Hamming Codes

We have seen that a binary $[n, k]$ code will correct any single error if all the columns in its parity-check matrix are non-zero and distinct.

One way to achieve this: Make the $n-k$ bits in successive columns be the binary representations of the integers 1, 2, 3, etc.

This is one way to get a parity-check matrix for the $[7,4]$ Hamming code:

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

When n is a power of two minus one, the columns of H contain binary representations of all non-zero integers up to $2^{n-k}-1$. These are the binary Hamming codes.

Hamming Codes are Perfect

For each positive integer c, there is a binary Hamming code of length $n = 2^c - 1$ and dimension $k = n - c$. These codes all have minimum distance 3, and hence can correct any single error.

They are also perfect, since

$$2^n/(1+n) \;=\; 2^{2^c-1}/(1+2^c-1) \;=\; 2^{2^c-1-c} \;=\; 2^k$$

which is the number of codewords.

One consequence: A Hamming code can correct any single error, but if there is more than one error, it will not be able to give any indication of a problem — instead, it will "correct" the wrong bit, making things worse.

The *extended Hamming codes* add one more check bit (ie, they have one more row of all 1s to the parity-check matrix). This allows them to detect when two errors have occurred.

Encoding Hamming Codes

By rearranging columns, we can put the parity-check matrix for a Hamming code in systematic form. For the $[7,4]$ code, we get

$$H \;=\; \begin{pmatrix} 0 & 1 & 1 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 & 0 & 0 & 1 \end{pmatrix}$$

Recall that a systematic parity check matrix of the form $[-P^T \,|\, I_{n-k}]$ goes with a systematic generator matrix of the form $[I_k \,|\, P]$. We get

$$G \;=\; \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 \end{pmatrix}$$

We encode a message block, $\mathbf{a}$, of four bits, by computing $\mathbf{u} = \mathbf{a}G$. The first four bits of $\mathbf{u}$ are the same as $\mathbf{a}$; the remaining three bits are "check bits". Note: The order of bits may vary depending on how the code is constructed.

Decoding Hamming Codes

Consider the 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 $\mathbf{u}$ is sent, but $\mathbf{v} = \mathbf{u} + \mathbf{e}$ is received. The receiver can compute the *syndrome* for $\mathbf{v}$:

$$\mathbf{s} \;=\; \mathbf{v}H^T \;=\; (\mathbf{u}+\mathbf{e})H^T \;=\; \mathbf{u}H^T + \mathbf{e}H^T \;=\; \mathbf{e}H^T$$

Note that $\mathbf{u}H^T = \mathbf{0}$ since $\mathbf{u}$ is a codeword.

If there were no errors, $\mathbf{e} = \mathbf{0}$, so $\mathbf{s} = \mathbf{0}$.

If there is one error, in position i, then $\mathbf{e}H^T$ 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.

If we rearranged H to systematic form, we modify this procedure in corresponding fashion.

Syndrome Decoding

For any linear code with parity-check matrix H, we can find the nearest-neighbor decoding of a received block, $\mathbf{v}$, using the syndrome, $\mathbf{s} = \mathbf{v}H^T$.

We write the received data as $\mathbf{v} = \mathbf{u} + \mathbf{e}$, where $\mathbf{u}$ is the transmitted codeword, and $\mathbf{e}$ is the *error pattern*, so that $\mathbf{s} = \mathbf{e}H^T$.

A nearest-neighbor decoding can be found by finding an error pattern, $\mathbf{e}$, that produces the observed syndrome, and which has the smallest possible weight. Then we decode $\mathbf{v}$ as $\mathbf{v} - \mathbf{e}$.

Building a Syndrome Decoding Table

We can build a table indexed by the syndrome that gives the error pattern of minimum weight for each syndrome.

We initialize all entries in the table to be empty.

We then consider the non-zero error patterns, $\mathbf{e}$, in some order of non-decreasing weight. For each $\mathbf{e}$, we compute the syndrome, $\mathbf{s} = \mathbf{e}H^T$, and store $\mathbf{e}$ in the entry indexed by $\mathbf{s}$, *provided* this entry is currently empty. We stop when the table has no empty entries.

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.

Example: 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:

s	**e**
001	00001
010	00010
011	00100
100	01000
101	10000
110	10100
111	01100

The last two entries are not unique.

Product Codes

A *product code* is formed from two other codes $\mathcal{C}_1$, of length n_1, and $\mathcal{C}_2$, of length n_2. The product code has length $n_1 n_2$.

We can visualize the $n_1 n_2$ symbols of the product code as a 2D array with n_1 columns and n_2 rows.

Definition of a product code: An array is a codeword of the product code if and only if

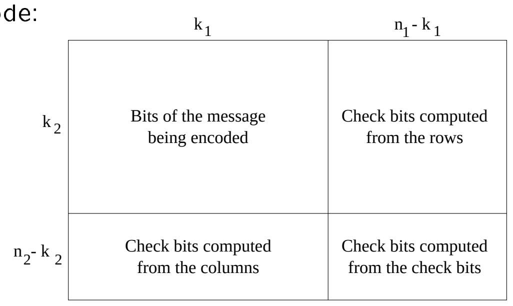
- all its rows are codewords of $\mathcal{C}_1$
- all its columns are codewords of $\mathcal{C}_2$

We will assume here that $\mathcal{C}_1$ and $\mathcal{C}_2$ are linear codes, in which case the product code is also linear. (Why?)

Dimensionality of Product Codes

Suppose $\mathcal{C}_1$ is an $[n_1, k_1]$ code and $\mathcal{C}_2$ is an $[n_2, k_2]$ code. Then their product will be an $[n_1 n_2, k_1 k_2]$ code.

Suppose $\mathcal{C}_1$ and $\mathcal{C}_2$ are in systematic form. Here's a picture a codeword of the product code:


k_1
$n_1 - k_1$
k_2
$n_2 - k_2$
Bits of the message being encoded
Check bits computed from the rows
Check bits computed from the columns
Check bits computed from the check bits

The dimensionality of the product code is not more than $k_1 k_2$, since the message bits in the upper-left determine the check bits. We'll see that the dimensionality equals $k_1 k_2$ by showing how to find correct check bits for any message.

Encoding Product Codes

Here's a procedure for encoding messages with a product code:

1. Put $k_1 k_2$ message bits into the upper-left k_2 by k_1 corner of the n_2 by n_1 array.
2. Compute the check bits for each of the first k_2 rows, according to $\mathcal{C}_1$.
3. Compute the check bits for each of the n_1 columns, according to $\mathcal{C}_2$.

After this, all the columns will be codewords of $\mathcal{C}_2$, since they were given the right check bits in step (3). The first k_2 rows will be codewords of $\mathcal{C}_1$, since they were given the right check bits in step (2). But are the *last* $n_2 - k_2$ rows codewords of $\mathcal{C}_1$?

Yes! Check bits are linear combinations of message bits. So the last $n_2 - k_2$ rows are linear combinations of earlier rows. Since these rows are in $\mathcal{C}_1$, their combinations are too.

Minimum Distance of Product Codes

If $\mathcal{C}_1$ has minimum distance d_1 and $\mathcal{C}_2$ has minimum distance d_2, then the minimum distance of their product is $d_1 d_2$.

Proof:

Let $\mathbf{u}_1$ be a codeword of $\mathcal{C}_1$ of weight d_1 and $\mathbf{u}_2$ be a codeword of $\mathcal{C}_2$ of weight d_2. Build a codeword of the product code by putting $\mathbf{u}_1$ in row i of the array if $\mathbf{u}_2$ has a 1 in position i. Put zeros elsewhere. This codeword has weight $d_1 d_2$.

Furthermore, any non-zero codeword must have at least this weight. It must have at least d_2 rows that aren't all zero, and each such row must have at least d_1 ones in it.

Decoding Product Codes

Products of even small codes (eg, $[7, 4]$ Hamming codes) have lots of check bits, so decoding by building a syndrome table may be infeasible.

But if $\mathcal{C}_1$ and $\mathcal{C}_2$ can easily be decoded, we can decode the product code by first decoding the rows (replacing them with the decoding), then decoding the columns.

This will usually **not** be a nearest-neighbor decoder (and hence will be sub-optimal, assuming a BSC and equally-likely messages).

One advantage of product codes: They can correct some *burst errors* — errors that come together, rather than independently.