10-704: Information Processing and Learning **Spring 2015**

Lecture 25: LDPC Codes and Belief Propagation

Lecturer: Akshay Krishnamurthy *Scribe: Akshay Krishnamurthy*

Note: *LaTeX template courtesy of UC Berkeley EECS dept.*

Disclaimer: *These notes have not been subjected to the usual scrutiny reserved for formal publications. They may be distributed outside this class only with the permission of the Instructor.*

In channel coding, we want to send a message x over a noisy channel. Typically we add some redundancy to the message so that a receiver can reliably decode the message even in the presence of noise. For this lecture, we will send a message $x \in \{0,1\}^k$ over a channel by mapping it to a *codeword* $w \in \{0,1\}^n$. This codeword is sent over a channel and the reciever observes $y \in \{0,1\}^n$. The schematic is:

$$X \overset{\text{Encoding}}{\to} W \to \text{Channel} \to Y \overset{\text{Decoding}}{\to} \hat{X}$$

Since $w \in \{0,1\}^n$ and $x \in \{0,1\}^k$, this is a rate $R = k/n$ code.

25.1 Coding Schemes

Here we will briefly discuss some coding schemes.

25.1.1 Random Codes

Random codes are implicit in Shannon's proof of the Channel Capacity Theorem. We pick 2^k vectors $w_1, \ldots, w_{2^k}$ at random from $\{0,1\}^n$ and use these as the codewords. Encoding involves mapping x to w_x where x is interpreted as an integer in $[2^k]$. Decoding is based on maximum likelihood decoding, we find the codeword x that maximizes $\mathbb{P}[y|w = w_x]$, which involves the channel characteristics.

The nice thing about this coding scheme is that it is known to achieve capacity. There are two downsides however. First, there is no structure to the codebook, so both the sender and the receiver must store all 2^k words in the codebook, which is often intractable from a space perspective. Secondly, decoding is known to be NP-hard in general, so there is also computational complexity.

25.1.2 Linear Codes

In a linear code, we pick a k-dimensional subspace of $\{0,1\}^n$ (Treated as a finite field) and use all of the 2^k vectors in that subspace as the codebook. Let $z_1, \ldots, z_k \in \{0,1\}^n$ be a basis for the subspace that we chose, then coding amounts to taking $w(x) = \sum_{i=1}^k x_i z_i$ where x_i is the ith bit of the input x. This is just taking linear combinations. Decoding is again based on the maximum likelihood approach, we find the vector in the subspace that is most likely to have generated the observation y.

Random linear codes (where the subspace is chosen at random) are also known to achieve capacity. The advantage over random codes is that there is a lot of structure to the codebook, so rather than store all of the 2^k codewords, we can just store the basis vectors $z_1, \ldots, z_k$ and generate the codewords as we need to. However the decoding procedure is still known to be NP-hard, so there is still some difficulty.

25.1.3 Low-Density Parity-Check (LDPC) Codes

LDPC codes attack this problem from the other angle. Instead of building a code that tries to achieve capacity, we instead build a code for which decoding is fairly simple.

Another way to specify a linear code is as the null space of a full row-rank matrix $H \in \{0,1\}^{n-k \times n}$. That is, the codewords are all of the vectors w such that $Hw = 0$. This is known as a parity check code, since each row of h is saying that the parity of some bits of w must be zero. H is known as the parity check matrix for the code. If the ith row of h is $(1,0,0,1,1)$ then the parity check is $h_i(w) = w_1 \oplus w_4 \oplus w_5$ and the linear system demands that this is zero for any codeword.

This is a linear code, just expressed in a different way, so encoding is the same as before. We can perform gaussian elimination to identify a basis for the null space of H and encode x by taking the linear combination with this basis.

For decoding, it is useful to think of a graphical representation of the parity check matrix. We form a bipartite graph, where nodes on one side corresponds to bits in the codeword and nodes on the other side correspond to the parity check equations h_i. We place an edge between a codeword bit and a parity check equation if that bit is involved in the check equation.

Decoding is based on a message passing algorithm on this graph. It is easiest to see what is going on in the binary erasure channel.

25.1.3.1 Binary Erasure Channel

In the BEC, each bit in the y is erased with some probability p, and there are no other corruptions. So we might receive $y = w_1, \star, w_3, w4, \star$, meaning that we lost both the second and fifth bits of the message. The message passing algorithm is fairly straightforward here. The algorithm looks like:

1. If there are no unknown bits, we are done, and have successfully decoded.
2. If there is a parity check equation with only one unknown bit, we can correctly determine the value of that bit. This is a just a single equation in one variable, so we it can be solved. Fill in the value for that bit and go back to step 1.
3. If all remaining parity check equations have at least two unknown bits, then we cannot hope to recover the true message, so just guess among any satisfying assignments.

Another way to think about the algorithm is the following, in the graph, each code word variable sends its value to each parity check variable. Each parity check variable uses all of this information to figure out assignments for unknown variables and sends these assignments back to the variables. The variables update their assignments and repeat the process.

As a small comment on theory, you can show that with appropriate degree distributions (you want parity checks to have low degrees), and with enough parity check equations, this algorithm will succeed with high probability.

25.1.3.2 Binary Symmetric Channel

In the binary symmetric channel (BSC), each bit y is flipped with some probability q. Decoding is done by a softer version of the algorithm for the BEC.

Each codeword variable w_i will send a message $m_{w_i \to h_j}$ to each parity check equation h_j that it is involved in. The message will be two numbers, reflecting the probability that w_i is 0 or 1, conditioned on some information from the rest of the network. Each parity check equation h_j will the send a message $m_{h_j \to w_i}$ to each codeword variable w_j. These messages are also two numbers, reflecting the probability that w_i is 0 or 1 based on this parity check equation.

The algorithm is iterative, and we initialize the algorithm with:

$$m^{(0)}_{w_i \to h_j}(0) = P(w_i = 0 | y_i) = \begin{cases} q \text{ if } y_i = 1 \\ (1-q) \text{ if } y_i = 0 \end{cases}$$

$$m^{(0)}_{w_i \to h_j}(1) = P(w_i = 1 | y_i) = \begin{cases} q \text{ if } y_i = 0 \\ (1-q) \text{ if } y_i = 1 \end{cases}$$

These are just the probabilities that w_i is one or zero based on the channel distribution and the observation y_i.

So these messages are passed to the parity check variables and used to compute messages to send back to the variables. Computing these messages involves a short calculation. Assuming x_1 and x_2 are independent random variables, we can compute the distribution of the random variable $x_1 \oplus x_2$ by:

$$\begin{aligned} 2P[x_1 \oplus x_2 = 0] - 1 &= 2P[x_1 = 1 \wedge x_2 = 1] + 2P[x_1 = 0 \wedge x_2 = 0] - 1 \\ &= 2P[x_1 = 1]P[x_2 = 1] + 2P[x_1 = 0]P[x_2 = 0] - 1 \\ &= 2P[x_1 = 1]P[x_2 = 1] + 2(1 - P[x_1 = 1])(1 - P[x_2 = 1)] - 1 \\ &= 4P[x_1 = 1]P[x_2 = 1] - 2P[x_1 = 1] - 2P[x_2 = 1] + 1 \\ &= (2P[x_1 = 1] - 1)\,(2P[x_2 = 1] - 1) \end{aligned}$$

And a similar calculation shows that:

$$2P[x_1 \oplus \cdot \oplus x_m = 0] - 1 = \prod_{l=1}^{m} (2P[x_l = 1] - 1)$$

Now if a parity check equation h_j involves a set $S \subset [n]$ of bits, we can compute the probability that this equation is satisfied. If w_i is a variable in that parity check equation, then the probability that w_i is zero is the same as the probability that $\oplus_{l \in S \setminus \{i\}} w_l$ is zero. In words, if all of the other bits sum (modulo 2) to zero, then it is should be the case that w_i is zero. Therefore the message that h_j sends to w_i is:

$$m^{(t)}_{h_j \to w_i}(0) = P[\oplus_{l \in S \setminus \{i\}} w_l = 0] = \frac{1}{2}\left(1 + \prod_{l \in S \setminus \{i\}} \left(2m^{(t)}_{w_l \to h_j}(1) - 1\right)\right)$$

And $m^{(t)}_{h_j \to w_i}(1) = 1 - m^{(t)}_{h_j \to w_i}(0)$ because these are probabilities.

For a codeword w_i, each of these messages is independent information about the assignment to w_i, so we update its probability as:

$$P[w_i = a] = P(w_i = a | y_i) \prod_{j \in \mathcal{N}(w_i)} m^{(t)}_{h_j \to w_i}(a)$$

where $\mathcal{N}(w_i)$ are the parity check equations that involve w_i. The messages for the next round of the algorithm are updated as:

$$m^{(t+1)}_{w_i \to h_j}(0) = P(w_i = 0 | y_i) \prod_{l \in \mathcal{N}(w_i) \setminus h_j} m^{(t)}_{h_l \to w_i}(0).$$

That is LDPC codes decoding algorithm. We iteratively updated the probabilities for the w_i random variables for a set number of iterations or until the probabilities become highly assymetric (close to zero or one), and then we decode based on these probabilities. Some remarks:

1. The entire algorithm can be implemented instead by using the log-likelihoods instead of the raw probabilities. The log-likelihood variant was discussed in the lecture and is more common in practice because working in the log domain avoids numerical underflow issues.

2. There is some theory for this algorithm for LDPC decoding but the arguments are fairly technical. The idea is that if the graph is randomly generated with low-degree then it looks locally tree-like, and the message passing algorithm is known to be exact for trees.

25.2 Belief Propagation and Inference in Graphical Models

The message passing decoding algorithm for LDPC codes is an instantiation of a more general algorithm, known as *belief propagation*, that is used for inference in graphical models. In this section we describe the inference problem and describe the belief propagation (BP) algorithm.

Definition 1. *A **factor graph** is a collection of random variables $X_1, \ldots, X_d$ along with factors $f_1, \ldots, f_m$ and associated subsets $S_1, \ldots, S_m \subset [d]$. Each factor f_j associates a real number with each assignment to the subset S_j of random variables, which we denote by x_{S_j}. We use lower case x to denote realizations of the random vector X, so that x_{S_j} is the realization of the random variables indexed by S_j. The factor graph specifies a probability distribution as:*

$$p(x_1, \ldots, x_d) = \prod_{j=1}^{m} f_j(x_{S_j}) \tag{25.1}$$

An example of a factor graph is the ising model for binary random variables:

$$p(x_1, \ldots, x_d) \propto \exp\left\{ \beta_1 \sum_{(i,j)\in E} x_i x_j + \beta_0 \sum_i x_i \right\}$$

This distribution can be decomposed into a set of unary and binary factors, along with a factor encoding the normalizing constant.

Important inference questions in a factor graph are:

1. How do we compute the marginal distributions $p(x_i)$ for a random variable x_i?

2. How do we compute conditional distributions $p(x_i | X_S = x_S)$, conditioned on a set of variables X_S taking a particular value x_S?

3. How do we compute the most likely assignment (conditionally and unconditionally)?

It turns out that variants of the Belief Propagation (BP) algorithm can be used to answer these questions.

There are two types of messages in BP: $\mu_{X_i \to f_j}(\cdot)$ is the unnormalized belief about X_i that is relevant to f_j

and $\mu_{f_j \to X_i}(\cdot)$ is the information that factor f_j has about assignment to X_i. The recursive updates are:

$$\mu_{X_i \to f_j}(x) = \prod_{f_k \in \mathcal{N}(X_i) \setminus \{f_j\}} \mu_{f_k \to X_i}(x) \qquad \text{Multiply probabilities from incoming factors}$$

$$\mu_{f_j \to X_i}(x) = \sum_{x_{S_j \setminus \{i\}}} f_j(X_i = x, X_{S_j \setminus \{i\}} = x_{S_j \setminus \{i\}}) \prod_{X_l \in \mathcal{N}(f_j) \setminus \{X_j\}} \mu_{X_l \to f_j}(x_l)$$

The first expression is fairly straightforward: we multiply the incoming messages from all of the factors except for the outgoing one f_j, evaluated at our choice for X_i, which is x in the expression. The second one is more complicated: we are marginalizing out all neighboring variables in the factor f_j. So we sum over all possible assignments to all other neighboring variables $X_{S_j \setminus \{i\}}$ and we evaluate the factor for the assignment (with $X_i = x$) and then take into account the incoming probabilities from the neighboring variables in the factor.

This is the general BP algorithm. We initialize by having variable nodes send $\mu_{X \to f}(\cdot) = 1$. The marginals are computed by multiplying all incoming factors for any individual variable $p(X_i = x) = \prod_{f_j \in \mathcal{N}(X_i)} \mu_{f_j \to X_i}(x)$. In a tree, if we start from the leaves, work inward and then work back outward, it is known that this algorithm is exact. In a general graph, it is more difficult to say what happens. Inference in discrete graphical models is NP-hard in general.

To condition on a random variable, say $X_j = x_j$ we fix it's value in all messages sent by that node, i.e. we always set $\mu_{X_j \to f}(x_j) = 1$ and $\mu_{X_j \to f}(x'_j) = 0$ for all $x'_j \neq x_j$. Alternatively you can condition by folding the assignment into the factors.

Some remarks:

1. This algorithm is also known as the *sum-product* algorithm. It is used to compute marginal and conditional distributions. A related algorithm known as the *max-product* algorithm can be used to compute the most-likely assignment, marginally or conditionally. To do this, replace the sum in the update equation from factors to variables with a max. This algorithm is a generalization of the Viterbi algorithm used for Hidden Markov Models.

2. In general graphs, we don't have convergence guarantees for this algorithm. One option is to make your graph look more like a tree, and this is known as the Junction-Tree approach. The high-level idea is to cluster the random variables in the graph so that graph induced on the clusters is a tree. Then treat each cluster as a single random variable and run BP on this new tree.

   The issue with this approach is that the clusters could be quite large, and the running time depends exponentially on the size of the largest cluster. This is because we have to sum over all possible assignments to all nodes in that cluster to marginalize in the factor update equation. The size of this largest cluster is known as the *tree-width* of the graph and so we can do inference efficiently on graphs with low tree-width.

3. The theory for LDPC codes is based on the fact that random, sparse, bipartite graphs are locally tree-like so we can exploit the fact that the algorithm is exact on trees.