Error Control Coding

Professor Richard Harvey FBCS

1st February 2022

I'm writing this on a computer with a 1TB drive and 64GB of memory. A modern computer constantly juggles data from disc to memory at speeds of around 500 Mbits per second. So, if you work ten hours a day, and I modestly remark that is a typical working day for a computer scientist[1], then that means your computer is shuffling $500 \times 60 \times 60 \times 10$ Mbits = 1.8×10^{13} bits (20 peta bits of information) per day. If one of those bits is wrong, your computer will crash. And to the question "when was the last time your computer crashed?" most people might answer many months ago.

A moment's thought shows that it is a wildly improbable situation – humans have designed a system with billions of electrical parts that is operating at an error rate of less than 1 in 10^{13}. It's hard to visualize, isn't it? There are roughly 8 billion people on planet earth (8×10^9). If they each tossed a coin two-thousand times each day, then that would be roughly 10^{12} events per day. Imagine asking everyone to record those results on some enormous website – what's the chance that we could do that without error? Well, your humble PC does that every day. What makes this even more surprising is that all electronic devices and communication channels make errors all the time. We don't notice those errors because of a brilliant bit of Information Theory known as Error Control Coding.

As far as I can tell, the early history of error control coding is not very well described. Not much seems to have happened until 1948 when Claude Shannon published one of the most famous papers in computer science (Shannon, 1948) and the whole of information theory was born. I'll come to Shannon's contributions later, which were immense, but it's certainly the case that error detection was known about pre-Shannon. Early systems almost certainly used a form of error control called ARQ or Automatic Repeat Request – it's the commonest form of visible error control in human communication "I'm sorry but could you repeat that." If you have been following these lectures, then you may recall that the internet uses this mechanism extensively: the internet protocols contain a mechanism that, should the receiver think there is a need, then it can request the transmitter to resend a packet of data.

What I would like to focus on in this lecture is forward error correction or FEC. This has no precise analogue in human communication, but it is easy to describe – the idea is for the transmitter to send additional information, often called side information, so that the receiver can, firstly, check if an error has occurred and, secondly, maybe correct the error.

One of the oldest ideas is called a *checksum*. Checksums abound in computer science and the idea is a simple one: in addition to the message, we will send a count of the number of informational items in the message. Some people claim that old fashioned telegrams used to transmit the number of words in the message so that the receiving operator could cross-check the message was received correctly. In current technology every packet of information sent over the internet has a header. The header declares the source address, the destination address, the protocol and so on. It also includes a checksum which is a count of the number of ones in the header. At the receiver, the checksum is recomputed. If it does not match the transmitted checksum, then the packet is discarded and, depending on the protocol, a request is sent for a

[1] I should also acknowledge that experts in machine learning, security and other computer science specialities are indeed piad quite reasonably nowadays.

new one. The checksum does not allow a message to be corrected, it's an error detection protocol that allows the communication to take appropriate action to correct the message (usually via ARQ).

A similar idea is parity. Parity comes in two flavours, *even parity* is where the parity digit is appended so that there is always an even number of ones. So, if we had data digits 100011, which contains three ones, then we would append a parity digit. Often the parity digit is appended to the end of the word so 100011 would become 1000111 or 1100011. *Odd parity* is arranged such that the total number of 1s is odd. Parity is everywhere – computers use parity checks when writing and reading data from memory, your hard disc uses parity.

The description I have just given for computing parity: add up the numbers of ones; determine if that number is odd or even; and then append a 1 so that the total sum is either odd or even; looks like it would be not so easy to implement in low-level hardware as summing large blocks of data uses quite a bit of silicon area. However, there is a cunning trick. The fundamental operations of computer logic are usually reported as the OR and the AND operator. The OR gives a 1 if either of the inputs are 1. The AND only if both inputs are 1. We can define these via truth tables in which we list the possibilities of two inputs[2] *A* and *B*

A	*B*	OR(A,*B*)
0	0	0
0	1	1
1	0	1
1	1	1

A	*B*	AND(A,*B*)
0	0	0
0	1	0
1	0	0
1	1	1

But there is another operator, which is just as simple, called the Exclusive-OR or XOR. XOR outputs a 1 if either input is 1 but not if both are 1. XOR is perhaps closer the English language use of "either … or" as "Richard is either a poet or an engineer" (but not both).

A	*B*	XOR(A,*B*)
0	0	0
0	1	1
1	0	1
1	1	0

XOR is also the same as even parity. And since XOR is very simple to implement in hardware, we can simply XOR the digits together and generate a parity.

Important point – XOR is so common in electronics and computer science that instead of XOR(*A*,*B*) you will often see *A* + *B*. The justification is that EXOR is also equivalent to adding two binary numbers without the carry. Thus 0+0 = 0; 0+1 = 1; 1+0 = 1 and 1+1? Well, that would equal 2 in decimal, or 10 in binary, the 1 gets carried so, if we ignore the carry, then 1+1 = 0. Mathematicians call this modulo-2 addition. One can define the standard operations of addition, subtraction, multiplication and division on these numbers to form what is known in the patois as a *Galois Field* or GF(2). Subtraction is 1-0 = 1; 0-1 = -1 or, ignoring the carry, 1. 1-1 = 0. So, in GF(2), subtraction is identical to addition which is very handy.

Simple parity was used in early high-reliability disc drives. RAID[3] 5 splits the data into blocks or units. The first unit is written to the first drive, the second to the third drive and the third drive contains a block that is the parity of the first two units. Thus, if the first block, written to the first drive, was 00101010 and the second, written to the second drive, was 10100110 then the bitwise EXOR or modulo 2 sum is

00101010

[2] We can define these for more than two inputs too but it's simple to split multiple inputs into collections of two inputs.
[3] RAID stands for "redundant array of independent drives" and was originally devised to speed-up reading and writing from disc. Nowadays it is usually used to increase reliability.

+ 10100110

= 10001100

We write 10001100 to the third drive. Let's say the second drive fails, in which case we can recover the missing block, by subtracting (which is the same as adding modulo 2) the block from the first drive from the third:

10001100

- 00101010

= 10100110

Clearly this comes at a cost – we have bought 3 1TB drives but they are providing only 2 TB of storage – the remainder being spent on parity digits. However, this is the first example we have seen where the parity is being used to, not only detect an error, but also correct it. We have achieved our first feed-forward error correcting code. Feed-forward because we are not merely detecting an error and asking for another version of the data via feed-back (or ARQ). Of course, we can be fairly sure we know when a drive has failed, which makes it easy to determine which of the two units to subtract. In more general communication, we are streaming data, and it may not be clear how the channel has failed – we merely have errors in unknown positions.

An interesting thought experiment is to think about merely repeating the data. The transmitter and receiver agree to repeat the data, say, three times. The receiver stacks up the three received versions of the data and takes a majority vote. So, if our original error rate[4] was *e* then for the majority vote to fail, there have to be two errors for the output to be in error. Let's imagine the channel was quite noisy and was flipping one bit in 10 (a 10% error rate or $p = 0.1$) then the majority vote system fails if either all three bits are flipped or, more likely, any two bits are flipped. If you are keen on probabilities, then you can probably work out the output error rate is $p^3 + 3p^2(1-p)$. So, a 10% error rate becomes 0.001 + 0.027 =0.028 which is around 3%. The three-repeats plus majority voting have lowered the error rate, but there has been a cost: our information rate has dropped (in this case to a third of its raw value)[5].

Engineers often display the tension between bandwidth and error rate using a graph in which bandwidth is on the *x*-axis and probability of error is on the *y*-axis. Clearly as we increase the number of repetitions, the error rate drops but so does the bandwidth. It's not a linear drop because we have to calculate the number of ways a majority vote can go wrong – and that turns out to be a polynomial but, even so, the diagram would definitely lead you to the impression that there is a what optimization experts would call a "no free lunch" situation – we can achieve whatever error rate we like but it will come at a cost which is greatly reduced bandwidth.

But surely we can devise something that uses only a few bits of error correction, like the parity digit, but, unlike parity, allows us to correct at least some errors? This was exactly the situation considered by Richard Hamming in the late 1940s (Hamming R. W., 1950). Hamming was conducting much computation but became infuriated by errors he was seeing on his punched tapes[6]. Hamming argued as follows. Detecting errors is easy – we can use a checksum (or, if you compute a checksum over GF(2), then a parity digit) but to correct errors we need to point to a location where is an error. How many positions could *n* parity digits point to? Well, 2^n but we probably also need to use one position no indicate no error so 2^n-1. So, we need a neat way to compute those parity digits and to make them point to the location of the error. Let's see how this works with three parity bits

[p0 p1 p2]

If we have [0 0 0] then let's use that to indicate no errors, [0 0 1] should indicate an error in the first bit, [0 1 0] which is 2 in decimal should indicate an error in the second bit, [0 1 1] = 3 an error in the third bit and so on. [1 1 1] is the number 7 so we can handle up to 7 positions. Three of the positions are already parity digits so with three parity digits we have 4 data bits. This is referred to as (7,4) code – 7 bits in the block or which 4 are data. We can decide to compute even parity, so that's one decision made. The remaining decision is

[4] I'm assuming here that the channel is something called a binary symmetric memoryless channel.

[5] This example and some subsequent ones come from a book by the late David McKay. McKay was an eccentric and entertaining man and his book, which available free for online reading, is a fun read (McKay, 2003). I think it's fair to say if you manage to read all of McKay's book then you will have a strong grasp of information theory.

[6] Hamming is an interesting man because unlike other inventors who are often give rather bland descriptions of how they came up with ideas, in later life he decided to write-up how the process of invention (Hamming R. R., 1977).

how to compute the parity bits and in which position can they sit. Let's have a 7-bit block of data: [b_1 b_2 b_3 b_4 b_5 b_6 b_7]. Hamming devised an ingenious scheme which interleaved the parity bits with the data[7] [p_1 p_2 d p_3 d d d]. The first parity digit in the left-hand position is the even parity of all the odd-numbered positions. The second covers positions 2, 3, 6 and 7 and the third covers positions 4, 5, 6 and 7.

Let's look at an example in which the data was decimal 12 or binary 1100. Hamming then pops this into the word to give [p_1 p_2 1 p_3 1 0 0]. Computing the parities gives [0 1 1 1 1 0 0]. This is transmitted but there is an error in, say the fifth bit, giving a received word [0 1 1 1 **0** 0 0]. When we recompute the parities, we get a one for the first parity (but we received a zero), we get a one for the second parity (correct) and the third is incorrect. We now assign ones or zeros for correct/incorrect hence 101 of five – the error is in the fifth position[8].

In the slides I have plotted the new code, our (7,4) Hamming code, on our previous diagram of probability of error (after coding) versus rate. I've spared you the details of me working out what the probability of error is, but it's around 7% and since (7,4) Hamming code passes 4 bits of information in a block of 7 the rate is 4/7 = 0.57.

What if we have more than one error? Clearly the (7,4) Hamming code does not handle that situation but there is a simple tweak, which is add another parity-of-party bit (thus it becomes an (8,4) code) which is often called an extended Hamming code. Again, it would seem that we can reduce the output error rate[9] but the cost is a reduction in bandwidth – more parity bits in this case.

So far, so good. But in 1947 Claude Shannon came up with a rather startling result – he claimed that however poor the channel was, it was still possible to find a code such that we could transmit with zero error. Furthermore, he could predict the rate of the channel. His formula for the capacity of a binary symmetric channel[10] was

$C = 1 - H_2$

Where H_2 is entropy[11] $H_2 = -\log_2 p - (1-p)\log_2(1-p)$. For our example p = 0.1 (10% error) so C = 0.53. What an astonishing result. Instead of adding more and more error control, we need only double-up the data and we can operate with zero error. Or, more practically we can choose the error rate we want and hence infer the bandwidth we need.

However, it will not have escaped your beady eye that none of the codes I have derived so far are very close to this limit. This is the problem of error control coding – we do not have a system for constructing error control codes that hit this limit. This means that a lot of human ingenuity has been devoted to constructing better and better codes.

In the pantheon of codes that people have found useful over the years we ought to mention a couple of classes of codes: the *cyclic codes* which have some repetitious structure which makes them suitable for implementation on cheap hardware; there are *convolutional codes* where the current parity is a function of not only the current data but the previous data stream (they have feedback) and there are *block codes*. If I had been giving this lecture ten years ago, to find a code that gets close to the Shannon limit I would be forced to explain some rather fancy codes such as Reed Solomon, BCH, Viterbi codes and so on. However, more recently the technical community has rediscovered some codes proposed by Gallager in the 1970s (Gallager, 1962). These low-density parity-check (LPDC) codes work in a similar way to that already discussed but by using large data blocks, it is possible to get pretty efficient coding. Another recent innovation are polar codes which are also linear block codes (Arikan, 2009). And modern data transmissions standards

[7] Nowadays we will usually place the parity digits at the end of the word as it makes the mathematics a bit tidier. We don't need the mathematics in this lecture so I'm sticking to Hamming's original order.

[8] A massive warning if you are trying to follow yourself – some authors put all the parity digits at the end of the code, some put the data into position with the least significant bit on the left (Hamming puts the data into the word the other way round which is very annoying). Before following an example, be sure how they are packing the data into the word. The order of the data has no effect on the theoretical properties of the code.

[9] In some places the output error rate is known as the Viterbi rate in honour of Andrew Viterbi who invented a class of codes and a cunning back-tracking algorithm for decoding them and other things.

[10] Binary because we are transmitting zeros and ones and symmetric because we are assuming that probability of a zero being flipped to a one is the same as a probability of flipping a one to zero.

[11] More information on entropy and the starting genius of Claude Shannon can be found in other Harvey lectures particularly (Harvey, 2018). The subscript 2 on the logarithm means log to the base 2 ($\log_2(x) = \log_{10}(x)/\log_{10}(2)$)

have been rushing to use LPDC or polar codes. The block sizes are often large — digital video is transmitted using the Digital Video Broadcasting (DVB) standard in a (64800,21600) LDPC code.

In principle, a LPDC is no different from other block codes - on transmission one exors some data bits to create the parities, and on reception one re-computes the parities and hence corrects the data. However, computing 20,000 parity bits and making appropriate corrections may not be that simple in practice and the code may need to be constructed in a way to either make the decoder work swiftly or for other practical reasons. For example, specifying 21600 parity digits and their computation is a recipe for typing mistakes. The latest wifi standard 802.11n specifies, among others, a (648,486) block code via a (27 by 27) block which is then munged to create the matrices necessary for the full code. This issue bedevilled LDPC codes for a while — Gallager essentially presented some huge random parity check matrices with some good error correcting properties whereas Computer Scientists want a short and tidy algorithm for creating such matrices. This is one of the advantages of polar codes, developed by Erdal Arikan, they have a deterministic construction method which implies simpler communication. Your 5G mobile telephone system, when it's not giving you covid[12], is designed to use polar codes.

A couple of technical terms come up frequently in error control coding. A *linear* code is one in which the sum of any two code words is also a code word where we are using + to mean modulo-2 addition as before. Since the effect of errors is to flip digits in code words, we sometimes refer to the *Hamming distance* or *distance* between code words which is the number of bits by which they differ. Is it obvious that if a code has a minimum Hamming distance of 3 then it can tolerate 3 errors before one code word is mutated into another? And, if we look at the mutated word then, if we measure the Hamming distance between it and the closest code word, then we can choose that word and correct up to 1 error and detect 2. Such is the parlance of error control coding – we might say the (23,12) Golay code[13] has a minimum distance of 7 so can correct 3 or fewer errors per block. It's worth adding that both Hamming and Golay codes are known as "perfect" codes because they fill the space perfectly so that each code word has the same minimum distance. This is rather unusual, most codes are not "perfect" which means that certain code words have more error correction possibilities than others which, if the errors are not equally likely can be used to an advantage.

A common misconception is that, if a code cannot guarantee to correct more than *t* errors then it's not worth trying to correct more. A decoder that acts like this is called a *bounded distance decoder* and via a conjecture, not yet proved but widely believed, called the Gilbert-Varshimov conjecture, it can be shown that such a decoder cannot reach the Shannon limit.

So far, we have assumed that the probability of error is equally likely throughout the data stream. That is a bad assumption for many situations and it's quite common to have burst errors where the signal-to-noise ratio of the channel is degraded badly for a short interval. One common solution is interleaving in which the transmitted signal is spread out so that the bursts are spread over error blocks. Interleaving can cause delays, however. Another alternative is to use a code that is specially designed to be resistant to, and to spot, burst errors. There are quite a few of these but the Fire code is one of the better-known examples.

When it comes to decoding block codes, then we have already encountered a system known as *hard decoding* in which the parity bits are computed, and those bits can be converted into an address or *syndrome* that points to the bits to be corrected. Hard decoding is so called because it takes place in the digital part of the receiver where we have no choice but to make a firm decision if we have a received a zero or a one. *Soft decoding* works on the received voltages or signal strengths. That allows the receiver to reason that while this bit might have crossed the threshold to become a zero, it only just crossed the threshold, so we ought to remain open to the possibility that it was a one and vice versa. Such a system becomes worthwhile when the algorithm is iterating through possibilities. Soft decoding out-performs hard decoding but is a lot more computation.

If you have been following others lectures in this series, you may have noticed that I have been able to consider a variety of today's ingenious and useful technologies and able to trace back from the present day to often pre-electronic computers. Error control is not like that. It arose from a sudden burst of intellectual brilliance in the late 1940s driven by Claude Shannon – the father of the information age. Not only is error control ubiquitous in today's systems it is vital – without it physicists would be forced to produce impossibly

[12] Joke!

[13] Golay codes actually pre-date Hamming codes by a few months. They were introduced in a single-page paper. E R Belekamp called this "the best single published page" in coding. If he was right about that, then there must be some very awful papers in coding theory, but it is an interesting curiosity (Golay, 1949).

accurate devices.

References

Arikan, E. (2009). Channel polarization: a method for constructing capacity-achieving codes for symmtric binary-input memoryless channels. *IEEE Transactions on Information Theory, 55*, 3051--3073.

Gallager, R. G. (1962). Low-Density Parity-Check Codes. *IRE Transactions on Information Theory, 8*(1), 21--28.

Golay, M. J. (1949). Notes on Digital Coding. *Proc IRE, 37*, 657.

Hamming, R. R. (1977). *The Art of doing science and engineering: learning to learn.* CRC Press.

Hamming, R. W. (1950). Error Detecting and Error Correcting Codes. *Bell System Technical Journal, XXIX*(2), 147--160.

Harvey, R. (2018, October 23). *It from BIT: the science of information.* Retrieved Nov 19, 2021, from https://www.gresham.ac.uk/lectures-and-events/it-from-bit-science-of-information

McKay, D. (2003). *Information Theory, Inferenence and Learning Algorithms.* Cambridge University Press.

Shannon, C. E. (1948). A mathematical theory of communication. *The Bell System Technial Journal, 27*, 379--423.