CSC310 – Information Theory Sam Roweis

Lecture 20:

Low Density Parity Check Codes

November 20, 2006

Reminder: Hamming Codes

- Hamming Codes are *perfectly packed* linear codes which are guaranteed to correct any single-bit transmission error.
- For every integer $c \geq 2$, there is a Hamming code which encodes messages of $K = 2^c - c - 1$ bits into transmissions of length $N = 2^c - 1$. We have seen the [3,1] Hamming code (aka the repetition code) and the [7,4] code; the next code is [15,11], etc.
- To make a Hamming code of size N, we construct a $K \times N$ parity check matrix H whose N columns are the $K - bit$ binary expansions of the integers from 1 to N.
- To encode a source message $\mathbf{s}$, we compute the generator matrix G from H, and transmit $\mathbf{t} = \mathbf{s}G$.
- To decode, we use the clever trick called *syndrome decoding*.

Syndrome Decoding for Hamming Codes

- Consider the original (non-systematic) parity-check matrix:

$$H = \begin{pmatrix} 0\,0\,0\,1\,1\,1\,1 \\ 0\,1\,1\,0\,0\,1\,1 \\ 1\,0\,1\,0\,1\,0\,1 \end{pmatrix}$$

- Suppose $\mathbf{t}$ is sent, but $\mathbf{r} = \mathbf{t} + \mathbf{n}$ is received ($\mathbf{n}$ is channel noise).
- The receiver can compute the *syndrome* for $\mathbf{r}$:

$$\mathbf{z} = \mathbf{r}H^T = (\mathbf{t}+\mathbf{n})H^T = \mathbf{t}H^T + \mathbf{n}H^T = \mathbf{n}H^T$$

  Note that $\mathbf{t}H^T = \vec{0}$ since $\mathbf{t}$ is a codeword.
- If there were no errors, $\mathbf{n} = \vec{0}$, so $\mathbf{z} = \vec{0}$.
- If there is one error, in position i, then $\mathbf{n}H^T$ will be the ith column of H — which contains the binary representation of the number i!
- So to decode, we compute the syndrome, and if it is non-zero, we flip the bit it identifies. Easy!

Syndrome Decoding by Table Lookup

- For any linear code with parity-check matrix H, we start decoding by computing the syndrome $\mathbf{z} = \mathbf{r}H^T$, from the received block $\mathbf{r}$.
- If we received $\mathbf{r} = \mathbf{t} + \mathbf{n}$, where $\mathbf{t}$ is the transmitted codeword and $\mathbf{n}$ is the *noise pattern*, then $\mathbf{z} = \mathbf{n}H^T$ (since $\mathbf{t}H^T = \vec{0}$).
- Nearest-neighbor decoding involves finding a noise pattern, $\mathbf{n}$, that produces the observed syndrome $\mathbf{z}$ (and which has the smallest possible weight). Then we decode $\mathbf{r}$ as $\mathbf{r} - \mathbf{n}$.
- To do this, we build a table indexed by the syndrome $\mathbf{z}$ that gives the noise pattern $\mathbf{n}$ of minimum weight for each syndrome.
- Initialize all entries in the table to be empty. Then consider the non-zero noise patterns, $\mathbf{n}$, in some order of non-decreasing weight. For each $\mathbf{n}$, we compute the syndrome, $\mathbf{z} = \mathbf{n}H^T$, and store $\mathbf{n}$ in the entry indexed by $\mathbf{z}$, *provided* this entry is currently empty. Stop when the table has no empty entries left to fill.

Example: Syndrome Decoding for the $[5,2]$ Code

- Recall the $[5,2]$ code with this parity-check matrix:

$$\begin{pmatrix} 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 \end{pmatrix}$$

- Here is a syndrome decoding table for this code:

$\mathbf{z}$	$\mathbf{n}$
001	00001
010	00010
011	00100
100	01000
101	10000
110	10100
111	01100

The last two entries are not unique, ie there are multiple noise vectors of weight 2 corresponding to each of those syndromes.

Decoding in General

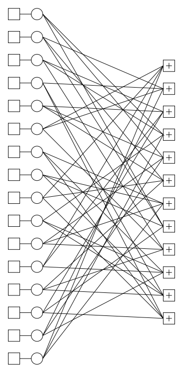
- Syndrome decoding for general linear codes (other than Hamming codes) requires building a table of all possible syndromes and doing reverse lookup to find the best codeword.
- Problem: The size of the table is exponential in the number of check bits — it has $2^{N-K} - 1$ entries for an $[N,K]$ code.
- Maximum Likelihood/Nearest Neighbour decoding is even more expensive in general, and requires solving a linear system.
- For example, in an $[N,K]$ code, decoding for the BEC requires solving a system of $N-K$ equations and will take time proportional to $(N-K)^3$ using the most obvious algorithm. This time can be reduced somewhat by using cleverer algorithms, but for values of $N-K$ in the thousands, the time could still be quite substantial.
- Can we find a class of codes which is both compact to represent, easy to encode *and* easy to decode?

What a Good Linear Code Looks Like

- We don't want our linear codes to have very low-weight code words, because this means they have very small minimum distance.
- So the generator matrix for a good code should not be sparse — each row should have many 1s, so that encoding a message with only a few 1s still produces a codeword that has many 1s.
- The decoder's perspective: To be confident of decoding correctly, getting even *one* bit wrong should produce a large change in the codeword, which will be noticeable (unless we're very unlucky).
- Thus we should avoid sparse generator matrices. But can we use a sparse parity-check matrix? Doing so isn't *quite* optimal, but such "Low Density Parity Check" (LDPC) codes can be very good.
- **The big advantage of LDPC codes:**
  There is a *computationally feasible* way of decoding them that is very good, though not exactly optimal.

Building Low Density Parity Check Codes

- LDPC codes are built using sparse random parity check matrices.
- We can construct LDPC codes randomly, in various ways.
  One way: to make an $[N,K]$ code, randomly generate columns of H with exactly three 1s in them.
- For better results, equalize the number of 1s in each row (as much as possible) by randomly picking the position of the three 1s in the next column from among rows that don't already have $3N/(N-K)$ 1s in them.
- If each row (as well as each column) has exactly the same number of 1s, the code is called a *regular Gallager code*, after its inventor (1961).



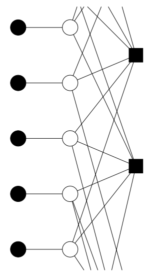
Example: A $[50, 25]$ LDPC Code

Here's the parity-check matrix for a small LDPC code (three 1s in each column, six in each row and a systematic generator matrix obtained from the parity-check matrix (with columns re-ordered):

```
0011000001010000000000000001000000000000000000001000     10000000000000000000000000110000011111011101001010
0000000000000000000101000000100001001000000010000        01000000000000000000000001001010101001101110111000
0010101000000000000010001000000100000000000000000000     00100000000000000000000000111001100110000100111000
0000010100000000010000000000000000000000110010000000     00010000000000000000000000001011010111111110010010
0000000001000000000100000000000110000000000001000010     00001000000000000000000001010011000110001000100001
0000010000001000000001000000000000100000000000001010     00000100000000000000000000111000001111010001011011
0000000000010100010000000000000000010010000001000000     00000010000000000000000001111010111111010001011011
0100001000010000001000000000000000010000000000000001     00000001000000000000000001011011011011100010011011
0000000000000000110100000000000100000000100001000000     00000000100000000000000000100101000110011001000100
0100000000001000000000000110010000000000001000000000     00000000010000000000000001001011001111010000110000
0000000000101010000000000000000000000000011000010000     00000000001000000000000001000010101001100110100011
0000000000000000000010000000001000010000000000001101     00000000000100000000000000100001000000000000001101
0001000000000000100000001100100000000000100000000000     00000000000010000000000000011000000000000000110010
1000000100000000100010000000001010000000000000000000     00000000000001000000000000101001010000010011010011
0001001000100000000000000010000001010000000000000000     00000000000000100000000000010011001101011100000011
0000000100000100000001000000000001000000000011000000     00000000000000010000000000111101100100011101001001
1000000001000010000000000000001000000000000000000101     00000000000000001000000000100000100110101010100000
0000000000000000000100010001000010000100000010000000     00000000000000000100000001010010100110001001100011
0010100010000000010000000000000000000100001000000000     00000000000000000010000000111011100000000110001011
1000000000100000000000000000001000000100100000000100     00000000000000000001000000110001001100110011001100
0000000010000000000000110000001000000011000000000000     00000000000000000000100001110010000010101000101000
0000010000000000100110000010000000000000001000000000     00000000000000000000010000000110011110101000010000
0000000010000110000000000010000000001000000000100000     00000000000000000000001000111101000101101100000000
0100000000000000000001001000000000000000001100100000     00000000000000000000000100110010000011101011111100
0000100000000000000000000000000110000000000000110010     00000000000000000000000011001110100000101110101000
```

Decoding LDPC Codes

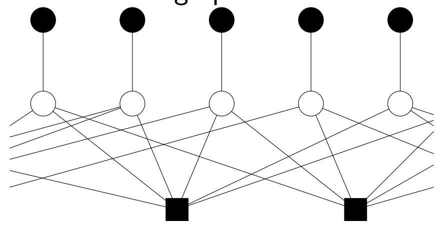
- To encode a message with an LDPC, we just multiply it by the generator matrix. But how do we decode?
- The optimal method is to do maximum likelihood decoding, which often reduces to nearest neighbour decoding, i.e. picking the codeword nearest to what was received.
- But both maximum likelihood and nearest neighbour are computationally infeasible in general
- The reason LDPC codes are interesting is that the sparseness of their parity-check matrices allows for an *approximate* (good, but not optimal) decoding method that works by *propagating messages* through a graph.



Graphical Representation of a Code

- We can represent a code by a graph:
  - – Empty circles represent true bits of the original codeword.
  - – Black circles represent received bits (message + check).
  - – Black squares represent parity check equations.

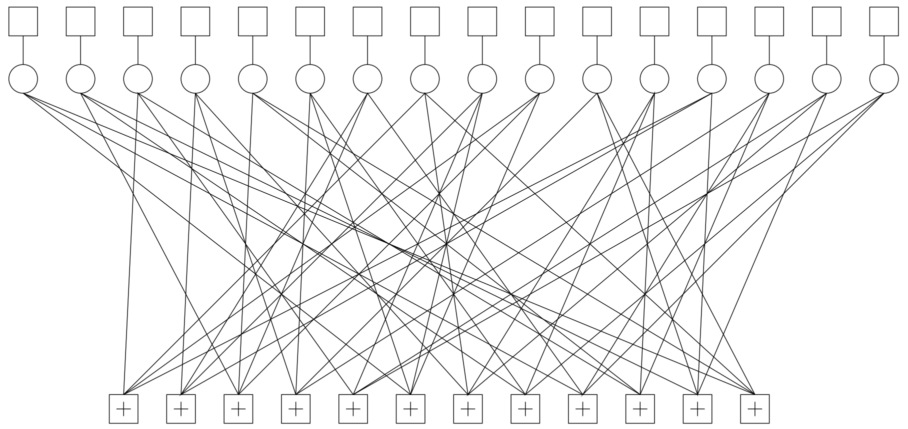
  Here's a fragment of such a graph:



Notice that each codeword bit connects to three parity checks — corresponding to the three 1s in each column of H.
Each parity check connects to six codeword bits.
Our task: Fill in the empty circles.

A Larger Example

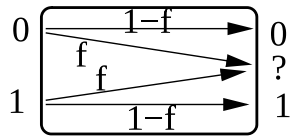
Here's the graph for a [16,12] code.



This is called the *Tanner Graph* for the code. The nodes at the top are the variable nodes and the nodes at the bottom are the check nodes.

Decoding LDPC codes over the BEC

- Let's consider decoding a LDPC code when the transmission uses a Binary Erasure Channel (BEC).
- Reminder: for the BEC, the input alphabet is $\{0, 1\}$, but the output alphabet is $\{0, ?, 1\}$. The "?" output represents an "erasure" (corruption), in which the transmitted symbol is lost, but the receiver knows it was lost. (eg error correcting memory)
- An erasure happens with probability f; otherwise, the symbol is received correctly.

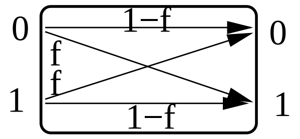


Decoding by Passing Messages

- If the correct value for a bit in a codeword is known — either because it was received correctly through the channel, or because its value has been determined in the decoding process — this bit in the codeword sends a message to all parity checks of which it is a part, telling these parity checks what its value is.
- A parity check waits until it has received messages from all but one of the bits that participate in the parity check, at which point it can send a message to the remaining bit, telling it that its value must be the value needed for the parity check to come out correctly (given the values of the other bits, which are now known).
- This process of exchanging messages continues until no further messages can be sent. At this point, the codeword bits may all be determined, in which case decoding was successful, or some codeword bits may still be unknown, in which case decoding was not completely successful (though some of the erased bits may have been filled in with their correct values).

Decoding LDPC codes over the BSC

- A harder problem is decoding a LDPC code when the transmission uses a Binary Symmetric Channel (BSC), since there are no bits of the codeword we are certain about.
- For the BSC, the input and output alphabets are both $\{0, 1\}$.
- With probability f, the symbol received is different from the symbol transmitted. With probability $1 - f$, the symbol is received correctly.



- In this case, we can still do iterative decoding by passing messages, but the messages have to represent *soft decisions* or *probabilities* rather than hard decisions as before.

Decoding by Propagating Probabilities

- We can't be absolutely sure of the codeword bits, but we can keep track of the *odds* in favour of 1 over 0 (the ratio of the probability of 1 over the probability of 0).
- Each black node will send each codeword bit it connects to a message giving its idea of what the odds for 1 over 0 should be for that bit.
- All the messages a codeword receives are multiplied to give the current idea of what the odds are for that bit — used to guess the codeword once these odds have stabilized.
- But first, we iterate: Each codeword bit sends each parity check it connects to a message with its current odds, which the parity check node uses to update its messages to other codeword bits. Messages propagate until the odds have stabilized.

Details of the Messages

- **Received data bit to codeword bit:** For a BSC, odds sent are $(1-f)/f$ if the received data is 1, $f/(1-f)$ if the received data is 0. (For a BEC, the odds are either 0, 1, or ∞, which produces the simple message passing algorithm used in the last assignment.)
- **Parity check to codeword bit:** Message is the probability of the parity check being satisfied if that bit is 1, divided by the probability if that bit is 0. These probabilities are calculated based on that parity check's idea of the odds for the *other* bits in the parity check being 1 versus 0.
- **Codeword bit to a parity check:** Message is the odds of the bit being 1 versus 0, based on the received data, and on the messages from the *other* parity checks the codeword bit is involved in.

Avoiding Double-Counting Information

- Messages send between codeword bits and parity checks exclude information obtained from the node the message is being sent to. This avoids undesirable "double-counting" of information when a message comes back from that node.
- **But:** This works perfectly only if the graph is a tree. If there are cycles in the graph, information can return to its source indirectly.
- This is why probability propagation is only an *approximate* decoding method. It works well up to a point, but doesn't have as low an error rate as nearest-neighbor (maximum likelihood) decoding would achieve.

Performance of LDPC Codes

- Rate $1/2$ LDPC codes with three bits in each column of H, with varying codeword lengths, tested using a BSC with varying error probability, f, and hence capacity, $C = 1 - H_2(f)$.
- Here are the block error rates for three such codes, estimated from 1000 simulated messages:

f	C	$[100, 50]$	$[1000, 500]$	$[10000, 5000]$
0.02	0.86	0.000	0.000	0.000
0.03	0.81	0.012	0.000	0.000
0.04	0.76	0.059	0.000	0.000
0.05	0.71	0.108	0.000	0.000
0.06	0.67	0.213	0.005	0.000
0.07	0.63	0.327	0.104	0.000
0.08	0.60	0.482	0.404	0.125

Tests were done with software available from Radford Neal's web page, http://www.cs.utoronto.ca/~radford/.

History of LDPC and Related Codes

- – Gallager, LDPC codes — 1961.
  True merits not realized? Computers too slow? Largely ignored and forgotten.
  – Berrou, *et al*, TURBO codes — 1993.
  Surprisingly good codes, practically decodable, but not really understood.
  – MacKay and Neal — 1995.
  Reinvent LDPC codes, slightly improved. Show they're almost as good as TURBO codes. Decoding algorithm related to other probabilistic inference methods.
  – Many (Richardson, Frey, etc.) — ongoing.
  Further improvements in LDPC codes, relationship to TURBO codes, theory of why it all works.