Lecture 17, Friday w9, 2014-11-14

The noisy channel coding theorem proof imagined generating a large collection of random codewords. How do practical state-of-the art codes work? This lecture aimed to give a sketch.

Firstly, codes *don't* work, by identifying roughly how may bits we'll need to correct (e.g., Nf for the BSC) and then creating a code with *distance* twice that number of bits. It's not possible to approach the Shannon limit that way (as argued in the slides, and MacKay p212–).

Reed–Solomon codes: I would have been negligent not to mention these widely used codes. However I just gave a sketch of an idea behind them (final slide week7.pdf). I can't think of an exam question I could ask, given how little I said about them. The course text only mentions them (e.g., p185 and p589). There's more on https://en.wikipedia.org/wiki/Reed_Solomon which says: *"Today, RS codes are widely implemented in digital storage devices and digital communication standards, though they are being slowly replaced by more modern low-density parity-check (LDPC) codes or turbo codes."*

LDPC codes: Low density parity check codes. These codes use large block lengths, and many parity checks, but each only involving a few of the transmitted bits. A *graphical model* called a *factor graph* or a *Tanner graph* describes the posterior distribution over transmitted bits given received bits. This posterior distribution is intractable, so an approximate inference algorithm, also used in machine learning, called loopy-belief propagation (or the sum-product algorithm) is used. The sum-product algorithm works when the connections are low-density. The low-density connections also make codeword construction from a parity check matrix practical.

Factor graphs: Each factor (a square in the diagram) has a positive function of the variables (round nodes) that it is connected to. The joint probability of the variables is proportional to the product of the factors: $P(\mathbf{x}) = \frac{1}{Z}\prod_k f_k(\mathbf{x}_{C_k})$, where C_k is the group of variables attached to factor k. The LDPC code has factors encoding the prior: one for each parity check, each stating that a group of variables must sum to zero modulo 2. There are also factors for the likelihood: one connected to each bit, biasing it towards the bit that was observed. The product of factors, proportional to prior times likelihood, gives a function proportional to the posterior over the transmitted bits.

Belief propagation: Each variable receives messages from attached factors, giving beliefs about the setting of the variable. The normalized product of these messages is an approximation (only correct if the graph is a tree) to the marginal distribution $P(x_n)$.

The variable sends its current beliefs about its setting to the attached factors, but removing the part of these beliefs that came from the factor that is receiving the message. The message from a variable says "this is what other factors think I should be".

The factors send messages to variables that result from a more complicated sum-product operation (giving the algorithm its name). The factor tells a variable how much it wants it to take on a particular setting. The factor considers how well the variable setting would fit in with each of the possible settings of the attached variables (the sum). Each term in the sum is weighted by how plausible these settings are given the beliefs of the other variables and the factor's function (a product of beliefs).

If you're keen (i.e., you can take it on trust if you like) you can show this algorithm is exact on a tree. For a chain of variables it results in the well-known forward-backward algorithm for hidden Markov models, which you may have seen in another course.

The use of the sum-product algorithm is a little weird. We're not really interested in the probabilities of the bits. We want to optimize, to return a single bit pattern that fits all the parity checks that seems most plausible given the data. The natural approximate algorithm for finding the most probable setting of the bits is *max*-product (called *Viterbi* for a hidden Markov model if you've seen that), but sum-product works better on LDPC's loopy graphs.

Check your understanding

In an early tutorial we estimated that repetition codes would need a block length of around $N = 60$ to get a low probability of error of around 10^{-15}. LDPC codes can get close to the limits given by the noisy channel coding theorem. So how would their rate compare to the repetition code? Would LDPC codes need a smaller or larger block length than a repetition code to achieve the same probability of error?

Why are the parity checks "low-density", when a uniformly random linear code (where each variable has a half chance of connecting to each factor) gives very good codes for some channels? Similarly, why did digital fountain codes only involve a few source packets in each transmitted packet?

Can you give an example of a distribution where maximizing the marginal probabilities of each variable doesn't give a probable joint setting of the variables? Any ideas why maximizing marginals might be ok for LDPC codes?

In sum-product, what would happen if the messages didn't remove the part of the local belief that came from the receiver of the message? Looking at an example with one variable and one factor may help.

Extra reading

My exposition of the sum-product algorithm was based on MacKay p336.

MacKay's treatment of LDPC codes starts on p556. However, I'll only expect knowledge at the level given by the brief overview given in class. The Chapter gives practical details for making sum-product fast in this context, and for creating a generator matrix, so we can make codewords given a check matrix. The references at the end include some history.