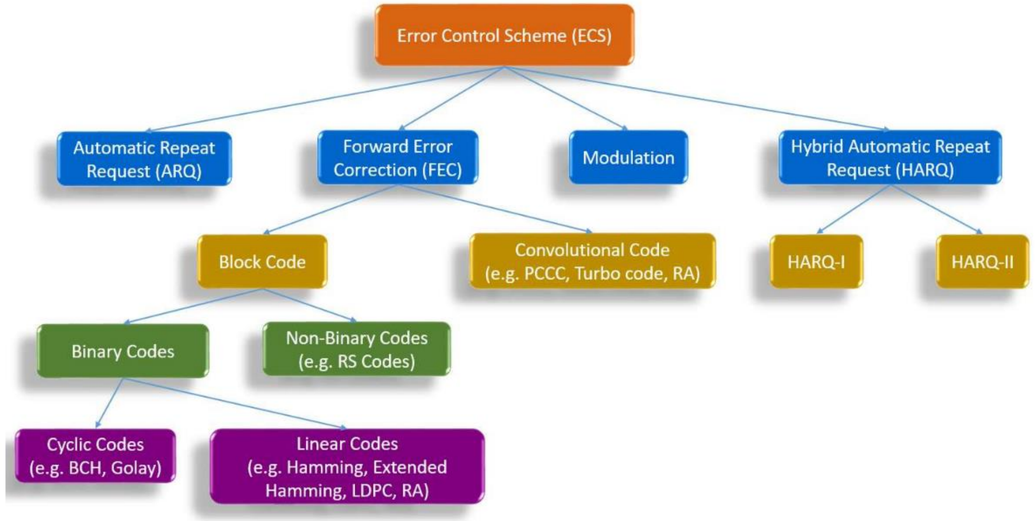
Some approaches in Neural Networks Assisted Error Correction Coding

RONZHIN DMITRY

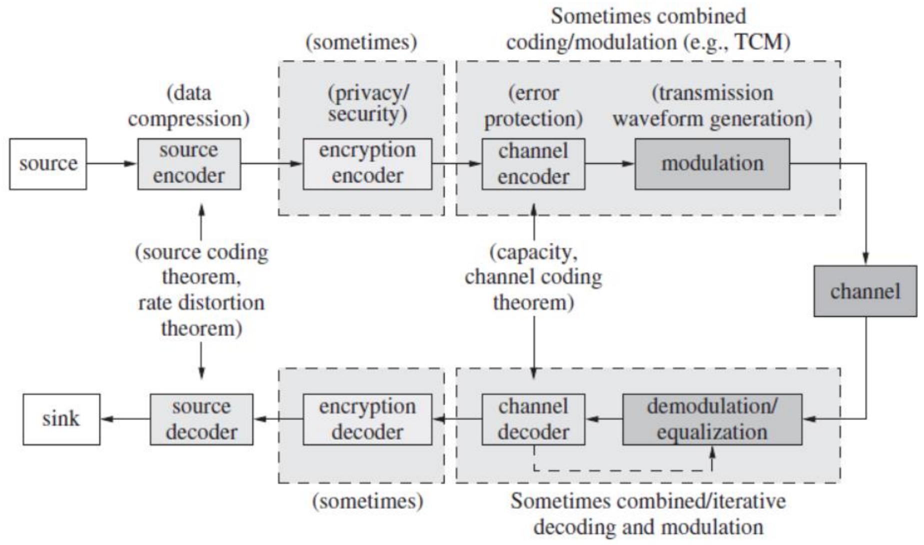
RONZHINDV@MY.MSU.RU

What is ECC?

Error Correction Coding (ECC) – technique that is used in telecommunication and computing systems that allows to fix errors that appear in message while transferring through some noisy channel. Different names for this research field and its subfields can be met in literature, i.e. Forward Error Correction (FEC), Channel Coding, Erasure Coding (EC, actually sub-direction of ECC), Error Controlling Codes etc. In these slides we will focus on block coding.


Error Control Scheme (ECS)
Automatic Repeat Request (ARQ)
Forward Error Correction (FEC)
Modulation
Hybrid Automatic Repeat Request (HARQ)
Block Code
Convolutional Code (e.g. PCCC, Turbo code, RA)
HARQ-I
HARQ-II
Binary Codes
Non-Binary Codes (e.g. RS Codes)
Cyclic Codes (e.g. BCH, Golay)
Linear Codes (e.g. Hamming, Extended Hamming, LDPC, RA)

Scheme from "Opportunities and Challenges for Error Correction Scheme for Wireless Body Area Network—A Survey" by Rajan Kadel et.al.


Sometimes combined coding/modulation (e.g., TCM)
(sometimes)
(data compression)
(privacy/ security)
(error protection)
(transmission waveform generation)
source
source encoder
encryption encoder
channel encoder
modulation
(source coding theorem, rate distortion theorem)
(capacity, channel coding theorem)
channel
sink
source decoder
encryption decoder
channel decoder
demodulation/ equalization
(sometimes)
Sometimes combined/iterative decoding and modulation

General scheme of the telecommunication channel from T.Moon “Error Correction Coding”, 2nd ed.

Is it useful?

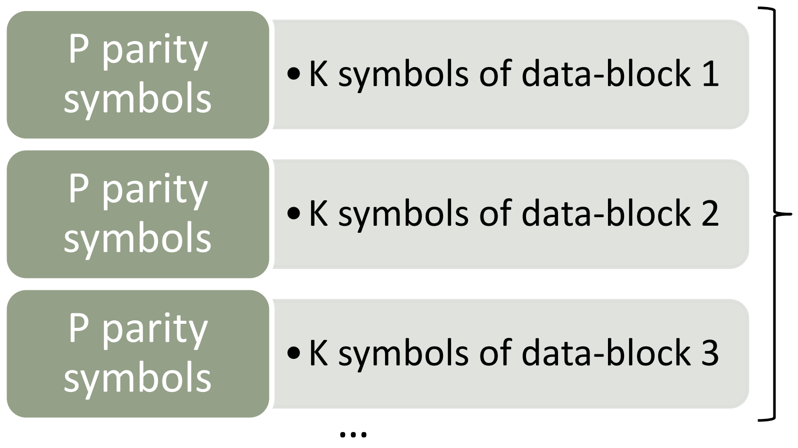
ECC techniques are used in quite a lot of directions and applications:

- Space and satellite communications;
- Digital mobile communications (4G, 5G);
- High-speed computer memory (NAND Flash memory);
- File transfer protocols (including TCP/IP stack);
- Quantum computing;
- and much more…

For more examples you can refer to the article “Applications of Error-Control Coding” by Daniel J. Costello et.al. (*although it is rather old one*)

What’s the intuition behind ECC?

Common block-ECC techniques work “against” compression, **i.e. some amount of additional information is added to the initial user data**, which allows you to find and fix errors inside the noisy message after transmission. In case of systematic code original data is unmodified, in not systematic case data is also modified before transmitting (but can be recovered). **Rate of the code is the fraction of data symbols length to total block length after adding parity.**


P parity symbols
• K symbols of data-block 1
P parity symbols
• K symbols of data-block 2
P parity symbols
• K symbols of data-block 3
...

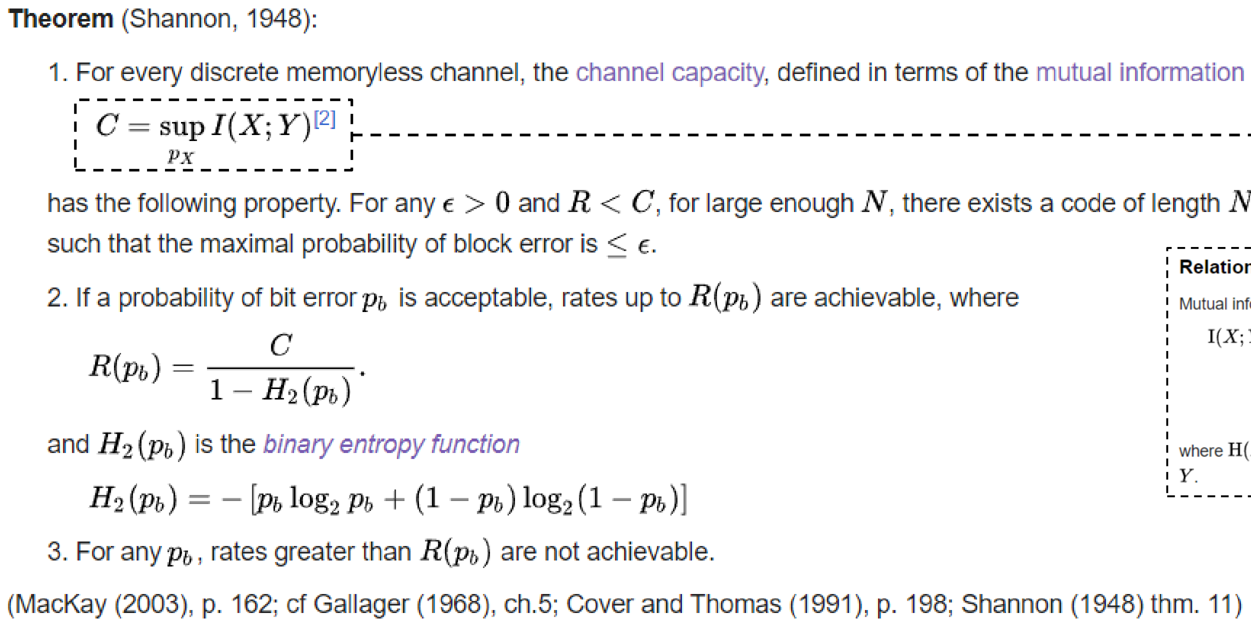
$$\text{Rate } R = \frac{K}{K+P}$$

General scheme for systematic encoding *– to each block of message symbols some amount of parity added before transmitting in channel, to ensure reliability*

In the end each K data symbols are matched to K+P data symbols, which means that there is a mapping between K-dimensional space (in binary case – just vectors of {0,1} of length K) and K+P dimensional space. **Error correction properties of the code are achieved if the resulting blocks of length K+P have some “space” between each other and can be distinguished after some noise is added from channel.**

What are rate limits?

The first and probably most well-known limit to the performance of ECC was proved by **Claude Shannon in 1948** in his noisy-channel coding theorem. This theorem in its first part states that not any rate is achievable in reliable data transmission, given some specific channel with some level of noise that it adds to the transmitted message (*X – channel input, Y – channel output, p_X - marginal distribution of input*). **For finite-block regime Shannon bound is not tight**, and better estimation (2nd order) bounds can be used (see Polyanskiy-Poor-Verdu bounds).



Theorem (Shannon, 1948):

1. For every discrete memoryless channel, the channel capacity, defined in terms of the mutual information $I(X;Y)$ as

$$\boxed{C = \sup_{p_X} I(X;Y)^{[2]}}$$

has the following property. For any $\epsilon > 0$ and $R < C$, for large enough N, there exists a code of length N and rate $\geq R$ and a decoding algorithm, such that the maximal probability of block error is $\leq \epsilon$.

2. If a probability of bit error p_b is acceptable, rates up to $R(p_b)$ are achievable, where

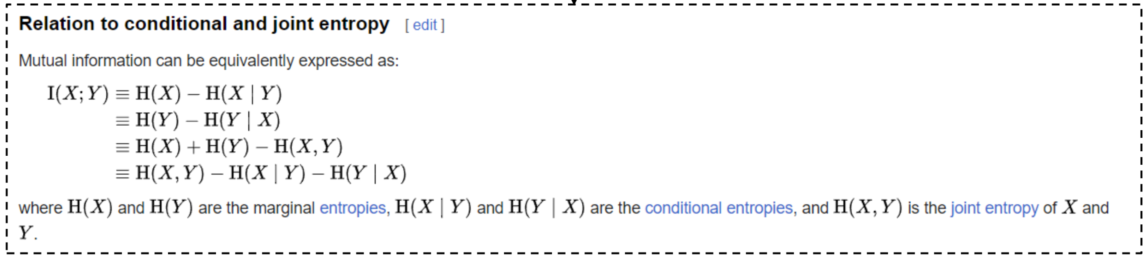
$$R(p_b) = \frac{C}{1 - H_2(p_b)}.$$

and $H_2(p_b)$ is the *binary entropy function*

$$H_2(p_b) = -\left[p_b \log_2 p_b + (1 - p_b) \log_2 (1 - p_b)\right]$$

3. For any p_b, rates greater than $R(p_b)$ are not achievable.

(MacKay (2003), p. 162; cf Gallager (1968), ch.5; Cover and Thomas (1991), p. 198; Shannon (1948) thm. 11)



Relation to conditional and joint entropy [edit]

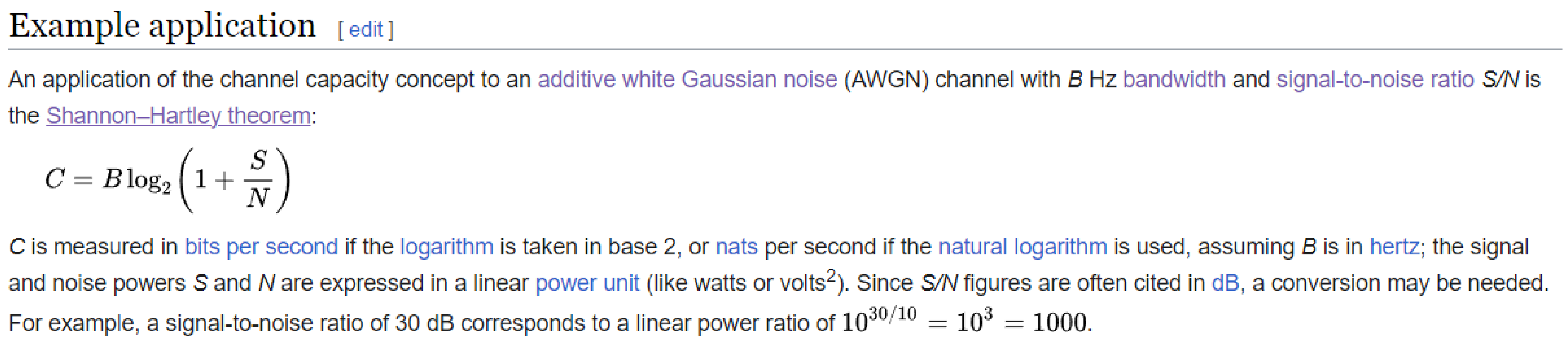
Mutual information can be equivalently expressed as:

$$\begin{aligned} \mathrm{I}(X;Y) &\equiv \mathrm{H}(X) - \mathrm{H}(X \mid Y) \\ &\equiv \mathrm{H}(Y) - \mathrm{H}(Y \mid X) \\ &\equiv \mathrm{H}(X) + \mathrm{H}(Y) - \mathrm{H}(X,Y) \\ &\equiv \mathrm{H}(X,Y) - \mathrm{H}(X \mid Y) - \mathrm{H}(Y \mid X) \end{aligned}$$

where $\mathrm{H}(X)$ and $\mathrm{H}(Y)$ are the marginal entropies, $\mathrm{H}(X \mid Y)$ and $\mathrm{H}(Y \mid X)$ are the conditional entropies, and $\mathrm{H}(X,Y)$ is the joint entropy of X and Y.

How the noise is measured?

Shannon theorem states that rate should be less than capacity for some reliable coding scheme to exist, but capacity is calculated for specific channel with some level of noise. To express the level of noise usually the specific metric Signal-to-Noise Ratio is used. Signal-to-noise ratio is defined as the ratio of the power of a signal (meaningful input) to the power of background noise (meaningless or unwanted input): $SNR = \frac{P_{signal}}{P_{noise}}$; however, measuring this fraction "as-is" is not very convenient in applications, so usually SNR is expressed in decibels: $SNR_{db} = 10\log_{10}(SNR)$. For some specific channel models (i.e. Additive White Gaussian Noise Channel) SNR is a very convenient metric to express channel-capacity at once:



Example application [edit]

An application of the channel capacity concept to an additive white Gaussian noise (AWGN) channel with *B* Hz bandwidth and signal-to-noise ratio *S/N* is the Shannon–Hartley theorem:

$$C = B\log_2\left(1+\frac{S}{N}\right)$$

C is measured in bits per second if the logarithm is taken in base 2, or nats per second if the natural logarithm is used, assuming *B* is in hertz; the signal and noise powers *S* and *N* are expressed in a linear power unit (like watts or volts^2). Since *S/N* figures are often cited in dB, a conversion may be needed. For example, a signal-to-noise ratio of 30 dB corresponds to a linear power ratio of $10^{30/10} = 10^3 = 1000$.

How to make a code?

Suppose we identified the upper bound for a rate based on channel characteristics and we want to construct some ECC scheme for reliable transmission (*we have to take into consideration that Shannon bound is very optimistic in finite block coding regime, though*).

If we just select a random mapping from the K-dimensional space (where K is data size in block) to K+P dimensional scheme, so that $R = \frac{K}{K+P} < C$, the code may appear good

(*and most probably will in case of very long values K+P, that's what Shannon's proof is about*),

but **we will have to organize a good encoding and decoding algorithm for any input data stream**.

We could go with the following algorithm(*input data in vector space over some finite field*):

*For every vector in K-dimensional space store the corresponding vector of size K+P. After receiving some message find the closest vector (i.e. closest by Hamming in finite fields) from the listed vectors in K+P dimensional space, and say that this was an originally sent message (**Maximum Likelihood approach**).*

However, this is a terribly complex approach with extremely high computational and memory requirements, even in case of moderate-length codes (i.e. $K + P > 100$).

What makes some code good?

Thus, ECC scheme is in general some mapping and algorithms for decoding and encoding.

A good ECC scheme is such a mapping, that:

- has reasonable computational complexity;
- has low memory requirements;
- is able to decode enough amount of errors to achieve reliable transmission.

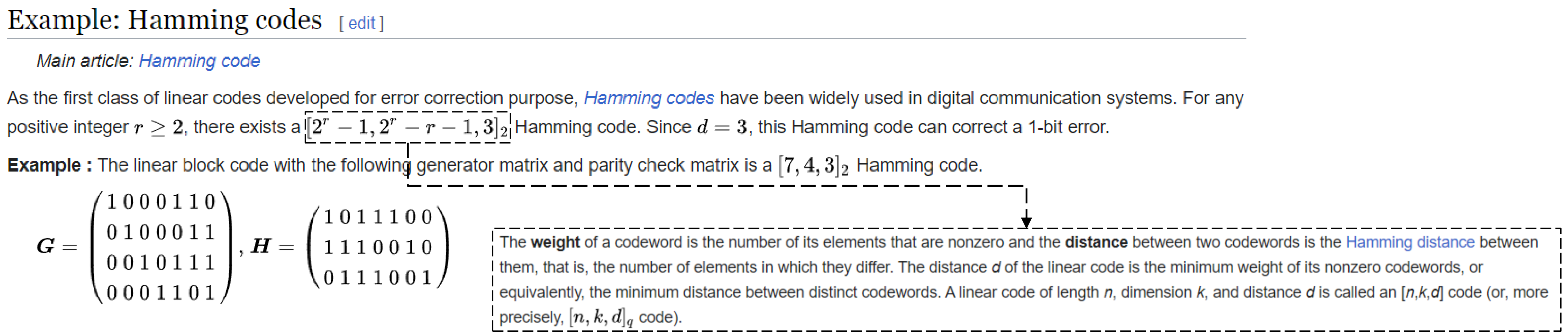
That is why usually code constructions under consideration have some good properties, and scientists put a lot of work to construct codes from some big class of codes. One of the most valuable classes of good codes (*if not the most valuable*) is **the class of linear codes**.

What are linear codes?

In coding theory, a linear code is an error-correcting code for which any linear combination of codewords is also a codeword.

A linear code of length n and dimension k is a linear subspace C with dimension k of the vector space F_q^n, where F_q is the finite field with q elements. Such a code is called a q-ary code. If q = 2 the code is described as a binary code. The vectors in C are called codewords. The size of a code is the number of codewords and equals qk.

Since it is a linear subspace, it is possible to generate a basis for this subspace, and thus make a matrix which will generate any possible element of the linear code using right-multiplication. Such a matrix is called a **generator matrix for the linear code**. If we consider an orthogonal subspace for C and construct a generator matrix for it, we will receive a so-called **parity-check matrix for the linear code**. It is obvious that left-multiplication of parity check matrix on any codeword from C results in zero (so kernel of parity check matrix is code C).



Example: Hamming codes [edit]

Main article: Hamming code

As the first class of linear codes developed for error correction purpose, *Hamming codes* have been widely used in digital communication systems. For any positive integer $r \geq 2$, there exists a $[2^r - 1, 2^r - r - 1, 3]_2$ Hamming code. Since $d = 3$, this Hamming code can correct a 1-bit error.

Example : The linear block code with the following generator matrix and parity check matrix is a $[7, 4, 3]_2$ Hamming code.

$$\boldsymbol{G} = \begin{pmatrix} 1\,0\,0\,0\,1\,1\,0 \\ 0\,1\,0\,0\,0\,1\,1 \\ 0\,0\,1\,0\,1\,1\,1 \\ 0\,0\,0\,1\,1\,0\,1 \end{pmatrix}, \boldsymbol{H} = \begin{pmatrix} 1\,0\,1\,1\,1\,0\,0 \\ 1\,1\,1\,0\,0\,1\,0 \\ 0\,1\,1\,1\,0\,0\,1 \end{pmatrix}$$

The **weight** of a codeword is the number of its elements that are nonzero and the **distance** between two codewords is the Hamming distance between them, that is, the number of elements in which they differ. The distance *d* of the linear code is the minimum weight of its nonzero codewords, or equivalently, the minimum distance between distinct codewords. A linear code of length *n*, dimension *k*, and distance *d* is called an [*n*,*k*,*d*] code (or, more precisely, $[n, k, d]_q$ code).

How does codec work?

Using a generator matrix G for each input data block of size k in finite field we create a message using multiplication by G:

$message = input\_data\ * G$

After data transfer through some noisy channel, part of the message may become corrupted. First check that is made is multiplication by parity-check matrix H to see, if the received message a codeword from our code:

$H * message == 0$

If this equality is not holding, we can state that error was detected. If it is holding, it is either the correct message or the undetected error. In case if we can resend data when we detect some errors this would be it, and we would just ask for new message from the sender. However, usually we expect to be able to fix some amount of errors, and that is when the hard part starts:

We have to design such an algorithm for a given code (mapping) that would allow us to:

1) Detect the positions of the errors in the incoming message (*this is where all heavy math usually appears*)

2) Fix errors on these positions

If we somehow know **the positions of errors (and we can treat them like erasures)** we have half of the work done, and this is the easy case.

Here the concept of code distance plays the crucial part. If the code distance is d, and we have t errors and e erasures in the incoming message, the following should hold to guarantee the error-correction capability:

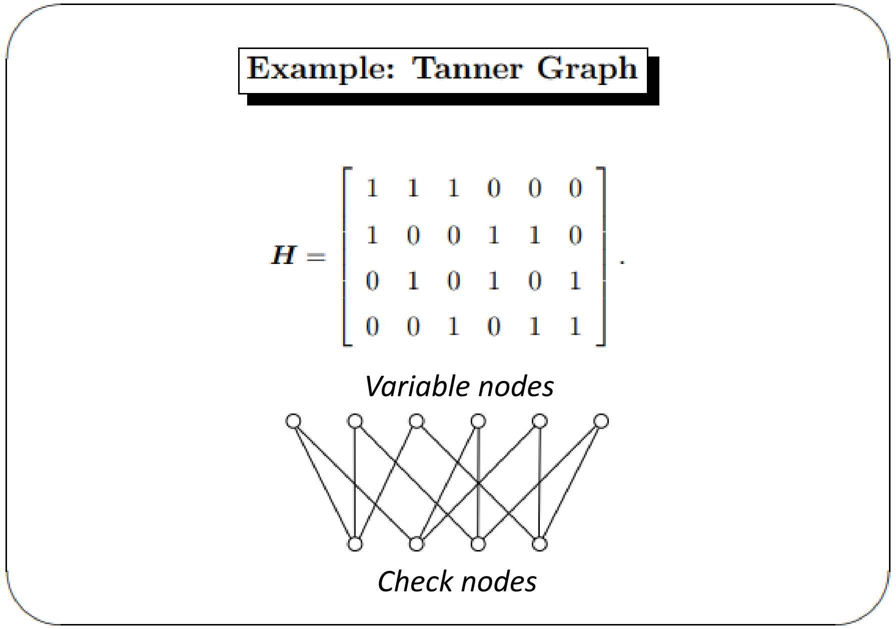
$2 * e + t < d$

How else codes can be represented?

In coding theory, a **Tanner graph**, named after Michael Tanner, is a bipartite graph **used to state constraints or equations which specify error correcting codes**.

For linear codes Tanner graph would be just a very neat representation of a parity-check matrix. As we already know, parity check matrix H is just a special matrix that allows us to check if the incoming message a codeword or not. In case is it is not a codeword – we are sure some errors happened during transmission, if not – we cannot be 100% sure that it is a correct message, but with some probability.

In case of binary codes edges in tanner graph need to additional weights, and using Tanner graph only one can reconstruct the linear code.



$$\boldsymbol{H} = \begin{bmatrix} 1 & 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 \end{bmatrix}.$$

Example of a Tanner graph from these slides

What’s LDPC?

In information theory, a **low-density parity-check (LDPC)** code is a **linear error correcting code**, a method of transmitting a message over a noisy transmission channel. An LDPC code is **constructed using a sparse Tanner graph**. LDPC codes are capacity-approaching codes, which means that practical constructions exist that allow the noise threshold to be set very close to the theoretical maximum (the Shannon limit) for a symmetric memoryless channel.

LDPC codes are also known as Gallager codes, in honor of Robert G. Gallager, who developed the LDPC concept in his doctoral dissertation at the Massachusetts Institute of Technology in 1960.LDPC codes have also been shown to have ideal combinatorial properties. In his dissertation, Gallager showed that LDPC codes achieve the Gilbert–Varshamov bound for linear codes over binary fields with high probability.

$$H = \left[\begin{array}{cccccccccccc}
1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\
\hline
1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 1 \\
0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\
\hline
1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 \\
0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 1
\end{array}\right].$$

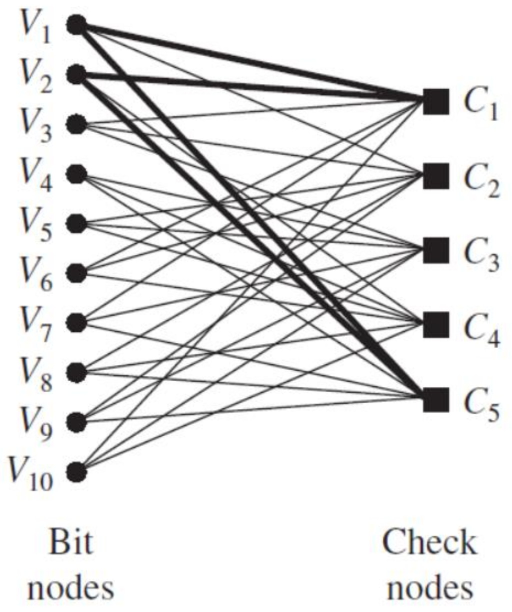
$$H = \left[\begin{array}{cccccccccccc}
1 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 \\
1 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & \mathbf{1} & 0 \\
0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & \mathbf{1} & 1 \\
0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 1 \\
0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & \mathbf{1} & 0 \\
1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 \\
0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1
\end{array}\right].$$

Gallager (upper) and MacKay (lower) examples of LDPC constructions.
See this article in Russian for more gentle introduction to LDPC.

How LDPC is constructed?

LDPC codes can be **regular or irregular**. **Regular codes have the good property that in each row in parity-check matrix has exactly d_c non-zero elements, and each column has exactly d_v non-zero elements**. In that case, matrix can be extremely sparse, and the structure of Tanner graph is very regular too – any check node and any variable node will have the same amount of edges coming in. **A lot of algorithms exist to construct some good LDPC, which are out of scope of these slides**. One of the approaches can be described like this:

we can first construct a smaller matrix with zeroes and ones, and then instead of each non-zero element put a circulant matrix (such construction is called protograph quasi-circular matrix)



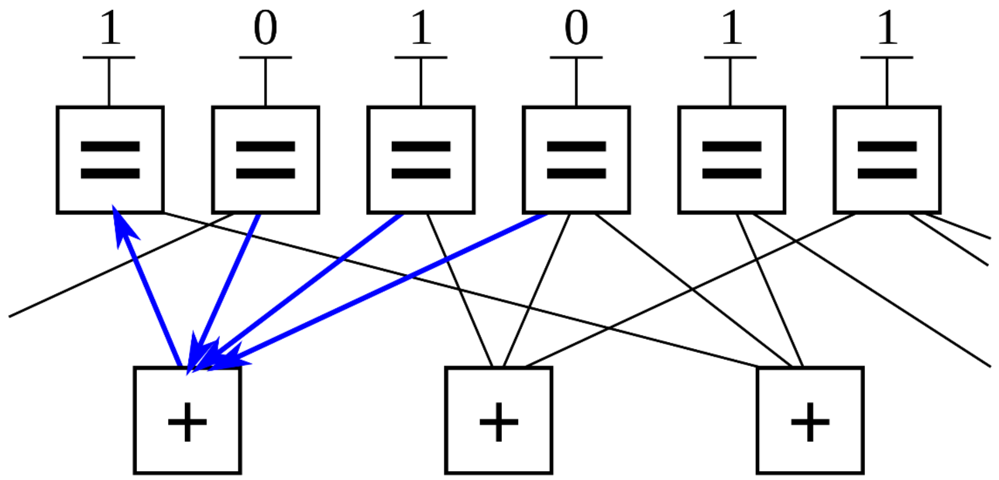
$$H = \begin{bmatrix} \underline{1} & \underline{1} & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 & 1 & 0 & 1 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 0 & 1 \\ \underline{1} & \underline{1} & 0 & 1 & 0 & 0 & 1 & 1 & 1 & 0 \end{bmatrix}$$

Example of parity-check matrix and corresponding Tanner graph. Bold lines correspond to cycle of length 4, which are crucial for decoders (as we will see later).
"Good" parity check matrices usually result in fast algorithms for both decoding and encoding (non-quadratic, as it would be expected). Please refer to T.Moon "Error correction coding", 2nd ed. for more details and algorithms.

What’s the intuition behind LDPC codec?

LDPC encoding if our of scope of these slides, although generally encoding is based on some good parity-check matrices constructions (*i.e. for full-rank parity-check matrix systematic encoding would be simply multiplication by the inverse parity part; or some Gaussian elimination approach may be used to make the parity-check matrix lower-triangular, for easy parity part calculation*). For more details you can refer to this article.

In these slides **we will consider LDPC decoding process**. Unlike many well-known and quite complex algebraic codes (Reed-Solomon, Reed-Muller, BCH etc.) whose efficiency can be precisely calculated and who base decoding process on algebraic operations over finite fields (*i.e. find an error-locating polynomial with some optimizations like Berlekamp-Massey algorithm*), **LDPC codecs usually base on different class of decoders**, which proceed iteratively **with some belief-propagation** to share information between check nodes and parity nodes in Tanner graph. That is, we consider our input message from channel as some “weight” of the variable nodes in Tanner graph, and start sending messages from variable nodes to check nodes and vice-versa and use some simple calculations to modify weights of the variable nodes in hope that process finally converges and errors are corrected (*and that is why we do not like short cycles in Tanner graphs*).



Simplistic scheme of hard-decoding belief propagation from Wiki

What decoder receives as input?

Let us focus on binary case from now on, i.e. will have only 0 and 1 in the parity check matrix and the message that is sent via channel is only some sequences of 0 and 1 (although channel can add some non-binary noise, of course). For any decoder it is crucial what information is given at the input. There are actually two types of input data:

- ❑ **Hard information**. It is just 0 and 1 for each bit of data on the input stream. In case of error in some bit, the value of bit can be switched, so we can receive different input stream, that was originally sent. This is the most simple case to understand and work with, although decoders may be not so complex for this type of input, and for a lot of applications it is crucial for decoder to work good with hard information.
- ❑ **Soft information**. Some measure of our certainty that input bit sent was either 0 or 1. Although this measure can be modulation-specific, usually all results in some kind of log likelihood ratio (LLR) information. For each bit b of binary input stream after transmission through noisy channel LLR can be defined as:

$$L_b = \log \frac{P(b=0)}{P(b=1)};$$

The sign of L_b shows how much we are sure that input value is 0 or 1, i.e. if $L_b > 0$ then $b = 0$ is more probable and if $L_b < 0$ then $b = 1$ is more probable. In case of L_b close to zero, we have less certainty about initial bit value before transmission.

What is required from good decoder?

We have several requirements for the “good” decoder (and they are quite the same as just for “good” codes, since decoder is a major part of a code):

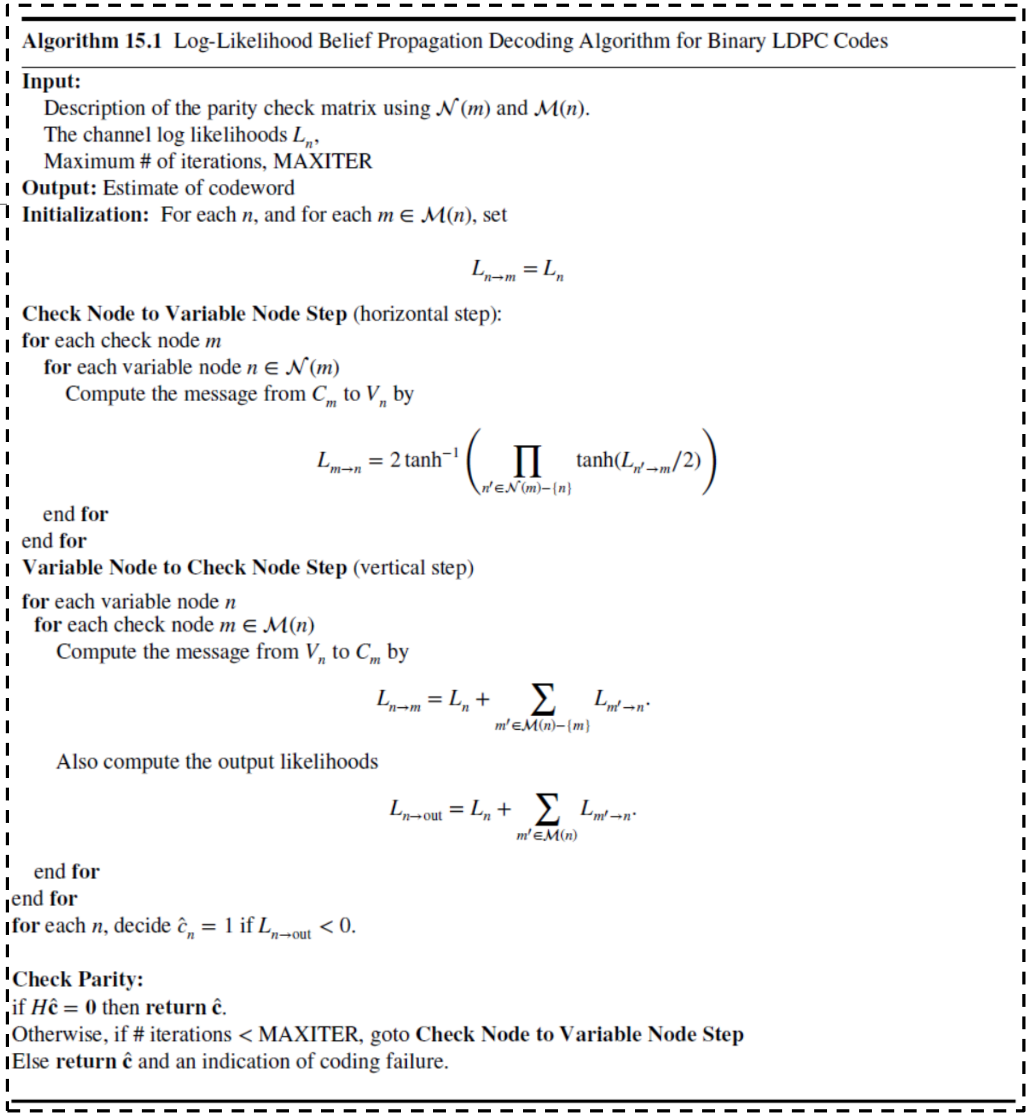
- ❑High error correction capability;
- ❑High throughput and low latency;
- ❑Low memory consumption.

It is easy to see that soft input decoder can work with hard information and vice-versa (after minor modification of input stream), however it is often the case that good hard-decoding algorithms for LDPC perform not so good with soft information and vice-versa(and it is quite a natural thing, actually). **In these slides we will focus on soft decoding**, although for anyone interested in more details about LDPC and hard decoding it may be a good starting point to read T.Moon or to look through these introductory slides.

How does good soft-decoder work?

One of the best (and first) LDPC soft-decoders is **log-likelihood belief propagation algorithm.** Sometimes this algorithm is called sum-product algorithm (SPA), although it is not quite correct, since SPA is a bit different, but in perfect case this algorithm is numerically equivalent to SPA. On this slide you can see a very concise and neat description of the log-likelihood belief propagation algorithm from T.Moon “Error correction coding” 2nd edition (and there you can read some more about probabilistic intuition behind this seemingly unreasonable set of operations with LLRs). If you want some more gentle introduction in Russian, you can read through [this small article](#).

- N: number of variable nodes (length of code).
- K: number of input bits (dimension of code).
- M: number of check nodes $= N - K$.
- n: index on variable nodes.
- m: index on check nodes.
- $\mathcal{N}(m)$: set of variable nodes (the set of n indices) which are neighbors to check node m.
- $\mathcal{M}(n)$: set of check nodes (the set of m indices) which are neighbors to variable node n.
- $\mathcal{N}(m) - \{n\}$: the neighbors to V_m, excluding n.
- $L_{m' \to n}$: The message (log probability ratio) from check node m' to variable node n.
- $L_{n' \to m}$: The message (log probability ratio) from variable node n' to check node m.



Algorithm 15.1 Log-Likelihood Belief Propagation Decoding Algorithm for Binary LDPC Codes

```
Input:
  Description of the parity check matrix using $\mathcal{N}(m)$ and $\mathcal{M}(n)$.
  The channel log likelihoods $L_n$,
  Maximum # of iterations, MAXITER
Output: Estimate of codeword
Initialization: For each $n$, and for each $m \in \mathcal{M}(n)$, set
        $L_{n \to m} = L_n$
Check Node to Variable Node Step (horizontal step):
for each check node $m$
  for each variable node $n \in \mathcal{N}(m)$
    Compute the message from $C_m$ to $V_n$ by
        $L_{m \to n} = 2 \tanh^{-1} \left( \prod_{n' \in \mathcal{N}(m) - \{n\}} \tanh(L_{n' \to m}/2) \right)$
  end for
end for
Variable Node to Check Node Step (vertical step)
for each variable node $n$
  for each check node $m \in \mathcal{M}(n)$
    Compute the message from $V_n$ to $C_m$ by
        $L_{n \to m} = L_n + \sum_{m' \in \mathcal{M}(n) - \{m\}} L_{m' \to n}.$
    Also compute the output likelihoods
        $L_{n \to \text{out}} = L_n + \sum_{m' \in \mathcal{M}(n)} L_{m' \to n}.$
  end for
end for
for each $n$, decide $\hat{c}_n = 1$ if $L_{n \to \text{out}} < 0$.

Check Parity:
if $H\hat{\mathbf{c}} = \mathbf{0}$ then return $\hat{\mathbf{c}}$.
Otherwise, if # iterations < MAXITER, goto Check Node to Variable Node Step
Else return $\hat{\mathbf{c}}$ and an indication of coding failure.
```

How to make soft-decoder faster?

Calculation of hyperbolic tangent function (tanh) is too expensive for decoder, so people spent some efforts to have a good approximations for this operation to make it computationally cheaper and not too less efficient in terms of decoding capability. Nowadays **a lot of decoders on real devices use some variation of Min-Sum algorithm** (Normalized Min-Sum, Min Sum with offset etc.).

The basic idea behind Min-Sum algorithm is very easy: keep everything just like it is in log-likelihood belief propagation algorithm, except instead of using complex tanh function use as an approximation the following combination of min and sign functions:

$$L_{m\to n} \cong \left(\prod\nolimits_{n'\in N(m)-\{n\}} sign(L_{n'\to m})\right) \cdot min_{n'}(|L_{n'\to m}|).$$

Modifications of this approach usually add some multipliers to min function (*normalization factor*) or some offset to arguments inside min function. For more details behind why it is possible and what modifications of algorithm exist one can refer to Moon's book or to [this article in Russian]().

Where are neural networks?

Min-sum approach makes LDPC decoder less complex, however error-correction capability is decreased compared to log-likelihood belief propagation algorithm. A lot of research efforts is devoted to receive better code performance with error-correction capability close to BP approach. **Two directions of neural network application for LDPC decoding that will be discussed in these slides are**:

- ❑Improving LDPC decoder itself (MS and BP);
- ❑Improving LLR information on the input of decoder;

These approaches, although not widely used in product solutions, appear to be interesting novel research directions.

What is NNMS decoder?

Neural Normalized Min-Sum (NNMS) decoder is way to use neural networks for LDPC NMS decoding. We will base NNMS description on the article *“A recipe of training neural network-based LDPC decoders” by Guangwen Li, Xiao Yu.*

$$b_{v_i} = \log\left(\frac{p(y_i|c_i=0)}{p(y_i|c_i=1)}\right) = \frac{2y_i}{\sigma^2}$$

For example let’s consider AWGN with zero mean and variance σ^2.

$$x^{(l)}_{v_i \to c_j} = b_{v_i} + \sum_{\substack{c_p \to v_i \\ p \in \mathcal{C}(i)/j}} x^{(l-1)}_{c_p \to v_i}$$

$$x^{(l)}_{c_j \to v_i} = \left(\prod_{\substack{v_q \to c_j \\ q \in v(j)/i}} sgn\left(x^{(l)}_{v_q \to c_j}\right)\right) \min_{\substack{v_q \to c_j \\ q \in v(j)/i}} \left|x^{(l)}_{v_q \to c_j}\right|$$

Normalized Min-Sum message passing

$$x^{(l)}_{v_i \to c_j} = \alpha^{(l)}_i b_{v_i} + \sum_{\substack{c_p \to v_i \\ p \in \mathcal{C}(i)/j}} \beta^{(l)}_{p,i} x^{(l-1)}_{c_p \to v_i} \quad (6)$$

$$x^{(l)}_{c_j \to v_i} = \left(\prod_{\substack{v_q \to c_j \\ q \in v(j)/i}} sgn\left(x^{(l)}_{v_q \to c_j}\right)\right)\left(\gamma^{(l)}_{q,j} \min_{\substack{v_q \to c_j \\ q \in v(j)/i}} \left|x^{(l)}_{v_q \to c_j}\right|\right) \quad (7)$$

NNMS message passing

$$x^{(l)}_{v_i} = b_{v_i} + \sum_{\substack{c_p \to v_i \\ p \in \mathcal{C}(i)}} x^{(l-1)}_{c_p \to v_i}$$

What is NNMS decoder?

Algorithm 1 NNMS decoding

Input: channel signals $\boldsymbol{b}$, $\boldsymbol{H}, T$, and well trained $\boldsymbol{\alpha}, \boldsymbol{\beta}, \boldsymbol{\gamma}$

Output: estimated binary vector $\widehat{\mathbf{c}}$

```
 1: For any $i \in \{1, 2, ..., N\}, j \in \{1, 2, ..., M\}$
 2: $x^{(0)}_{c_j \to v_i} = 0, l = 1;$
 3: repeat
 4:     calculate $v_i -> c_j$ message with 6;
 5:     calculate $c_j -> v_i$ message with 7;
 6:     $\widehat{\mathbf{c}}^{(l)} = (\hat{c}_1, \hat{c}_2, ..., \hat{c}_N), \hat{c}_i = (1 - sgn(x^{(l)}_{v_i}))/2$ by 4;
 7:     if $\boldsymbol{H}\widehat{\mathbf{c}}^{(l)} = \mathbf{0}$ then
 8:         return $\widehat{\mathbf{c}}^{(l)}$;
 9:     else
10:         $l = l + 1;$
11:     end if
12: until $(l > T)$
13: return $\widehat{\mathbf{c}}^{(T)}$;
```

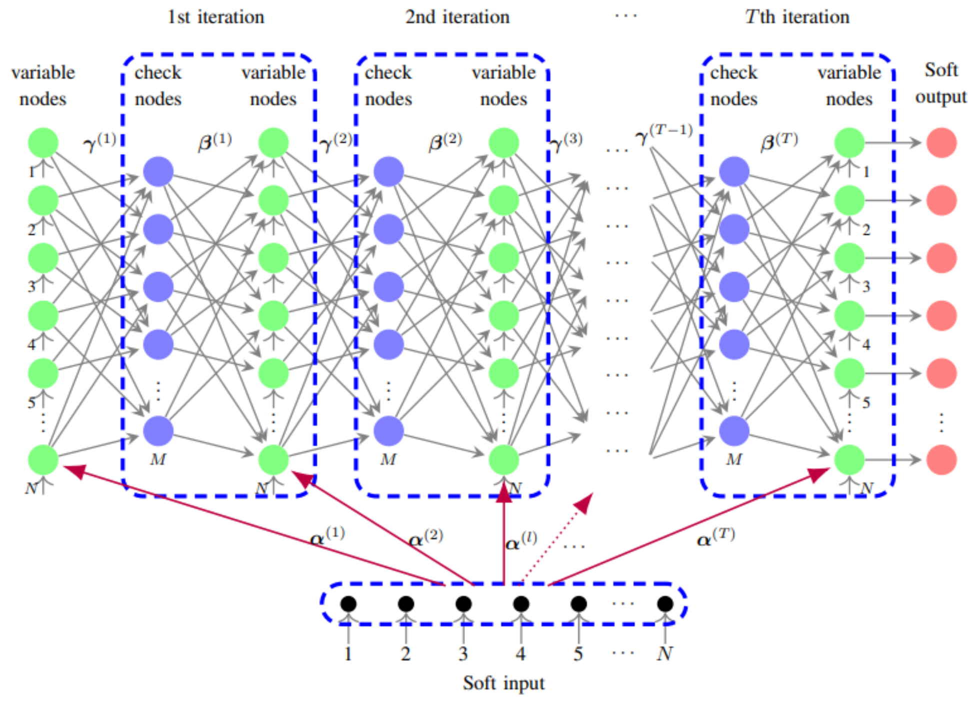


Fig. 1. A full-loaded NNMS framework in which the trainable parameters $\boldsymbol{\alpha}, \boldsymbol{\beta}, \boldsymbol{\gamma}$ assigned for each edge can be shared each other or trimmed off to meet the need of applications, and the edge connections is in line with placement of non-zero elements of check matrix $\boldsymbol{H}$

From the article *"A recipe of training neural network-based LDPC decoders" by Guangwen Li, Xiao Yu.*

How to train NNMS?

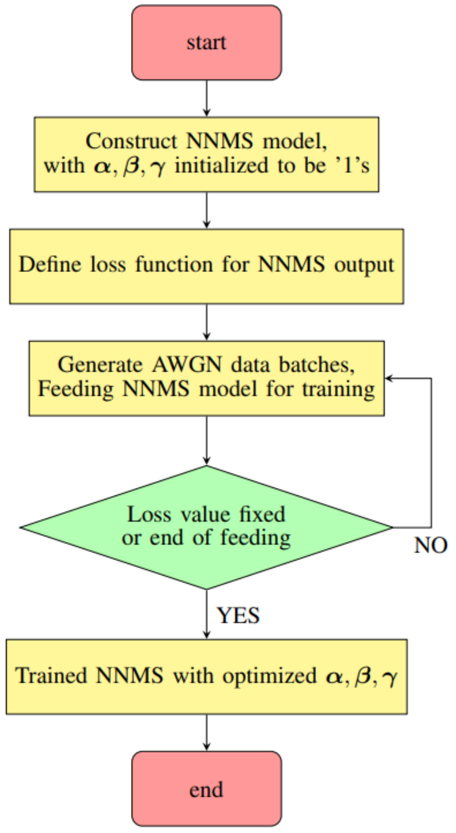

start
Construct NNMS model, with $\boldsymbol{\alpha}, \boldsymbol{\beta}, \boldsymbol{\gamma}$ initialized to be '1's
Define loss function for NNMS output
Generate AWGN data batches, Feeding NNMS model for training
Loss value fixed or end of feeding
NO
YES
Trained NNMS with optimized $\boldsymbol{\alpha}, \boldsymbol{\beta}, \boldsymbol{\gamma}$
end

Fig. 2. Flow chart of training NNMS

1) Choice of loss function: One prerequisite of training is to select a viable loss function among the ones off the shelf, on which we apply a stochastic gradient descent (SGD) method to optimize these trainable parameters.

Given the authentic and estimated binary vectors $\boldsymbol{c}$ and $\widehat{\boldsymbol{c}}^{(j)}, j = 1, 2, ...T$, the hybrid loss function is defined as follows,

$$\ell(\boldsymbol{c}, \widehat{\boldsymbol{c}}) = \rho \ell_{ce}(\boldsymbol{c}, \widehat{\boldsymbol{c}}) + (1-\rho)\kappa \ell_{mse}(\boldsymbol{c}, \widehat{\boldsymbol{c}}) \qquad (8)$$

where the weight factor $\rho = 0.2$ and balance factor $\kappa = 100$, and its cross entropy term is

$$\ell_{ce}(\boldsymbol{c}, \widehat{\boldsymbol{c}}) = \frac{1}{NT} \sum_{i=1}^{N} \sum_{j=1}^{T} \sum_{z=0}^{1} \left(p(c_i = z) \log \frac{1}{p(\widehat{c}_i^{(j)} = z)} \right)$$

and the other mean squared error (MSE) term is

$$\ell_{mse}(\boldsymbol{c}, \widehat{\boldsymbol{c}}) = \frac{1}{N} \sum_{i=1}^{N} \sum_{z=0}^{1} p(c_i = z)(p(\widehat{c}_i^{(T)} = z) - z)^2$$

2) Generating of training data: Notably, the assumption of all-zeros codeword transmitted in training, has no impact on the validness of the followed inferences, in the sense the trained NNMS can deal with the cases of any codeword sending equally well, This unique property, attributed to satisfying the message passing symmetry conditions [23], greatly simplifies the training process. Correspondingly, the loss definition 8 reduces to

$$\ell(\boldsymbol{c}, \widehat{\boldsymbol{c}}) = \frac{\rho}{NT} \sum_{i=1}^{N} \sum_{j=1}^{T} \log \frac{1}{p(\widehat{c}_i^{(j)} = 0)} + \frac{(1-\rho)\kappa}{N} \sum_{i=1}^{N} p^2(\widehat{c}_i^{(T)} = 0) \quad (9)$$

$$Y_i \sim \mathcal{N}(1, \sigma_i^2)$$

$$Z_i = \frac{2}{{\sigma_i}^2} Y_i \sim \mathcal{N}\left(\frac{2}{{\sigma_i}^2}, \frac{4}{{\sigma_i}^2}\right)$$

$$(\mathrm{SNR})_i = \left(\frac{E_b}{N_0}\right)_i = 10 \log_{10} \frac{N}{2K\sigma_i^2}$$

$$f(Z) = \frac{1}{I} \sum_{i=1}^{I} f_{Z_i}(Z)$$

$$\mu_a = E[Z] = \frac{1}{I} \sum_{i=1}^{I} E[Z_i] = \sum_{i=1}^{I} \frac{2}{I\sigma_i^2}$$
$$= \frac{1}{\sigma_e - \sigma_s} \int_{\sigma_s}^{\sigma_e} \frac{2}{x^2} dx, I \to +\infty$$

$$\sigma_a^2 = D[Z] = \frac{1}{I} \sum_{i=1}^{I} \left((\frac{2}{\sigma_i^2})^2 + \frac{4}{\sigma_i^2} \right) - \mu_a^2$$
$$= \frac{1}{\sigma_e - \sigma_s} \int_{\sigma_s}^{\sigma_e} 4(\frac{1}{x^4} + \frac{1}{x^2}) dx, I \to +\infty$$

Or just use equal probability sampling from quantized SNR interval – a simplistic approach

From the article *"A recipe of training neural network-based LDPC decoders" by Guangwen Li, Xiao Yu.*

TABLE I
TRAINING SETTINGS OF THREE CODES

Three Codes	SNR Range (dB)	Minibatch		# of epochs	T
		number	***size***		
code A	[3.2,3.8]	2000	64	6	20
code B	[2.8,3.2]	2000	64	6	10
code C	[1.8,2.4]	2000	32	6	15/20

What’s NNMS efficiency?

Experimental results from the mentioned article: code A: a WiMAX (802.16) LDPC code (1056,880), **code B**: a finite geometry LDPC code (1023,781), **code C**: a Gallager LDPC code (1008,504). The abbreviation **’SNNMS**’ refers to the neural decoder with a shared trainable parameter for each iteration at the check nodes side, that is, $\alpha(l) = 1$, $\beta(1) = 1$, $\gamma(l) = \gamma(l)$. Likewise, **’UNNMS**’ is the one with a unique trainable parameter across all iterations, or $\alpha(l) = 1$, $\beta(1) = 1$, $\gamma(l) = \gamma$, while **’ANNMS**’ denotes a full-loaded NNMS decoder without any trimming.

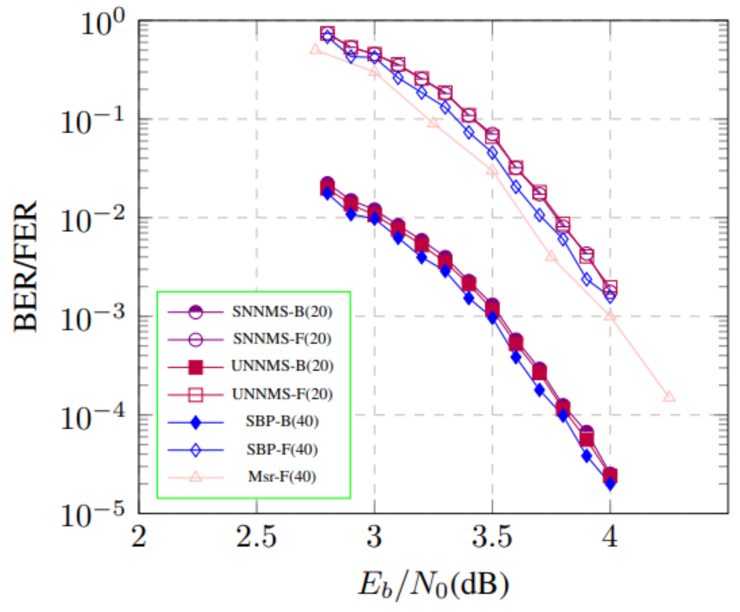


Fig. 8. BER/FER comparison for the decoding schemes of code A

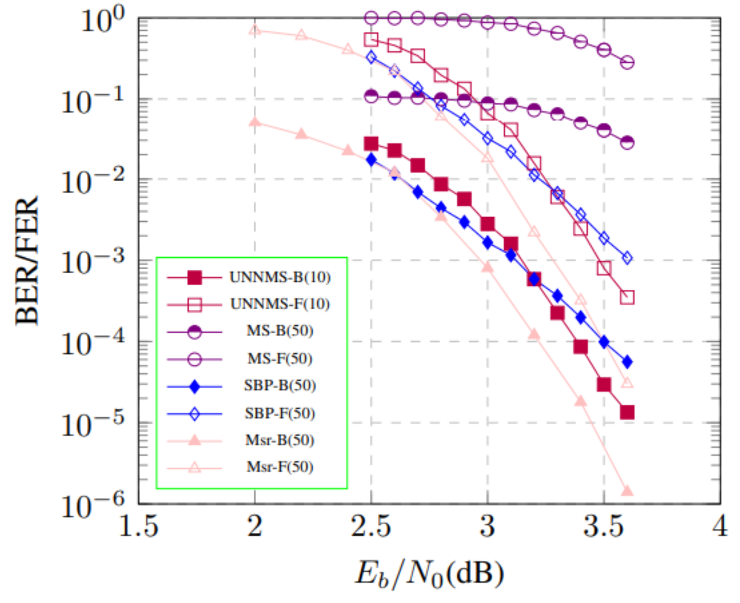


Fig. 9. BER/FER comparison for the decoding schemes of code B

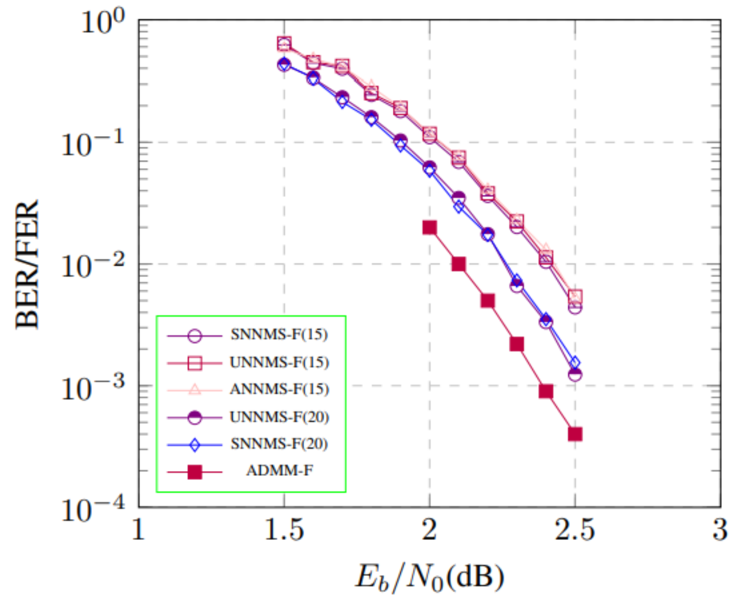


Fig. 10. BER/FER comparison for the decoding schemes of code C

From the article *“A recipe of training neural network-based LDPC decoders” by Guangwen Li, Xiao Yu.*

How else can we modify NNMS?

In the article *"Learning to Decode Protograph LDPC Codes" by Jincheng Dai, et.al.* a lot of modifications of NNMS algorithm for the protograph-based LDPC codes are described. The baseline idea behind all the approaches in the article is to share weights between edges in single protograph, but we can share other weights too, and so other variations are proposed to modify the training process of NNMS:

$$\ell_{e=(v,c)}^{(i_v)} = \ell_v + \sum_{e'=(v,c'),c'\neq c} \ell_{e'}^{((i-1)_c)},$$

$$\ell_{e=(v,c)}^{(i_c)} = \left(\prod_{e'=(v',c),v'\neq v} \operatorname{sgn}\left(\ell_{e'}^{(i_v)}\right)\right) \times$$
$$\mathbf{ReLU}\left(\alpha_e^{(i)} \times \min_{e'=(v',c),v'\neq v} \left|\ell_{e'}^{(i_v)}\right| - \beta_e^{(i)}\right),$$

$$\mathbf{ReLU}(x) = \max(x, 0).$$

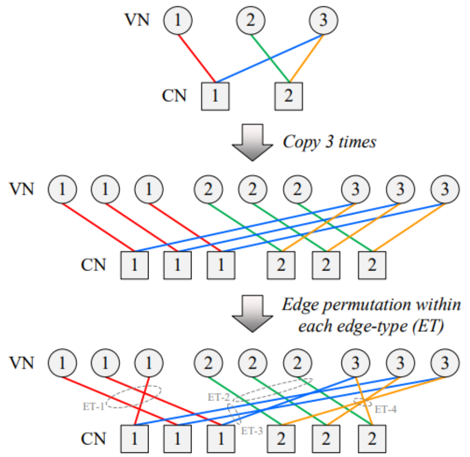


Fig. 1. A graphical demonstration of protograph LDPC codes.

And relaxation approach:

$$\ell_{e=(v,c)}^{(i_v)} = \gamma_e^{(i)} \ell_e^{((i-1)_v)} + \left(1 - \gamma_e^{(i)}\right) \tilde{\ell}_e^{(i_v)},$$

where

$$\tilde{\ell}_e^{(i_v)} = \ell_v + \sum_{e'=(v,c'),c'\neq c} \ell_{e'}^{((i-1)_c)},$$

TABLE I
FOUR TYPES OF NEURAL MS DECODER.

Type	Description	Definition (i' and i'' denote two different iterations)
Type-I	neural NOMS	$\boldsymbol{\alpha}^{(i')} \neq \boldsymbol{\alpha}^{(i'')}$, $\boldsymbol{\beta}^{(i')} \neq \boldsymbol{\beta}^{(i'')}$ with $i' \neq i''$
Type-II	simplified neural NOMS	$\boldsymbol{\alpha}^{(i)} = \alpha^{(i)}$, $\boldsymbol{\beta}^{(i)} = \beta^{(i)}$, $\alpha^{(i')} \neq \alpha^{(i'')}$, $\beta^{(i')} \neq \beta^{(i'')}$ with $i' \neq i''$
Type-III	simplified neural NMS	$\boldsymbol{\alpha}^{(i)} = \alpha^{(i)}$, $\boldsymbol{\beta}^{(i)} = 0$, $\alpha^{(i')} \neq \alpha^{(i'')}$ with $i' \neq i''$
Type-IV	simplified neural OMS	$\boldsymbol{\alpha}^{(i)} = 1$, $\boldsymbol{\beta}^{(i)} = \beta^{(i)}$, $\beta^{(i')} \neq \beta^{(i'')}$ with $i' \neq i''$

TABLE II
TWO TYPES OF NEURAL MS DECODER WITH DAMPING FACTORS.

Type	Description	Definition (i' and i'' denote two different iterations)
Type-V	neural NOMS with damping	$\boldsymbol{\alpha}^{(i')} \neq \boldsymbol{\alpha}^{(i'')}$, $\boldsymbol{\beta}^{(i')} \neq \boldsymbol{\beta}^{(i'')}$, $\boldsymbol{\gamma}^{(i')} \neq \boldsymbol{\gamma}^{(i'')}$ with $i' \neq i''$
Type-VI	simplified neural NOMS with damping	$\boldsymbol{\alpha}^{(i')} \neq \boldsymbol{\alpha}^{(i'')}$, $\boldsymbol{\beta}^{(i')} \neq \boldsymbol{\beta}^{(i'')}$, $\boldsymbol{\gamma}^{(i)} = \gamma^{(i)}$, $\gamma^{(i')} \neq \gamma^{(i'')}$ with $i' \neq i''$

How else can we modify NNMS?

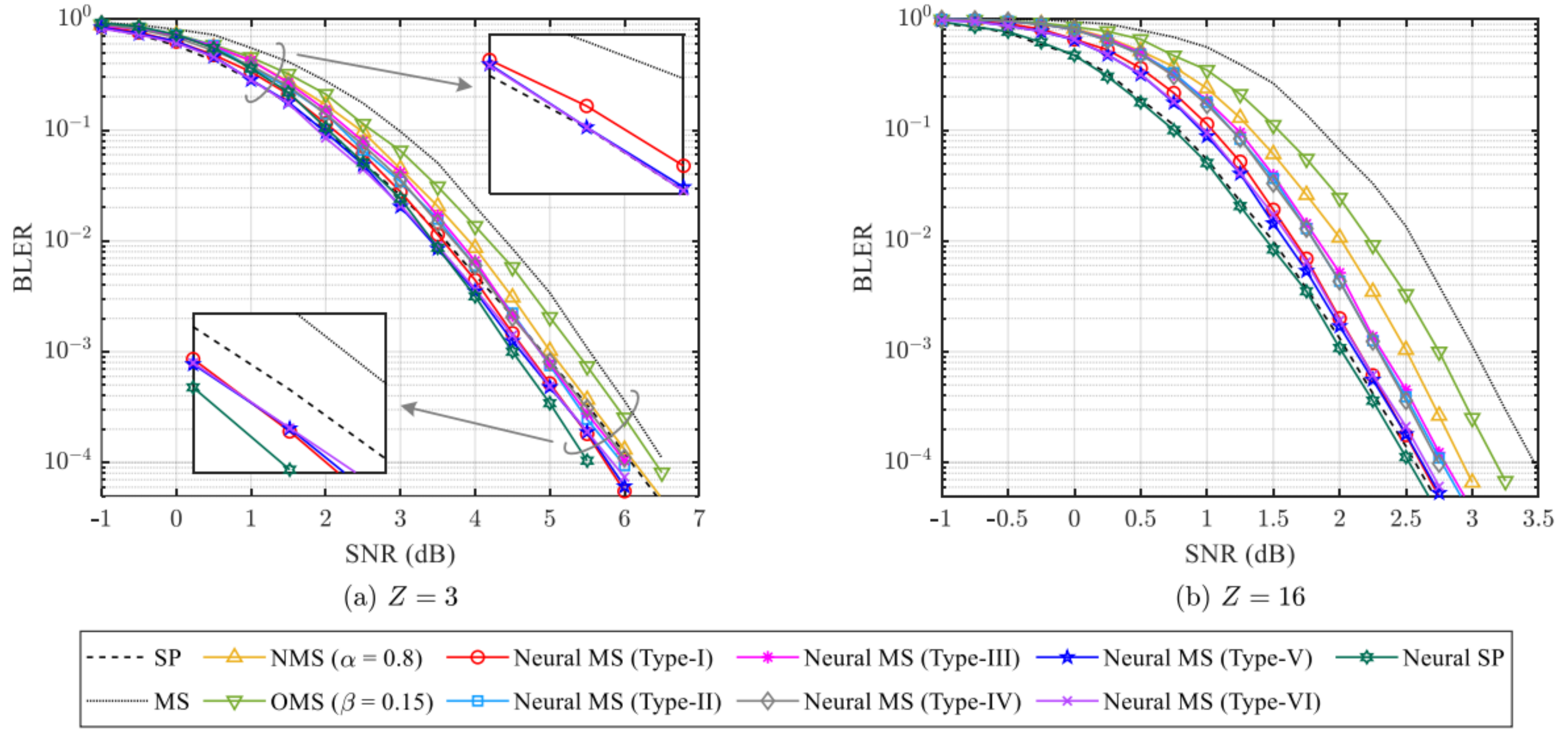


Fig. 10. BLER performance of BG2 codes with lifting sizes $Z = 3$ in (a) and $Z = 16$ in (b) under the AWGN channel, where the number of iterations is $I = 25$.

From the article *“Learning to Decode Protograph LDPC Codes” by Jincheng Dai, et.al.*; *Project on GitHub: https://github.com/KyrieTan/Neural-Protograph-LDPC-Decoding*

What else to read on NNMS decoder?

"Deep Learning Methods for Improved Decoding of Linear Codes" by Eliya Nachmani et.al. may be considered as one of the origins for the experiments with NNMS, most of evaluations are BCH based. Some recurrent networks architectures are also evaluated.

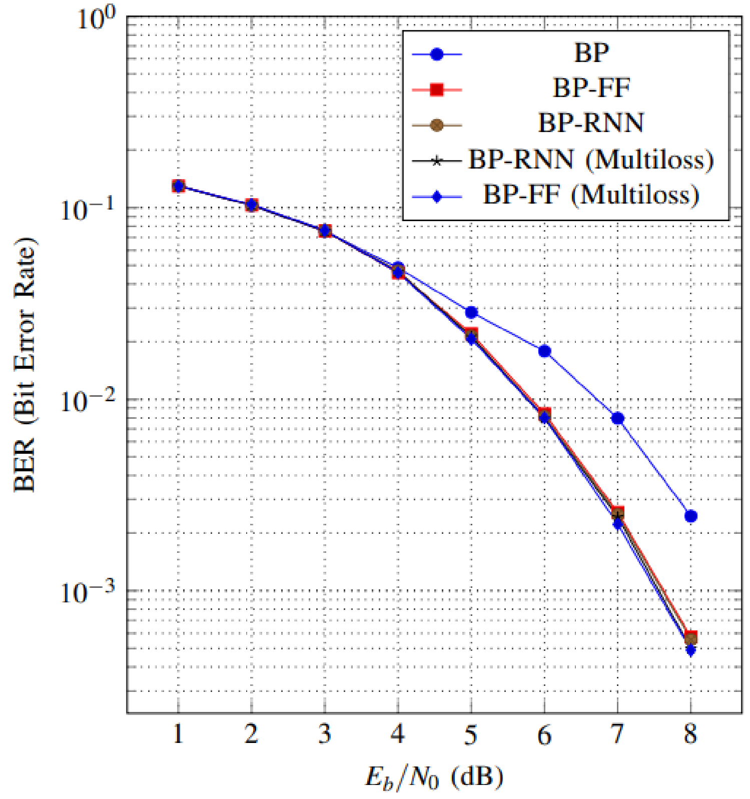


Fig. 9. BER results for BCH(127,64) code trained with right-regular parity check matrix.

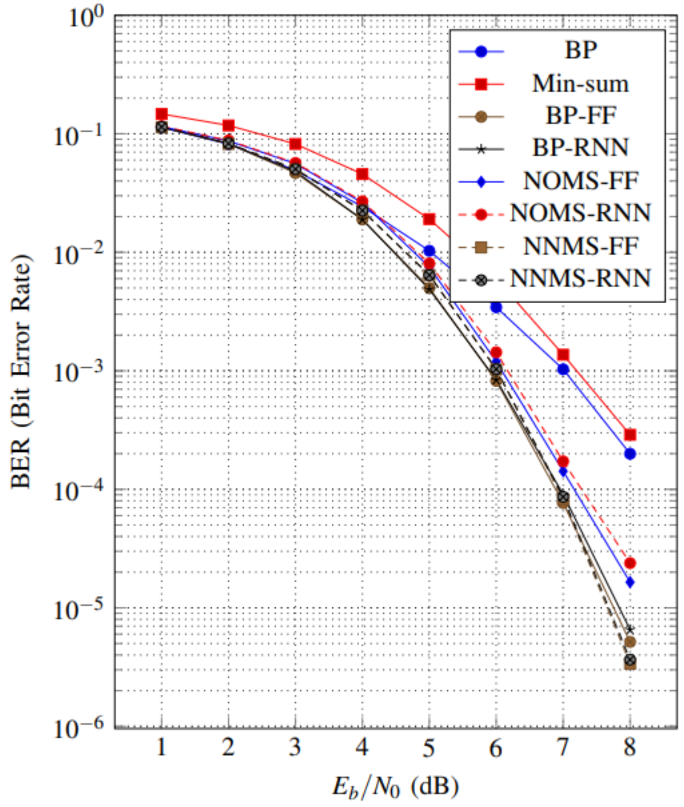
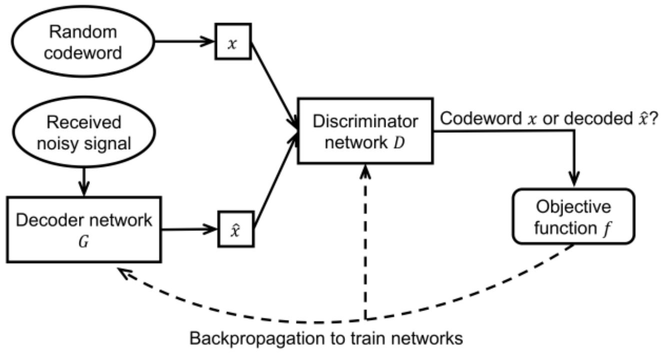


Fig. 13. Performance comparison of BP and min-sum decoders for BCH (63,36) code.

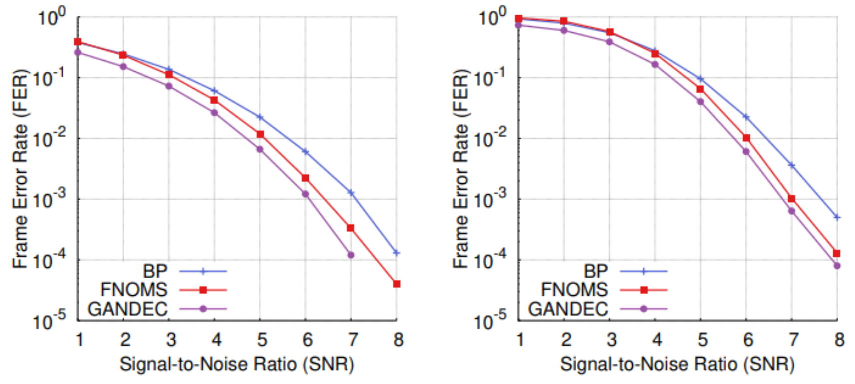
What else to read on NNMS decoder?

"Adversarial Neural Networks for Error Correcting Codes" by Hung T. Nguyen et.al is devoted to using GAN architecture to improve NNMS decoder.



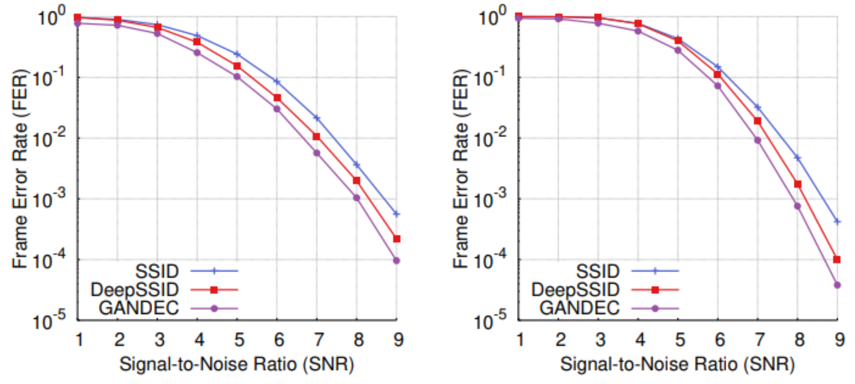
Random codeword
x
Received noisy signal
Decoder network G
$\hat{x}$
Discriminator network D
Codeword x or decoded $\hat{x}$?
Objective function f
Backpropagation to train networks

Fig. 2: Proposed framework with decoder and discriminator networks competing and improving each other.



(a) BCH(15,11) (b) BCH(63,45)

Fig. 3: Performance of different decoders on BCH codes.

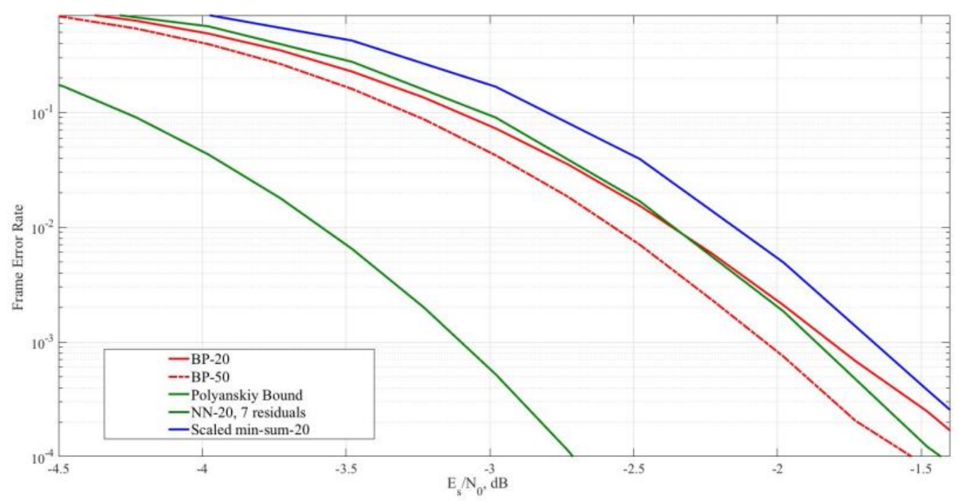


(a) RS(15,11) (b) RS(31,27)

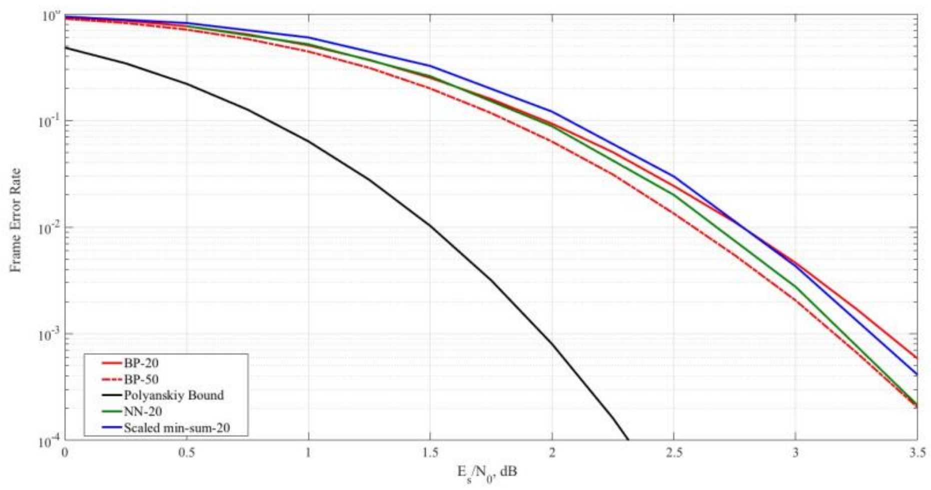
What else to read on NNMS decoder?

[General overview of the NNMS approaches from Skoltech (Frolov, Andreev et.al.).]()

Simulation results, 5G LDPC codes, BG2, K=120, R=0.2



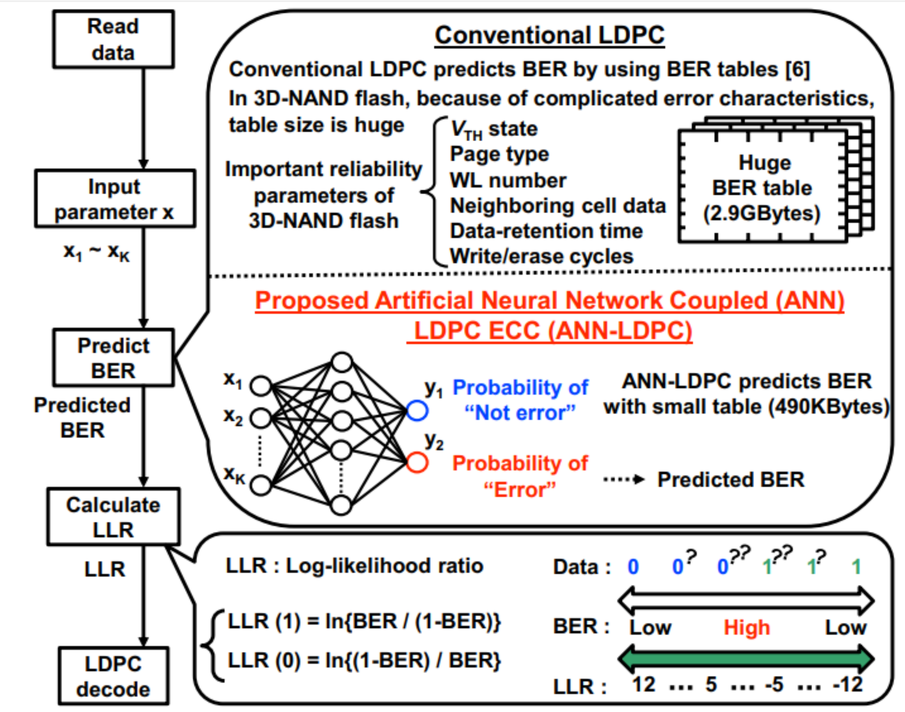
Simulation results, 5G LDPC codes, BG2, K=120, R=0.5



How to improve LLRs with NN?

Interesting NN usage is proposed in article [“9.1x Error Acceptable Adaptive Artificial Neural Network Coupled LDPC ECC for Charge-trap and Floating-gate 3D-NAND Flash Memories” by Toshiki Nakamura et. al.]()

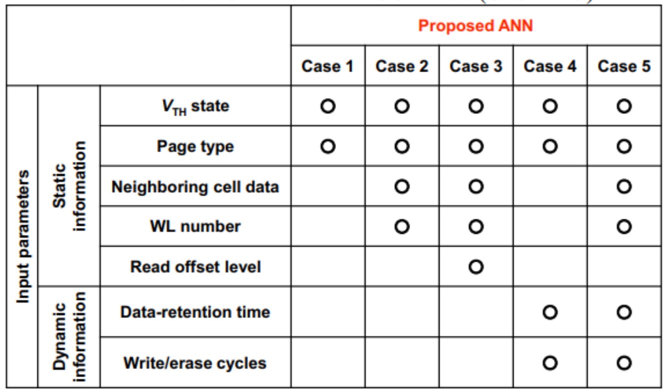
Application scope is 3D NAND memory cells, which also adapt LDPC in modern SSD, however input LLR highly affect SSD performance (and it degrades quickly with SSD wearing out in some amount of time). Authors propose to modify SSD controller, which is currently storing BER of the cells (to improve decoding) to a NN model which can predict RBER and propose LLR as LDPC input.



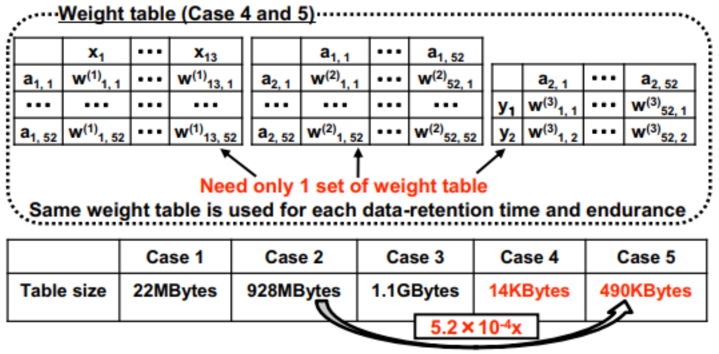
Read data
Input parameter x
$x_1 \sim x_K$
Predict BER
Predicted BER
Calculate LLR
LLR
LDPC decode
Conventional LDPC
Conventional LDPC predicts BER by using BER tables [6]
In 3D-NAND flash, because of complicated error characteristics, table size is huge
Important reliability parameters of 3D-NAND flash
V_{TH} state
Page type
WL number
Neighboring cell data
Data-retention time
Write/erase cycles
Huge BER table (2.9GBytes)
Proposed Artificial Neural Network Coupled (ANN) LDPC ECC (ANN-LDPC)
x_1
x_2
x_K
y_1
Probability of “Not error”
y_2
Probability of “Error”
ANN-LDPC predicts BER with small table (490KBytes)
Predicted BER
LLR : Log-likelihood ratio
LLR (1) = ln{BER / (1-BER)}
LLR (0) = ln{(1-BER) / BER}
Data : 0 0? 0?? 1?? 1? 1
BER : Low High Low
LLR : 12 ··· 5 ··· -5 ··· -12

Fig. 2. Proposed Artificial Neural Network Coupled (ANN) LDPC ECC (ANN-LDPC ECC) which precisely and automatically estimates BER and LLR. By the precise LLR, LDPC decoder effectively corrects errors.

TABLE I. INPUT PARAMETERS OF ANN (CASE 1 - 5)



			Proposed ANN				
			Case 1	Case 2	Case 3	Case 4	Case 5
Input parameters	Static information	V_{TH} state	○	○	○	○	○
		Page type	○	○	○	○	○
		Neighboring cell data		○	○		○
		WL number		○	○		○
		Read offset level			○		
	Dynamic information	Data-retention time				○	○
		Write/erase cycles				○	○



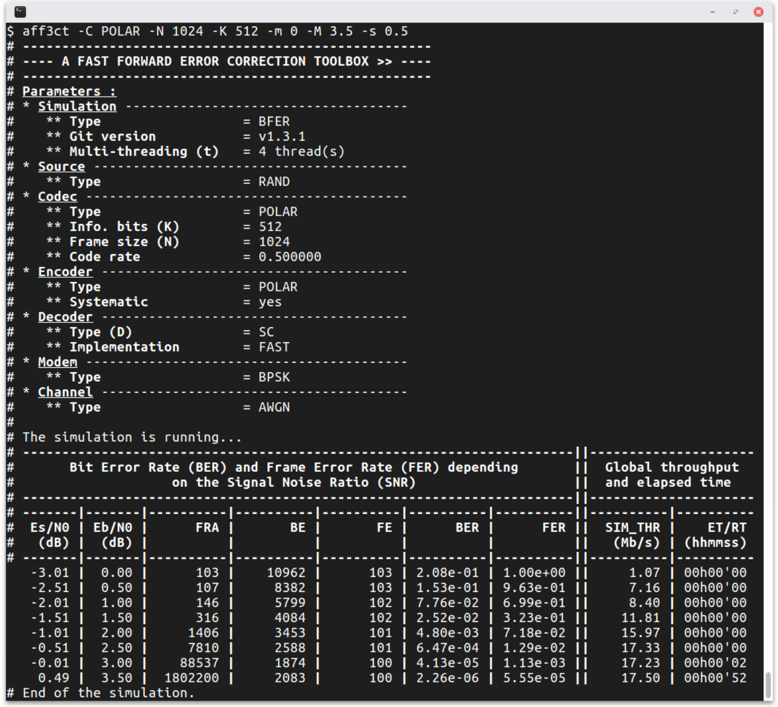
Weight table (Case 4 and 5)
x_1 ··· x_{13}
$a_{1,1}$ $w^{(1)}_{1,1}$ ··· $w^{(1)}_{13,1}$
$a_{1,52}$ $w^{(1)}_{1,52}$ ··· $w^{(1)}_{13,52}$
$a_{1,1}$ ··· $a_{1,52}$
$a_{2,1}$ $w^{(2)}_{1,1}$ ··· $w^{(2)}_{52,1}$
$a_{2,52}$ $w^{(2)}_{1,52}$ ··· $w^{(2)}_{52,52}$
$a_{2,1}$ ··· $a_{2,52}$
y_1 $w^{(3)}_{1,1}$ ··· $w^{(3)}_{52,1}$
y_2 $w^{(3)}_{1,2}$ ··· $w^{(3)}_{52,2}$
Need only 1 set of weight table
Same weight table is used for each data-retention time and endurance
5.2×10^{-4}x

	Case 1	Case 2	Case 3	Case 4	Case 5
Table size	22MBytes	928MBytes	1.1GBytes	14KBytes	490KBytes

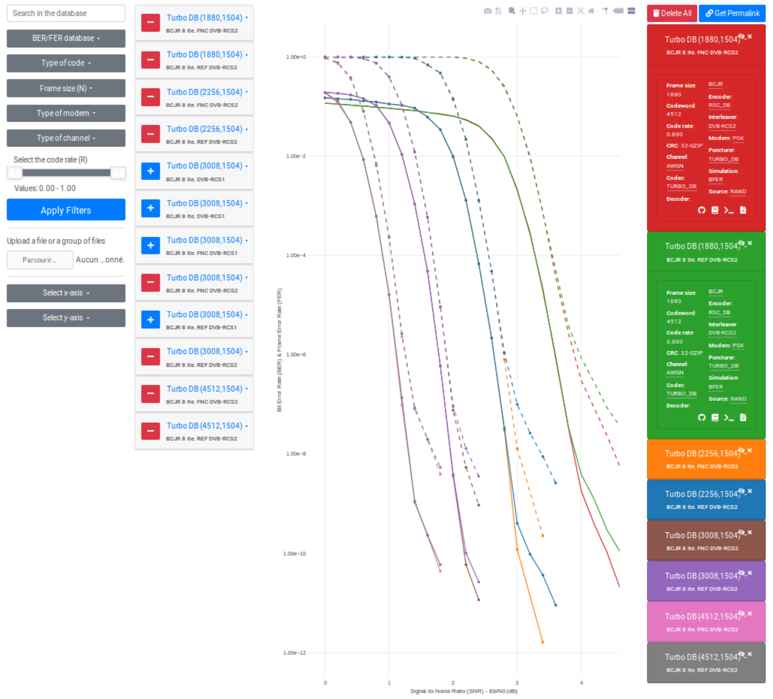
What can be used for ECC modeling?

If you want to try out some ECC approaches you can experiment with open-source simulation tool called AFF3CT.

AFF3CT is an Open-source software (MIT license) dedicated to the Forward Error Correction (FEC or channel coding) simulations. It is written in C++11 and it supports a large range of codes: from the well-spread Turbo codes to the new Polar codes including the Low-Density Parity-Check (LDPC) codes. A particular emphasis is given to the simulation throughput performance (hundreds of Mb/s on today's CPUs) and the portability of the code.



```
$ aff3ct -C POLAR -N 1024 -K 512 -m 0 -M 3.5 -s 0.5
# -------------------------------------------------
# ---- A FAST FORWARD ERROR CORRECTION TOOLBOX >> ----
# -------------------------------------------------
# Parameters :
# * Simulation ----------------------------------------
#    ** Type                    = BFER
#    ** Git version             = v1.3.1
#    ** Multi-threading (t)     = 4 thread(s)
# * Source --------------------------------------------
#    ** Type                    = RAND
# * Codec ---------------------------------------------
#    ** Type                    = POLAR
#    ** Info. bits (K)          = 512
#    ** Frame size (N)          = 1024
#    ** Code rate               = 0.500000
# * Encoder -------------------------------------------
#    ** Type                    = POLAR
#    ** Systematic              = yes
# * Decoder -------------------------------------------
#    ** Type (D)                = SC
#    ** Implementation          = FAST
# * Modem ---------------------------------------------
#    ** Type                    = BPSK
# * Channel -------------------------------------------
#    ** Type                    = AWGN
#
# The simulation is running...
# ---------------------------------------------------------------------||---------------------
#       Bit Error Rate (BER) and Frame Error Rate (FER) depending      ||  Global throughput
#                    on the Signal Noise Ratio (SNR)                   ||  and elapsed time
# ---------------------------------------------------------------------||---------------------
# -------|-------|----------|----------|----------|----------|----------||----------|----------
#  Es/N0 | Eb/N0 |      FRA |       BE |       FE |      BER |      FER ||  SIM_THR |    ET/RT
#   (dB) |  (dB) |          |          |          |          |          ||   (Mb/s) | (hhmmss)
# -------|-------|----------|----------|----------|----------|----------||----------|----------
   -3.01 |  0.00 |      103 |    10962 |      103 | 2.08e-01 | 1.00e+00 ||     1.07 | 00h00'00
   -2.51 |  0.50 |      107 |     8382 |      103 | 1.53e-01 | 9.63e-01 ||     7.16 | 00h00'00
   -2.01 |  1.00 |      146 |     5799 |      102 | 7.76e-02 | 6.99e-01 ||     8.40 | 00h00'00
   -1.51 |  1.50 |      316 |     4084 |      102 | 2.52e-02 | 3.23e-01 ||    11.81 | 00h00'00
   -1.01 |  2.00 |     1406 |     3453 |      101 | 4.80e-03 | 7.18e-02 ||    15.97 | 00h00'00
   -0.51 |  2.50 |     7810 |     2588 |      101 | 6.47e-04 | 1.29e-02 ||    17.33 | 00h00'00
   -0.01 |  3.00 |    88537 |     1874 |      100 | 4.13e-05 | 1.13e-03 ||    17.23 | 00h00'02
    0.49 |  3.50 |  1802200 |     2083 |      100 | 2.26e-06 | 5.55e-05 ||    17.50 | 00h00'52
# End of the simulation.
```



Thank you for your attention!