Error Correcting Code

Robin Schriebman

April 13, 2006

Motivation

Even without malicious intervention, ensuring uncorrupted data is a difficult problem. Data is sent through noisy pathways and it is common for an occasional bit to flip. This effect has always been present, but its effects have heightened as technology increased. With the telephone, a few swapped bits means a slightly noisier conversation, but with computer data it could mean the difference between a working process and a destroyed hard drive.

The general strategy for pin-pointing errors is to send messages with repetition. This repetition allows for some of the data to be corrupted while still retaining the ability to decode the original message. Using this repetition optimally is the challenge for modern day error codes. There is a provable maximum accuracy (called the Shannon limit), but while it is theoretically interesting to know that there is fundamentally a 'best' code, it does not create a practical code that functions at this limit.

This paper will look into the different error correcting codes that have practical value. I will be dealing with the advantages and disadvantages associated with these codes rather than their specifics.

Repetition

The straight forward solution to this problem is to repeat each bit a certain n times and then use the majority as the correct bit. For example, with $n = 3$ the message 101 would be sent as 111000111. In order for this to be useful, the message must be repeated at least three times.

It is easy to see that if less then $\frac{n}{2}$ errors occur in each group then the message will be decoded correctly. As the number of repetitions increases, the probability of an undecipherable error decreases proportionately. Unfortunately, this also correlates with an increase in the amount of data being sent.

This is a very crude method of error correction. It works most effectively when there is unlimited bandwidth and it is essential that the data is recoverable. But it is unacceptably costly on a practical scale. Also if a user needs to encrypt the data for security, knowing that the information is repeated allows for significantly easier cracks. Therefore more sophisticated repetition algorithms are needed.

Parity Checkers

Rather then repeating every bit, it would be more efficient if we encoded and verified chunks of data. A solution to that is to introduce a parity-check bit. Parity-check bits are bits that demonstrate whether another set of bits is even or odd. By tradition the parity-check bits attempt to make the number of 1's even. For example, if we were attempting to send the string 001 then adding a parity-check bit to the end to check the entire string would cause the new string to become 0011.

Therefore if one bit is flipped, a parity-check bit would be able to tell. However, it would not be able to determine what bit was flipped or if two bits were flipped. Therefore a single parity-check bit could only inadequately check the accuracy of a message and would be unable to fix it. More sophisticated methods layer multiple

parity bits to gain corrective capabilities.

Hamming Code(4)

What is generally consider the first leap towards error correcting codes was done by R. W. Hamming during his employment at Bell Labs. He wrote his paper in 1950, but improved Hamming Codes are still in use today. He began by detailing the use of a single parity-check bit, but quickly determined that more sophisticated methods would be needed in order to correct errors.

Hamming Codes send m information bits padded with a specific k parity-check bits. They have the ability to correct any single mistake. They manage this by having the k parity-check bits set at positions $1, 2, \ldots, 2^{k-1}$ and checking every element whose binary representation has a "1" in position $k_i - 1$. For example, bit 4 would check the sum of the parities in positions $100, 101, 110, 111, 1100, 1101, \ldots = 4, 5, 6, 7, 12, 13, \ldots$

Encoding a message in this manner is computationally simple and understandable. Decoding it and determining where there is an error turns out to be just as simple. Assume the bits to be sent are 100110. Then the encoded string would be $k_1, k_2, 1, k_3, 0, 0, 1, k_4, 1, 0$ where $k_1 = 1, k_2 = 0, k_3 = 1, k_4 = 1$. So the final sent string would be 1011001110. Now let us simulate introducing a random error at position 5, causing the string to become 1011101110. To decode, you would check each parity bit accuracy (if they are not correct make $k_i = 1$). So you get $k_1 =$ (incorrect) $= 1, k_2 =$ (correct) $= 0, k_3 = 1, k_4 = 0$ which numerically is 0101 = 5. Therefore you can determine that the flipped bit occurred at position 5.

This code is significantly more efficient then pure repetition. You only have to add on an addition $\log n$ bits rather then xn, but it only has the capacity to correct one error per block of code. If two bits are flipped it will incorrectly decode to a

different string. Hamming added a further adjustment to his code to allow for the detection of up to two errors. He did this by adding an additional parity-checking bit to the end of the string which checks the entire string. Hamming codes can detect any two bit errors even though they can still only correct a single error.

Hamming is also know for developing the concept of a "distance". The distance is defined as the minimum number of bits that would have to flip in order to go from one codeword (error free word) to another. All error correcting mechanisms correct to the nearest codeword to the received string. Therefore the maximum number of errors a code can reliably fix is less then half of the distance between the closest two codewords. Any more mistakes and it could potentially decode incorrectly. Hamming proves that this is the most efficient use of parity checks for single detection, correction and double detection since he maximizes the distance between codewords.

Hamming codes have one distinct problem. They are relatively inefficient when sending small amounts of data, but they get increasingly inaccurate as the number of bits increases. They can only correctly locate one flipped bit for each codeword regardless of its length. Therefore you almost always have to encode strings of length n with about n parity checks in order to ensure accuracy of information. However, this is still a rather large step up from pure repetition.

Reed-Muller (2) and Reed-Solomon Codes(6)

Reed-Muller and Reed-Solomon Codes use vectors that partially span a vector space as their way of inducing repetition. They create a polynomial using the data bits as the coefficients for the spanning vectors. Then they send an over-sampled section of this polynomial. The original polynomial can be reconstructed by multiplying the data points with vectors perpendicular to the spanning vectors.

Then, the original data can be reconstructed[1].

This description is slightly misleading. Both Reed-Muller and Reed-Solomon codes use this idea as the basis, but their implementations vary widely. In particular, Reed-Muller codes are designed to only handle binary representation and they approach the matter differently. The theoretical aspect of the above paragraph is gleamed from Reed-Solomon codes as they generalize Reed-Muller's algorithm, but for the purposes of this paper I will treat them identically.

This initially seems overly complicated, but it has the very convenient fact that the larger the code word, the more errors this code can correct. It also has the ability to tailor exactly how many correctable errors at the cost of more sent data.

Without going into too much detail, Reed-Muller codes are described as $\mathscr{R}(r, m)$, where m is the number of spanning vectors (causing the space to have 2^m dimensions) and r is the depth of linear combinations of spanning vectors. For example, $\mathscr{R}(2, 4)$ has 16 dimensions, and is partially spanned by vectors x_1, x_2, x_3, x_4 and $x_1x_2, x_1x_3, x_1x_4, x_2x_3, x_2x_4, x_3x_4$. It can therefore encode a message of up to 10 bits long and the sent message would be 16 bits long. The minimum distance between codewords is 2^{m-r}. So the exact number of errors any Reed-Mullen code can correct is $\frac{2^{m-r}-1}{2}$ rounded down. In the above example the 10 bit code could have corrected $\frac{2^{4-2}-1}{2} = 1$ error. However, if we had been working with $\mathscr{R}(1, 4)$ it would only be able to use x_1, x_2, x_3, x_4 and thereby encode a message of length 4 bits into 16 bits while correcting any $\frac{2^{4-1}-1}{2} = 3$ errors.

Looking at small examples it becomes apparent that this code is significantly less efficient then the Hamming code. It can correct as many errors, but often requires sending much more information to do so. On top of that, the encoding and decoding algorithm is significantly more complex then the Hamming Code.

[1]It took me a long time to fully understand the mathematics behind it. When I did it was very cool, but it is not particularly necessary for this paper that the reader understand the specifics, only the consequences of it. If you desire more information, check out(2)(6). They explain it as well as I could be able to.

Therefore since this code is highly inefficient for sending small blocks of data it seems logical that this code would excel at sending larger chunks of data.

However, after examining how this functions on large chunks of data it becomes apparent that it is similarly inefficient. Let us examine $\mathscr{R}(15, 25)$. This gives us the capability to encode a string of length 29703675 with 33554432 bits with at most 511 errors. It is highly improbably that only 511 errors occurs in a string of that size so even though this is highly efficient it will probably return an incorrect result. However, if you increase the number of errors to a more logical number like 32767 (or ≈ 1 in 250) by using $\mathscr{R}(8, 25)$ then you can only encode at most 1807780 bits. Therefore to get a reasonable number of error checking you are forced to send 18 times the amount of data. For comparison, using the most common Hamming Code (7 bits: 4 parity, 3 data) you would have the ability to correct about 146 times more errors and send 8 times more information.

Through these calculations it does not seem advantageous to ever use Reed-Mullen codes. However, there is one very large practical advantage I am glossing over. The Hamming code has the ability to correct 146 times as many errors, but it still can only correct one error every seven bits (although it can detect up to two). When assuming that each bit has the same chance of flipping regardless of its surrounding bits this is not an issue. If every bit has significantly less then a $\frac{1}{250}$ chance of flipping then the likelihood that more then two errors will occur in a set of seven is probabilistically impossible. However, in real situations errors can tend to cluster. This causes the Hamming Code option to be infeasible in certain situation (despite it being nice theoretically). Reed-Mullen codes sacrifice overall efficiency for broad accuracy. If the errors are known to cluster then this is a necessary sacrifice.

Low-Density Parity-Check Codes (LDPC) (3)

Low-density parity-check (LDPC) codes are encodings that use specific parity bits. They are designed in such a way that all bits act equivalently. Each parity-check bit checks some small fixed $k \in \mathbb{Z}$ bits and each bit is checked by some small fixed $j \in \mathbb{Z}$ parity-check bits. This has the nice property that every codeword has the same set of distances to other code words. Therefore the minimum distance from any one codeword to another is the same for all codewords. LDPC codes are also highly efficient.

The major contribution of LDPC codes at the time they were invented was their decoding algorithm. Every prior decoding method was strict and deterministic. Every potential outcome had an exact procedure to follow that lead to a single solution. LDPC implements two different "softer" decoding algorithms. The first decoding algorithm cycles through the digits and checks each one versus its parity check operators. If the majority of them contradict what the bit is, then it is flipped. This process is repeated until all bits are static. The second approach is more accurate and computationally intensive. It proceeds similarly to the first approach, but instead of using a majority process it computes the probability that a certain bit is a 1, by taking into account all the other bits. Then it iterates through until it reaches a static position. In both these, the results depend upon the order the elements are examined.

These decoding algorithms were computationally impractical at the time they were created. LDPC codes were also very erratic in how effectively they worked and there was no computationally feasible methods for creating effective ones. Because of those factors, LDPC codes were ignored for thirty years.

Convolutional Codes(5)

This brings us to the next paradigm in error correcting codes: Convolutional codes. Up until now, each individual block has been encoded as an independent entity. Convolution codes encode bits based upon a state which is determined by summing a fixed set of previously bits. Each input bit is manipulated in a few different ways to produce several outputs bits. Therefore each output bit conveys the combined information of many different input bits. The state is initialized to a key that is initially passed from encoder to decoder. Due to the integral part this key plays in decoding, these codes are often used for cryptography.

There are many different algorithms for encoding data, but they are all dependent upon two variables, how many bits in the state and how many output bits are produced per input bit (called the *rate*). The codes can be further subdivided into systematic and recursive: a systematic code has an output that is the input; a recursive algorithm uses a prior output as part of the new input. Figure 1 depicts three diagrams of convolution encodings. They all use a rate of $\frac{1}{2}$ (or 1 input, 2 outputs) and have a state that is dependent upon seven inputs. The top image depicts a systematic nonrecursive, the middle a nonsystematic nonrecursive, and the bottom a systematic recursive.

Since there are so many different types of convolution codes it is difficult to do direct comparisons. But overall, convolution codes are significantly better at approaching the theoretical Shannon limit then prior error correcting codes. They are fast, efficient and generally accurate. Unfortunately their accuracy varies significantly depending on the input. In specific, convolution codes have specific codewords where their accuracy plummets. Some codewords are only separated by a distance of one. Much research in this field attempts to reduce the quantity of these trouble words. However, creating a convolution that is free of them has yet to be done.

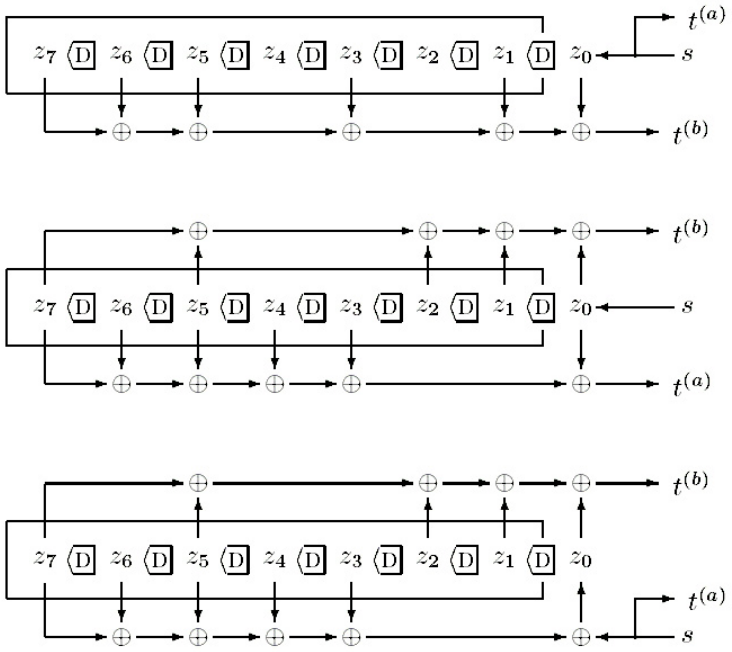

Figure 1: Different methods of generating convolutional codes with rate $\frac{1}{2}$. The symbol $\boxed{\text{D}}$ indicates a copying with a delay of one clock cycle. The symbol $\oplus$ denotes linear addition modulo 2 with no delay. The symbol s denotes the input. The symbol $t(b)$ and $t(a)$ represent the two outputs. (5, p. 587)

Another area of variable accuracy is in the initial configuration. Certain initial states will yield more distance for certain codewords then others. Therefore if the codeword is know in advance the initial state can be tailored to ensure that it will not be a problem word. This is not an easy process. Also, any attempt to suit the initialization to the data causes the codeword to be easier to crack. This reduces its effectiveness in cryptography.

Convolution codes allow for the addition of other checksums. It is common practice to integrate a form of Reed-Mullen parity-checks into the code before convoluting it. This gives double error-protection, but sacrifices some of the speed associated with encoding and decoding convolution codes.

Turbo Codes (1)

Turbo codes are a rather recent innovation being first documented 1993. They use a hybridization of many previously discovered codes. The simplest turbo codes work through a series of simultaneous steps. The input is split into as many copies as desired. Then a copy is sent directly through a convolution code. Simultaneously another copy is permuted and sent through a potentially different convolution code. This process is repeated using different permutations and potentially different convolution codes until all the copies are sent. Then it appends/sums mod 2 certain combinations of outputs and that is the message that is sent. For an example see Figure 2.

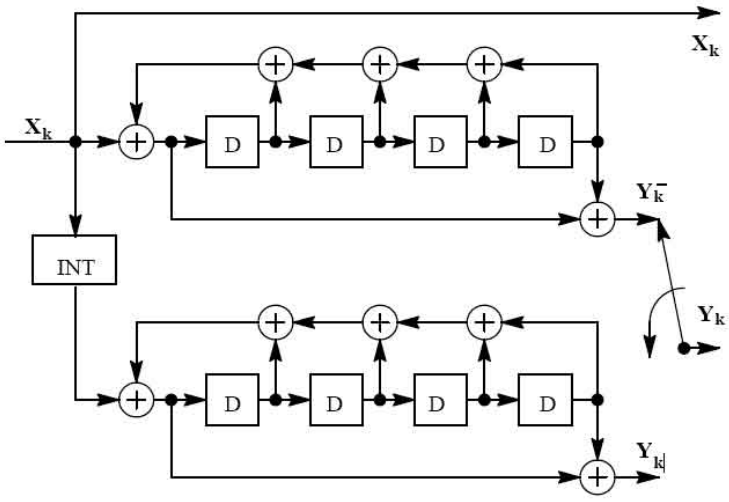

Figure 2: This is a rate $\frac{1}{2}$, systematic recursive turbo code. The symbol INT represents a permutation. The input is X_k and the two outputs are X_k and Y_k. Y_k is created by summing $Y_{\bar{k}}$ and $Y_{k|}$ (1, p. 9)

Alternately this process could be done in series. In that situation, one input is convoluted into n outputs, then those n outputs are permuted and sent through a second encoder, designed to take n inputs and produce m outputs. This process can be repeated as many times as desired. Some turbo codes use a combination of parallel and series configurations.

There are various factors that create the most effective turbo codes: using more convolutions; using recursive convolutions; using "softer" decoders; and, rather nonintuitively, using convolutions with fewer bits in the state. Turbo codes are most effective on longer codewords and are consistently close to Shannon's limit. Further work is currently investigating whether integrating Reed-Solomon codes increases performance as well as investigating the optimum permutation functions.

LDPC codes revisted

At the same time that turbo codes were researched, LDPC codes were being reexamined. Since the level of technology had increased, LDPC codes had now become feasible. Through reexamination it was discovered that certain specific LDPC codes outperformed turbo codes as can be seen in Figure 3. Surprisingly, the closest codes to reaching the theoretical Shannon's limit are currently LDPC codes.

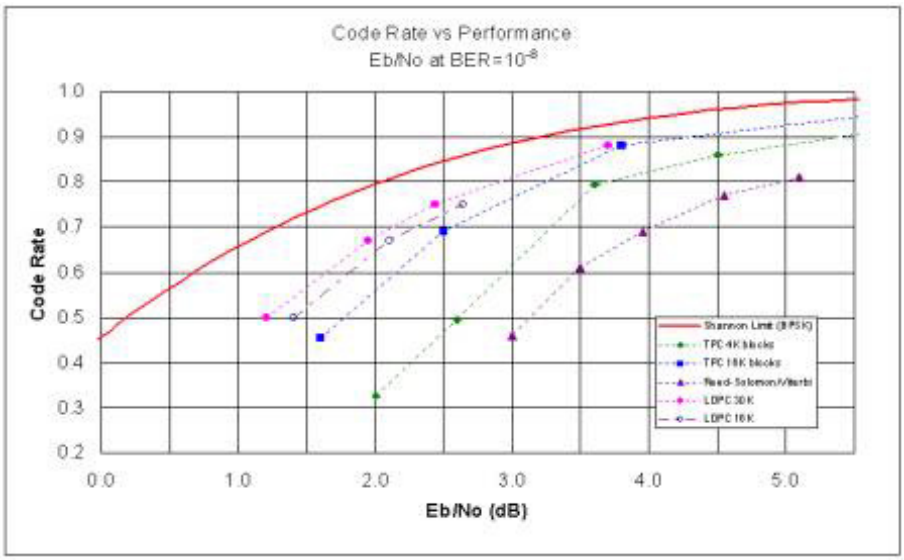

Figure 3: Side by side comparison of two turbo codes, two LDPC codes, and a Reed-Solomon/Muller code. Both LDPC codes outperformed all the others. (7)

Conclusion

Error-correction code has been mainly an iterative process. Each code examines the successes and failing of the previous code and builds upon it. They have been getting more complex and effective over time. However, surprisingly, the best code we currently have is one that is theoretically simple and was invented in 1960.

The two best codes utilize methods that allow for flexibility. Both turbo codes and LDPC codes require the use of a ”soft” decoder and they have can be adjusted to better work with specific codewords. I would hypothesize that it is this flexibility that makes the difference and that future improvements will come from using increased amounts of flexibility.

References

[1] Barbulescu, S. Adrian and Pietrobon, Steven S. "TURBO CODES: a tutorial on a new class of powerful error correcting coding schemes Part I: Code Structures and Interleaver Design." Revised October 26, 1998.

[2] Cooke, Ben. "Reed Muller Error Correcting Codes", http://www-math.mit.edu/phase2/UJM/vol1/COOKE7FF.PDF. Viewed, April 8, 2006.

[3] Gallager, Robert G. "Low-Density Parity-Check Codes" pp. 1-20. Monograph, M.I.T. Press, 1963.

[4] Hamming, R. W. "Error Detecting and Error Correcting Codes", *The Bell System Technical Journal*,J Soc, Indust. Appl. Math. Vol. 26, No.2, April 1950.

[5] MacKay, David J.C. *Information Theory, Inference, and Learning Algorithms*, Chapters 1,2, 48. Cambridge University Press, Version 7.0 August 25, 2004.

[6] Reed, I. S. and Solomon, G. "Polynomial Codes Over Certain Finite Fields", J Soc, Indust. Appl. Math. Vol. 8, No. 2, June 1960.

[7] Summers, Tony "LDPC: Another Key Step Toward Shannon". 14 October 2004. <http://www.commsdesign.com/design_center/broadband/design_corner /showArticle.jhtml?articleID=49901136>.