 Proof of detailed balance for slice sampling

In order to show that the slice sampling Markov chain Monte Carlo procedure for $p(z)$ described in Section 2.2.4 has $p(z)$ has a stationary distribution, I will show that the procedure satisfies detailed balance:

$$p(z)q(y|z) = p(y)q(z|y), \tag{A.28}$$

where $q(y|z)$ is the p.d.f. that the procedure chooses the new value $y = z^{\text{new}}$ from the old value $z = z^{\text{old}}$. Factor this transition probability using the two steps taken by the procedure: choosing a slice at height s given z and then choosing y given the slice and z:

$$q(y|z) = \int_s q(y|s,z)q(s|z)ds. \tag{A.29}$$

The equation for detailed balance can be written

$$\int_s q(y|s,z)q(s|z)p(z)ds = \int_s q(z|s,y)q(s|y)p(y)ds. \tag{A.30}$$

In order to prove detailed balance I show that

$$q(s|z)p(z) = q(s|y)p(y) \quad \text{and} \quad q(y|s,z) = q(z|s,y). \tag{A.31}$$

Let $f(z) = \alpha p(z)$. The p.d.f. for s given z is uniform over the interval $[0, f(z)]$, so $q(s|z) = 1/f(z) = 1/(\alpha p(z))$, and $q(s|z)p(z) = p(z)/(\alpha p(z)) = 1/\alpha$. Similarly the p.d.f. for s given y is uniform over the interval $[0, f(y)]$, so $q(s|y) = 1/f(y) = 1/(\alpha p(y))$, and $q(s|y)p(y) = p(y)/(\alpha p(y)) = 1/\alpha$. Therefore, $q(s|z)p(z) = q(s|y)p(y)$.

To prove $q(y|s,z) = q(z|s,y)$, first consider the case where y and z are in the same segment of the slice. Given that y and z are in the same segment of the slice, the procedure for picking y does not depend on z, and *vice versa*. It follows trivially that $q(y|s,z) = q(z|s,y)$. Reasoning by symmetry, it can be shown that $q(y|s,z) = q(z|s,y)$ in the case where y and z are in different segments

Appendix B

The BNC Software Package

BNC (Bayesian Networks for coding) is a Tcl-based interactive program that performs probability propagation. The package is tailored to coding applications, but can be used to propagate probabilities in any Bayesian network where the real-valued variables are observed and where the conditional probabilities for the real-valued variables are mixtures of Gaussians. BNC uses vectors, so for example a group of information variables can be handled quite easily. Also, instead of defining one conditional distribution (link) for each variable in the network, BNC uses prototypical links that can be reused for different variables. Some of the commands that BNC uses are

- `crVars`: creates a vector of discrete or real variables
- `crLink`: creates a prototypical link
- `addtoLink`: adds to a prototypical link a parent-child configuration that has non-zero probability
- `linkVars`: links a variable to its parents
- `shNet`: shows the network connectivity
- `clVal`: clamps a variable to a given value
- `sndrcvMsg`: passes a probability message from one variable to another
- `shProb`: shows the current estimate of the marginal probability for a variable

After describing where to find the software and how to install it, I give a simple example of how BNC can be used to propagate probabilities in the burglar alarm network from Section 2.1.2. Then, I give the BNC scripts that I used to obtain the turbo-code results described in Chapter 5.

B.1 Installing the software

BNC makes use of the Tcl and extended Tcl libraries, which can be downloaded from

`http://sunscript.sun.com`

The BNC tar file `bnc.tar` can be obtained from my web page,

`http://www/cs.utoronto.ca/~frey`

Untar this file with a command like `tar xf bnc.tar`, and you will get a directory called `./bnc` which will contain the source for BNC and some helpful scripts (*e.g.*, copies of the commands given in the following tutorials). You should see the following files in the directory `./bnc`: `berrou.tcl`, `bnc.c`, `lfsr.tcl`, `Makefile`, `man.ps`, `rsc.tcl`, and `sr.tcl`. `man.ps` contains a tutorial and a *BNC command reference*.

Before making `bnc`, you'll need to know which version of Tcl you have installed on your machine. Change into the directory `./bnc`, and edit `Makefile` in order to set the `BNC_INCLUDE_MAIN` flag as described in `Makefile`. You may also need to edit the include directory and library links to get the Tcl and extended Tcl libraries working right.

B.2 An example: The burglar alarm problem

The burglar alarm network described in Section 2.1.2 consists of three variables b (burglar), e (earthquake) and a (alarm), and the following conditional probabilities:

$$\begin{aligned} P(b=1) = 0.1, \qquad P(e=1) = 0.1, \\ P(a=1|b=0, e=0) = 0.001, \quad P(a=1|b=1, e=0) = 0.368, \\ P(a=1|b=0, e=1) = 0.135, \quad P(a=1|b=1, e=1) = 0.607. \end{aligned} \tag{B.1}$$

In this section, I show how BNC can be used to propagate probabilities in response to the observation $a = 1$.

I suggest that `bnc` be run with `fep` so that command lines can be easily modified:

```
> fep bnc

 H-------------------------------- bnc --------------------------------H
   Bayesian Networks for Coding.  Type 'cmds' for a list of commands.
                  Copyright (c) 1996 Brendan J. Frey.
```

Next, create three binary vector variables containing one element each:

```
bnc>crVars d b 2 1
bnc>crVars d e 2 1
bnc>crVars d a 2 1
```

With the option `d`, the command `crVars d` v m n creates a vector called v of n discrete variables, each of which can take on the values $\{0, \ldots, m-1\}$. The elements of a vector are referred to with a hyphen. For example, to find out about variable `b-0`, type

```
bnc>shVar b-0
Variable name: b-0
Real-valued: No
Observed: No
Number of states: 2
Value: 32320
Currently unlinked
```

Note that the value of `b-0` is undefined since `b-0` has not been observed.

Now create the prototypical conditional probability links, using the values given in (B.1):

```
bnc>crLink P(b) d 2; addtoLink P(b) 0.9 0; addtoLink P(b) 0.1 1
bnc>
bnc>crLink P(e) d 2; addtoLink P(e) 0.9 0; addtoLink P(e) 0.1 1
bnc>
bnc>crLink P(a|b,e) d 2 2 2
bnc>addtoLink P(a|b,e) 0.999 0 0 0; addtoLink P(a|b,e) 0.001 1 0 0
bnc>addtoLink P(a|b,e) 0.865 0 0 1; addtoLink P(a|b,e) 0.135 1 0 1
bnc>addtoLink P(a|b,e) 0.632 0 1 0; addtoLink P(a|b,e) 0.368 1 1 0
bnc>addtoLink P(a|b,e) 0.393 0 1 1; addtoLink P(a|b,e) 0.607 1 1 1
```

BNC interprets `P(b)` as a string representing the name of the link, and does *not* parse characters such as `(`, `)`, and `|`. In particular, at this point BNC does not relate the link `P(b)` to the burglar variable vector `b` created above. In the command `crLink P(b) d 2`, `d` indicates the child variable for the link is discrete, and `2` indicates the child can take on two values, $\{0, 1\}$. The command `addtoLink P(b) 0.1 1` adds to the link `P(b)` a probability mass of 0.1 for the child having the value 1. Note that it is only necessary to specify the parent-child configurations that have non-zero probability. `addtoLink P(a|b,e) 0.368 1`

`1 0` specifies a probability of 0.368 that the child has the value 1 given that the first parent has the value 1 and the second parent has the value 0.

Now that the prototypical links have been defined, link together the network and take a look at it:

```
bnc>linkVars P(b) b-0
bnc>linkVars P(e) e-0
bnc>linkVars P(a|b,e) a-0 b-0 e-0
bnc>shNet
NULL -> b-0
NULL -> e-0
b-0 e-0 -> a-0
```

Here, `NULL` indicates that the variable does not have any parents. This completes the specification of the Bayesian network.

The next series of commands clamps the values of `a-0` to 1 and propagates messages across the network in the fashion shown in Figure 2.2:

```
bnc>clVal a-0 1
bnc>sndrcvMsg b-0 a-0 v
(0.9000,0.1000)
bnc>sndrcvMsg a-0 e-0 v
(0.1714,0.8286)
bnc>sndrcvMsg e-0 a-0 v
(0.9000,0.1000)
bnc>sndrcvMsg a-0 b-0 v
(0.0354,0.9646)
```

The flag `v` in the command `sndrcvMsg` means "verbose", and causes the command to print out the probability message. Note that these messages are normalized versions of the ones shown in Figure 2.2.

Finally, examine the marginal probabilities for `b-0` and `e-0` given that `a-0` is clamped to 1:

```
bnc>shProb b-0
(0.2485,0.7515)
bnc>shProb e-0
(0.6506,0.3494)
```

```
bnc>
bnc>exit
```

In general, it is up to the user to decide in what order the probability messages should be passed.

See `man.ps` for a complete BNC command reference.

B.3 Scripts used to decode a turbo-code

Before listing the main BNC script `berrou.tcl` used to simulate a turbo-code, I list the script `lfsr.tcl` that is used to build the conditional probability links for linear feedback shift registers, given the feedforward and feedback delay taps.

B.3.1 `lfsr.tcl`

```
# usage: buildLFSR <num> <den> <u->s> <us->s> <s->x> [<s->s> [<s->u>]]
#
# Copyright 1996 Brendan J. Frey.
#
# This bnc script defines a procedure for building a binary I/O
# LFSR given the coefficients of the z-transform numerator and
# denominator polynomials. num and den are lists of bits. This
# procedure returns -1 if there is an error and otherwise
# returns the number of states for the state variable. Note
# that the z^0 coefficients in num and den (right-most bits)
# must be 1. Also, outputs are a function of the state *alone*,
# so the input is included as part of the state; this approach
# differs from the trellis-based approach (where outputs are
# associated with state *transitions*), but gives a Bayesian
# network that is singly-connected and so can be processed
# using probability propagation. So, the current state contains
# the true state of the LFSR plus the input bit to be used in
# determining the next state. The procedure creates three
# links and two more optional ones. The link names are
# passed to buildLFSR by the user. utos links the first input
# to the LFSR. In contrast, ustos links an input and a previous
# state to the next state. stox links the state to the output.
```

```
# The optional link stos links a previous state to the next
# state so that the state will eventually reach 0. (E.g.,
# in coding applications, this link can be used to implement
# trellis termination.) The second optional link stou determines
# which input bit is stored in the state. (E.g., if systematic
# trellis termination is being used, this link can be used to
# obtain the input bit that was needed to help terminate the
# trellis.)

proc buildLFSR {num den utos ustos stox {stos "NULL"} {stou "NULL"}} {
    set lnum [llength $num]; set lden [llength $den]
    set stmem $lden; set stsz [expr 1 << $stmem]

    # Check that coefficients make sense.
    if { [expr $lden != $lnum] } { return -1 }
    if { [expr [lindex $num [expr $lnum - 1]] != 1] } { return -1 }
    if { [expr [lindex $den [expr $lden - 1]] != 1] } { return -1 }

    # Build the link used for the first state.
    crLink $utos d $stsz 2
    addtoLink $utos 1.0 0 0; addtoLink $utos 1.0 1 1

    # Make the other links, by examining the LFSR transfer function.
    crLink $ustos d $stsz 2 $stsz; crLink $stox d 2 $stsz
    if {[expr {"NULL" != "$stos"}]} { crLink $stos d $stsz $stsz }
    if {[expr {"NULL" != "$stou"}]} { crLink $stou d 2 $stsz }
    loop st 0 $stsz {
        # Get the input bit from the state.
        set w [expr $st & 1]
        if {[expr {"NULL" != "$stou"}]} { addtoLink $stou 1.0 $w $st }

        # XOR the input bit with the feedback bit.
        loop i 1 $stmem {
            if {[expr [lindex $den [expr $lden - $i - 1]] == 1]} {
                set w [expr $w + (($st >> $i) & 1)]
            }
        }
```

```
        set w [expr $w % 2]

        # Compute the next state and add to the ustos link, for each
        # possible input bit (bt) at the next time step.
        loop bt 0 2 {
            set nst [expr (((($st >> 1) << 1) + $w) << 1) % $stsz + $bt]
            addtoLink $ustos 1.0 $nst $bt $st
        }

        # Compute the next state for the stos link and add to the link.
        if {[expr {"NULL" != "$stos"}]} {
            set nst [expr (((($st >> 1) << 1) + $w) << 1) % $stsz]
            set bt 0
            loop i 1 $stmem {
                if {[expr [lindex $den [expr $lden - $i - 1]] == 1]} {
                    set bt [expr $bt + (($nst >> $i) & 1)]
                }
            }
            set bt [expr $bt % 2]; set nst [expr $nst + $bt]
            addtoLink $stos 1.0 $nst $st
        }

        # Compute the output bit from w and the state, and add to the
        # stox link.
        if {[expr [lindex $num [expr $lnum-1]]==1]} {
            set x $w } else { set x 0 }
        loop i 1 $stmem {
            if {[expr [lindex $num [expr $lnum - $i - 1]] == 1]} {
                set x [expr $x + (($st >> $i) & 1)]
            }
        }
        set x [expr $x % 2]
        addtoLink $stox 1.0 $x $st
    }

    return $stsz
}
```

B.3.2 berrou.tcl

The following BNC script is for a specific E_b/N_0 (0.6 dB) and for a specific number of transmitted blocks (530). The value of E_b/N_0 was varied to obtain BER-E_b/N_0 curves.

```
# Results for a punctured rate 1/2 turbo-code. Since the all-zero
# codeword is always sent, a decoder network is built, plus a noise
# vector network (independent Gaussian units).

# Set up the constants
set LOGFILE berrou0.6.log
set SNR 0.6
set K 65536
set NBLOCKS 530
set NITERS 18
set NUM {1 0 0 0 1}
set DEN {1 1 1 1 1}

set K2 [expr 2*$K]; set Km1 [expr $K-1]
set RATE [expr 1.0*$K/$K2]; set VAR [expr pow(10.0,-($SNR/10.0))/2.0/$RATE]

# Build the recursive convolutional encoder link.
source lfsr.tcl
set S [buildLFSR $NUM $DEN u->s us->s s->x]
if { [expr $S == -1] } { puts "Error: Could not build encoder link."; exit }


# Create information bit variables, state variables, codeword bit variables,
# and received signal variables for constituent codes 1 and 2.
crVars d du 2 $K; crVars d ds1 $S $K; crVars d ds2 $S $K; crVars d dx 2 $K2
crVars r dy $K2

# Create the noise vector variables.
crVars r ns $K2

# Create a 50/50 prior link for the info bits
crLink u d 2; addtoLink u 0.5 0; addtoLink u 0.5 1
```

```
# Create a link used for the systematic codeword bits.
crLink u->x d 2 2; addtoLink u->x 1.0 0 0; addtoLink u->x 1.0 1 1

# Create the channel link (Gaussian distribution).
crLink x->y r 2
addtoLink x->y 1.0 -1.0 $VAR 0; addtoLink x->y 1.0 1.0 $VAR 1

# Create the noise vector link (Gaussian distribution).
crLink ns r; addtoLink ns 1.0 -1.0 $VAR

# Build the interleaver.
set P [permute $K]

# Connect up the noise links to the noise variables.
loop i 0 $K2 { linkVars ns ns-$i }

# Connect up the variables for the decoder network. (Don't forget to
# puncture the two constituent convolutional codes.)
loop i 0 $K { linkVars u du-$i; linkVars u->x dx-[expr 2*$i] du-$i }
loop i 0 $K2 { linkVars x->y dy-$i dx-$i }

linkVars u->s ds1-0 du-0;
linkVars u->s ds2-0 du-[lindex $P 0]
loop i 1 $K {
    linkVars us->s ds1-$i du-$i ds1-[expr $i-1]
    linkVars us->s ds2-$i du-[lindex $P $i] ds2-[expr $i-1]
}

loop i 0 $K {
    set ip1 [expr $i+1]; set i2p1 [expr (2*$i)+1]
    if { [expr ($i%2) == 0] } { linkVars s->x dx-$i2p1 ds1-$ip1
    } else { linkVars s->x dx-$i2p1 ds2-$i }
}

# Define which variables are clamped in the decoder.
loop j 0 $K2 { clVal dy-$j }
```

```
# Define the transmit procedure.
proc transmit {} { drVal ns; transfer ns dy }

# Define the schedule and procedure for initializing the decoder.
loop j 0 $K2 { addtoSched init sndrcv dy-$j dx-$j }
loop i 0 $K {
    set ip1 [expr $i+1]; set i2 [expr 2*$i]; set i2p1 [expr $i2+1]
    if { [expr ($i%2) == 0] } { addtoSched init sndrcv dx-$i2p1 ds1-$ip1
    } else { addtoSched init sndrcv dx-$i2p1 ds2-$i }
}
loop j 0 $K { addtoSched init sndrcv dx-[expr 2*$j] du-$j }
proc initDecoder {} {
    initMsgs du; initMsgs ds1; initMsgs ds2; initMsgs dx; initMsgs dy
    exSched init
}

# Define the fb1 and fb2 schedules.
loop j 0 $K { addtoSched fb1 sndrcv du-$j ds1-$j }
loop j 0 $Km1 { addtoSched fb1 sndrcv ds1-$j ds1-[expr $j+1] }
loop j $Km1 0 -1 { addtoSched fb1 sndrcv ds1-$j ds1-[expr $j-1] }
loop j 0 $K { addtoSched fb1 sndrcv ds1-$j du-$j }

loop j 0 $K { addtoSched fb2 sndrcv du-[lindex $P $j] ds2-$j }
loop j 0 $Km1 { addtoSched fb2 sndrcv ds2-$j ds2-[expr $j+1] }
loop j $Km1 0 -1 { addtoSched fb2 sndrcv ds2-$j ds2-[expr $j-1] }
loop j 0 $K { addtoSched fb2 sndrcv ds2-$j du-[lindex $P $j] }

# Simulate many block transmissions, printing the current BER estimate out
# as we go. Also, save the noise patterns that cause problems.
set fID [open $LOGFILE w]; seed 0
set nerr {}; loop k 0 [expr $NITERS+1] { lappend nerr 0 }
loop i 0 $NBLOCKS {
    transmit; initDecoder; detMAP du; set dst [shNorm du]
    set nerr [lreplace $nerr 0 0 [expr $dst + [lindex $nerr 0]]]
    set a [format "%8.2le " [expr 1.0 * [lindex $nerr 0]/$K/($i+1)]]
    puts -nonewline $fID $a
```

```
    loop k 1 [expr $NITERS+1] {
        exSched fb1; exSched fb2; detMAP du; set dst [shNorm du]
        set nerr [lreplace $nerr $k $k [expr $dst + [lindex $nerr $k]]]
        set a [format "%8.2le " [expr 1.0 * [lindex $nerr $k]/$K/($i+1)]]
        puts -nonewline $fID $a
    }
    puts $fID ""; flush $fID
}
close $fID
```

Bibliography

Bahl, L. R., Cocke, J., Jelinek, F., and Raviv, J. (1974). Optimal decoding of linear codes for minimizing symbol error rate. *IEEE Transactions on Information Theory*, 20:284–287.

Baum, L. E. and Petrie, T. (1966). Statistical inference for probabilistic functions of finite state markov chains. *Annals of Mathematical Statistics*, 37:1559–1563.

Benedetto, S., Divsalar, D., Montorsi, G., and Pollara, F. (1996). Soft-ouput decoding algorithms in iterative decoding of parallel concatenated convolutional codes. Submitted to *IEEE International Conference on Communications.*

Benedetto, S. and Montorsi, G. (1996a). Iterative decoding of serially concatenated convolutional codes. *Electronics Letters*, 32:1186–1188.

Benedetto, S. and Montorsi, G. (1996b). Serial concatenation of block and convolutional codes. *Electronics Letters*, 32:887–888.

Benedetto, S., Montorsi, G., Divsalar, D., and Pollara, F. (1997). Serial concatenation of interleaved codes: Performance analysis, design, and iterative decoding. To appear in *IEEE Transactions on Information Theory.*

Berrou, C. and Glavieux, A. (1996). Near optimum error correcting coding and decoding: Turbo-codes. *IEEE Transactions on Communications*, 44:1261–1271.

Berrou, C., Glavieux, A., and Thitimajshima, P. (1993). Near Shannon limit error-correcting coding and decoding: Turbo codes. In *Proceedings of the IEEE International Conference on Communications.*

Bishop, C. M. (1995). *Neural Networks for Pattern Recognition.* Oxford University Press Inc., New York NY.

Bishop, C. M., Svensén, M., and Williams, C. K. I. (1997). Gtm: the generative topographic mapping. To appear in *Neural Computation.*

Blahut, R. E. (1990). *Digital Transmission of Information.* Addison-Wesley Pub. Co., Reading MA.

Breiman, L., Friedman, J. H., Olshen, R. A., and Stone, C. J. (1984). *Classification and regression trees.* Wadsworth, Blemont CA.

Calderbank, A. R. and Sloane, N. J. A. (1987). New trellis codes based on lattices and cosets. *IEEE Transactions on Information Theory*, 33:177.

Chandler, D. (1987). *Introduction to Modern Statistical Mechanics.* Oxford University Press, New York NY.

Chow, C. K. (1957). An optimum character recognition system using decision functions. *IRE Transactions on Electronic Computing*, 6:247–254.

Collins, O. M. (1993). Determinate state convolutional codes. *IEEE Transactions on Communications*, 41(12):1785–1794.

Cooper, G. F. (1990). The computational complexity of probabilistic inference using Bayesian belief networks. *Artificial Intelligence*, 42:393–405.

Cover, T. M. and Thomas, J. A. (1991). *Elements of Information Theory.* John Wiley & Sons, New York NY.

Dagum, P. (1993). Approximating probabilistic inference in Bayesian belief networks is NP-hard. *Artificial Intelligence*, 60:141–153.

Dagum, P. and Chavez, R. M. (1993). Approximating probabilistic inference in bayesian belief networks. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 15(3):246–255.

Daut, D. G., Modestino, J. W., and Wismer, L. D. (1982). New short constraint length convolutional code constructions for selected rational rates. *IEEE Transactions on Information Theory*, 28(5):794–800.

Dayan, P. and Hinton, G. E. (1996). Varieties of Helmholtz machine. *Neural Networks*, 9:1385–1403.

Dayan, P., Hinton, G. E., Neal, R. M., and Zemel, R. S. (1995). The Helmholtz machine. *Neural Computation*, 7:889–904.

Dayan, P. and Zemel, R. S. (1995). Competition and multiple cause models. *Neural Computation*, 7:565–579.

Dempster, A. P., Laird, N. M., and Rubin, D. B. (1977). Maximum likelihood from incomplete data via the EM algorithm. *Proceedings of the Royal Statistical Society*, B-39:1–38.

Devroye, L. (1986). *Nonuniform Random Variate Generation*. Springer-Verlag, New York NY.

Divsalar, D. and Pollara, F. (1995). Turbo-codes for PCS applications. In *Proceedings of the International Conference on Communications*, pages 54–59.

Duane, S., Kennedy, A. D., Pendleton, B. J., and Roweth, D. (1987). Hybrid monte carlo. *Physical Letters B*, 195:216–222.

Duda, R. O. and Hart, P. E. (1973). *Pattern Classification and Scene Analysis*. John Wiley, New York NY.

Everitt, B. S. (1984). *An Introduction to Latent Variable Models*. Chapman and Hall, New York NY.

Feygin, G. (1995). *Arithmetic Coding: Parallel Algorithms and Architectures*. Department of Electrical and Computer Engineering, University of Toronto, Toronto Canada. Doctoral dissertation.

Fletcher, R. (1987). *Practical methods of optimization*. John Wiley & Sons, New York NY.

Foldiak, P. (1990). Forming sparse representations by local anti-hebbian learning. *Biological Cybernetics*, 64:165–170.

Forney, Jr., G. D. (1973). The Viterbi algorithm. *Proceedings of the IEEE*, 61(3):268–277.

Forney, Jr., G. D. (1988). Coset codes - Part I: Introduction and geometrical classification. *IEEE Transactions on Information Theory*, 34:1123.

Forney, Jr., G. D. (1997). Approaching the capacity of the AWGN channel with coset codes and multilevel coset codes. Submitted to *IEEE Transactions on Information Theory*.

Frey, B. J. (1997a). Continuous sigmoidal belief networks trained using slice sampling. In Mozer, M. C., Jordan, M. I., and Petsche, T., editors, *Advances in Neural Information Processing Systems 9*. MIT Press, Cambridge MA.

Frey, B. J. (1997b). Variational inference for continuous sigmoidal bayesian networks. In *Sixth International Workshop on Artificial Intelligence and Statistics*. Ft. Lauderdale FL.

Frey, B. J. and Hinton, G. E. (1996). Free energy coding. In Storer, J. A. and Cohn, M., editors, *Proceedings of the Data Compression Conference 1996*. IEEE Computer Society Press. Available at `http://www.cs.utoronto.ca/~frey`.

Frey, B. J. and Hinton, G. E. (1997). Efficient stochastic source coding and an application to a Bayesian network source model. To appear in *The Computer Journal*.

Frey, B. J. and Kschischang, F. R. (1996). Probability propagation and iterative decoding. In *Proceedings of the* 34^{th} *Allerton Conference*. Available at `http://www.cs.utoronto.ca/~frey`.

Gallager, R. G. (1963). *Low-Density Parity-Check Codes*. MIT Press, Cambridge MA.

Geman, S. and Geman, D. (1984). Stochastic relaxation, Gibbs distribution and the Bayesian restoration of images. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 6:721–741.

Ghahramani, Z. and Jordan, M. I. (1996). Factorial hidden Markov models. MIT Computational Cognitive Science Technical Report 9502.

Gilks, W. R. and Wild, P. (1992). Adaptive rejection sampling for Gibbs sampling. *Applied Statistics*, 41:337–348.

Goodman, L. A. (1970). The multivariate analysis of qualitative data: Interaction among multiple classifications. *Journal of the American Statistical Association*, 65:226–256.

Hagenauer, J., Offer, E., and Papke, L. (1993). Improving the standard coding system for deep space missions. In *Proceedings of IEEE International Conference on Communications*, pages 1092–1097.

Hagenauer, J., Offer, E., and Papke, L. (1996). Iterative decoding of binary block and convolutional codes. *IEEE Transactions on Information Theory*, 42(2):429–445.

Hammersley, J. M. and Handscomb, D. C. (1964). *Monte Carlo Methods*. Chapman and Hall, London England.

Heckerman, D. and Geiger, D. (1995). Learning Bayesian networks: a unification for discrete and gaussian domains. In Besnard, P. and Hanks, S., editors, *Proceedings of the Eleventh Conference on Uncertainty in Artificial Intelligence*. Morgan Kaufmann.

Hinton, G. E., Dayan, P., Frey, B. J., and Neal, R. M. (1995). The wake-sleep algorithm for unsupervised neural networks. *Science*, 268:1158–1161.

Hinton, G. E. and Sejnowski, T. J. (1986). Learning and relearning in Boltzmann machines. In Rumelhart, D. E. and McClelland, J. L., editors, *Parallel Distributed Processing: Explorations in the Microstructure of Cognition*, volume I, pages 282–317. MIT Press, Cambridge MA.

Hinton, G. E. and Zemel, R. S. (1994). Autoencoders, minimum description length and Helmholtz free energy. In Cowan, J. D., Tesauro, G., and Alspector, J., editors, *Advances in Neural Information Processing Systems 6*. Morgan Kauffmann.

Hofmann, R. and Tresp, V. (1996). Discovering structure in continuous variables using bayesian networks. In Touretzky, D., Mozer, M., and Hasselmo, M., editors, *Advances in Neural Information Processing Systems 8*. MIT Press.

Huffman, D. A. (1952). A method for the construction of minimum redundancy codes. *Proceedings of the Institute of Radio Engineers*, 40:1098–1101.

Imai, H. and Hirakawa, S. (1977). A new multilevel coding method using error-correcting codes. *IEEE Transactions on Information Theory*, 23:371–377. Correction, Nov. 1977, p. 784.

Jaakkola, T. and Jordan, M. I. (1997). A variational approach to Bayesian logistic regression models and their extensions. In *Sixth International Workshop on Artificial Intelligence and Statistics*.

Jaakkola, T., Saul, L. K., and Jordan, M. I. (1996). Fast learning by bounding likelihoods in sigmoid type belief networks. In Touretzky, D. S., Mozer, M. C., and Hasselmo, M. E., editors, *Advances in Neural Information Processing Systems 8*. MIT Press.

Jacobs, R. A., Jordan, M. I., Nowlan, S. J., and Hinton, G. E. (1991). Adaptive mixtures of local experts. *Neural Computation*, 3:79–87.

Jordan, M. I. (1995). Why the logistic function? A tutorial discussion on probabilities and neural networks. Technical Report Computational Cognitive Science 9503, MIT, Cambridge MA.

Jordan, M. I. and Jacobs, R. A. (1994). Hierarchical mixtures of experts and the EM algorithm. *Neural Computation*, 6:181–214.

Kalos, M. H. and Whitlock, P. A. (1986). *Monte Carlo Methods, Volume I: Basics*. John Wiley, New York NY.

Kinderman, R. and Snell, J. L. (1980). *Markov Random Fields and Their Applications*. American Mathematical Society, Providence USA.

Kschischang, F. R. and Frey, B. J. (1997). Iterative decoding of compound codes by probability propagation in graphical models. To appear in *IEEE Journal on Selected Areas in Communications*, available at `http://www.cs.utoronto.ca/~frey`.

Lauritzen, S. L., Dawid, A. P., Larsen, B. N., and Leimer, H. G. (1990). Independence properties of directed Markov fields. *Networks*, 20:491–505.

Lauritzen, S. L. and Spiegelhalter, D. J. (1988). Local computations with probabilities on graphical structures and their application to expert systems. *Journal of the Royal Statistical Society B*, 50:157–224.

Lauritzen, S. L. and Wermuth, N. (1989). Graphical models for associations between variables, some of which are qualitative and some quantitative. *Annals of Statistics*, 17:31–57.

Le Cun, Y., Boser, B., Denker, J. S., Henderson, D., Howard, R. E., Hubbard, W., and Jackel, L. D. (1989). Back-propagation applied to handwritten zip code recognition. *Neural Computation*, 1:541–551.

Lee, E. A. and Messerschmitt, D. G. (1994). *Digital Communication.* Kluwer Academic Publishers, Norwell MA.

Lee, L. (1977). Concatenated coding systems employing a unit-memory convolutional code and a byte-oriented decoding algorithm. *IEEE Transactions on Communications*, 25(10):1064–1074.

Lin, S. and Costello, Jr., D. J. (1983). *Error Control Coding: Fundamentals and Applications.* Prentice-Hall Inc., Englewood Cliffs NJ.

Lodge, J., Young, R., Hoeher, P., and Hagenauer, J. (1993). Separable MAP 'filters' for the decoding of product and concatenated codes. In *Proceedings of IEEE International Conference on Communications*, pages 1740–1745.

MacKay, D. J. C. (1995). Bayesian neural networks and density networks. *Nuclear Instruments and Methods in Physics Research*, 354:73–80.

MacKay, D. J. C. (1997). Good codes based on very sparse matrices. Submitted to *IEEE Transactions on Information Theory.*

MacKay, D. J. C., McEliece, R. J., and Cheng, J. F. (1997). Turbo-decoding as an instance of Pearl's 'belief propagation' algorithm. To appear in *IEEE Journal on Selected Areas in Communications.*

MacKay, D. J. C. and Neal, R. M. (1995). Good codes based on very sparse matrices. In Boyd, C., editor, *Cryptography and Coding. 5th IMA Conference*, number 1025 in Lecture Notes in Computer Science, pages 100–111. Springer, Berlin Germany.

MacKay, D. J. C. and Neal, R. M. (1996). Near Shannon limit performance of low density parity check codes. *Electronics Letters*, 32(18):1645–1646. Due to editing errors, reprinted in *E*lectronics Letters **33**, March 1997, 457–458.

McCullagh, P. and Nelder, J. A. (1983). *Generalized Linear Models*. Chapman and Hall, London England.

McEliece, R. J. (1996). On the BJCR trellis for linear block codes. *IEEE Transactions on Information Theory*, 42.

Meng, X. L. and Rubin, D. B. (1992). Recent extensions of the EM algorithm (with discussion). In Bernardo, J. M., Berger, J. O., Dawid, A. P., and Smith, A. F. M., editors, *Bayesian Statistics 4*. Clarendon Press, Oxford England.

Metropolis, N., Rosenbluth, A. W., Rosenbluth, M. N., Teller, A. H., and Teller, E. (1953). Equation of state calculation by fast computing machines. *Journal of Chemical Physics*, 21:1087–1092.

Movellan, J. R. and McClelland, J. L. (1992). Learning continuous probability distributions with symmetric diffusion networks. *Cognitive Science*, 17:463–496.

Neal, R. M. (1992). Connectionist learning of belief networks. *Artificial Intelligence*, 56:71–113.

Neal, R. M. (1993). Probabilistic inference using Markov chain Monte Carlo methods. Unpublished manuscript available over the internet by ftp at `ftp://ftp.cs.utoronto.ca/pub/radford/review.ps.Z`.

Neal, R. M. (1996). *Bayesian Learning for Neural Networks*. Springer-Verlag, New York NY.

Neal, R. M. (1997). Markov chain Monte Carlo methods based on "slicing" the density function. In preparation.

Neal, R. M. and Hinton, G. E. (1993). A new view of the EM algorithm that justifies incremental and other variants. Unpublished manuscript available over the internet by ftp at `ftp://ftp.cs.utoronto.ca/pub/radford/em.ps.Z`.

Pearl, J. (1986). Fusion, propagation, and structuring in belief networks. *Artificial Intelligence*, 29:241–288.

Pearl, J. (1987). Evidential reasoning using stochastic simulation of causal models. *Artificial Intelligence*, 32:245–257.

Pearl, J. (1988). *Probabilistic Reasoning in Intelligent Systems.* Morgan Kaufmann, San Mateo CA.

Peterson, C. and Anderson, J. R. (1987). A mean field theory learning algorithm for neural networks. *Complex Systems*, 1:995–1019.

Potamianos, G. G. and Goutsias, J. K. (1993). Partition function estimation of Gibbs random field images using Monte Carlo simulations. *IEEE Transactions on Information Theory*, 39:1322–1332.

Rabiner, L. (1989). A tutorial on hidden Markov models and selected applications in speech recognition. *Proceedings of the IEEE*, 77:257–286.

Rasmussen, C. E. (1996). *Evaluation of Gaussian Processes and Other Methods for Non-Linear Regression.* Department of Computer Science, University of Toronto, Toronto Canada. Doctoral dissertation (`ftp://ftp.cs.toronto.edu/pub/carl/thesis.ps.gz`).

Rasmussen, C. E., Neal, R. M., Hinton, G. E., van Camp, D., Revow, M., Ghahramani, Z., Kustra, R., and Tibshirani, R. (1996). *The DELVE Manual.* University of Toronto, Toronto Canada. `http://www.cs.utoronto.ca/~delve`.

Ripley, B. D. (1987). *Stochastic Simulation.* John Wiley, New York NY.

Rissanen, J. (1989). *Stochastic Complexity in Statistical Inquiry.* World Scientific, Singapore.

Rissanen, J. and Langdon, G. G. (1976). Arithmetic coding. *IBM Journal of Research and Development*, 23:149–162.

Saul, L. K., Jaakkola, T., and Jordan, M. I. (1996). Mean field theory for sigmoid belief networks. *Journal of Artificial Intelligence Research*, 4:61–76.

Saund, E. (1995). A multiple cause mixture model for unsupervised learning. *Neural Computation*, 7:51–71.

Schubert, L. K. (1976). Extending the expressive power of semantic networks. *Artificial Intelligence*, 7:163–198.

Shannon, C. E. (1948). A mathematical theory of communication. *Bell System Technical Journal*, 27:379–423, 623–656.

Sheykhet, I. I. and Simkin, B. Y. (1990). Monte Carlo method in the theory of solutions. *Computer Physics Reports*, 12:67–133.

Spiegelhalter, D. J. (1986). Probabilistic reasoning in predictive expert systems. In Kanal, L. N. and Lemmer, J. F., editors, *Uncertainty in Artificial Intelligence*, pages 47–68. North Holland, Amsterdam.

Spiegelhalter, D. J. (1990). Fast algorithms for probabilistic reasoning in influence diagrams, with applications in genetics and expert systems. In Oliver, R. M. and Smith, J. Q., editors, *Influence Diagrams, Belief Nets, and Decision Analysis*, pages 361–384. John Wiley & Sons, New York NY.

Spiegelhalter, D. J. and Lauritzen, S. L. (1990). Sequential updating of conditional probabilities on directed graphical structures. *Networks*, 20:579–605.

Tanner, R. M. (1981). A recursive approach to low complexity codes. *IEEE Transactions on Information Theory*, 27:533–547.

Tibshirani, R. (1992). Principal curves revisited. *Statistics and Computing*, 2:183–190.

Ungerboeck, G. (1982). Channel coding with multilevel/phase signals. *IEEE Transactions on Information Theory*, 28(1).

Viterbi, A. J. and Omura, J. K. (1979). *Principles of Digital Communication and Coding.* McGraw-Hill, New York NY.

Vorobev, N. N. (1962). Consistent families of measures and their extensions. *Theory of Probability and Applications*, 7:147–163.

Wachsmann, U. and Huber, J. (1995). Power and bandwidth efficient digital communication using turbo-codes in multilevel codes. *European Transactions on Telecommunications*, 6(5):557–567.

Wallace, C. S. (1990). Classification by minimum-message-length inference. In S. G. Akl, *et. al.*., editor, *Advances in Computing and Information — ICCI 1990*, number 468 in Lecture Notes in Computer Science. Springer, Berlin Germany.

Wiberg, N. (1996). *Codes and Decoding on General Graphs.* Department of Electrical Engineering, Linköping University, Linköping Sweden. Doctoral dissertation.

Wiberg, N., Loeliger, H.-A., and Kötter, R. (1995). Codes and iterative decoding on general graphs. *European Transactions on Telecommunications*, 6:513–525.

Wicker, S. (1995). *Error Control Systems for Digital Communications and Storage*. Prentice-Hall Inc., Englewood Cliffs NJ.

Witten, I. H., Neal, R. M., and Cleary, J. G. (1987). Arithmetic coding for data compression. *Communications of the ACM*, 30:520–540.

Woods, W. A. (1975). What's in a link? Foundations for semantic networks. In Bobrow, D. and Collins, A., editors, *Representation and understanding*, pages 35–72. Academic Press, New York NY.

Wright, S. (1921). Correlation and causation. *Journal of Agricultural Research*, 20:557–585.

Zemel, R. E. (1993). *A minimum description length framework for unsupervised learning*. Department of Computer Science, University of Toronto, Toronto Canada. Doctoral dissertation.

Zhang, J. (1993). The mean field theory in EM procedures for blind Markov random field image restoration. *IEEE Transactions on Image Processing*, 2:27–40.