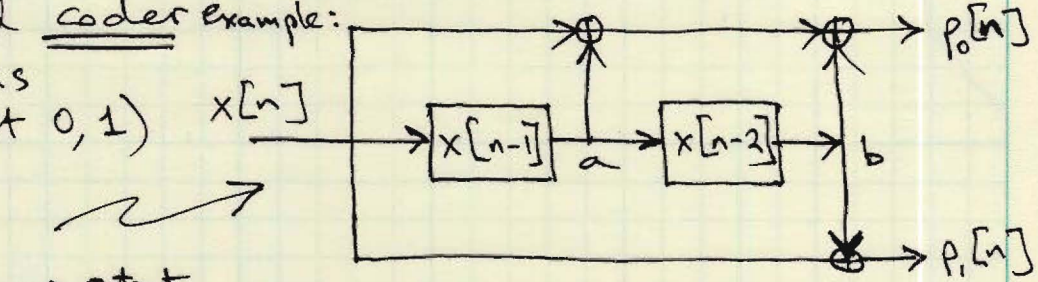
George Verghese
verghese@mit.edu

6.02
10/12/10

Rec. 9 (plus solutions to Rec. 8 questions)

Convolutional coder example: (assume signals in $\mathbb{F}_2$, i.e., just 0, 1)



Note that

represents one stage of a shift register — whatever value is at the input arrow at time n gets shifted into the register at time $n+1$, and is accessed at the output at that time.

So in the above block diagram, the value $x[n-1]$ is stored in the first element at time n, and is also the value seen at point 'a' of the figure. Similarly, $x[n-2]$ is stored in the second element at time n, and is also the value seen at point 'b' of the figure. At each clock cycle, the values simply shift along the register (to the right, in the above figure).

The 'parity' outputs (for this example) are defined by

$$p_0[n] = x[n] \oplus x[n-1] \oplus x[n-2]$$
$$p_1[n] = x[n] \oplus x[n-2]$$

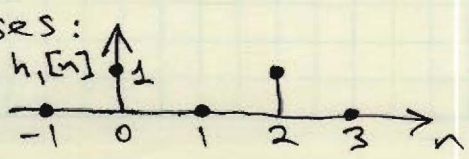
} LTI relations in $\mathbb{F}_2$

For every bit $x[n]$ at the input, the coder puts out two, namely $p_0[n]$, $p_1[n]$. So the rate of this code is 1/2.

What's 'convolutional' about the coder? → it's an LTI system, so $p_0[n]$ and $p_1[n]$ are found by convolving the input stream with the associated unit sample responses:

($\mathbb{F}_2$)

and



A nice text for further reading: 'Understanding Information Transmission' by J. B. Anderson & R. Johannesson

The constraint length of the coder is the time interval of input $x[\cdot]$ values that is involved in generating the parity outputs at any time. In our example: 3.

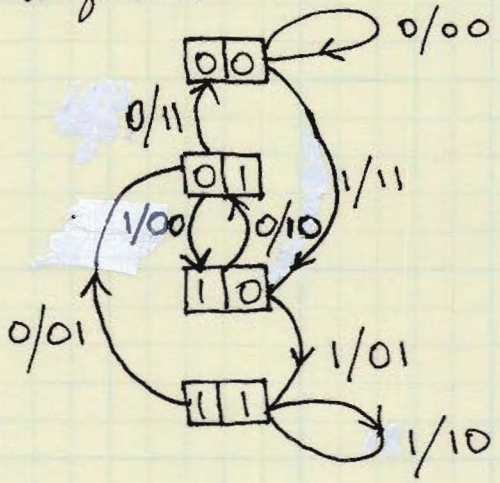
The outputs of the coder at time n are determined by the input $x[n]$ at that time, as well as the state of the coder. So evidently the state at n comprises the values in the registers at n: $x[n-1]$, $x[n-2]$ = 0 0 or 0 1 or 1 0 or 1 1.

State transition from n to $n+1$:

If the state is 0 0 and the input $x[n]$ is 0, then the parity outputs are $p_0[n] = 0$, $p_1[n] = 0$, and the next state is 0 0.

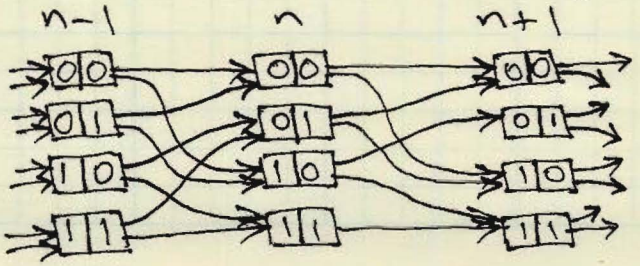
If the state is 0 0 and the input $x[n] = 1$, then $p_0[n] = 1$, $p_1[n] = 1$, and the next state is 1 0.

Continue similarly, to obtain the following diagram, where the labels on the arcs show $x[n]/p_0[n], p_1[n]$ associated with the indicated state transitions on the diagram: (from time n to $n+1$)



(This is arranged a little differently than in the lecture notes — to make the relation to the trellis a little more explicit.)

The associated trellis just stretches the state transition picture out along the time axis:

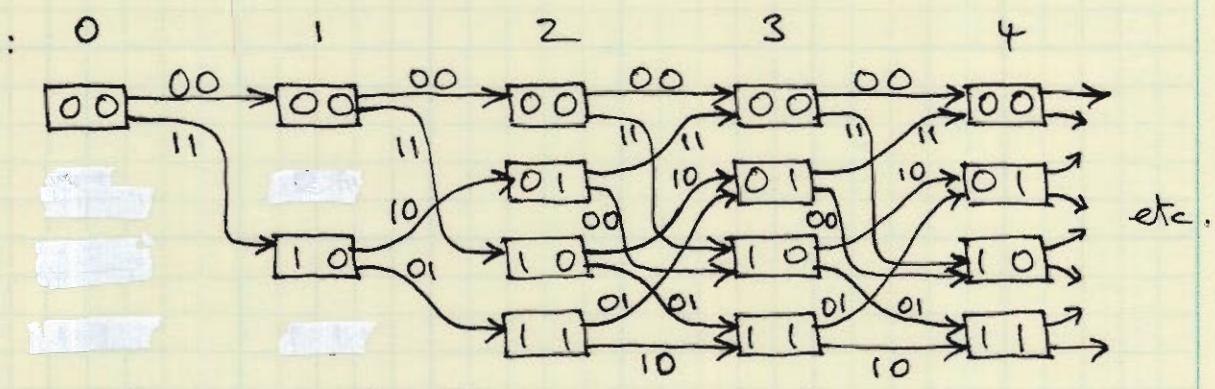


Every state box has two outputs, corresponding to the two states it can get to, depending on whether $x[n]=0$ or $x[n]=1$. The way we have arranged the states (i.e., binary counting order), the upper transition is always for $x[n]=0$, the lower for $x[n]=1$. So we shall drop $x[n]$ from the $x[n]/p_0[n],p_1[n]$ label on the state transition arcs → just label $p_0[n],p_1[n]$.

(Every state box also has two inputs, corresponding to the two choices for the bit that disappears off the end of the shift register.)

The associated trellis in more detail:

For concreteness, assume we start in state 0 0 — can be ensured by starting off with 0,0 in the input stream $x[\cdot]$; and also that we end in 0 0, ensured by ending with 0,0 in the input stream. May have thousands of message bits in between.



Recap: Every arc is labeled with the parity outputs transmitted when/if that arc is traversed.

{ depends on state and input at that time

The set of codewords is simply the set of strings of parity outputs obtainable by taking all possible routes through the trellis. This set is linear, i.e., $\mathbb{F}_2$-sum ($\oplus$) of two codewords is a codeword — simplest way to see this is note that the codewords are the outputs of an LTI system, with the corresponding inputs ranging over all possibilities (between the 0,0 at the start, and the other 0,0 at the end).

Since the code is linear, the minimum Hamming distance (HD) between two distinct codewords is the same as the min. distance between a nonzero codeword and the zero codeword (0), or equivalently the minimum weight among all nonzero codewords. Some exploration of the above trellis shows that this minimum HD = 5 (corresponding for example to the codeword 11 10 11 00 00 ..., which has weight 5). The interpretation of this in the context of convolutional codes is more subtle than for block codes. Roughly, though, a "burst" of ≤ 2 errors in a stretch of several bits of codeword (14 or so?) can be corrected, or ≤ 4 errors can be detected. (Bigger constraint length ⇒ bigger state space ⇒ greater min. HD possible)

Decoding a received codeword string:

Every uncorrupted codeword is easily decoded, because it corresponds to a unique path through the trellis, and whether we move up or down at step n indicates whether $x[n] = 0$ or $x[n] = 1$, respectively. For instance, receiving 11 10 00 10 ... would correspond to $x[n] = 1\ 0\ 1\ 0 \ldots$

Actually, in the noise free case we can do a more simple exact deconvolution: note for our case that $p_0[n] \oplus p_1[n] = x[n-1]$. Easy!

What about when there are errors? i.e., a value that should have been received as a 0, respectively 1, ends up being received as some other value, forcing us to make an educated guess. Two routes:

HDD: 'Hard-decision decoding': make a 'hard' decision on each received value, comparing it with a threshold to decide 0 or 1, as we have been doing so far, recognizing that there is a probability p of a bit error. Then find the codeword with minimum Hamming distance to the received sequence of 0's and 1's that results from the above thresholding.

(for any $p < \frac{1}{2}$)

This finds the most likely match to the received 0's and 1's that follow the thresholding, but (because our thresholding was a very local operation that had no regard for what was globally best) this 2-step process is suboptimum.

SDD: 'Soft-decision decoding': retain the received values as they are (apart from scaling so they would be of appropriate amplitude in the absence of noise), (i.e. height 0 or 1) with no thresholding to 0 or 1. Then find the codeword that is most likely to match this entire set of received values, for the particular noise characteristics of the channel. If the noise that causes the received samples to deviate from their ideal (0 or 1) values is Gaussian, independent from sample to sample, but of fixed variance and with mean 0, then instead of the Hamming distance metric of HDD, we use the Euclidean distance metric. More on this later.

HDD decoding in more detail:

Back to the trellis on p.③. Given some received sequence, say

00 01 01 10 01 10 ... (grouped in pairs for readability here)

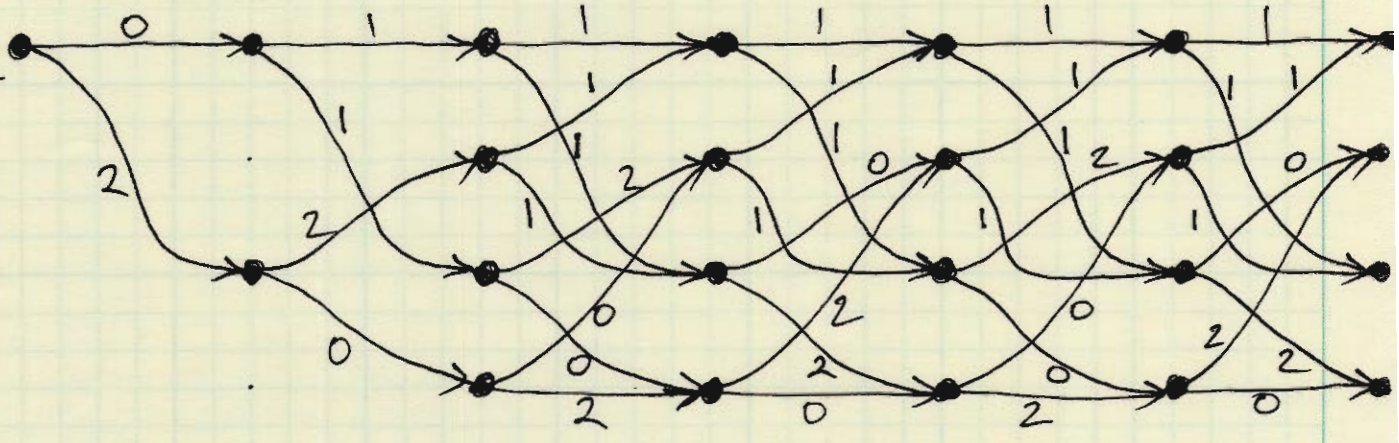
we need to know its Hamming distance (HD) from each codeword.

This is simply done: Mark each arc or branch on the trellis with the HD between its associated pair of parity bits $p_0[n]$, $p_1[n]$ and the received 0,1 - pair for that stage. For our example (refering to the trellis on p.③ and to the received 0,1 sequence listed above) we get:

Time: 0 1 2 3 4 5 6

Received pair: 00 01 01 10 01 10

Trellis, with arc weights = HD between received pair and p_0, p_1 on arc in trellis of p. (3)



So now the HD between a codeword and the received 0,1 sequence is just the sum of the branch weights (or costs, or metrics) on the arcs in the trellis path that corresponds to the codeword.

Our decoding task therefore is to find the path of minimum total weight through the above trellis. Impossibly difficult to do by enumeration, because total # of paths $= 2^L$, where L is the number of stages or time steps, which may be in the 100's or 1000's for a typical stretch of convolutional code.

Viterbi's clever idea (and also the idea behind 'dynamic programming' for a large variety of problems with such structure): Find the minimal-weight path from the start node to every state node at stage n, working from $n=1$ to the end, sequentially.

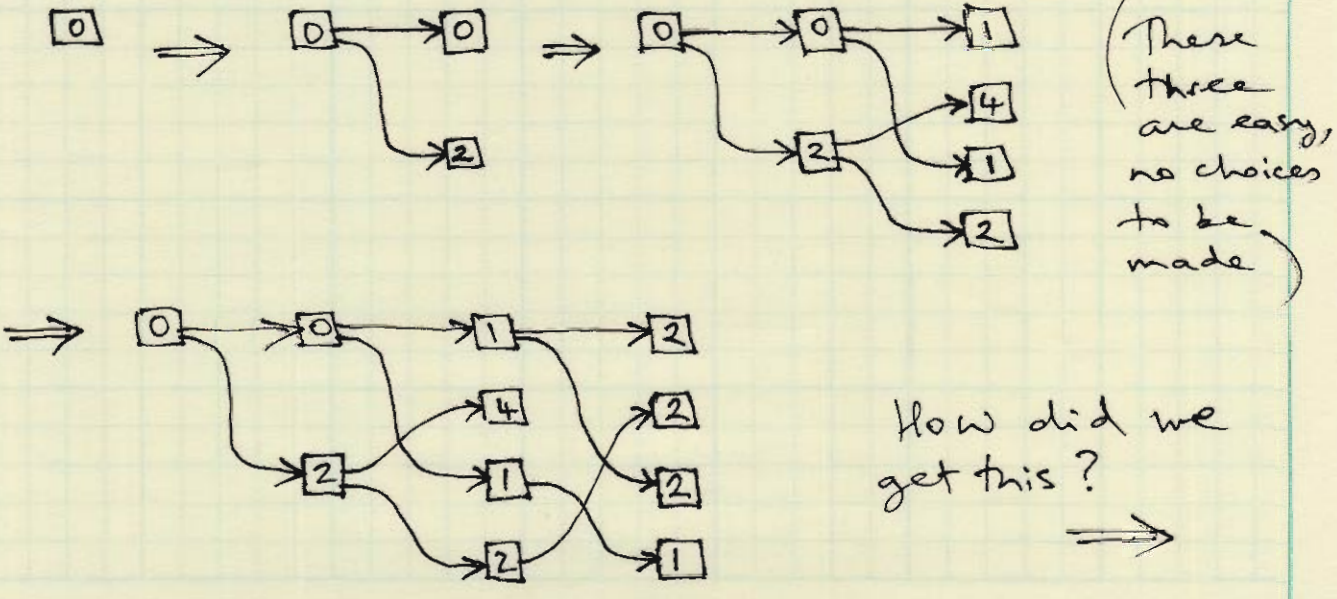
How does this help? Seems like more work! But note: We arrive at state S[n] at time n from some state S[n-1] at time n-1, and the minimum-weight path to S[n] must have first (over the first n-1 steps) used the minimum-weight path to S[n-1], because the path weights (metrics) are obtained by summing up the branch weights. We therefore only need to know the minimum-weight path to S[n-1] — all other paths (around 2^{n-1} of them!) are irrelevant!

↑ 00 or 01 or 10 or 11

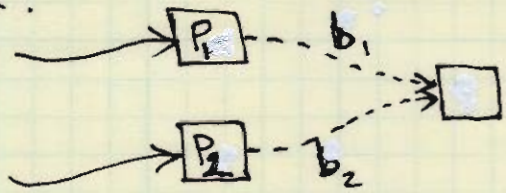
So, for a state-space of size 4, we only keep track of 4 paths as we proceed from left to right on the trellis. (Of course, larger constraint lengths — desirable for better code performance — means exponentially larger state space, and hence more work for Viterbi.)

Implementation: Redraw the trellis from p. 6, stage by stage from the left, but replacing each state node • by a box □ showing the minimum-weight cost to get to that node, and only showing the arc traversed at the previous stage to get this minimum weight:

i.e., the path metric

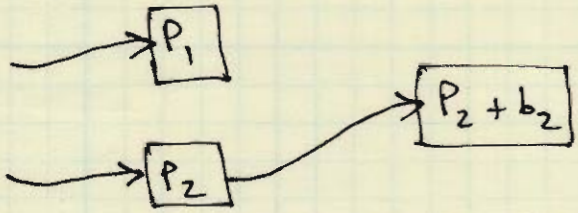


Consider a state box ☐ at stage n, after we're done up to stage n-1. There are two routes to any such box:



Stage: n-1 n

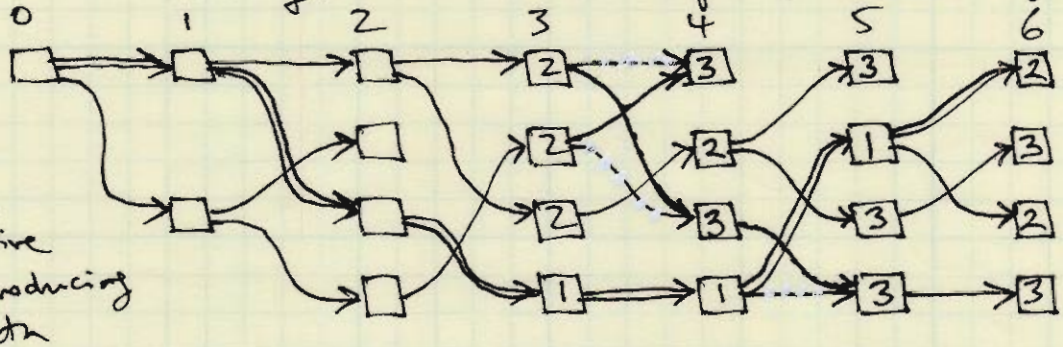
The upper left box is marked with its path metric P_1, representing the total weight (or cost) of the minimum-weight path to that box. Similarly for the lower left box. So the state box at stage n gets filled with the minimum of the two possibilities, $P_1 + b_1$ or $P_2 + b_2$, where b_1 and b_2 are the respective branch metrics. Suppose, for example, that $P_2 + b_2 < P_1 + b_1$. Our figure then becomes



(If $P_1 + b_1 = P_2 + b_2$, either can be chosen)

We repeat this for each of the 4 state boxes at stage n, then move to the next stage.

Continuing with our example for two stages further:



Dashed arcs indicate alternative choices, producing same path metric

If we knew that the state at time 6 was 010 (because the message was chosen that way, with 0 and 0 at the end), then we already see there is a unique minimum-weight path that leads to it — just trace back! (shown in ⟹ on figure above)

Our decoded message: 0 1 1 1 0 0 .

The codeword associated with this message is (from the trellis on p.(3)): 00 11 01 10 01 11
versus the received 00 01 01 10 01 10
So, assuming we have decoded correctly (i.e., assuming 011100 was actually sent), we have recovered from two bit-errors.

Back to 'soft-decision decoding' (SDD):

When the noise on a received sample is Gaussian, we have

$$-\log \text{PDF value of sample} \propto (\text{received sample value} - \text{nominal value})^2$$

('proportional to')

('(negative) log-likelihood' of sample)

smaller value of squared difference $\Leftrightarrow$ higher value of PDF

Under the SDD assumptions stated earlier, p.(5),

$$-\log \text{PDF value of string of received samples} \propto \sum_i (\text{received sample \#}i - \text{nominal value}\, i)^2$$

('negative log-likelihood' of received string of samples.)

So the process is the same as before, except that now the branch metrics are not computed via Hamming distance, but instead by the 'sum of squared deviations'. e.g. the first couple of stages of the trellis on p(6) become