Lecture 14: Error correction codes

ENGR 76 lecture notes — May 16, 2024

Ayfer Özgür, Stanford University

In this lecture we will see three important examples of error-correction codes that are widely used in practice.

Hamming code

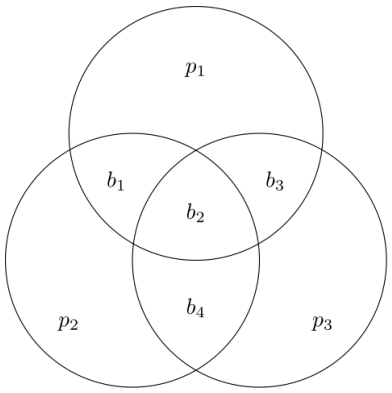
Question: What is the largest size (M) for a code of length 7 and distance 3?

To answer this question, we first look at a negative result that shows that M can only be so large. This is based on our earlier observation that the Hamming balls of radius 1 around the codewords must not intersect if we are to decode reliably. Note that we have 2^7 binary sequences of length 7, and each of them must lie in the Hamming ball of radius 1 of at most one codeword (it's fine if a sequence is not in the ball for any codeword since that will never be received given that there is only a single bit flip). Now for each codeword, the Hamming ball of radius 1 consists of 8 sequences (the original codeword along with 7 sequences that are produced by flipping one bit at each of the 7 positions). Thus we have

$$M \leq \frac{\text{Total number of sequences of length 7}}{\text{Number of sequences in each Hamming ball}}$$

which simply gives $M \leq \frac{2^7}{8} = 16$. Thus, we can have no more than 16 codewords of length 7 with distance 3. Equivalently, we can communicate at most 4 information bits in this scenario.

This type of *packing* argument, where we attempt to pack all possible sequences into distinct Hamming balls, is called the **Hamming bound**. The bound can be generalized to distance values greater than 3.



$$p_1 = b_1 \oplus b_2 \oplus b_3$$
$$p_2 = b_1 \oplus b_2 \oplus b_4$$
$$p_3 = b_2 \oplus b_3 \oplus b_4$$

Next, we define the **Hamming code** of length 7. The Hamming code is the code with largest size ($M = 16$) for length $L = 7$ and distance $d = 3$. The code was first introduced by Richard Hamming in the 1940's in the context of punch card based computation. It is still used in situations where the number of errors is quite small, e.g., in RAMs.

The codeword for a Hamming code is of the form $b_1b_2b_3b_4p_1p_2p_3$ where b_1, b_2, b_3, b_4 are the information bits and p_1, p_2, p_3 are the parity bits. The computation of the parity bits is performed to ensure that the number of `1`s in each circle is even, as shown in the figure above. For example, focusing on the top circle, the number of `1`s in p_1, b_1, b_2, b_3 must be even, which can achieved by setting p_1 to the XOR or the modulo 2 sum of b_1, b_2 and b_3. The decoding algorithm checks whether the number of ones in each circle is even or not. If all circles contain an even number of ones, the received sequence is a valid codeword, and we assume that no errors have occurred during transmission. If some of the circles do not satisfy the parity check, then the

bit in their intersection is in error and is corrected. Note that with this algorithm we can always correct a single bit flip. We did some examples of encoding and decoding in the lecture.

Recall the Hamming bound from above provides an upper bound on the size of the code for a given codeword length L and distance d. The Hamming code in fact achieves the Hamming bound with equality, i.e., $M(L+1) = 2^L$ (this is for $d = 3$). The family of Hamming codes consists of codes that satisfy this equality (with M always being a power of 2), and there are infinitely many codes in the family. Writing $M = 2^k$ where k is the number of information bits, we can manipulate the equality to the following form

$$L = 2^{L-k} - 1$$

Note that $L - k$ is the number of parity bits and based on the equation above, it grows logarithmically with L. Thus, the Hamming code has a very small fraction of bits that need to be added as parity bits, but is restricted to correcting only a single error. Next, we see another code that can be considered a dual to the Hamming code.

Hadamard code

The Hadamard code is defined via a recursive construction. We define

$$W_1 = \begin{bmatrix} 0 \end{bmatrix}$$

and

$$W_{2n} = \begin{bmatrix} W_n & W_n \\ W_n & \overline{W_n} \end{bmatrix}$$

for n a power of 2 ($\overline{W}$ is an element-wise complement of W). Some initial W matrices are shown below:

$$W_2 = \begin{bmatrix} 0 & 0 \\ 0 & 1 \end{bmatrix}$$

$$W_4 = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 \\ 0 & 1 & 1 & 0 \end{bmatrix}$$

$$W_8 = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 1 & 0 & 1 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 & 1 & 0 & 0 & 1 \end{bmatrix}$$

The code itself is defined by the rows of the matrix being the codewords. Thus, for each $r > 0$, we have a code with $M = 2^r$, $L = 2^r$ and $d = 2^{r-1}$. The distance property can be checked with a brute-force approach for W_8. Note that by construction all non-zero codewords (rows in matrix except for the first) have the same number of `1`s (Hamming weight).

Thus, the Hadamard code family consists of codes with increasing minimum distance, which is proportional to the length of the codeword. However the size of the code is the same as the length of the codeword, which means the number of information bits is logarithmic in the codeword length. Thus, the code can correct a lot of errors but has a very low rate (number of information bits per codeword bit). It is used in space communication by NASA.

Block codes

All the codes we have seen till now have been block codes. For example, the Hamming code can be thought of as a mapping from 4 information bits to 7 codeword bits, defined using the parity equations or through a lookup table. To encode a sequence of information bits with the Hamming code, we first need to chop the sequence into blocks of 4 bits and then apply the encoding to each block separately. For example, if the sequence of information bits was `101011100011`, then we would first split it into 3 blocks `1010`, `1110`, `0011`. Then we would apply the encoding to each block to get `1010011`, `1110100`, `0011110`. Finally we would concatenate the encoded blocks to get the final encoding `10100111101000011110`. Since the Hamming code is capable of correcting one bit flip, block coding in this manner can tolerate up to one bit flip per block.

Convolutional codes

Next, we look into convolutional codes, which are not block codes. Convolutional codes that are widely used in space communication and Wi-Fi. Convolutional codes are also the building blocks of more advanced codes like turbo codes, which combines two convolutional codes, and are used in space communication, 3G and 4G. We will also use convolutional codes in the audio communication project. We'll focus on a specific convolutional code in this lecture, study its encoding and view the code through a few different perspectives.

Encoding

The encoder takes a binary information sequence $b_1, b_2, \ldots, b_k \in \{0, 1\}$ and generates the encoded sequence $p_1, p_2, \ldots, p_n \in \{0, 1\}$. In general, we'll have $n > k$, for this specific code $n = 2k$ and the encoded sequence is obtained as follows:

For $j = 1, \ldots, k$:

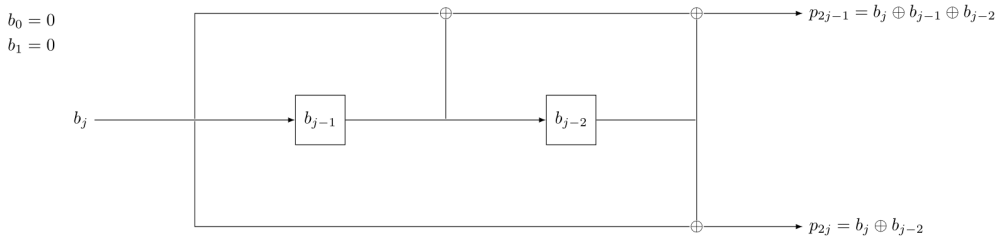
$$p_{2j-1} = b_j \oplus b_{j-1} \oplus b_{j-2}$$

$$p_{2j} = b_j \oplus b_{j-2}$$

Recall that the XOR or modulo 2 sum operation (denoted by $\oplus$) simply counts the number of `1`s and results in `0` if the number is even, and `1` otherwise. For example, `0`$\oplus$`0`=`1`$\oplus$`1`=`0` and `0`$\oplus$`1`=`1`$\oplus$`0`=`1`. The operation is commutative and associative, and hence can be computed in any order.

In the expression above, we assume that $b_0 = b_{-1} =$`0`, which can be thought of as the initial condition of the system. Note how the encoding happens in a streaming fashion - at each step j, the encoder takes b_j and produces p_{2j-1} and p_{2j} based on b_j, b_{j-1} and b_{j-2}.

Shift register view



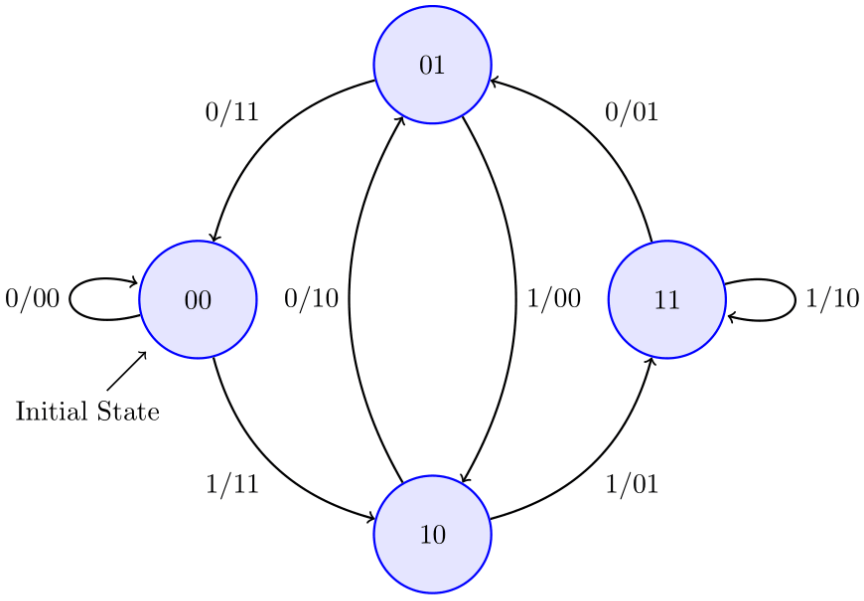
Next we see the encoding in a different way that is closely related to how the encoding works in hardware. As seen in the figure above, the shift register stores the previous two bits (b_{j-1}, b_{j-2}) in memory, and after each step the new symbol b_j pushes b_{j-1} to the right, and b_{j-2} is removed from the memory (the *shift* operation).

At step j, the outputs are computed with the $\oplus$ gate as shown in the figure. Initially, both memory elements store 0.

We can consider an example for the encoding. Say the information bit sequence is 01101. Then the encoding proceeds as follows. At step 1, we have $b_{j-1} = b_{j-2} =$0 and $b_j =$0. Thus, both output bits $p_1 = b_j \oplus b_{j-1} \oplus b_{j-2}$ and $p_2 = b_j \oplus b_{j-2}$ are 0. After the shift operation, we have $b_{j-1} = b_{j-2} =$0 and the new bit $b_j =$1 (where $j = 2$ now). Now we get $p_3 = p_4 = 1$. After the shift operation, we have $b_{j-1} =$1, $b_{j-2} =$0 and the new bit $b_j =$1 (where $j = 3$ now). Then we get $p_5 = 1 \oplus 1 \oplus 0 = 0$ and $p_6 = 1 \oplus 0 = 1$. The process continues until we are done with the input bits.

Note that the encoder produces two output bits for every input bit, and hence the information rate of the code is $\frac{1}{2}$.

State machine view



Label of each edge: $b_j / p_{2j-1}\, p_j$

The encoding process can be considered as a finite state machine, where we have multiple states, with each input bit leading to a transition from one state to another along with producing some output bits. The *state* of the convolutional is simply the memory in the shift register, i.e., state is $b_{j-1}b_{j-2}$ (note that the order of bits in the state is opposite the order of bits in the input, we use this convention to match the shift register view). The initial state is 00. The total number of states is just the number of values $b_{j-1}b_{j-2}$ can take, which is 4. The transitions are labeled with the input bit causing the transition and the output bits produced during the transition, denoted as $b_j / p_{2j-1}p_{2j}$.

Let's try to understand some of the transitions. If the current state is $b_{j-1}b_{j-2} =$ 00, and the input bit is $b_j =$ 0, then we can verify that the output bits are 00 and the next state is 00 (this is simply $b_j b_{j-1}$). This is shown as the self loop at the 00 state on the left. If the current state is $b_{j-1}b_{j-2} =$ 00, and the input bit is $b_j =$ 1, then we can verify that the output bits are 11 and the next state is 10 (this is simply $b_j b_{j-1}$). This is the arrow going from 00 to 10. Similarly the rest of the state diagram can be obtained. As we get input bits, we move from one state to another, and keep on producing output bits as shown in the diagram.