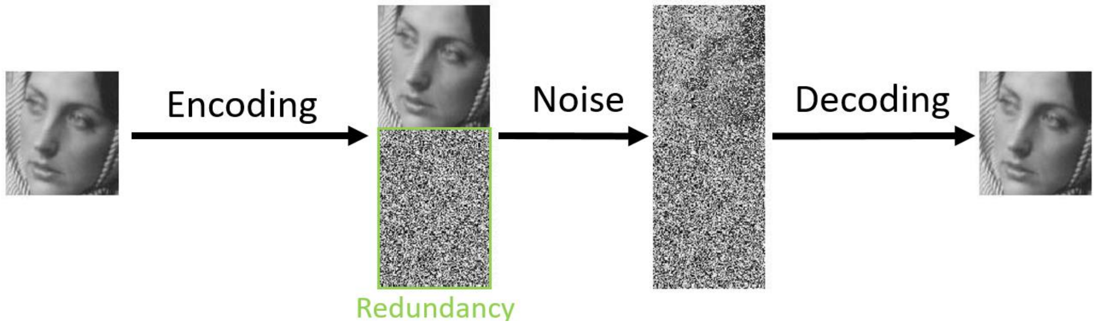
New Methods for Error Correction Codes with Deep Learning

Yoni Choukroun

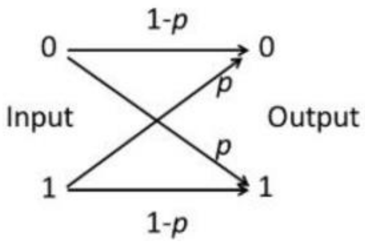
School of Computer Science, Tel Aviv University

Error Correction Code: Motivation

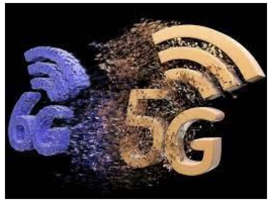
- Error-correcting codes (ECC) are clever ways of **representing data** so that one can **recover** the original information even if parts of it are corrupted.
- The basic idea is to **judiciously *add redundancy*** (**encoding**) so that the original information can be recovered (**decoding**) even when parts of the data have been corrupted (**noise**).

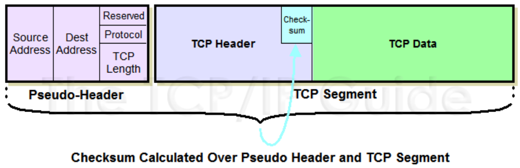


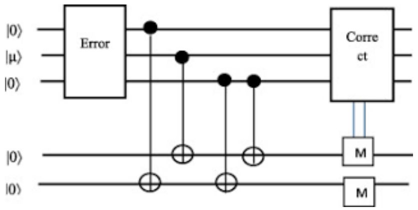
- E.g., assume we want to transmit **one single bit** b and the probability of bit flip is p.
  - Using the **3-repetition code** (sending $[b, b, b]$), the probability of error is **reduced** to ~ $3p^2$ (but transmission rate $1/3$).

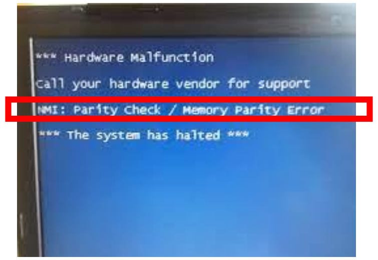


- ECC is present in all types of communication (transmission over *space*) and storage (transmission over *time*)



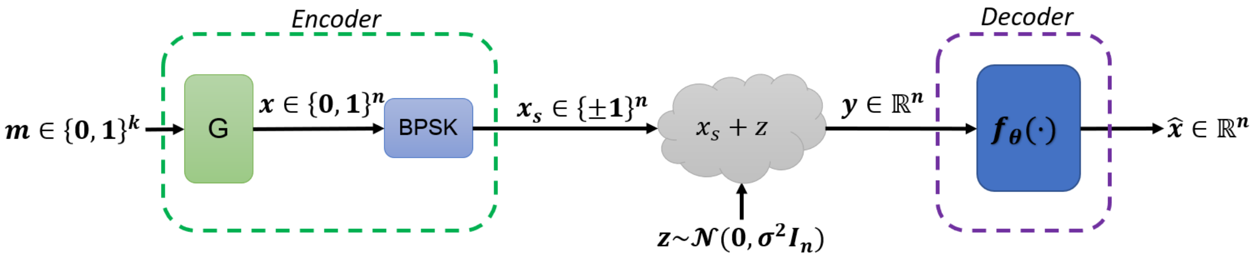






Error Correction Code: Setting

- **Goal**: *allow reliable data transmission over a noisy communication channel.*
  - For a *Binary Linear Block Code* $(\boldsymbol{n}, \boldsymbol{k})$
  - A desired *binary* message $\boldsymbol{m} \sim \boldsymbol{Bern}^{\boldsymbol{k}}(\boldsymbol{0.5})$ is *encoded* to a *redundant* **codeword**
    - $\boldsymbol{x} \equiv \mathbf{G}\boldsymbol{m}.$ Multiplication *over **GF(2)***, i.e., $\boldsymbol{x} = \mathbf{mod}(\mathbf{G}\boldsymbol{m}, \mathbf{2})$, $\mathbf{G} \in \{\mathbf{0}, \mathbf{1}\}^{\boldsymbol{n} \times \boldsymbol{k}}$
      - E.g., the natural **structure** of images allowing their denoising is here inserted ***artificially*** into the data to allow optimal decoding.
  - The codeword $\boldsymbol{x}$ is modulated (via BPSK) for transmission
    - $\boldsymbol{x_s} = \mathbf{1} - \mathbf{2}\boldsymbol{x}.$
  - The channel adds (AWGN) noise $\boldsymbol{z}$
    - $\boldsymbol{y} = \boldsymbol{x_s} + \boldsymbol{z}.$
  - The (parameterized) *decoder* $\boldsymbol{f_\theta}(\boldsymbol{y})$ *aims at retrieving the original codeword* $\boldsymbol{x}$ from $\boldsymbol{y}$.



- For the 3-repetition code (3,1):
  1. $m = 1 \in \{0,1\}$
  2. $\mathrm{G} = (1,1,1)^T$
  3. $x = x_s = Gm = (1,1,1)$
  4. $y = x_s \oplus z = (1, -1, -1)$
  5. $\hat{x} = f(y) = \left[\frac{\sum_i y_i}{3}\right] = (1,1,1)$

- For the (7,4) Hamming code:

$$x = Gm \equiv_2 \begin{pmatrix} 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 \\ 0 \\ 1 \\ 1 \end{pmatrix} = \begin{pmatrix} 2 \\ 3 \\ 1 \\ 2 \\ 0 \\ 1 \\ 1 \end{pmatrix} = \begin{pmatrix} 0 \\ 1 \\ 1 \\ 0 \\ 0 \\ 1 \\ 1 \end{pmatrix}$$

$$y = x \oplus z = \begin{pmatrix} 0 \\ 1 \\ 1 \\ 0 \\ 0 \\ 1 \\ 1 \end{pmatrix} + \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 1 \\ 0 \\ 0 \end{pmatrix} = \begin{pmatrix} 0 \\ 1 \\ 1 \\ 0 \\ \boxed{1} \\ 1 \\ 1 \end{pmatrix}$$

Error Correction Code: Setting

- **How can I detect that an error occurred?**
  - Check if this is one of the 2^k codewords
  - The **parity check** matrix $\boldsymbol{H} \in \{\mathbf{0},\mathbf{1}\}^{(\boldsymbol{n}-\boldsymbol{k})\times\boldsymbol{n}}$ kernel defines the codewords
    - $\mathbf{G}^{\mathrm{T}}\boldsymbol{H} \equiv \mathbf{0} \Rightarrow \boldsymbol{Hx} \equiv \mathbf{0}$.

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

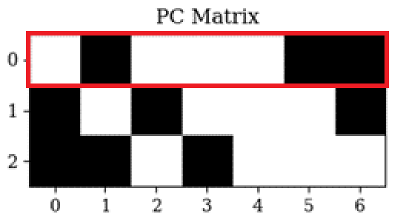
$$Hx \equiv_2 \begin{pmatrix} 1 & 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 \end{pmatrix} \begin{pmatrix} 0 \\ 1 \\ 1 \\ 0 \\ 0 \\ 1 \\ 1 \end{pmatrix} = \begin{pmatrix} 2 \\ 4 \\ 2 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}$$

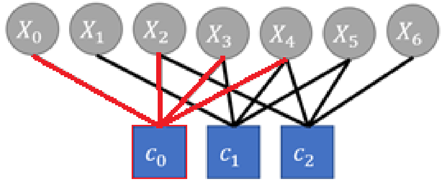
  - The parity check equations allow parity check ***errors** discovery*
    - $\boldsymbol{s} = \boldsymbol{H}(\boldsymbol{x} \oplus \boldsymbol{z_b}) = \boldsymbol{Hx} \oplus \boldsymbol{Hz_b} \equiv \boldsymbol{Hz_b} \in \{\mathbf{0},\mathbf{1}\}^{\boldsymbol{n}-\boldsymbol{k}}$.
    - This **binary** vector of parity check errors is called the ***SYNDROME***

$$s = Hy \equiv_2 \begin{pmatrix} 1 & 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 \end{pmatrix} \begin{pmatrix} 0 \\ 1 \\ 1 \\ 0 \\ \boxed{1} \\ 1 \\ 1 \end{pmatrix} = \begin{pmatrix} 3 \\ 4 \\ 3 \end{pmatrix} = \begin{pmatrix} \boxed{1} \\ 0 \\ \boxed{1} \end{pmatrix}$$

  - The **Tanner graph** is the (factor) graph representation of $\boldsymbol{H}$ (edge for 1 in each column)

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





Error Correction Code: Background

- Given a code (i.e., $\boldsymbol{H}$), the best possible decoder is defined by the **NP-hard maximum likelihood decoder** (search for the closest codeword among the $\mathbf{2}^{\boldsymbol{k}}$).

- Must have been done during the past 70 years in *information theory* for the design of **provably (asymptotical) optimal codes** and their efficient ($\leq$ polynomial time) decoding.

- Shannon's channel coding theorem (1948) shows that, given a noisy channel with capacity $\boldsymbol{C}$, if the information is transmitted at rate $\boldsymbol{R} = \boldsymbol{k}/\boldsymbol{n}$ such that $\boldsymbol{R} < \boldsymbol{C}$, then **there exists** a coding scheme that guarantees **negligible probability** of miscommunication.

- Recently, neural networks-based decoders (aka **neural decoders**) have shown promising results in this field.

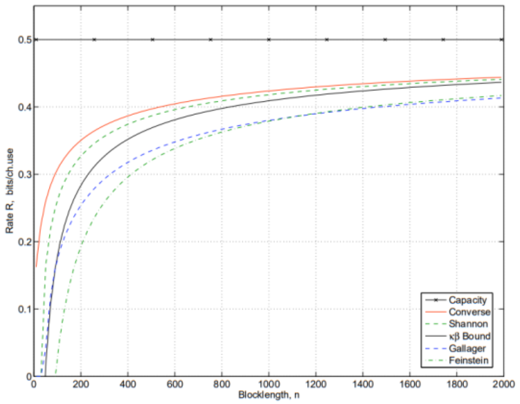
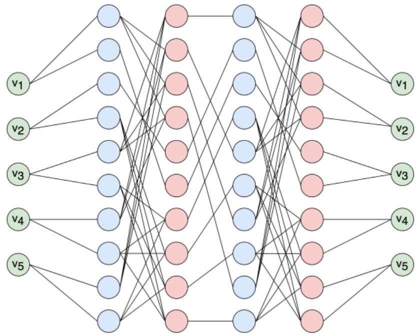


Fig. 6. Bounds for the AWGN channel, $SNR = 0$ dB, $\epsilon = 10^{-3}$.

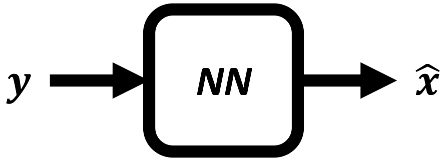
Neural Decoders

❖ ***Two** main families of neural decoders:*

- ***Model-based*** decoders implement *augmented* parameterized versions of the classical *Belief Propagation* decoder built upon the **Tanner graph** (graph representation of $\boldsymbol{H}$).
  - Pros:
    - Invariant to the codeword (robust to codewords overfitting)
    - Built on **iterative** (message-passing) legacy methods
  - Cons:
    - Suffers from ***heavy* and *restrictive* inductive bias**.
    - **Improvement vanishes** as the code length and the number of iterations increase



- ***Model-free*** decoders employ **general** types of neural network architectures (e.g., MLP, RNN)
  - Pros:
    - Total **freedom** in model design
  - Cons:
    1. **Overfitting** (*exponential number of codewords for training*)
    2. ***Difficulties for the model to learn the code.***
    3. ***Lack Iterative formulation***

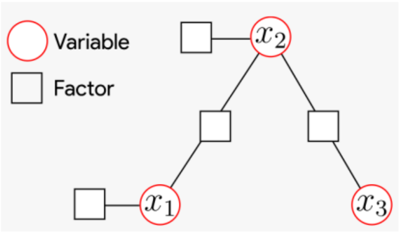


- Cons in Common :
  4. ***Lack Code invariance*** *(one decoder for each code)*
  5. ***Not adapted to modern ECC settings (e.g.,** quantum computing**)***
  6. ***Cannot design/learn the code***

Model-***BASED*** Decoders

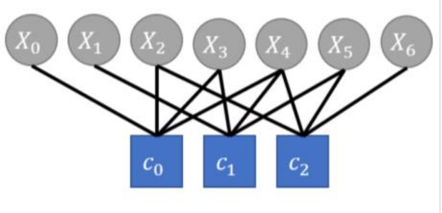
Belief Propagation (BP)

- A **factor graph** is a (bipartite) representation of a discrete probability distribution that takes advantage of conditional independencies between variables to make the representation *more compact*
  - *The Hammersley-Clifford theorem tells us that any positive joint distribution can be represented as a product of factors*



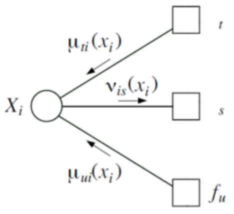
$p(x_1, x_2, x_3) \propto \phi_1(x_1)\phi_2(x_2)\,\phi_3(x_1, x_2)\,\phi_4(x_2, x_3)$

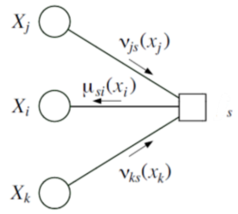
$$p(x_1, \dots, x_n) = \frac{\prod_{i=1}^{n-k} \phi_i(X_{S_i})}{Z}$$



- **Belief-Propagation** (Sum-product algorithm) allows efficient marginal inference $p(x_i)$ via variable elimination as *message-passing* over the factor graph.

$$\nu_{x_i \to \phi_k}(x_i) = \prod_{\phi_k \in N(i) \setminus \{\phi_k\}} \mu_{\phi_k \to X_i}(x_i) \quad \text{and} \quad \mu_{\phi_k \to X_i}(x_i) = \sum_{X_{S_k \setminus \{i\}}} \phi_s(X_{S_k}) \prod_{j \in S_k \setminus \{i\}} \nu_{X_j \to \phi_k}(x_j)$$





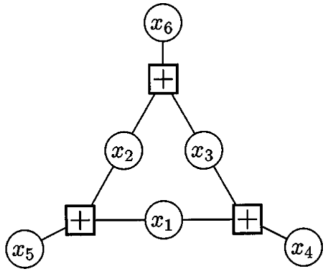
- For non-tree Bayesian graphs, the inference is not tractable. Thus, (**loopy**) belief-propagation is performed **iteratively** (until convergence of the beliefs to a local minimum)

$$b_i(x_i) = P(x_i) \propto \phi_i(x_i) \prod_{\phi_k \in N(i)} \mu^{T}_{\phi_k \to X_i}(x_i)$$

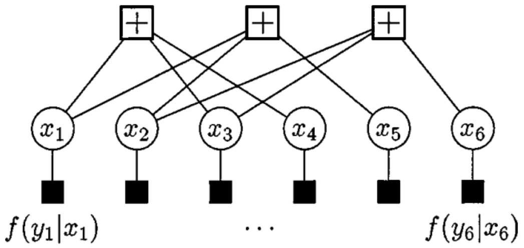
- Derivation:
  - Write the posterior factor graph $p(X) = \Pi_i \phi_i(X_{S_i})/Z$ and **the tree graph distribution** $q(X) = \Pi_i b_i(x_i)^{1-d_i} \Pi_j b_{S_j}\left(X_{S_j}\right)$
  - Minimize with respect to the (constrained) beliefs b_i, b_{S_j}: $KL(q||p) = H_q(X) - \sum_i \mathbb{E}_q \log(\phi_i(X_{S_i}))$

Belief Propagation Decoding

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



- Early (Gallager) application of BP to ECC with a Posteriori distribution
  - $P(x|y) \sim P(y|x)P(x) = \Pi_i f(y_i|x_i) P(\cup_k \ \bigoplus_{k \in N(j)} x_k = 0)$
  - $P(\cup_k \ \bigoplus_{k \in N(j)} x_k = 0) = \Pi_k P(\bigoplus_{k \in N(j)} x_k = 0)$
  - $\phi(X_{S_k}) = P(\bigoplus_{k \in N(j)} x_k = 0) = \frac{1}{2}(1 + \Pi_{k \in N(j) \setminus i} 2(v^{T}_{x_k \to C_j}(a)))$
    - Since the variables are independent



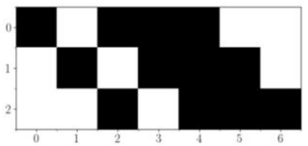
- Belief-propagation *decoding* is generally represented as a **Trellis** graph unrolling of the factor/**Tanner** graph (*log-likelihoods*).

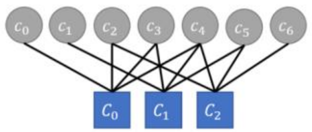
$$L_v = \log\left(\frac{\Pr(c_v = 1|y_v)}{\Pr(c_v = 0|y_v)}\right)$$

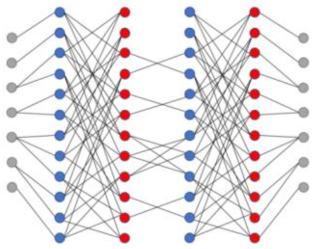
$$x_e^{2k+1} = x_{(c,v)}^{2k+1} = L_v + \sum_{e' \in N(v) \setminus \{(c,v)\}} x_{e'}^{2k}.$$

$$x_e^{2k} = x_{(c,v)}^{2k} = 2\text{arctanh}\left(\prod_{e' \in N(c) \setminus \{(c,v)\}} \tanh\left(\frac{x_{e'}^{2k-1}}{2}\right)\right)$$

$$o_v = L_v + \sum_{e' \in N(v)} x_{e'}^{2L}$$

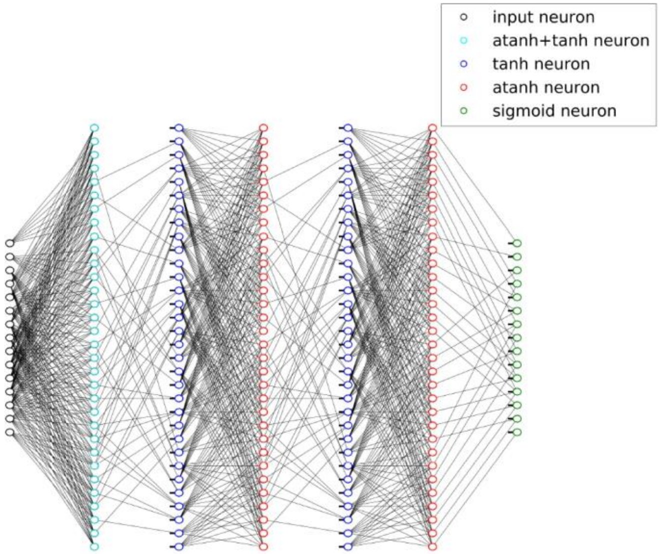






How can we augment BP decoders?

Learning to Decode Linear Codes Using Deep Learning, Nachmani et al. Allerton 2016



- Input - LLR

$$l_v = \log \frac{\Pr\left(C_v = 1 | y_v\right)}{\Pr\left(C_v = 0 | y_v\right)}$$

y_v is the channel output corresponding to the vth codebit, C_v.

- For odd i and $e = (v, c)$ -

$$x_{i,e=(v,c)} = l_v + \sum_{e'=(v,c'),\, c' \neq c} x_{i-1,e'}$$

- For even i and $e = (v, c)$ -

$$x_{i,e=(v,c)} = 2\tanh^{-1}\left(\prod_{e'=(v',c),\, v' \neq v} \tanh\left(\frac{x_{i-1,e'}}{2}\right)\right)$$

- The final vth output -

$$o_v = l_v + \sum_{e'=(v,c')} x_{2L,e'}$$

- For odd i and $e = (v, c)$ -

$$x_{i,e=(v,c)} = 2\tanh^{-1}\left(\prod_{e'=(v',c),\, v' \neq v} x_{i-1,e'}\right)$$

- For even i and $e = (v, c)$ -

$$x_{i,e=(v,c)} = \tanh\left(\frac{1}{2}\left(\boxed{w_{i,v}} l_v + \sum_{e'=(v,c'),\, c' \neq c} \boxed{w_{i,e,e'}} x_{i-1,e'}\right)\right)$$

- The final vth output -

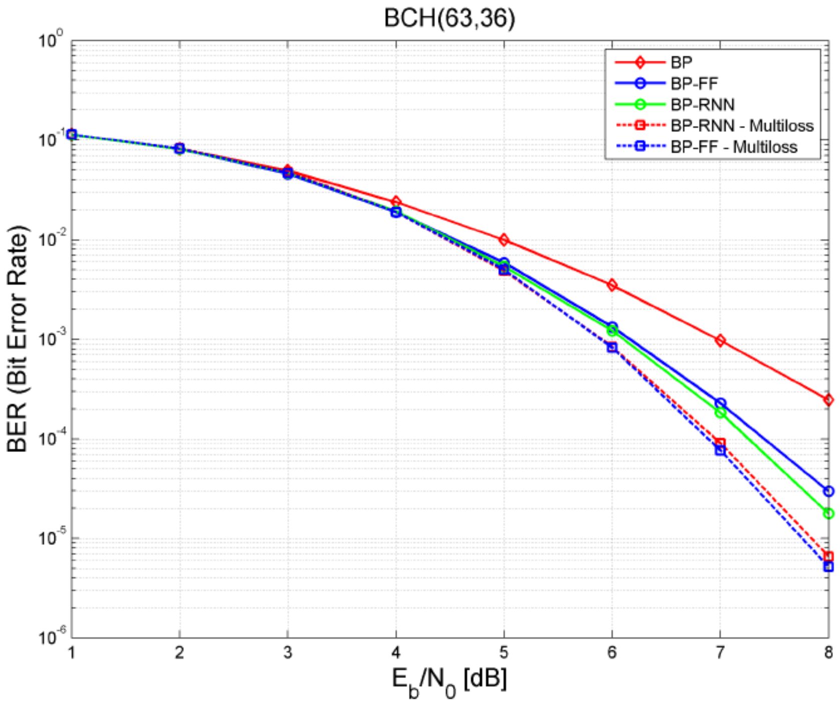
$$o_v = \sigma\left(\boxed{w_{2L+1,v}} l_v + \sum_{e'=(v,c')} \boxed{w_{2L+1,v,e'}} x_{2L,e'}\right)$$

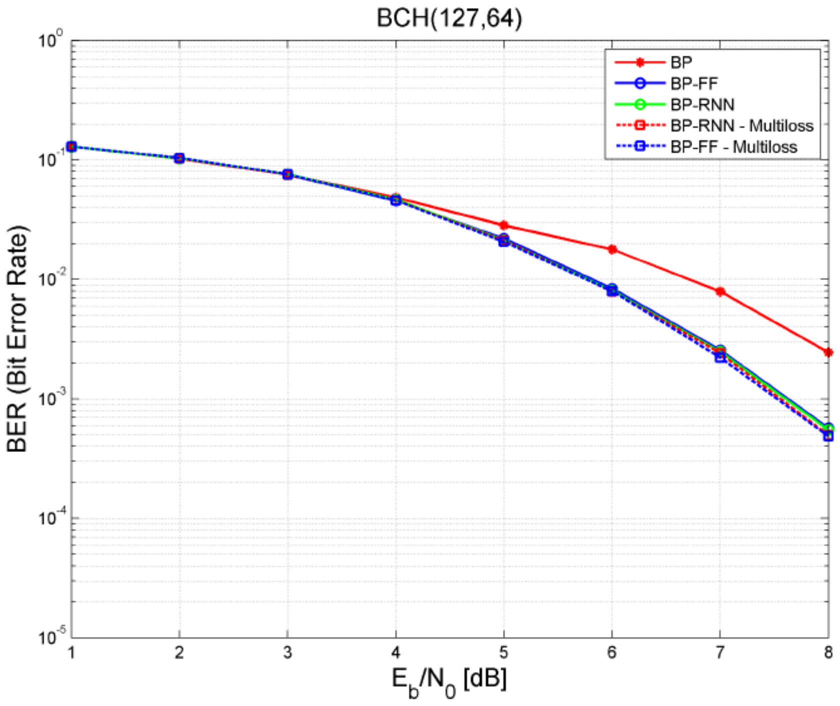
where $\sigma(x) \equiv (1 + e^{-x})^{-1}$

$$L(o, y) = -\frac{1}{N} \sum_{v=1}^{N} y_v \log(o_v) + (1 - y_v) \log(1 - o_v)$$

Learning to Decode Linear Codes Using Deep Learning,

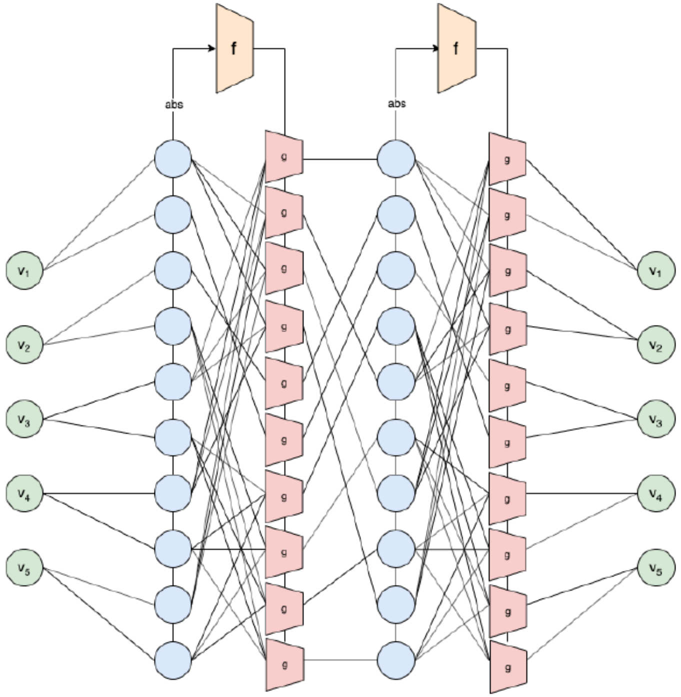
Nachmani et al.





- RNN == Shared parameters
- Multi loss $L(o,y) = -\frac{1}{N}\sum_{t=1}^{T}\sum_{v=1}^{N} y_v \log(o_{v,t}) + (1-y_v)\log(1-o_{v,t})$ for better gradient update (vanishing).
- Ensemble can also be applied

Hyper-graph-network decoders for block codes, Nachmani and Wolf. Neurips19

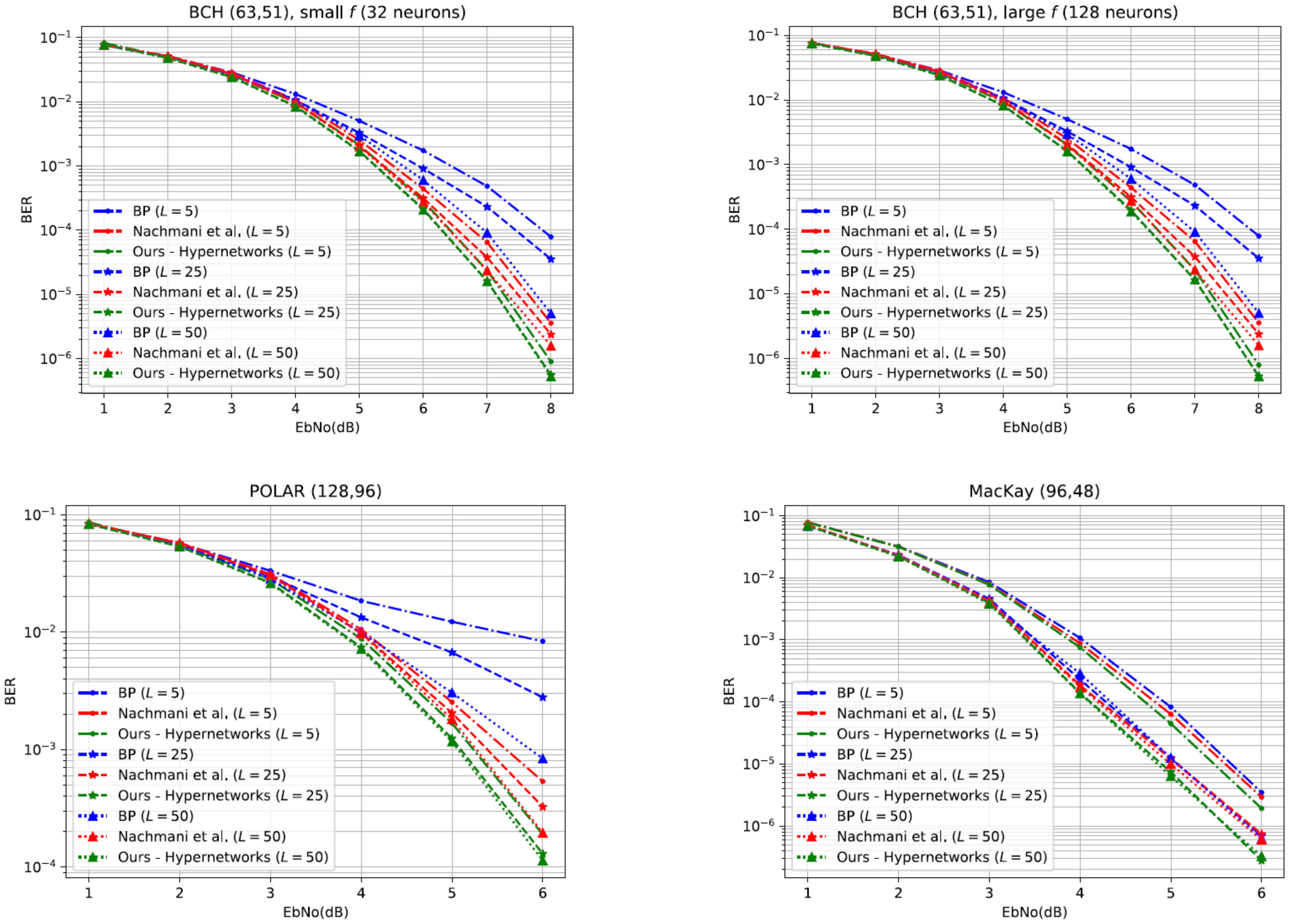


$$\theta_g^j = f(|x^{j-1}|, \theta_f)$$

$$x_e^j = x_{(c,v)}^j = g(l_v, x_{N(v,c)}^{j-1}, \theta_g^j)$$

- Replace arcthan with first order approximation for stable training
- The decoder maintains symmetry condition for zero codeword training (no overfitting)

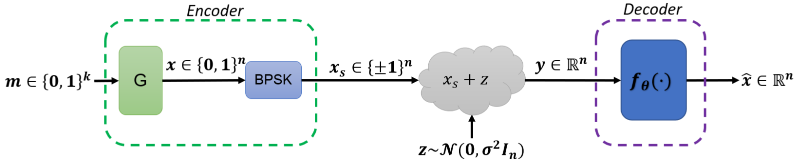
Hyper-graph-network decoders for block codes, Nachmani and Wolf. Neurips19



Model-***FREE*** Decoders

How can we remain robust to <u>overfitting</u>?

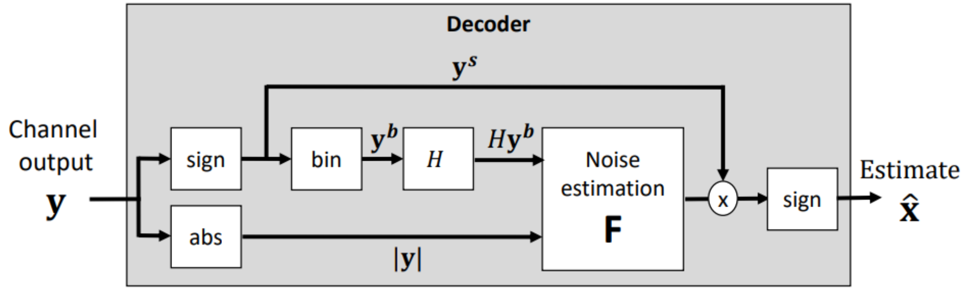
Solving *Model Free* Overfitting



- We want $h(\cdot)$ such that $P(h(y)|x) = P(h(y))$
- **Multiplicative noise**
  - The multiplicative noise $\tilde{z}$ is an **equivalent** statistical model to the true physical additive one z

$$y = x_s + z = x_s \odot \tilde{z} \Rightarrow \tilde{z} = y \odot x_s$$

- **Preprocessing for invariance**
  - We can remain **invariant to the transmitted codeword** by processing other measures of y
    - The **Magnitude**: $|y|$
      - ➢ $P(|y||x) = P(|x_s\,\tilde{z}|\,|x) = P(|x_s\,\tilde{z}||x) = P(|x_s||\,\tilde{z}||x) = P(|\tilde{z}||x) = P(|\tilde{z}|)$
    - The **Syndrome**: $s(y) = H\mathrm{bin}(\mathrm{sign}(y))$
      - ➢ $P(s(y)|x) = P(Hx + H\,z|x) = P(H\,z)$
  - ➢ Extends classical *syndrome decoding*



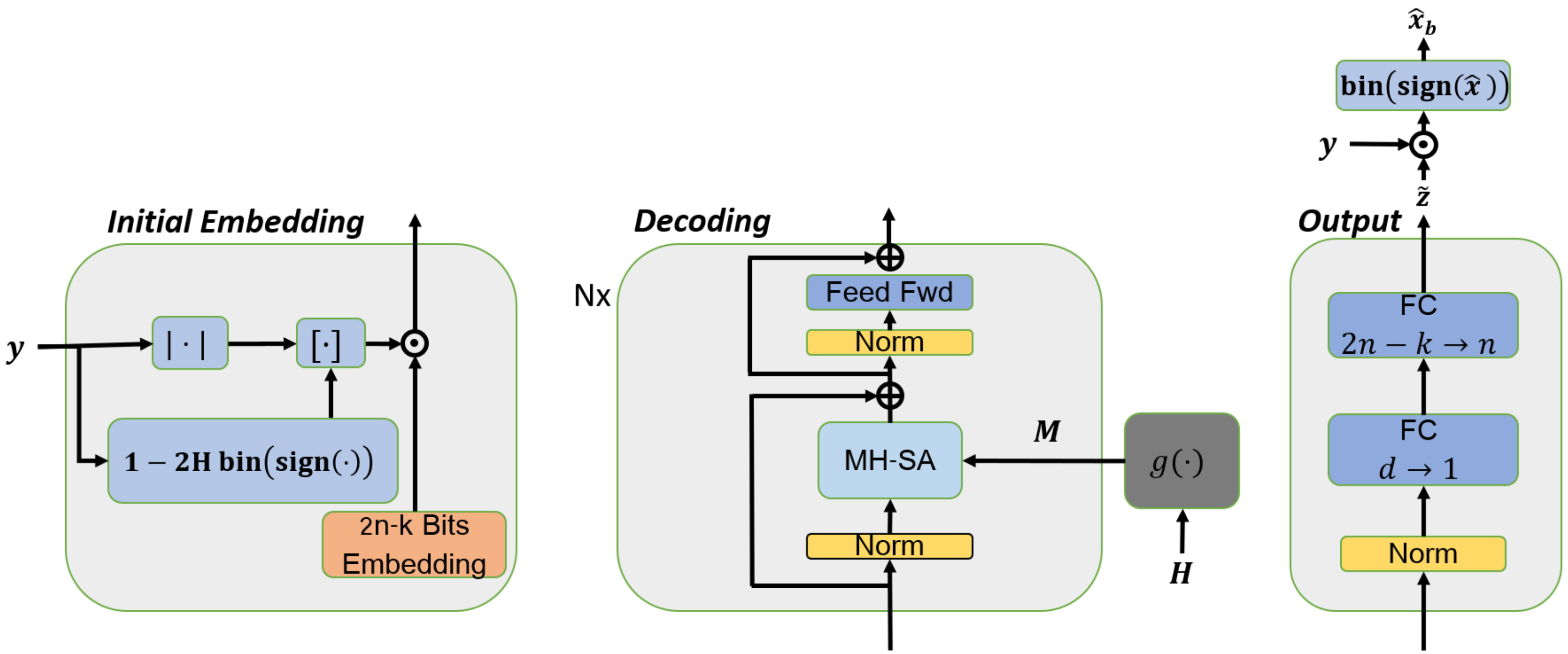
- **Result**
  - Even with loss of information
    - ➢ Does not involve any intrinsic performance penalty (in terms of BER and MSE)
    - ➢ Guarantees the generalization of performance obtained during training.

❖ The prediction $\hat{x} = f_\theta(y)$ is now defined by $f_\theta: \mathbb{R}_+^n \times \{0,1\}^{n-k} \rightarrow \mathbb{R}^n$ such that $f_\theta([|y|, s(y)]) \approx \tilde{z}$

Deep learning for decoding of linear codes-a syndrome-based approach, by A. Bennatan, Y. Choukroun* and P Kisilev, ISIT18*

How can we <u>insert</u> information about the <u>code</u> into the model free solutions?

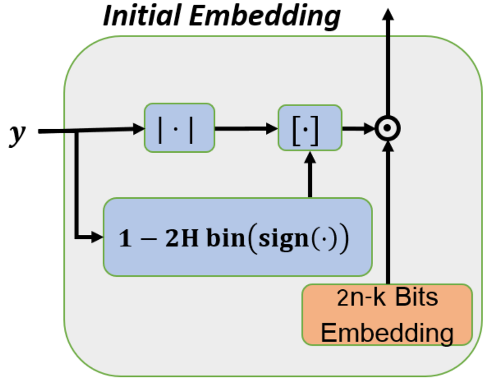
Error Correction Code *Transformer (ECCT)* Neurips22



Error Correction Code Transformer, Y. Choukroun and L. Wolf, Neurips22

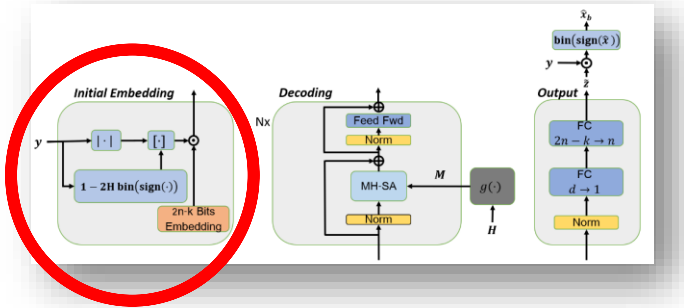
Positional Reliability Encoding

- Regular architectures (FC/RNN) lose every initial/positional information along the layers.
- The one-hot **high dimensional** embedding is ***modulated*** by the **code invariant** magnitude and syndrome values.
  $\Rightarrow$ *Less reliable elements* (i.e., low magnitude) *collapse to the origin* (≤ 0).
- Now the input is of *dimension* $(n+(n-k)) \times d = (\boldsymbol{2n-k}) \times \boldsymbol{d}$



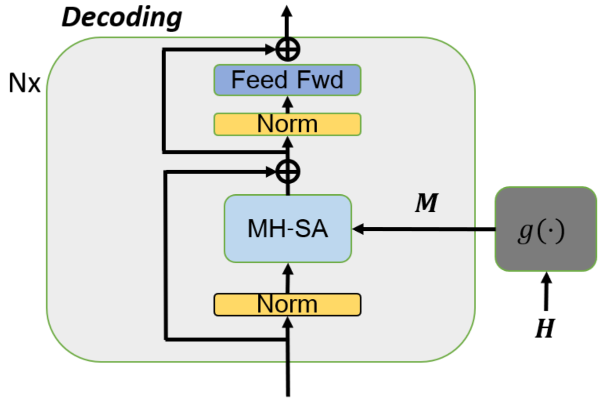
$$\phi_i = \begin{cases} |y_i| W_i, & \text{if } i \leq n \\ \big(1-2(s(y))_{i-n+1}\big) W_i, & \text{otherwise} \end{cases}$$

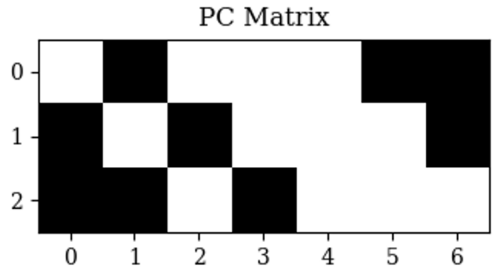
$$\boxed{\Phi = \big(h(y) \cdot 1_d^T\big) \odot W}$$

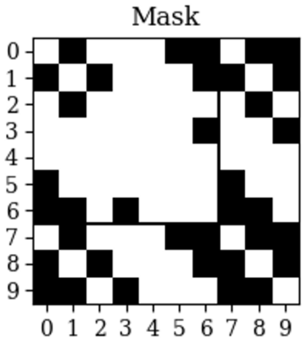


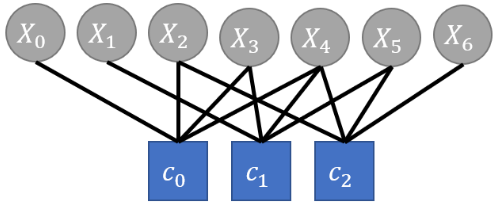
Code Aware Self-Attention

- Decoding requires the cross analysis between elements.
- *We propose to **insert the code** via an **adapted <u>masked</u> self-attention.***
- The proposed mask can be seen as the **<u>sparse</u>** *adjacency matrix of the Tanner graph* extended to a **two rings connectivity** for simultaneous cross analysis.



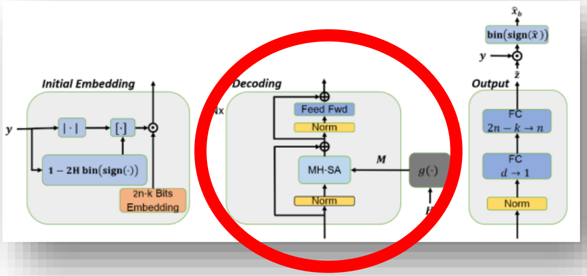






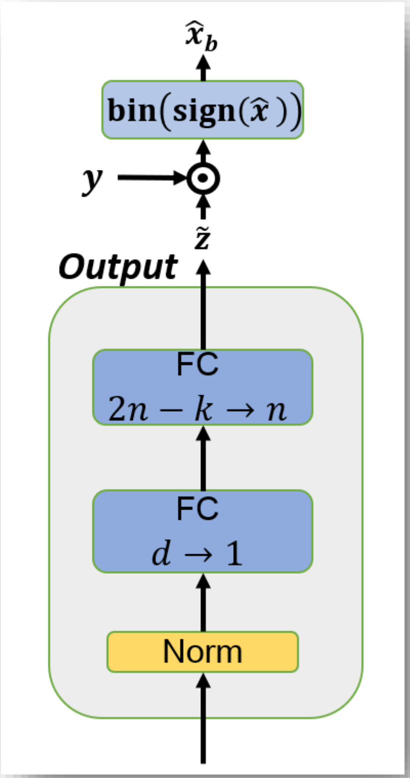
$$A_H(Q,K,V) = \text{Softmax}\left(\frac{QK^T + g(H)}{\sqrt{d}}\right)V.$$

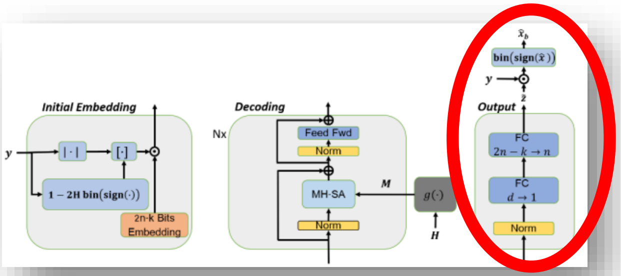
$$g(H) : \{0,1\}^{(n-k)\times k} \rightarrow \{-\infty, 0\}^{2n-k\times 2n-k}$$



Noise Prediction Module

In order to **predict the noise**, the embeddings are shrunk to a **one-dimensional scalar** representation and further reduced to **the code length.**

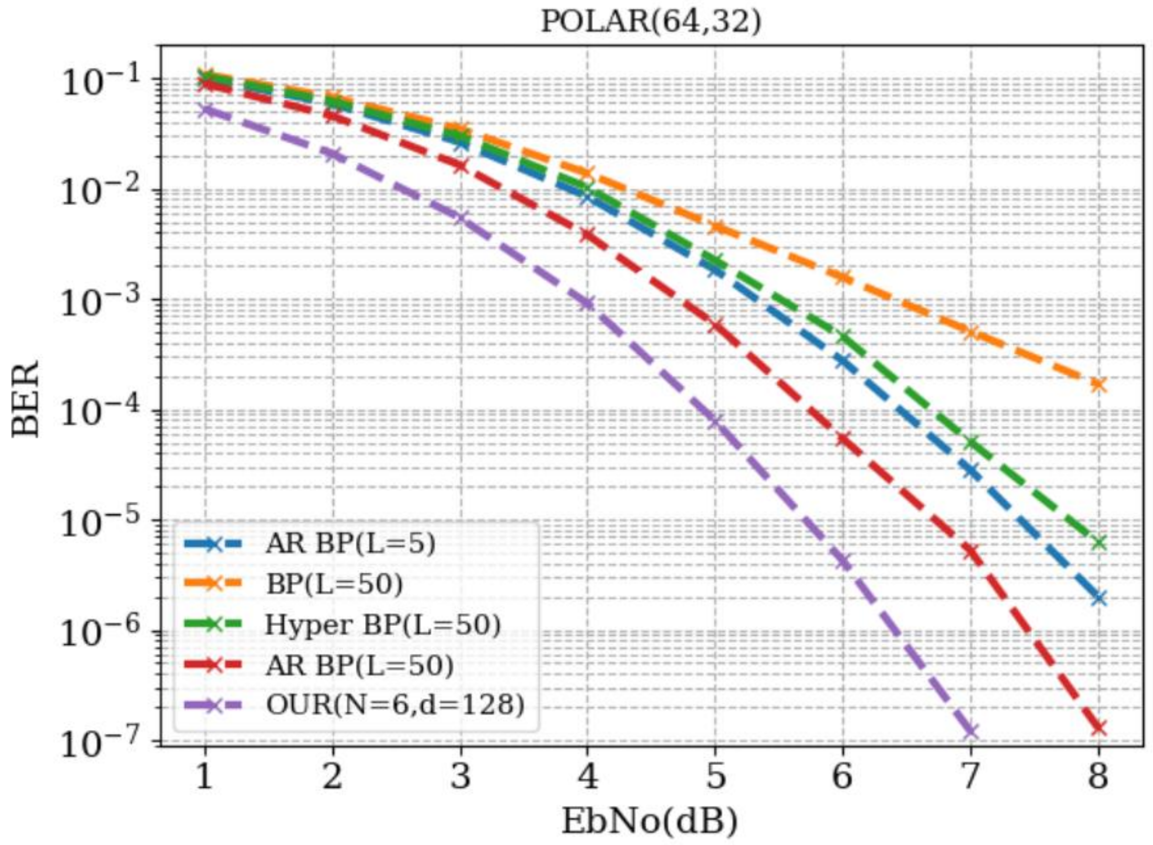




Experiments

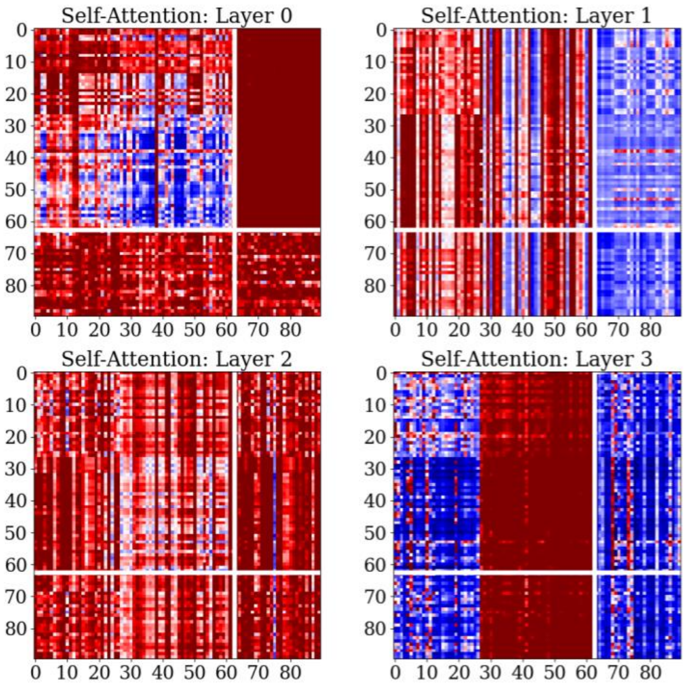
Method	BP[27]			Hyp BP[24]			AR BP[25]			Ours N=2			Ours N=6			Ours N=10		
	4	5	6	4	5	6	4	5	6	4	5	6	4	5	6	4	5	6
Polar(64,32)	3.52 4.26	4.04 5.38	4.48 6.50	4.25 4.59	5.49 6.10	7.02 7.69	4.77 5.57	6.30 7.43	8.19 9.82	4.27 4.57 4.87	5.44 5.86 6.2	6.95 7.50 7.93	*5.71* 6.48 **6.99**	*7.63* 8.60 **9.44**	*9.94* 11.43 **12.32**			
Polar(64,48)	4.15 4.74	4.68 5.94	5.31 7.42	4.91 4.92	6.48 6.44	8.41 8.39	5.25 5.41	6.96 7.19	9.00 9.30	4.92 5.14 5.36	6.46 6.78 7.12	8.41 8.9 *9.39*	*5.82* 6.15 **6.36**	*7.81* 8.20 **8.46**	10.24 10.86 **11.09**			
Polar(128,64)	3.38 4.10	3.80 5.11	4.15 6.15	3.89 4.52	5.18 6.12	6.94 8.25	4.02 4.84	5.48 6.78	7.55 9.30	3.51 3.83 4.04	4.52 5.16 5.52	5.93 7.04 7.62	4.47 *5.12* **5.92**	6.34 *7.36* **8.64**	8.89 *10.48* **12.18**			
Polar(128,86)	3.80 4.49	4.19 5.65	4.62 6.97	4.57 4.95	6.18 6.84	8.27 9.28	4.81 5.39	6.57 7.37	9.04 10.13	4.30 4.49 4.75	5.58 5.90 6.25	7.34 7.75 8.29	5.36 *5.75* **6.31**	*7.45* 8.16 **9.01**	*10.22* 11.29 **12.45**			
Polar(128,96)	3.99 4.61	4.41 5.79	4.78 7.08	4.73 4.94	6.39 6.76	8.57 9.09	4.92 5.27	6.73 7.44	9.30 10.2	4.56 4.69 4.88	5.98 6.20 6.58	7.93 8.30 8.93	*5.39* 5.88 **6.31**	*7.62* 8.33 **9.12**	*10.45* 11.49 **12.47**			
LDPC(49,24)	5.30 6.23	7.28 8.19	9.88 11.72	5.76 6.23	7.90 8.54	11.17 11.95	6.05 6.58	8.13 9.39	11.68 12.39	4.51 4.58 4.71	6.07 6.18 6.38	8.11 8.46 8.73	5.74 5.91 6.13	8.13 8.42 8.71	11.30 11.90 12.10	***6.68***	***9.53***	***13.30***
LDPC(121,60)	4.82 -	7.21 -	10.87 -	5.22 -	8.29 -	13.00 -	5.22 -	8.31 -	13.07 -	3.88 3.89 3.93	5.51 5.55 5.66	8.06 8.16 8.51	4.98 5.02 5.17	7.91 7.94 8.31	12.70 12.72 13.30	***5.73***	***9.35***	***15.01***

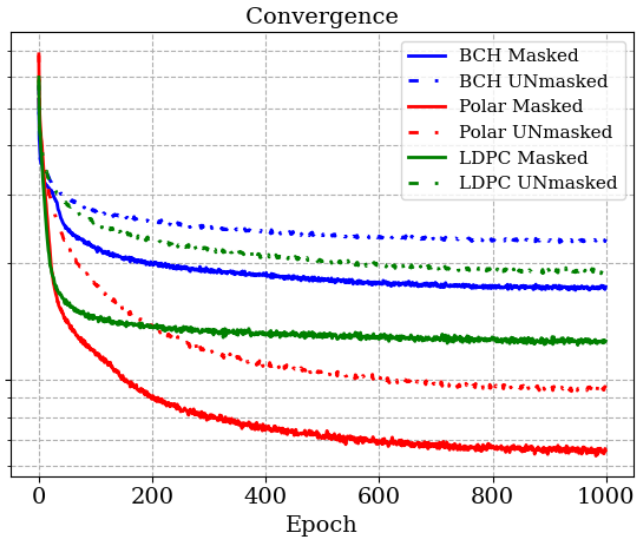
- $E_b/N_0 \sim SNR$
- $\text{Code}(n,k)$

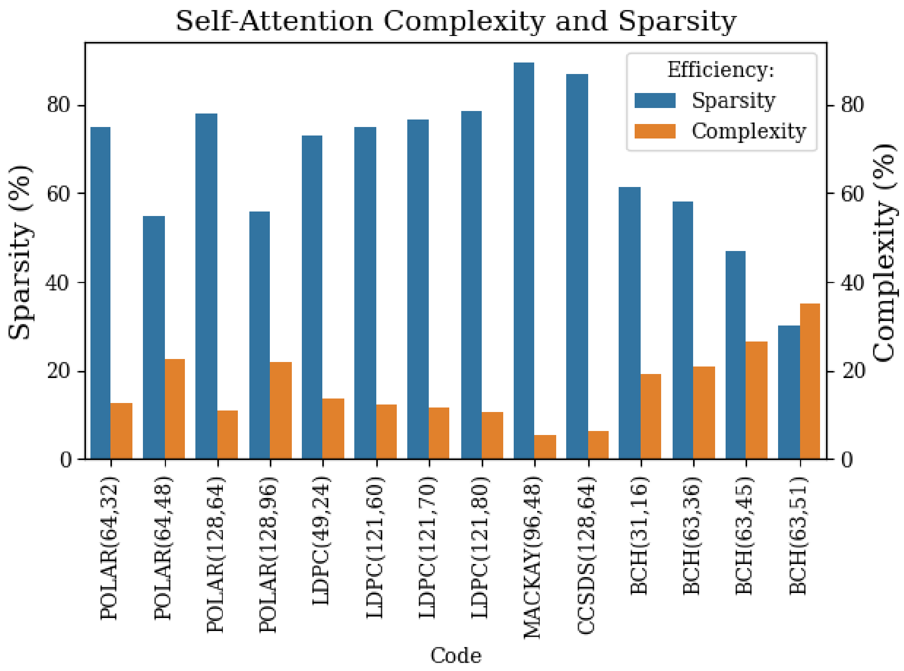


- Our method **surpasses** every existing neural decoder by very large margins (even with shallow ECCT) and at a **fraction of the complexity** of the previous SOTA method.

Analysis







- **Self-attention maps**:
  first layers have higher self-attention values in the **syndrome** part for the parity-check analysis to finally focus on the **information** bits part.

- **Impact of masking**:
  Masking improves the performance by orders of magnitude demonstrating the importance of **integrating code** information.

- **Sparsity and complexity ratio:**
  Sparsity and self-attention complexity can reach up to 80% and as low as 5, respectively.

“But it’s not very efficient…”

Accelerating Error Correction Code Transformers

- Accelerate and improve decoding via
  1. Adaptive absolute percentile **ternary** (0,1,-1) **quantization** (90% compression and >224x less energy)

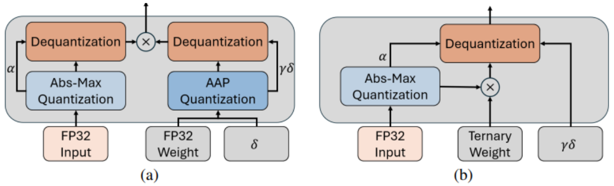
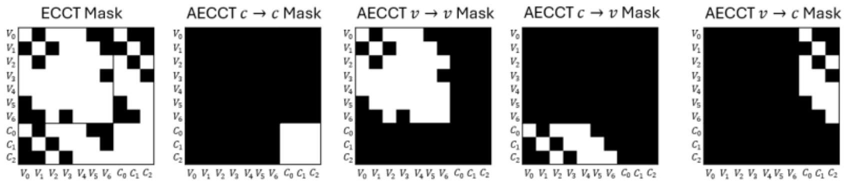


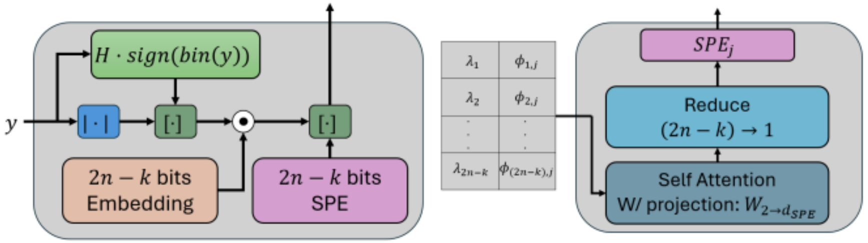
Figure 1: AAP Linear Layer: (a) QAT: Training with quantization noise; (b) Inference: Matrix multiplication using only integer additions with fixed ternary weights and fixed weight scale.

$$\mathrm{AAPLinear}(x, W, b) = \mathrm{Quant}(x) \cdot \mathrm{Ternary}(W) \cdot \frac{\gamma\delta\alpha}{Q_b} + b$$

  2. **Head partitioning** self attention



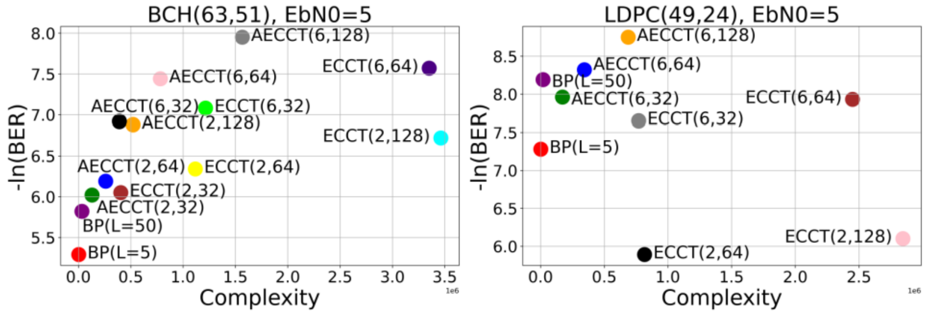
  3. **Positional encoding** of the tanner **graph**



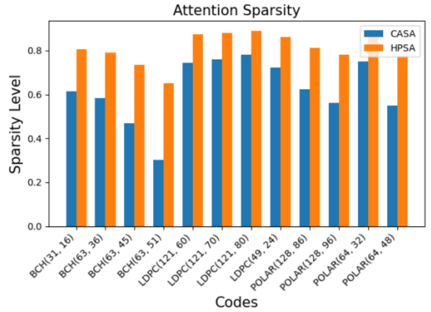
Accelerating Error Correction Code Transformers, M.Levy, Y. Choukroun and L. Wolf

Accelerating Error Correction Code Transformers

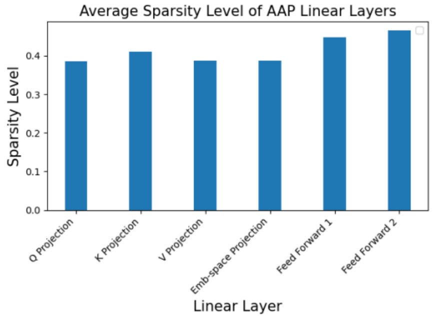
- Matches/improve over ECCT
- Close to BP's complexity



- Increase sparsity masking



- Linear Layers are extremely sparse



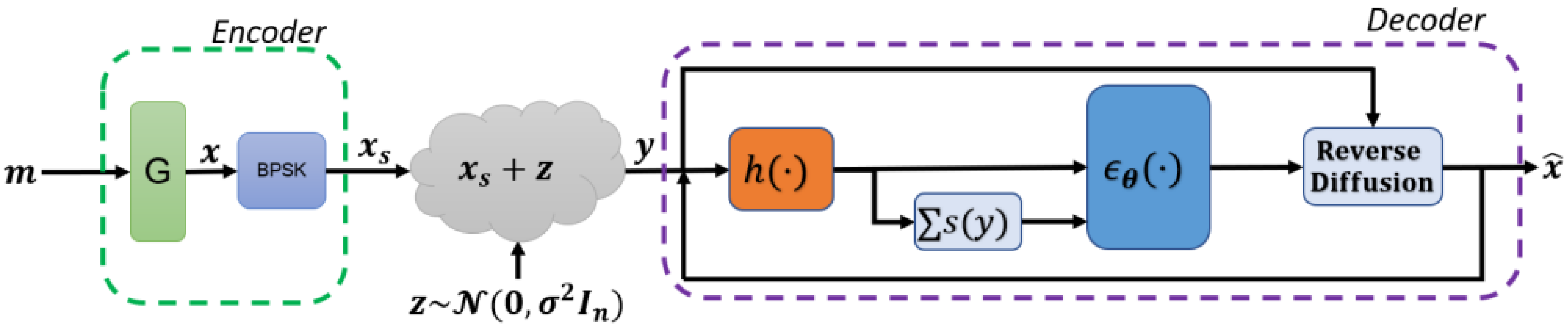
Method	BP			ECCT $N=6$			AECCT $N=6$			ECCT $N=10$			AECCT $N=10$		
	4	5	6	4	5	6	4	5	6	4	5	6	4	5	6
Polar(64,48)	3.52 4.26	4.04 5.38	4.48 6.50	6.36	8.46	11.09	**6.43**	**8.54**	**11.12**	6.43	8.40	11.00	**6.54**	**8.51**	**11.12**
Polar(128,86)	3.80 4.49	4.19 5.65	4.62 6.97	**6.31**	**9.01**	**12.45**	6.04	8.56	11.81	7.26	10.60	**14.80**	**7.28**	10.60	14.59
Polar(128,96)	3.99 4.61	4.41 5.79	4.78 7.08	**6.31**	**9.12**	**12.47**	6.11	8.81	12.15	**6.85**	**9.78**	12.90	6.79	9.68	**12.93**
LDPC(49,24)	5.30 6.23	7.28 8.19	9.88 11.72	5.79	8.13	11.40	**6.10**	**8.65**	**12.34**	6.35	9.01	12.43	**6.67**	**9.35**	**13.56**
LDPC(121,60)	4.82 -	7.21 -	10.87 -	5.01	7.99	12.78	**5.17**	**8.32**	**13.40**	5.51	8.89	14.51	**5.71**	**9.31**	**14.90**
LDPC(121,70)	5.88 -	8.76 -	13.04 -	6.19	9.89	15.58	**6.38**	**10.1**	**16.01**	6.86	11.02	16.85	**7.05**	**11.40**	**17.30**
LDPC(121,80)	6.66 -	9.82 -	13.98 -	7.07	10.96	16.25	**7.27**	**11.50**	**16.90**	7.76	12.30	17.82	**7.98**	**12.60**	**18.10**
BCH(31,16)	4.63 -	5.88 -	7.60 -	6.39	8.29	10.66	**7.01**	**9.33**	**12.27**	6.41	8.30	10.77	**7.21**	**9.47**	**12.45**
BCH(63,36)	3.72 4.03	4.65 5.42	5.66 7.26	4.68	6.65	9.10	**5.19**	**6.95**	**9.33**	**5.09**	**6.96**	**9.43**	4.90	6.64	9.19
BCH(63,45)	4.08 4.36	4.96 5.55	6.07 7.26	5.60	7.79	10.93	**5.90**	**8.24**	**11.46**	5.72	7.99	11.21	**5.83**	**8.15**	**11.52**
BCH(63,51)	4.34 4.50	5.29 5.82	6.35 7.42	5.66	7.89	11.01	**5.72**	**8.01**	**11.24**	5.38	7.40	10.50	**5.68**	**7.88**	**11.04**

Accelerating Error Correction Code Transformers, M.Levy, Y. Choukroun and L. Wolf

Both highly and slightly corrupted codewords go through the same computationally demanding *neural* decoding procedure.

How can we develop an adaptive and iterative decoding scheme?

Denoising Diffusion Error Correction Codes ICLR23

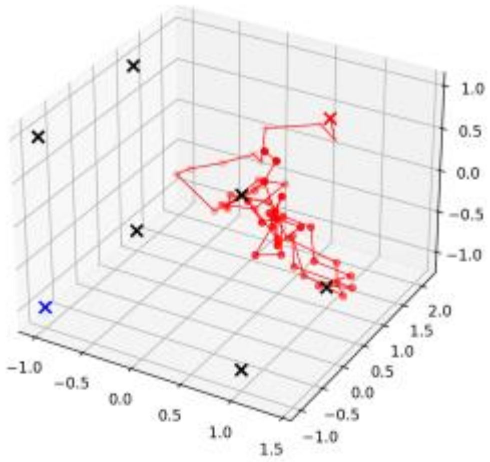


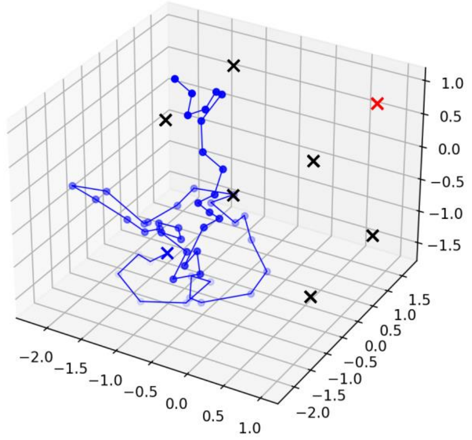
Denoising Diffusion Error Correction Codes, Y. Choukroun and L. Wolf, ICLR23

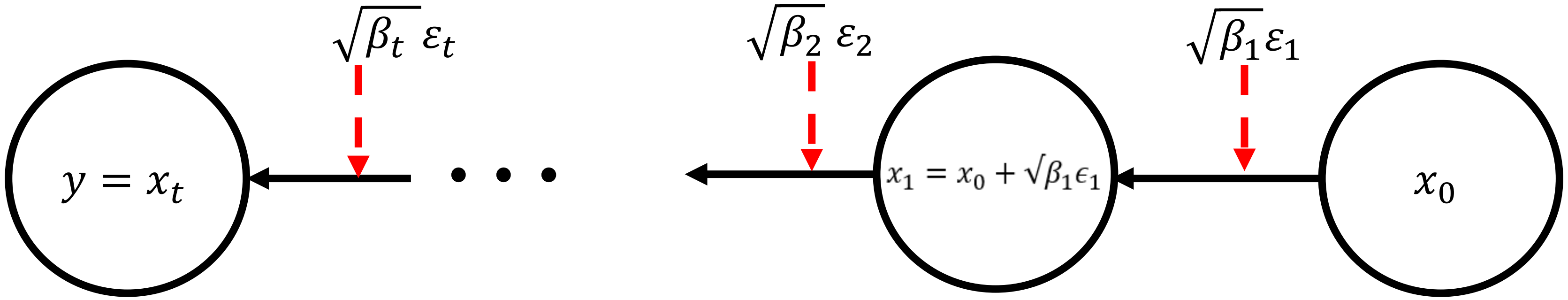
Channel Corruption as Diffusion

- We formulate the channel corruption process as an **iterative** *forward* **diffusion process**

$$
\begin{aligned}
y := x_t &= x_0 + \sigma\varepsilon,\ \varepsilon \sim \mathcal{N}(0, I) \\
&= x_0 + \sqrt{\beta_1}\varepsilon_1 + \cdots \sqrt{\beta_t}\varepsilon_t = x_0 + \sum_{i=1}^{t} \sqrt{\beta_i}\varepsilon_i, \\
&= x_0 + \sqrt{\bar{\beta}_t}\varepsilon \ \sim \mathcal{N}(x_t; x_0, \bar{\beta}_t I) \qquad \bar{\beta}_t = \textstyle\sum_{i=1}^{t} \beta_i
\end{aligned}
$$

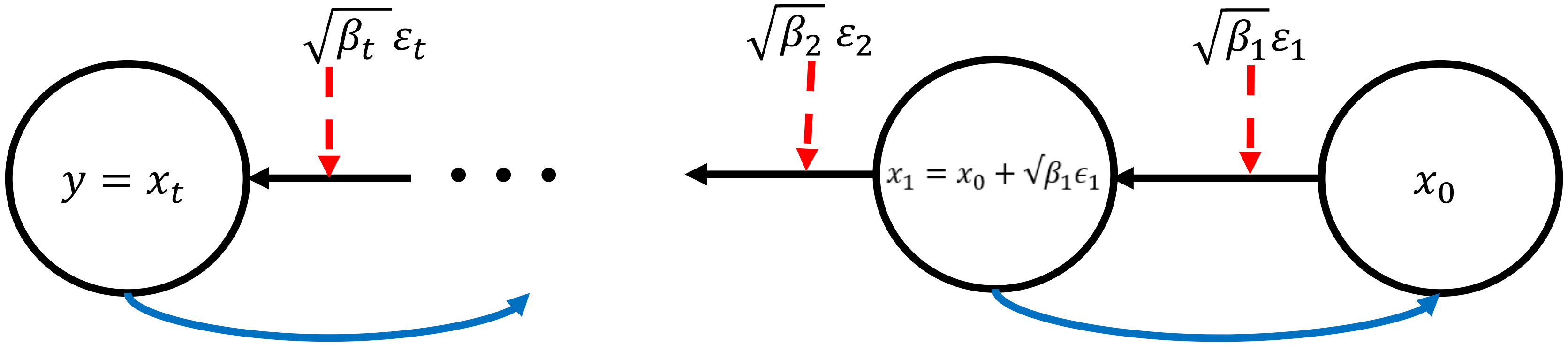


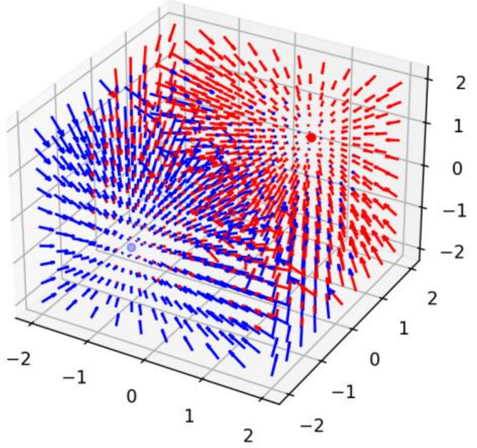




Decoding as Denoising Diffusion

- As in traditional *reverse* diffusion process, we are interested in learning to **iteratively denoise** the corrupted codeword $\boldsymbol{y}$.





- The original **scaled** setting is adapted to the ECC setting

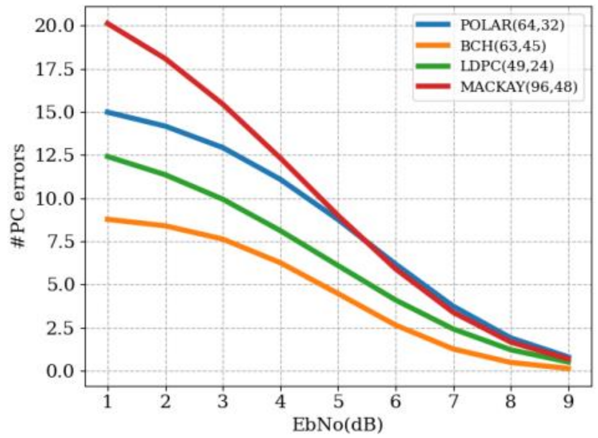
$$
\begin{aligned}
q(x_{t-1}|x_t, x_0) &= q(x_t|x_{t-1}, x_0)\frac{q(x_{t-1}|x_0)}{q(x_t|x_0)} \\
&\propto \exp\Big(-\frac{1}{2}\big(\frac{(x_t - x_{t-1})^2}{\beta_t} + \frac{(x_{t-1} - x_0)^2}{1-\bar{\beta}_t} - \frac{(x_t - x_0)^2}{1-\bar{\beta}_t}\big)\Big) \\
&= \exp\Big(-\frac{1}{2}\big((\frac{1}{\beta_t} + \frac{1}{\bar{\beta}_t})x_{t-1}^2 - (\frac{2}{\beta_t}x_t - \frac{2}{\bar{\beta}_t}x_0)x_{t-1} + C(x_t, x_0)\big)\Big)
\end{aligned}
$$

$$
\boxed{\tilde{\mu}_t(x_t, x_0) = \frac{\bar{\beta}_t}{\bar{\beta}_t + \beta_t}x_t + \frac{\beta_t}{\bar{\beta}_t + \beta_t}x_0 = x_t - \frac{\sqrt{\bar{\beta}_t}\beta_t}{\bar{\beta}_t + \beta_t}\varepsilon\,, \text{ and } \tilde{\beta}_t = \frac{\bar{\beta}_t\beta_t}{\bar{\beta}_t + \beta_t}.}
$$

Parity Check Conditioning and Reverse Diffusion

- The reverse denoising process of *traditional* DDPM is conditioned by the *time step*.

➢ Since we are not interested in generative models, we suggest **conditioning** the diffusion decoder according to the number of **parity check errors** which conveys information about the level of corruption.



$$\mathcal{L}(\theta) = -\mathbb{E}_{t,x_0,\varepsilon} \log \left(\epsilon_\theta(x_0 + \bar{\beta}_t^{1/2}\varepsilon, \boldsymbol{e_t}), \tilde{\varepsilon}_b \right)$$

- The diffusion model is trained to predict the binary (sign of the) **multiplicative** noise
- We obtain the traditional additive noise by *subtracting the predicted codeword*

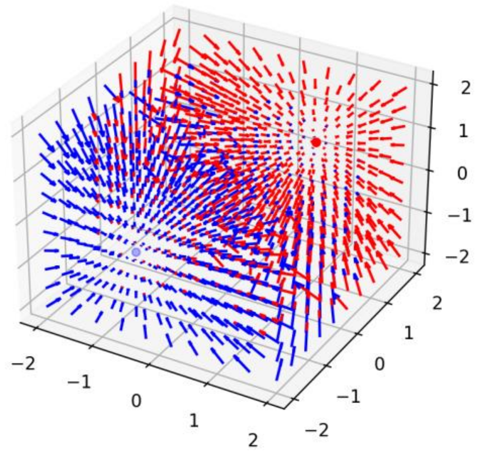
$$\hat{\varepsilon} = y - \operatorname{sign}(\hat{x}) = y - \operatorname{sign}(\hat{\tilde{\varepsilon}} y).$$

➢ The final *reverse* diffusion process is given by

$$\boxed{x_{t-1} = x_t - \frac{\sqrt{\bar{\beta}_t \beta_t}}{\bar{\beta}_t + \beta_t} \left(x_t - \operatorname{sign}(x_t \epsilon_\theta(x_t, e_t)) \right)}$$

Optimal Diffusion Step Size

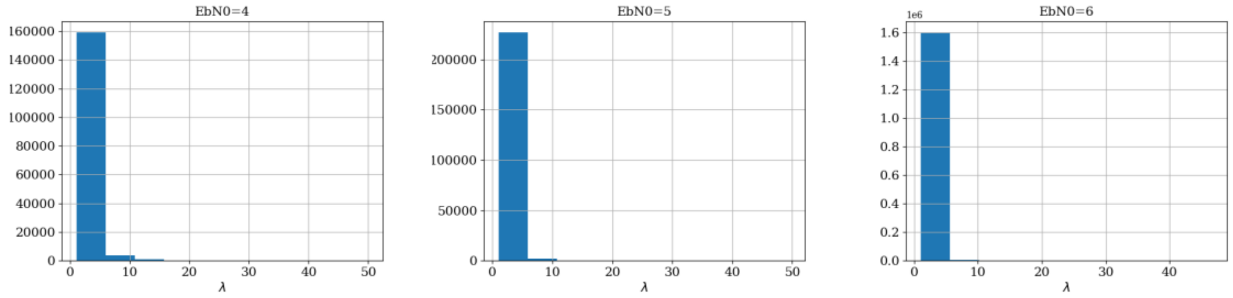
- One major limitation of the generative neural diffusion process is **the large number of steps** required - generally a thousand - to generate high-quality samples.
- The number of parity check errors conveys information about the level of corruption
  - ➢ Given the reverse diffusion direction/vector, we define the optimal step size as the one which **minimizes the number of parity check errors**.



- We propose to find the optimal step size λ by solving the minimum number of parity errors.

$$\lambda^* = \arg\min_{\lambda \in \mathbb{R}^+} \|s\left(x_t - \lambda \frac{\sqrt{\bar{\beta}_t \beta t}}{\bar{\beta}_t + \beta_t}\hat{\varepsilon}\right)\|_1,$$

- Since the objective is highly non-differentiable and non-convex (sign, modulo 2), we suggest using a grid **line-search** approach



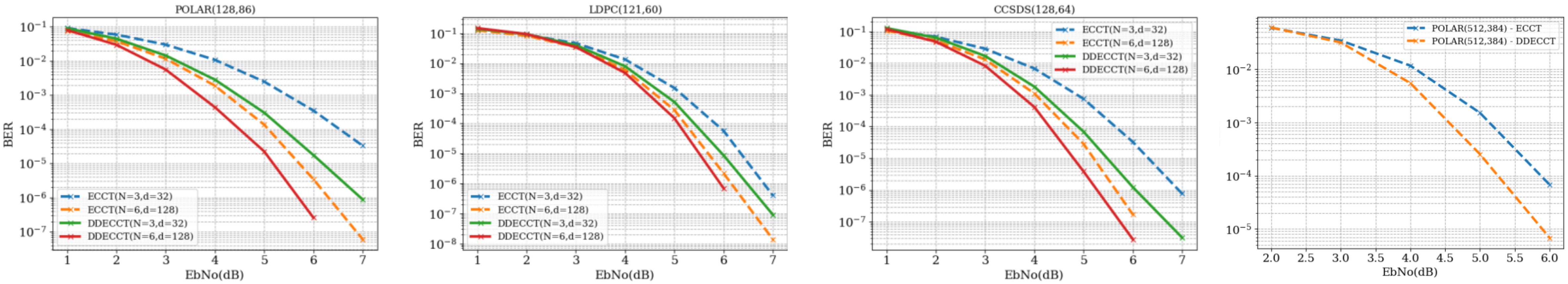
Experiments

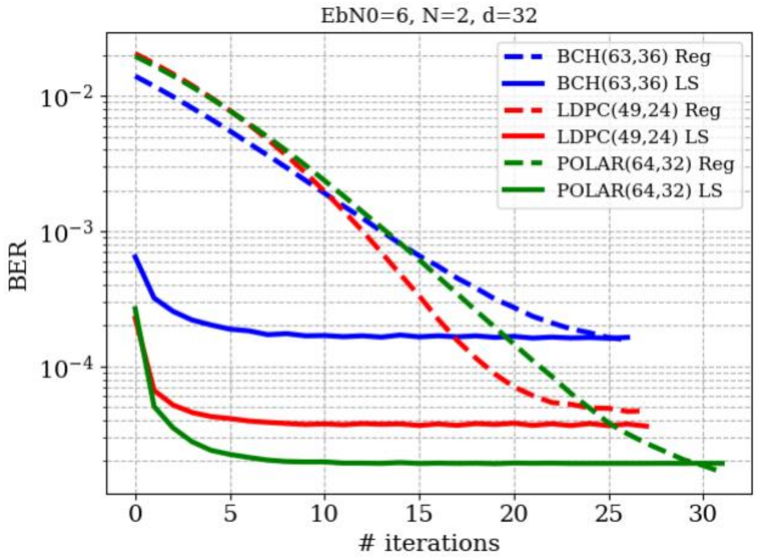
- The base noise estimator is an adapted (“time” conditioned) *ECCT*
- Our approach **outperforms** the current SOTA results (obtained by ECCT) by extremely large margins on several families of codes of different lengths and rates, at a **fraction** of the capacity.
- Especially for shallow models, the difference can be **orders of magnitude**.

Method	BP			ARBP			ECCT N=2			ECCT N=6			Ours N=2			Ours N=6		
E_b/N_0	4	5	6	4	5	6	4	5	6	4	5	6	4	5	6	4	5	6
Polar(64,32)	3.52	4.04	4.48	4.77	6.30	8.19	4.27	5.44	6.95	5.71	7.63	9.94	5.99	8.16	10.90	6.76	9.14	12.31
	4.26	5.38	6.50	5.57	7.43	9.82	4.57	5.86	7.50	6.48	8.60	11.43	6.23	8.52	11.23	6.90	9.43	12.85
							4.87	6.2	7.93	6.99	9.44	12.32	6.59	8.95	11.91	6.93	9.51	12.79
Polar(64,48)	4.15	4.68	5.31	5.25	6.96	9.00	4.92	6.46	8.41	5.82	7.81	10.24	5.55	7.67	10.08	5.98	8.02	10.94
	4.74	5.94	7.42	5.41	7.19	9.30	5.14	6.78	8.9	6.15	8.20	10.86	5.74	7.85	10.40	5.98	8.26	11.13
							5.36	7.12	9.39	6.36	8.46	11.09	5.77	7.94	10.64	5.96	8.04	10.98
Polar(128,64)	3.38	3.80	4.15	4.02	5.48	7.55	3.51	4.52	5.93	4.47	6.34	8.89	5.37	7.75	10.51	6.34	9.26	12.77
	4.10	5.11	6.15	4.84	6.78	9.30	3.83	5.16	7.04	5.12	7.36	10.48	5.97	8.52	11.72	7.24	10.70	14.56
							4.04	5.52	7.62	5.92	8.64	12.18	6.50	9.23	12.37	9.11	12.90	16.30
Polar(128,86)	3.80	4.19	4.62	4.81	6.57	9.04	4.30	5.58	7.34	5.36	7.45	10.22	5.61	7.76	10.42	6.52	9.21	12.64
	4.49	5.65	6.97	5.39	7.37	10.13	4.49	5.90	7.75	5.75	8.16	11.29	5.99	8.19	11.00	7.09	10.20	13.84
							4.75	6.25	8.29	6.31	9.01	12.45	6.27	8.64	11.61	7.6	10.81	15.17
Polar(128,96)	3.99	4.41	4.78	4.92	6.73	9.30	4.56	5.98	7.93	5.39	7.62	10.45	5.60	7.83	10.56	6.46	9.41	12.52
	4.61	5.79	7.08	5.27	7.44	10.2	4.69	6.20	8.30	5.88	8.33	11.49	5.95	8.42	11.38	6.83	9.99	13.36
							4.88	6.58	8.93	6.31	9.12	12.47	6.26	8.94	12.01	7.16	10.3	13.19
LDPC(49,24)	5.30	7.28	9.88	6.05	8.13	11.68	4.51	6.07	8.11	5.74	8.13	11.30	5.27	7.38	10.23	5.87	8.22	11.56
	6.23	8.19	11.72	6.58	9.39	12.39	4.58	6.18	8.46	5.91	8.42	11.90	5.31	7.35	10.40	5.84	8.29	11.85
							4.71	6.38	8.73	6.13	8.71	12.10	5.36	7.39	10.41	5.88	8.27	11.42
LDPC(121,60)	4.82	7.21	10.87	5.22	8.31	13.07	3.88	5.51	8.06	4.98	7.91	12.70	4.48	6.95	10.65	5.25	8.43	13.80
	-	-	-	-	-	-	3.89	5.55	8.16	5.02	7.94	12.72	4.56	7.02	10.64	5.32	8.69	13.82
							3.93	5.66	8.51	5.17	8.31	13.30	4.46	6.92	10.76	5.38	8.73	14.17
LDPC(121,70)	5.88	8.76	13.04	6.45	10.01	14.77	4.63	6.68	9.73	6.11	9.62	15.10	5.41	8.22	12.22	6.49	10.39	15.43
	-	-	-	-	-	-	4.64	6.71	9.77	6.28	10.12	15.57	5.52	8.47	12.63	6.64	10.65	16.21
							4.67	6.79	9.98	6.40	10.21	16.11	5.55	8.51	12.81	6.79	11.13	16.93
LDPC(121,80)	6.66	9.82	13.98	7.22	11.03	15.90	5.27	7.59	10.08	6.92	10.74	15.10	6.12	9.38	13.25	7.68	12.19	17.83
	-	-	-	-	-	-	5.29	7.63	10.90	7.17	11.21	16.31	6.26	9.41	13.41	7.39	11.46	17.65
							5.30	7.65	11.03	7.41	11.51	16.44	6.26	9.41	13.46	7.59	12.17	16.89
MacKay(96,48)	6.84	9.40	12.57	7.43	10.65	14.65	4.95	6.67	8.94	6.88	9.86	13.40	6.18	8.63	11.53	7.86	11.61	15.51
	-	-	-	-	-	-	5.04	6.80	9.23	7.10	10.12	14.21	6.28	8.8	11.78	7.93	11.65	15.51
							5.17	7.07	9.64	7.38	10.72	14.83	6.31	8.83	12.03	8.12	11.88	15.93
CCSDS(128,64)	6.55	9.65	13.78	7.25	10.99	16.36	4.35	6.01	8.30	6.34	9.80	14.40	5.79	8.48	12.24	7.28	11.66	17.02
	-	-	-	-	-	-	4.41	6.09	8.49	6.65	10.40	15.46	5.81	8.79	12.29	7.55	12.01	17.62
							4.59	6.42	9.02	6.88	10.90	15.90	5.77	8.7	12.49	7.81	12.48	17.66
BCH(63,36)	3.72	4.65	5.66	4.33	5.94	8.21	3.79	4.87	6.35	4.42	5.91	8.01	4.71	6.45	8.72	5.01	6.84	9.30
	4.03	5.42	7.26	4.57	6.39	8.92	4.05	5.28	7.01	4.62	6.24	8.44	4.84	6.65	9.01	5.07	7.02	9.85
							4.21	5.50	7.25	4.86	6.65	9.10	5.19	7.27	9.82	5.11	7.09	9.82
BCH(63,45)	4.08	4.96	6.07	4.80	6.43	8.69	4.47	5.88	7.81	5.16	7.02	9.75	5.12	7.16	9.95	5.49	7.71	10.86
	4.36	5.55	7.26	4.97	6.90	9.41	4.66	6.16	8.17	5.41	7.49	10.25	5.33	7.49	10.18	5.60	8.02	11.05
							4.79	6.39	8.49	5.60	7.79	10.93	5.41	7.61	10.46	5.61	7.94	11.36
BCH(63,51)	4.34	5.29	6.35	4.95	6.69	9.18	4.60	6.05	8.05	5.20	7.08	9.65	5.09	7.08	9.87	5.35	7.49	10.38
	4.5	5.82	7.42	5.17	7.16	9.53	4.78	6.34	8.49	5.46	7.57	10.51	5.19	7.23	10.20	5.39	7.48	10.53
							5.01	6.72	9.03	5.66	7.89	11.01	5.21	7.29	10.13	5.26	7.40	10.49

- $E_b/N_0 \sim SNR$
- Code(n,k)

Experiments and Analysis





LS: Our line search approach allows convergence within very few iterations.

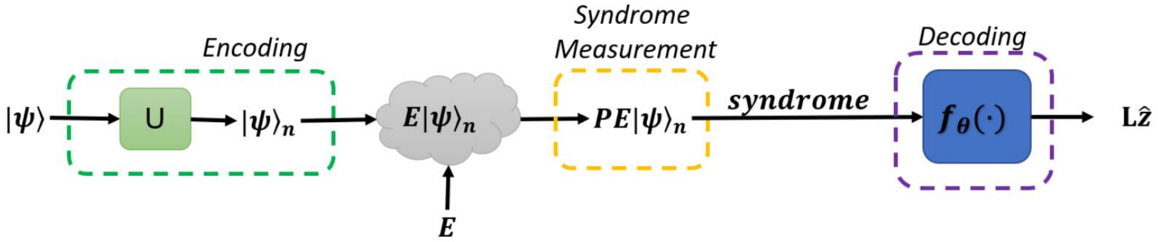
How to adapt neural decoders to <u>Quantum</u> error correction?

Deep Quantum Error Correction AAAI24

➢ **Goal**: *allow the protection of quantum information from quantum noise (e.g., quantum gates, decoherence).*

- A quantum bit (*qubit*) is defined as the superposition of two states

$$|\boldsymbol{\psi}\rangle = \boldsymbol{\alpha}|\mathbf{0}\rangle + \boldsymbol{\beta}|\mathbf{1}\rangle, \qquad \boldsymbol{\alpha},\boldsymbol{\beta} \in \mathbb{C}, \qquad |\boldsymbol{\alpha}|^2 + |\boldsymbol{\beta}|^2 = \mathbf{1}$$



- There are three major differences/challenges with classical error correction
    1. **Syndrome Decoding:** There is no arbitrary access to the current state (due to quantum wave measurement collapse) such that only partial information defined by the syndrome is available.
    It requires an adaptation of the existing neural decoders to <u>syndrome decoding</u>.
    2. **Logical Decoding**: We are interested in the logical qubits only, meaning we wish to predict the codeword **up to** the logical operators mapping (i.e., $\mathbb{L}\hat{\boldsymbol{z}}$ instead of $\hat{\boldsymbol{z}}$).
    However, this mapping is defined over the highly <u>non-differentiable</u> GF(2) (i.e., XOR).
    3. **Noisy Syndrome measurement**: The syndrome measurement itself being noisy, the decoding must be performed based on multiple noisy measurements of the syndrome is obtained upon **multiple** noisy syndrome observations.
    <u>Efficient</u> decoding methods must be developed.

- *These QECC challenges are at the core of our contributions.*

Overcoming Measurement Collapse by Prediction

- In the QECC setting, only the code **syndrome** is available, since classical measurements are not allowed due to the *wave function collapse phenomenon.* Thus, it's not possible to arbitrarily access/measure the quantum state.

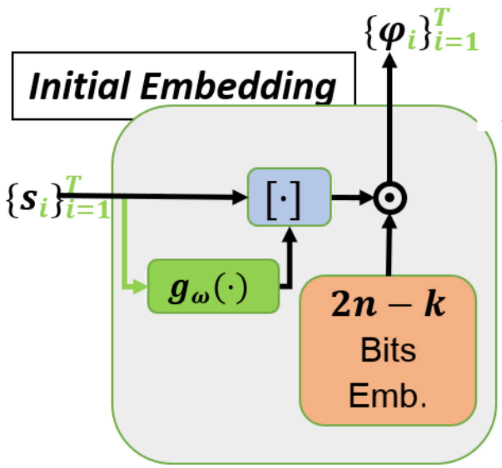
- We thus propose to *extend the ECCT*, by replacing the magnitude of the channel output (i.e., $|\boldsymbol{y}|$) with an **initial estimate of the noise** to be further refined by the code-aware network.
  - Reminiscent of MCMC methods.

- Given $g_\omega : \{0,1\}^{n_s} \rightarrow \mathbb{R}^n$ the **initial** parameterized **noise estimator** from a given syndrome

$$\hat{z} = f_\theta\big(h_q(s)\big) = f_\theta\big([g_\omega(s), s]\big)$$

- The initial estimator is trained to predict the noise from the syndrome (i.e., *syndrome decoding*)

$$\mathcal{L}_g = \mathrm{BCE}\big(g_\omega(s), \varepsilon\big)$$



Logical Decoding

- The logical error rate (LER) metric provides valuable information on the **practical** decoding performance.
- We wish to minimize the following LER objective

$$\mathcal{L}_{\text{LER}} = \text{BCE}\big(\mathbb{L} f_\theta(s), \mathbb{L}\varepsilon\big) \qquad \mathbb{L} \in \{0,1\}^{k\times n}$$

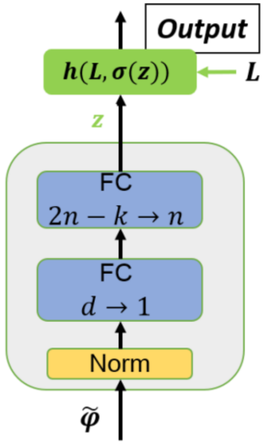
where the multiplications are performed over *the highly **nondifferentiable** GF(2).*

- We propose to optimize the objective using a differentiable equivalence mapping of the XOR operator via **bipolar mapping**
  - $\phi(u) = 1 - 2u$ induces $\phi(u \oplus v) = \phi(u)\phi(v)$ such that we have

$$\boxed{\big(\Lambda(\mathbb{L}, x)\big)_i \coloneqq \mathbb{L}_i \oplus x = \phi^{-1}\Big(\Pi_j \phi\big((\mathbb{L})_{ij} \cdot x_j\big)\Big)}$$

- The LER training objective becomes

$$\mathcal{L}_{\text{LER}} = \text{BCE}\Big(\Lambda\big(\mathbb{L}, bin(f_\theta(s))\big), \mathbb{L}\varepsilon\Big)$$



Noisy Syndrome Measurement

- In the presence of measurement errors, each syndrome measurement is repeated several times, and efficient noisy measurement decoding is required.

- At each time sample we have the syndrome defined as

$$s_t = \big(H(x \oplus \varepsilon_1 \oplus \cdots \oplus \varepsilon_t)\big) \oplus \tilde{\varepsilon}_t$$

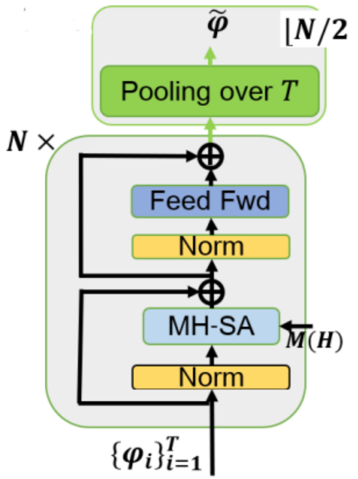
- We first analyze each measurement separately and then perform global decoding **at the embedding level** by applying a symmetric pooling function, e.g. an average, in the **middle** of the neural network.

- Given a neural decoder with N layers and the activations $\varphi \in \mathbb{R}^{T\times n\times d}$, the pooled embedding is given by $\tilde{\varphi} = \frac{1}{T}\sum_t \varphi_t$ at layer $l = \lfloor N/2 \rfloor$

- ***<u>Final Objective</u>***: Given $\mathcal{L}_{\text{BER}} = \text{BCE}\big(f_\theta(s), \varepsilon\big)$ the overall objective is

$$\mathcal{L} = \lambda_{\text{BER}}\mathcal{L}_{\text{BER}} + \lambda_{\text{LER}}\mathcal{L}_{\text{LER}} + \lambda_g\mathcal{L}_g$$

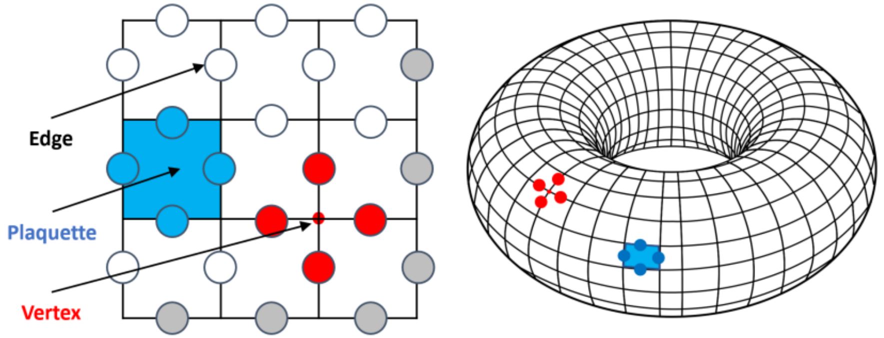
- The **BER regularization is important** since the *GF2* optimization induces severe *saddle point optimization.*



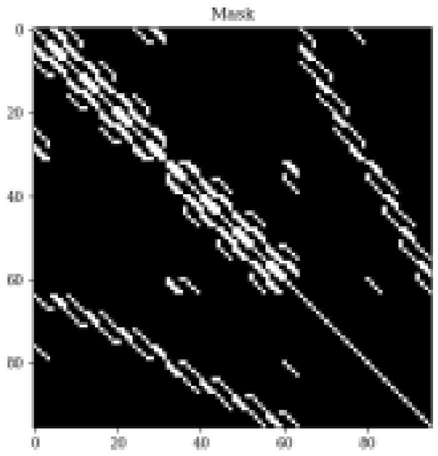
Experiments

- Experiments are performed on the popular **Toric and Surface** codes.
  - The performances are reported for lattice of up to length 10 (i.e., hundreds of qubits)

- Several noise settings are experimented
  - **Independent** noise
  - **Depolarization** noise
  - **Circuit** noise
  - With **faulty syndrome measurements**

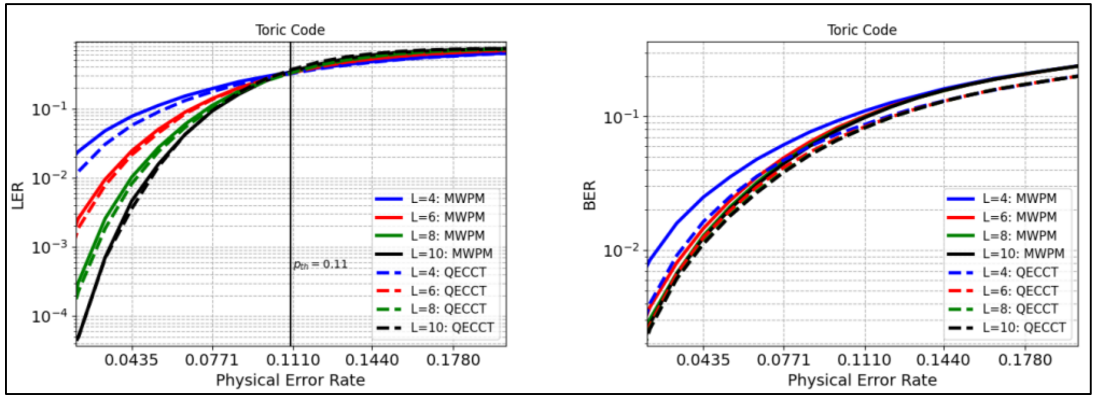


- The model is based on the *refined* ECCT with a *shallow* architecture of 6 layers with highly **sparse** self-attention.

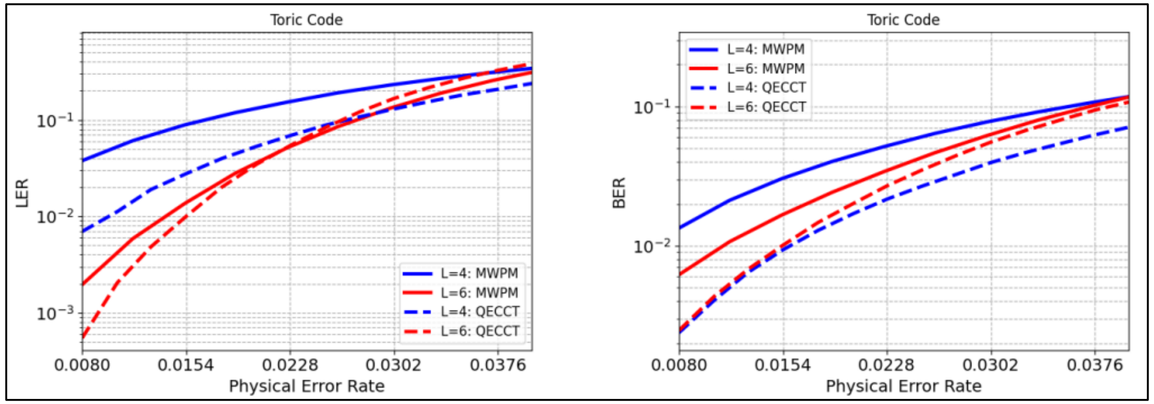


- The baseline is the very popular Minimum Weight Perfect Matching (**MWPM**) algorithm $(O((n^3+n^2)\,log(n))$ to $O(n^2))$

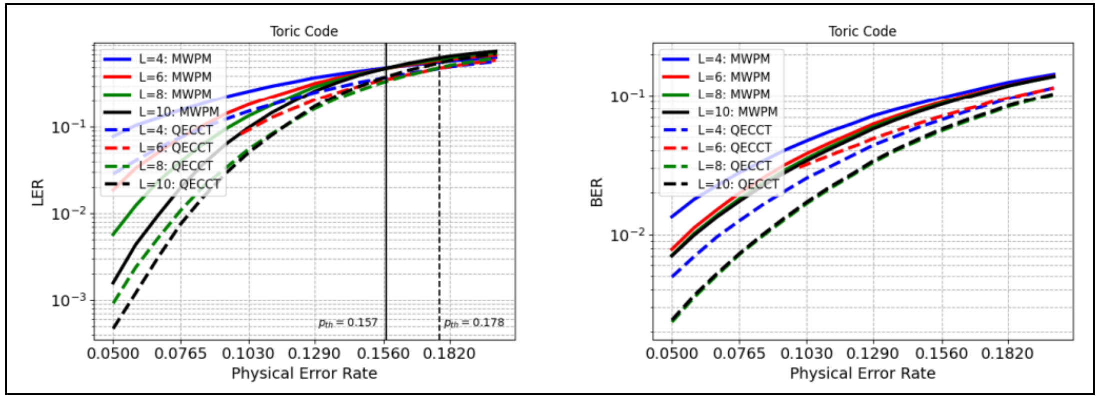
Toric Code Results



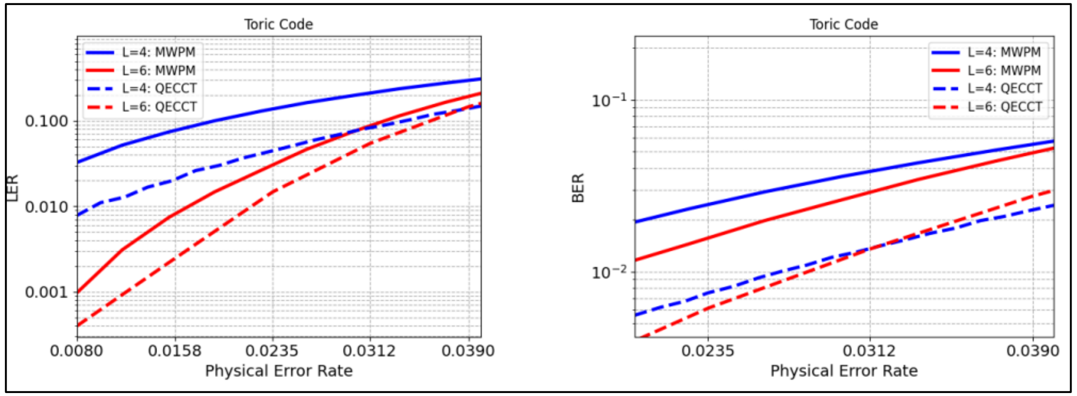
Independent noise



Independent noise
with faulty syndrome measurements

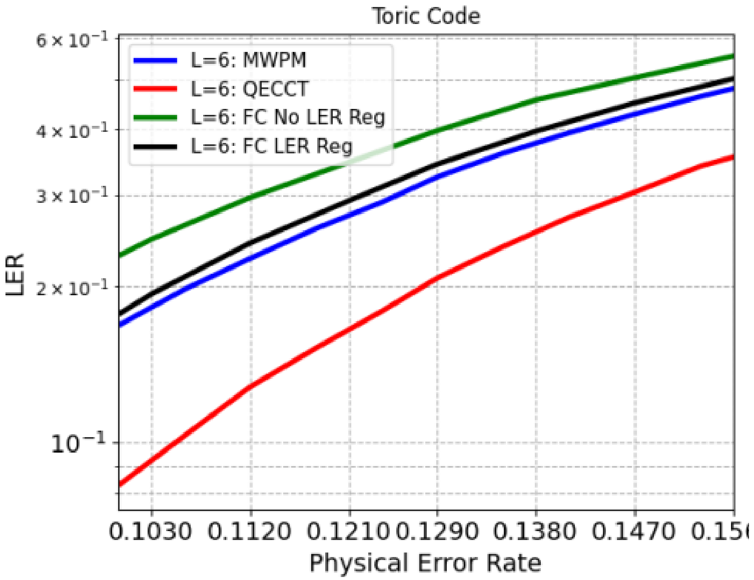


Depolarization noise

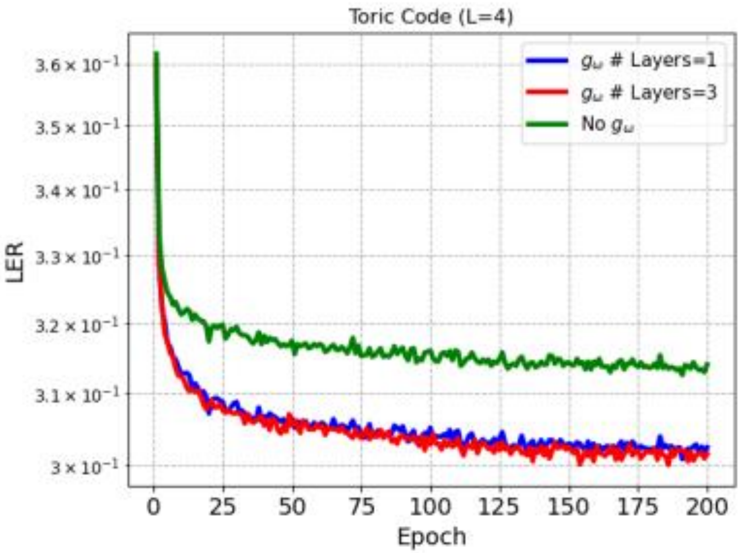


Depolarization noise
with faulty syndrome measurements

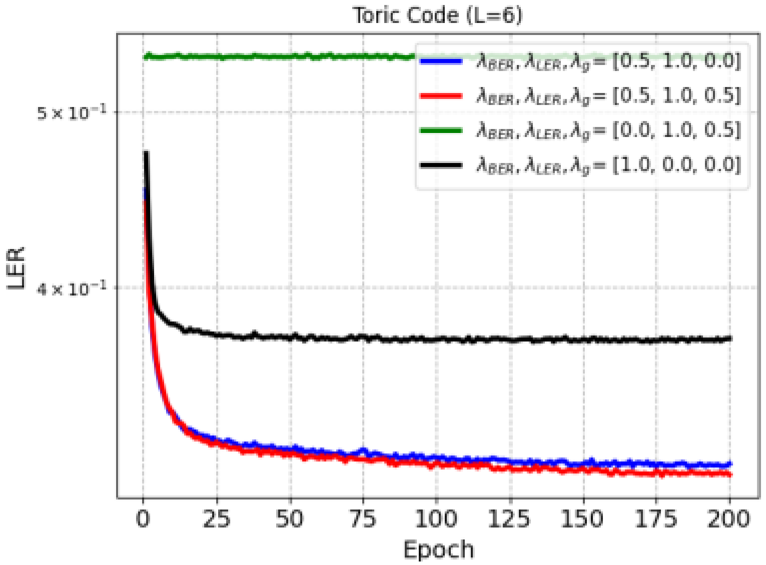
Analysis



Impact of the architecture
and of the LER optimization



Impact of the initial noise estimator

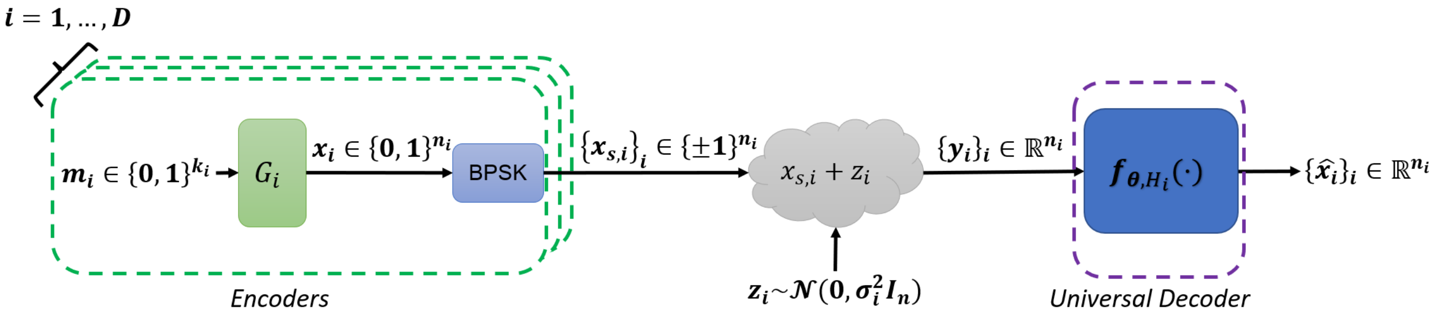


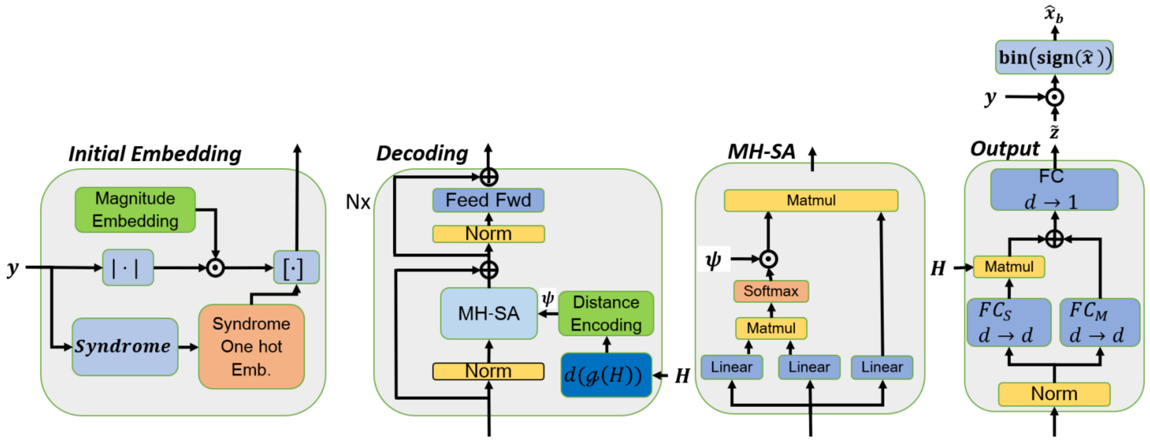
Impact of the BER regularization

One needs to develop, train, and deploy one (neural) decoder for each family of code, length, and rate.

How can we develop a <u>single</u> universal neural decoder which is code/length/rate invariant?

A Foundation Model for Error Correction Codes *ICLR24*



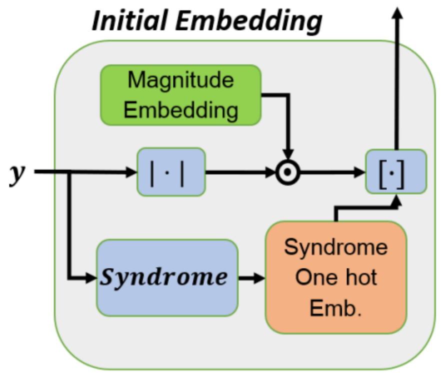


Code-Invariant Initial Embedding

- In ECCT, a unique model is crafted for every code and length where the initial embedding is designed such that **each input bit possesses its distinct embedding vector**, providing, as a byproduct, a **learned positional encoding.** $\boxed{\Phi = \left(h(y) \cdot 1_d^T\right) \odot W}$

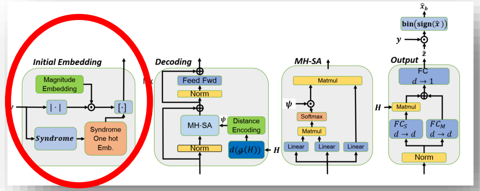
- In our **length-invariant** model (FECCT), we propose a new code-invariant embedding, where **a single embedding** is given for all **magnitude** elements, and **two embeddings** are given for every element of the binary **syndrome**.

$$\phi_i = \begin{cases} |y_i| W_M, & \text{if } \ i \leq n \\ W^S_{(s(y))_{i-n+1}}, & \text{otherwise} \end{cases}$$



With $\{W_M, W_0^S, W_1^S\} \in \mathbb{R}^d$.

- This new **length/rate-invariant initial encoding** requires **three** embedding vectors compared to the $\boldsymbol{2n - k}$ vectors of the ECCT.
- In contrast to ECCT which captures the bit position with learned embedding, our method **lacks positional information**.



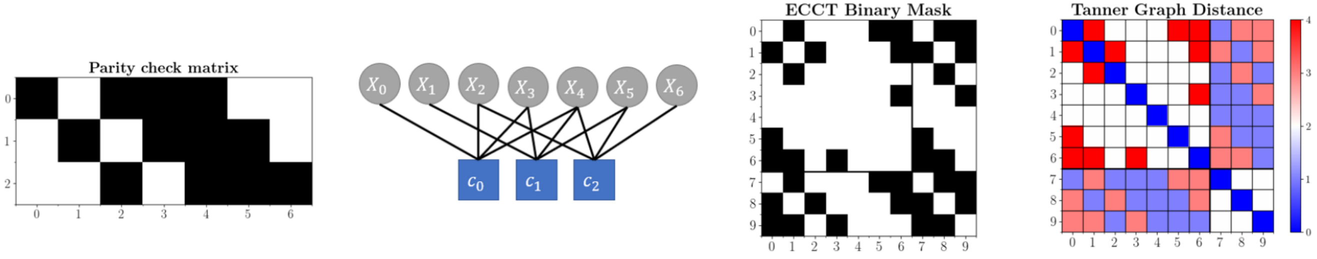
Tanner Graph Distance Masking as Code and Positional encoding

- FECCT's SA masking serves two purposes.
  - Similar to ECCT, it **integrates the code structure** into the transformer.
  - Adds the **relative position information** to the processed elements.
- The Tanner graph captures the **relations** between every two bits in the code (***relative** positional encoding*).
- We consider the distance matrix $\mathcal{G}(\boldsymbol{H}) \in \mathbb{N}^{(2n-k)\times(2n-k)}$, induced by the code (Tanner graph).
  - Each element (i, j) in this matrix is defined as the *length of the shortest path in the Tanner graph* between node i and node j.

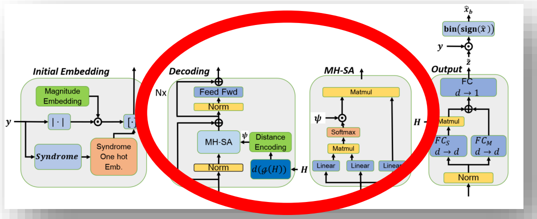
- We learn a *parameterized mapping* $\boldsymbol{\psi}: \mathbb{N} \rightarrow \mathbb{R}$ of the *distance matrix*, incorporated into the self-attention

$$A_H(Q,K,V) = \left(\mathrm{Softmax}\left(\frac{QK^T}{\sqrt{d}}\right) \odot \psi(\mathcal{G}(H))\right)V.$$

- This attention mechanism **generalizes** the ECCT which captures **only up to two rings** connectivity information.

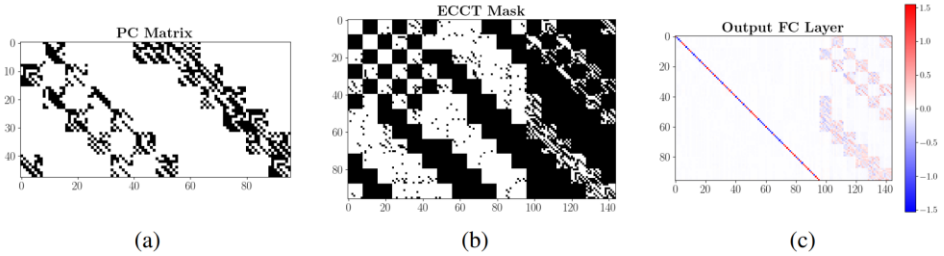


(a) (b) (c) (d)



Parity-Check Aware Prediction

- ECCT makes use of two *fully connected layers* (least length invariant modules) for the final prediction ($(2n-k) \rightarrow n$)
- ECCT's learned output layer is (surprisingly) greatly **induced** by the code/parity check matrix.

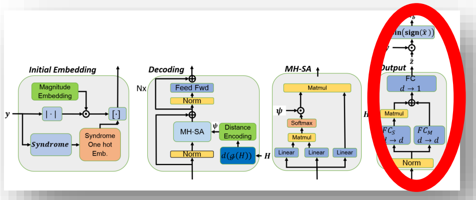


(a) (b) (c)

- Motivated by this phenomenon, we explicitly **enforce** a similar dependency structure.
- By *splitting* the syndrome and the channel output elements we integrate the remaining syndrome information by **aggregation** according to the parity check matrix connectivity

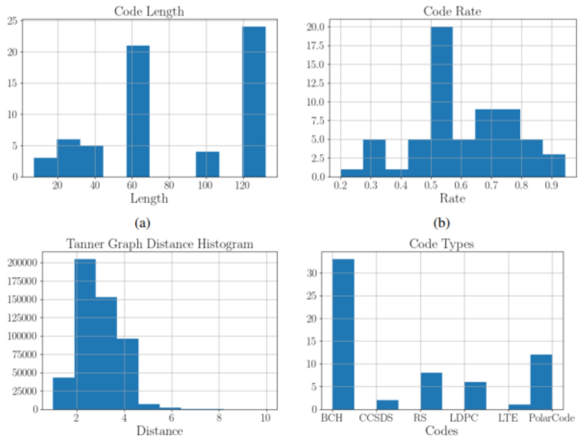
$$\hat{\tilde{\varepsilon}} = \left(\phi_{o,M} W_M + H^T(\phi_{o,S} W_S)\right) W_{d\rightarrow 1}$$

- This way, the final prediction is **code-aware** but also **code/length invariant**.
- Finally, the FECCT being invariant, its *number of parameters* is **independent of the code**.



Experiments

- Trained on multiple codes, **our _single_ decoder (_with smaller capacity_)** can match and even outperform **other methods _designed and trained separately on each code_**, in multiple scenarios
  - Pretrained codes
  - Zero-shot codes
  - Fine-tuned codes



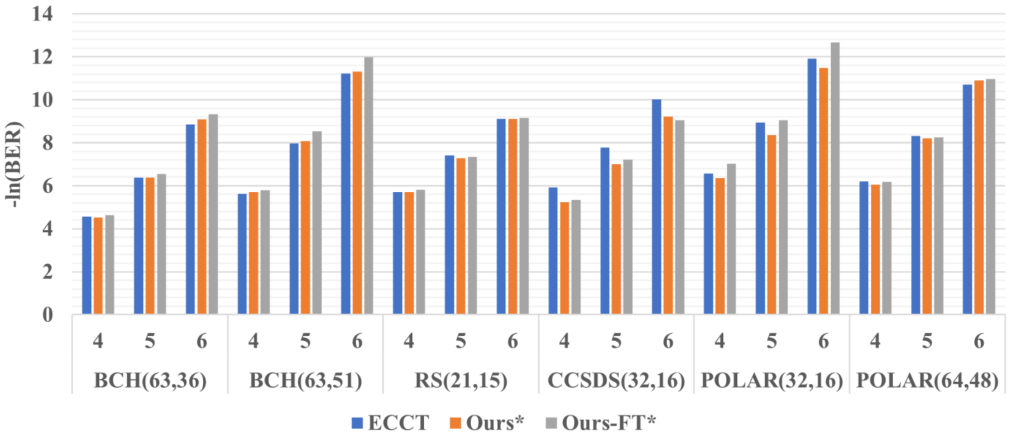
- $E_b/N_0 \sim SNR$
- Code(n,k)

Method	BP			Hyp BP			ARBP			ECCT			Ours		
E_b/N_0	4	5	6	4	5	6	4	5	6	4	5	6	4	5	6
BCH(63,36)	3.72 4.03	4.65 5.42	5.66 7.26	3.96 4.29	5.35 5.91	7.20 8.01	4.33 4.57	5.94 6.39	8.21 8.92	4.56	6.37	8.85	4.53	6.38	9.10
BCH(127,120)	NA	NA	NA	NA	NA	NA	NA	NA	NA	4.70	6.37	8.95	4.62	6.33	8.95
Reed Solomon(21,15)	NA	NA	NA	NA	NA	NA	NA	NA	NA	5.71	7.42	9.11	5.71	7.28	9.12
Reed Solomon(60,52)	NA	NA	NA	NA	NA	NA	NA	NA	NA	5.53	7.54	9.98	5.47	7.59	10.21
POLAR(32,16)	NA	NA	NA	NA	NA	NA	NA	NA	NA	6.57	8.94	11.91	6.36	8.36	11.49
POLAR(64,48)	3.52 4.26	4.04 5.38	4.48 6.50	4.25 4.59	5.49 6.10	7.02 7.69	4.77 5.57	6.30 7.43	8.19 9.82	6.21	8.32	10.71	6.06	8.21	10.90

Pretrained Codes

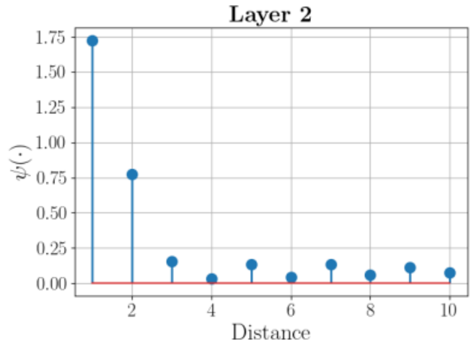
Supervision	Unlearned			Fully supervised									Zero-Shot		
Method	BP			Hyp BP			ARBP			ECCT			Ours		
E_b/N_0	4	5	6	4	5	6	4	5	6	4	5	6	4	5	6
BCH(63,45)	4.08 4.36	4.96 5.55	6.07 7.26	4.48 4.64	6.07 6.27	8.45 8.51	4.80 4.97	6.43 6.90	8.69 9.41	5.18	7.24	10.20	5.18	7.32	10.31
BCH(63,51)	4.34 4.50	5.29 5.82	6.35 7.42	4.64 4.80	6.08 6.44	8.16 8.58	4.95 5.17	6.69 7.16	9.18 9.53	5.63	7.96	11.22	5.71	8.07	11.31
BCH(127,92)	NA	NA	NA	NA	NA	NA	NA	NA	NA	4.10	5.71	8.38	4.11	5.84	8.79
BCH(255,163)	NA	NA	NA	NA	NA	NA	NA	NA	NA	3.34	4.13	5.80	3.34	4.13	5.76
CCSDS(128,64)	6.55 -	9.65 -	13.78 -	6.99 -	10.57 -	15.27 -	7.25 -	10.99 -	16.36 -	6.77	10.51	15.90	6.52	9.67	15.01
CCSDS(32,16)	NA	NA	NA	NA	NA	NA	NA	NA	NA	5.93	7.77	10.02	5.23	7.00	9.21
POLAR(128,86)	3.80 4.49	4.19 5.65	4.62 6.97	4.57 4.95	6.18 6.84	8.27 9.28	4.81 5.39	6.57 7.37	9.04 10.13	6.39	9.08	12.70	5.53	7.90	11.29
POLAR(64,32)	3.52 4.26	4.04 5.38	4.48 6.50	4.25 4.59	5.49 6.10	7.02 7.69	4.77 5.57	6.30 7.43	8.19 9.82	6.91	9.18	12.34	5.88	7.91	10.76

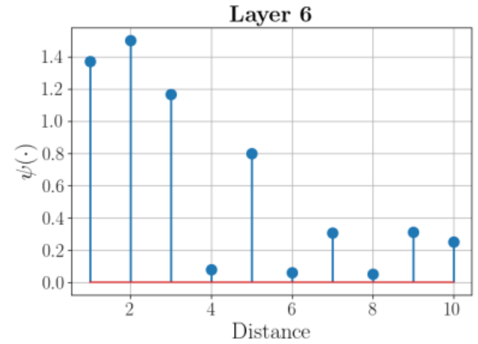
Zero-Shot Codes



Fine-Tuned Codes

Analysis





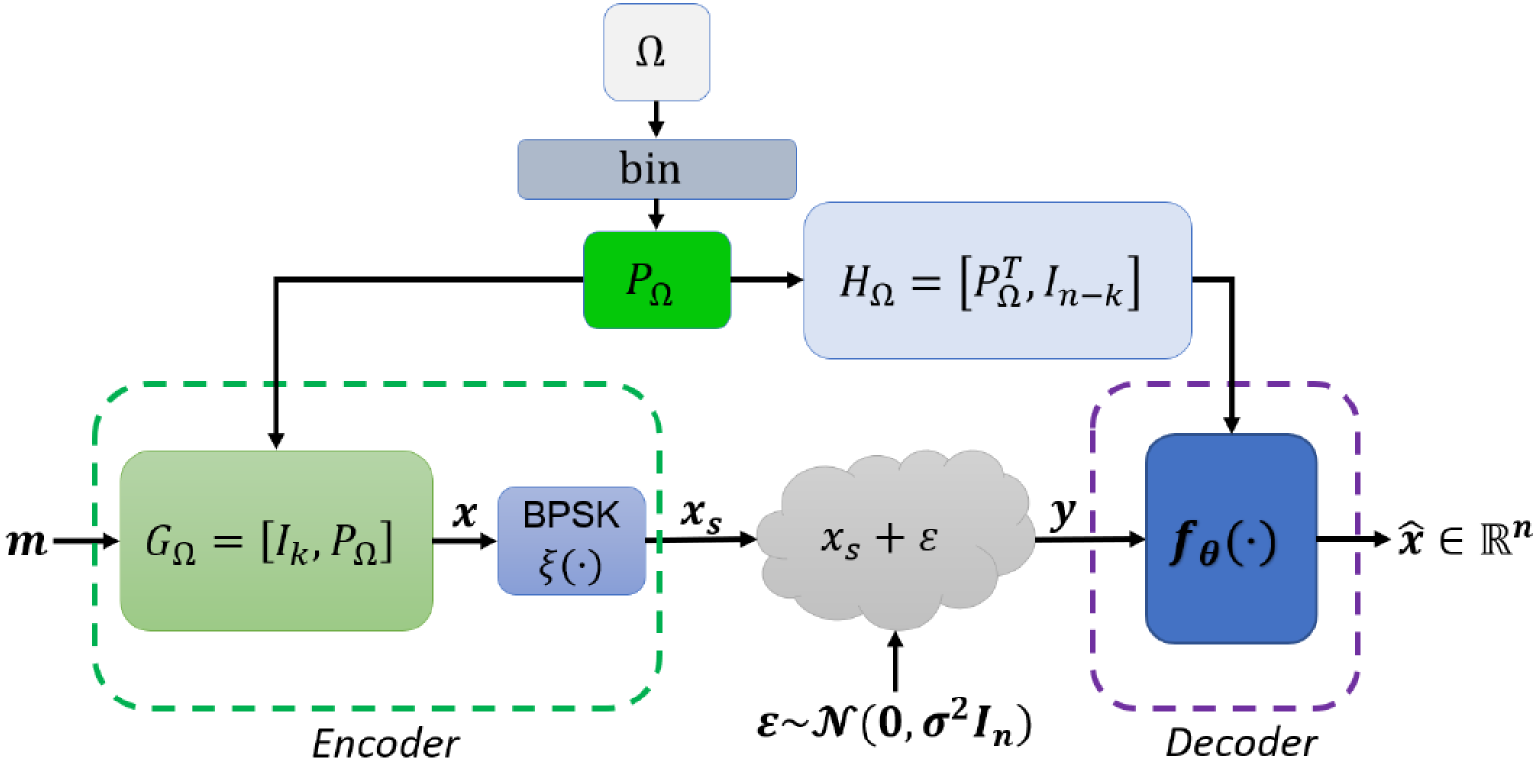
Method	POLAR(64,32)			BCH(63,45)		
	4	5	6	4	5	6
ECCT	4.12	5.22	6.67	4.45	5.81	7.65
ECCT + **II**	4.27	5.54	7.14	4.52	5.98	7.92
ECCT + **IO**	4.44	5.73	7.40	4.41	5.76	7.62
ECCT + **II** + **IO**	4.09	5.26	6.80	4.31	5.62	7.41
ECCT + **DM**	4.44	5.73	7.37	4.74	6.34	8.53
ECCT + DM + **II**	4.44	5.73	7.37	5.17	7.07	9.59
ECCT + DM + **IO**	4.36	5.64	7.32	4.53	6.01	8.03
FECCT: ECCT + DM + II + IO	4.36	5.64	7.32	4.52	5.98	8.05

Method	FECCT - single			FECCT		
	4	5	6	4	5	6
POLAR(64,48)	6.35	8.50	11.12	6.06	8.21	10.96
POLAR(128,86)*	3.90	5.36	7.57	5.53	7.90	11.29
BCH(63,36)	4.01	5.42	7.30	4.53	6.38	9.10
BCH(63,51)*	4.65	6.35	8.73	5.71	8.07	11.31
Reed Solomon(21,15)	4.25	4.62	4.97	4.56	6.83	10.51
Reed Solomon(60,52)	3.68	3.81	3.77	5.47	7.49	10.24
CCSDS(128,64)*	2.90	3.42	4.30	6.52	9.67	15.01
CCSDS(32,16)*	4.10	4.54	4.43	5.23	7.00	9.21

- **Learned Distance Mapping**: FECCT seems to assign the **most** impactful mapping for the elements distanced by one and two nodes, **remarkably matching the ECCT's two-ring heuristic.**
- **Architectural Ablation** The ablation demonstrate the beneficial impact of **each of the contributions** on the obtained accuracy compared to SOTA
- **Generalization:** To show the importance of dataset diversity, we show that training FECCT on one **single** code is slightly better on the trained code but totally lacks generalization

How can we co-learn binary linear block codes along with the neural decoder?

Learning Linear Block Error Correction Codes *ICML24*



End to End Optimization

- We assume the **standard** (/canonical/systematic) **form** of the code for differentiable and fast optimization

$$G = [I_k, P], \qquad P \in \{0,1\}^{k\times(n-k)}$$

$$\Rightarrow \; H = [P^T, I_{n-k}]$$

  - Using a *general form matrix* form is preferable but requires **fast and differentiable** computation of its inverse at each iteration.

- We want **trainable parameterization** of the code $\Omega \in \mathbb{R}^{k\times(n-k)}$ such that, with $\mathrm{bin}(\cdot)$ a binarization function, we have

$$P = P_\Omega = bin(\Omega),.$$

- The **end-to-end optimization objective** is now defined by

$$\mathcal{L}(\Omega, \theta) = \mathbb{E}_{m\sim\mathrm{Bern}^k(1/2),\varepsilon\sim\mathcal{Z}}\mathrm{BCE}\big(f_\theta(h_\Omega(y_\Omega)), \mathrm{bin}(\tilde{\varepsilon})\big)$$

$$y_\Omega = \xi(\phi(m, G_\Omega)) + \varepsilon$$
$$h_\Omega(y_\Omega) = [|y_\Omega|, H_\Omega \mathrm{bin}(y_\Omega)]$$

Optimization over *GF(2)*

❖ **Two** main **non-differentiable** modules in the pipeline

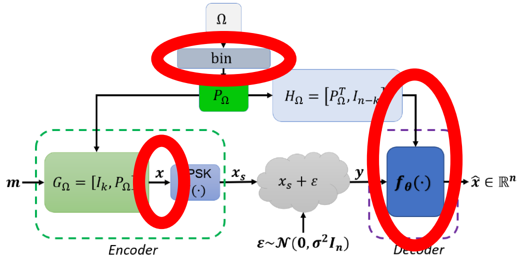
- **Binarization**
  - Performed via the Straight-Through-Estimator (STE)

$$\begin{cases} \text{bin}(u) = \xi^{-1}(\text{sign}(u)) \\ \frac{\partial \text{bin}(u)}{\partial u} = -\frac{\mathbb{1}_{|u|\leq\tau}}{2} \end{cases}$$

- **Dot products over *GF(2)***
  - Performed via polar transform [1] $\xi(u) = 1 - 2u, u \in \{0,1\} \rightarrow \xi(u \oplus v) = \xi(u)\xi(v), \forall u, v \in \{0,1\}$

$$\left(\phi(m, G_\Omega)\right)_i := G_{\Omega i} \oplus m = \xi^{-1}\left(\Pi_{j=1}^{k} \xi\left((G_\Omega)_{ij} \cdot m_j\right)\right)$$

- The new form defines a **multilinear polynomial** (potentially *inducing saddle-point optimization*) and the gradient can now be computed in a **differentiable** manner.



[1] Deep Quantum Error Correction, Y. Choukroun and L. Wolf, AAAI24

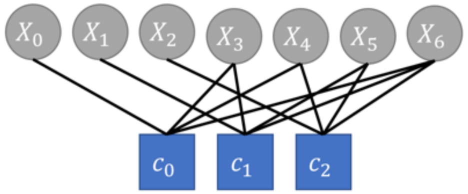
Differentiable Masking

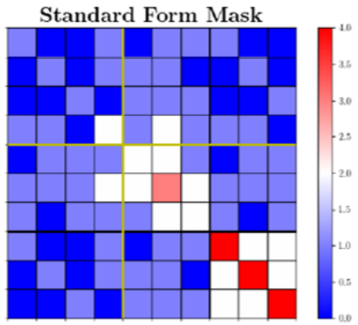
- We suggest to **extend FECCT** to support end-to-end training
- Existing masking methods induced from the code are **extracted once in a non-differentiable fashion**,
  - No information can be **backpropagated** during the optimization **from the mask to the code**
- We learn a parameterized mapping $\psi_\gamma : \mathbb{N} \to \mathbb{R}$ of the elements constituting the mask, which is **derived by the parity-check matrix**, such that

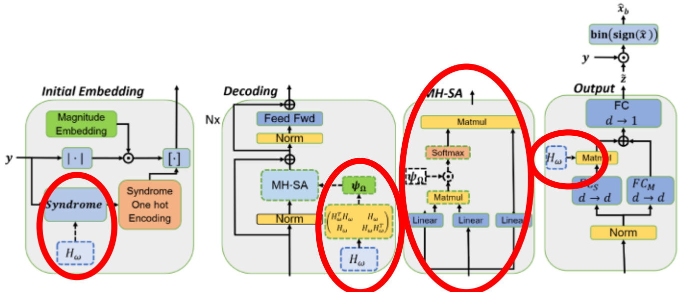
$$A_H(Q,K,V) = \text{Softmax}\left(\frac{QK^T + \psi_\gamma(g(H_\Omega))}{\sqrt{d}}\right)V,$$

$$g(H_\Omega) = \begin{pmatrix} H_\Omega^T H_\Omega & H_\Omega \\ H_\Omega^T & H_\Omega H_\Omega^T \end{pmatrix}$$

- Represents the **two-step transition matrix** between every two nodes





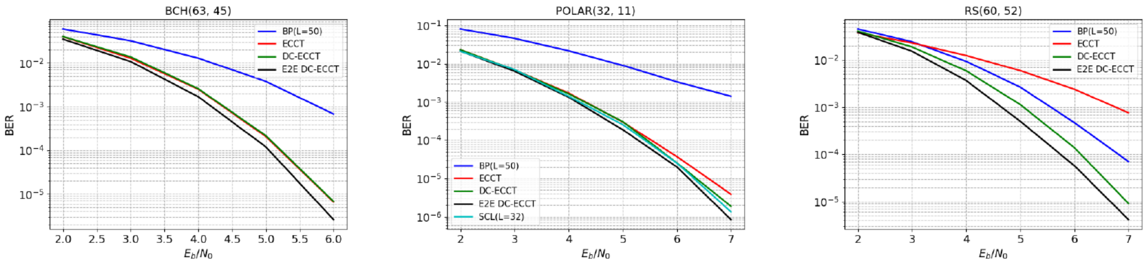


Experiments

- Even for fixed (not trained) code the **proposed neural decoder** outperforms the state-of-the-art neural decoder.

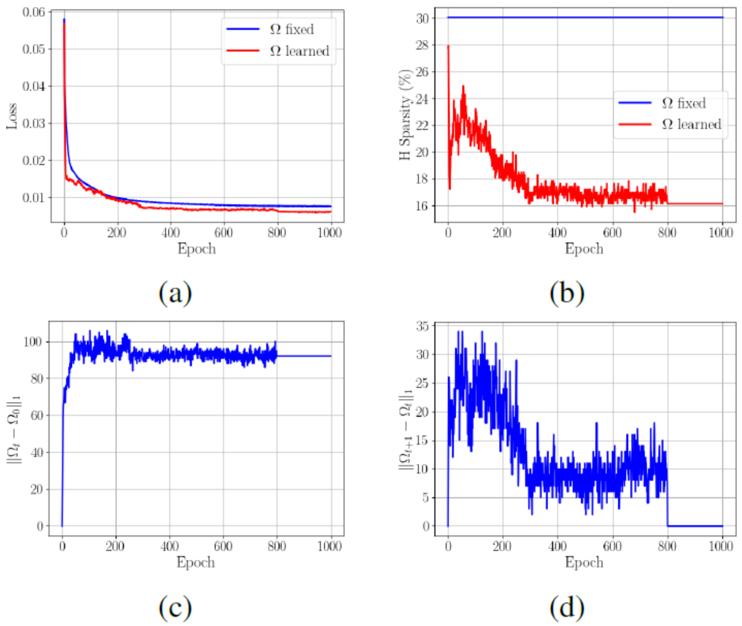
- The **end-to-end optimization** of the code improves the performance by very large margins.

Method	BP			SCL			ECCT			DC-ECCT			E2E DC-ECCT		
E_b/N_0	4	5	6	4	5	6	4	5	6	4	5	6	4	5	6
POLAR(32,11)	3.29	3.77	4.21	6.22	8.06	10.28	4.46	5.57	7.01	4.53	5.69	7.08	4.65	5.81	7.28
	3.84	4.71	5.70	*6.45*	*8.37*	*10.60*	6.37	8.12	10.19	6.41	8.09	10.57	**6.62**	**8.57**	**11.71**
POLAR(64,32)	3.53	4.02	4.45	7.24	9.74	12.91	4.56	5.93	7.75	4.40	5.76	7.67	4.72	6.22	8.13
	4.29	5.35	6.45	**8.16**	**10.73**	**13.98**	7.19	9.70	13.33	7.45	*10.49*	*13.74*	*7.59*	10.38	13.09
BCH(31,16)	4.59	5.87	7.57	NA			4.61	5.97	7.69	4.97	6.56	8.54	5.30	6.89	9.09
	5.12	6.87	9.27				6.37	8.32	10.63	*7.07*	**9.69**	*12.54*	**7.19**	*9.08*	**12.84**
BCH(63,45)	4.07	4.92	6.03	NA			4.59	6.07	8.13	4.54	6.07	8.14	4.98	6.69	8.97
	4.35	5.60	7.24				*6.25*	*8.78*	*12.45*	6.08	8.64	12.41	**6.37**	**9.09**	**13.12**
LDPC(49,24)	5.25	7.15	9.86	NA			4.21	5.32	6.56	4.08	5.29	6.55	4.95	6.46	8.33
	6.09	*8.75*	*11.91*				5.34	6.43	7.21	5.57	6.55	7.21	**6.28**	**8.77**	**12.33**
RS(60,52)	4.43	5.32	6.43	NA			4.37	5.11	6.03	5.04	6.68	8.82	5.12	6.80	9.02
	4.69	6.43	7.56				4.37	5.13	6.04	**5.61**	**7.59**	*9.82*	**5.61**	*7.56*	**9.90**



Analysis

Code	Baseline			Standard form			Random H			Random Ω			E2E Ω		
E_b/N_0	4	5	6	4	5	6	4	5	6	4	5	6	4	5	6
(31,16)	4.59	5.87	7.57	3.97	4.67	5.57	3.23	3.72	4.36	4.22	5.01	5.86	**6.13**	**7.95**	**9.90**
	5.12	6.87	9.27	4.64	5.89	7.37	3.42	4.18	5.33	4.89	6.14	7.38	**6.42**	**8.31**	**10.24**
(63,45)	4.07	4.92	6.03	4.37	5.33	6.42	3.72	4.26	4.93	3.96	4.61	5.41	**5.55**	**7.33**	**9.23**
	4.35	5.60	7.24	4.78	6.30	8.22	3.93	4.79	5.93	4.27	5.29	6.69	**6.15**	**8.60**	**11.32**
(60,52)	4.43	5.32	6.43	4.60	5.65	6.94	4.41	5.32	6.45	4.60	5.63	6.93	**4.73**	**5.79**	**7.01**
	4.69	5.95	7.56	4.83	6.21	8.00	4.65	5.90	7.53	4.83	6.19	7.97	**4.99**	**6.45**	**8.35**
(64,32)	3.53	4.02	4.45	3.82	4.37	5.03	2.92	3.37	3.73	3.34	3.90	4.64	**6.93**	**9.49**	**12.51**
	4.29	5.35	6.45	4.81	5.74	6.71	2.96	3.54	4.17	3.59	4.53	5.87	**7.69**	**10.04**	**12.94**

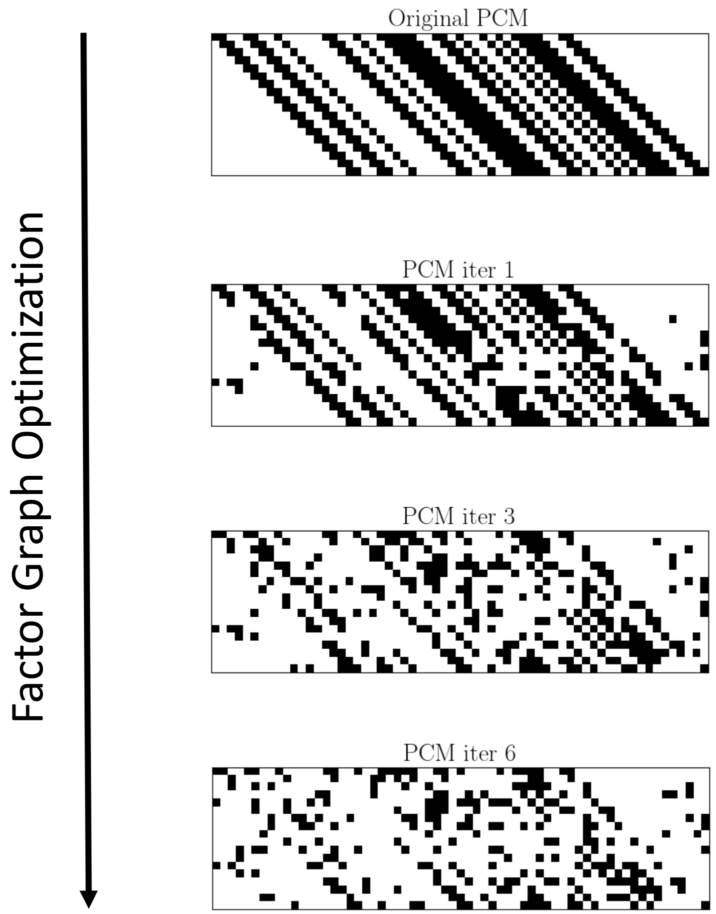


Code	(31,16)			(32,11)		
E_b/N_0	4	5	6	4	5	6
Our	5.16	6.51	8.19	4.40	5.54	7.04
Mask V2	4.83	6.18	7.85	4.04	5.05	6.28
STE	5.11	6.53	8.20	4.63	5.89	7.39
Mask S.G.	4.54	5.81	7.50	4.26	5.39	6.79
Random m	5.04	6.44	8.13	4.38	5.50	6.84

- **BP on learned codes**:
  The learned codes outperform other codes under the BP decoder by very large margins, *even if the code is presented in standard form*.
  Our method seems to provide ***good codes*** in a broader sense.
- **Training Dynamics**
  The encoder-decoder models enable faster and better training.
  Fast change during first stages.
  The learned code is *generally sparse*.
- **Ablation Study:**
  We show that the different elements of the **architecture** and **training** are crucial for performant results.

How can we design optimal codes according to Belief Propagation decoders' inductive bias?

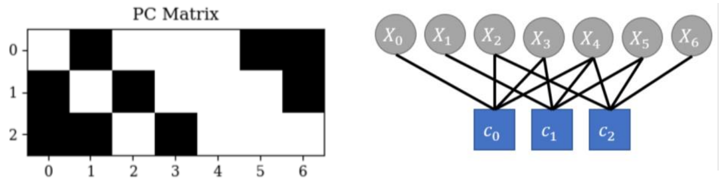
Factor Graph Optimization of Error-Correcting Codes for Belief Propagation Decoding *sub to Neurips24*



Belief Propagation Decoding

- A **factor graph** is a representation of a discrete probability distribution that takes advantage of conditional independencies between variables to make the representation *more compact*

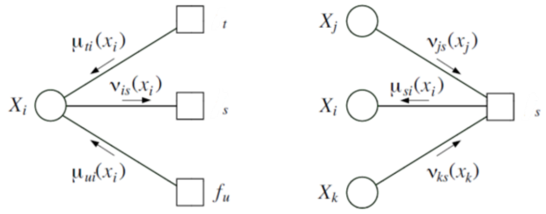
$$p(x_1, \dots, x_n) = \frac{\prod_{i=1}^{n-k} \phi_i(x_{S_i})}{Z}$$



- **Belief-Propagation** (Sum-product algorithm) allows efficient marginal inference $p(x_i)$ via variable elimination as *message-passing* over the factor graph.

$$\nu_{X_i \to \phi_t}(x_i) = \prod_{\phi_k \in N(i) \setminus \{\phi_t\}} \mu_{\phi_k \to X_i}(x_i)$$

$$\mu_{\phi_t \to X_i}(x_i) = \sum_{x_{S_t \setminus \{i\}}} \phi_s(x_{S_t}) \prod_{j \in S_t \setminus \{i\}} \nu_{X_j \to \phi_t}(x_j)$$

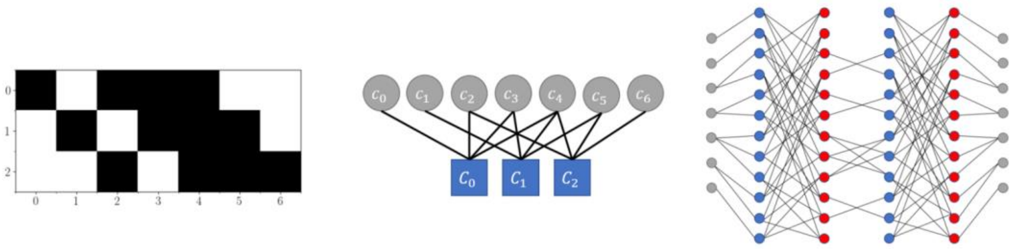


- For non-tree Bayesian graphs, the inference is not tractable. Thus, (**loopy**) belief-propagation is performed **iteratively** (until convergence)

$$P(x_i) = \phi_i(x_i) \prod_{\phi_k \in N(i)} \mu^{\boldsymbol{T}}_{\phi_k \to X_i}(x_i)$$

- Belief-propagation *decoding* is generally represented as a **Trellis** graph unrolling of the factor/**Tanner** graph (*log-likelihoods*).

1. $L_v = \log\left(\frac{\Pr(c_v = 1 | y_v)}{\Pr(c_v = 0 | y_v)}\right)$
2. $x_e^{2k+1} = x_{(c,v)}^{2k+1} = L_v + \sum_{e' \in N(v) \setminus \{(c,v)\}} x_{e'}^{2k}.$
3. $x_e^{2k} = x_{(c,v)}^{2k} = 2\text{arctanh}\left(\prod_{e' \in N(c) \setminus \{(c,v)\}} \tanh\left(\frac{x_{e'}^{2k-1}}{2}\right)\right)$
4. $o_v = L_v + \sum_{e' \in N(v)} x_{e'}^{2L}$



Belief Propagation Codes

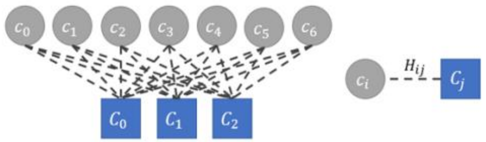
- We wish to obtain BP-optimized codes by solving the following **factor graph learning** (*structure learning*) objective

$$H^* = \underset{H \in \{0,1\}^{(n-k)\times n}}{\arg\min} \mathbb{E}_{m \sim \text{Bern}^k(1/2), \varepsilon \sim \mathcal{Z}, T \in \mathbb{N}_+} \mathcal{D}\Big(f_{H,T}\big(\phi(G(H)m) + \varepsilon\big), m \Big) + \mathcal{R}(H)$$

  - $\boldsymbol{f_{H,T}}$ is the **BP decoder** built upon $\boldsymbol{H}$ with $\boldsymbol{T}$ iterations
  - $\boldsymbol{\phi}(\cdot)$ denotes the codeword **modulation**
  - $\boldsymbol{\mathcal{D}}$ is the **metric** of interest and $\boldsymbol{\mathcal{R}}$ denotes a potential **regularization** of interest (e.g., sparsity or code structure)

- Multiple challenges arise
  1. The optimization is highly **non-differentiable** (NP-hard binary non-linear integer programming)
  2. $\boldsymbol{x} = \boldsymbol{G}(\boldsymbol{H})\boldsymbol{m} = \boldsymbol{Gm}$ is both **highly non-differentiable** (matrix-vector multiplication over $GF(2)$) and **computationally expensive** (inverse via Gaussian elimination of $\boldsymbol{H}$)
  3. The modulation $\boldsymbol{\phi}(\cdot)$ can be non-differentiable
  4. BP assumes a **fixed code** (i.e., the factor graph edges) upon which the decoder is implemented.

Structure Learning via *Tensor* Belief *Back*propagation



- BP decoders are generally implemented using *sparse graphs* via the *Trellis graph* depiction.
- We propose a *Tanner graph learning* approach, where the *bipartite graph* is assumed as **complete with binary weighted edges.**

- The two alternating stages of BP can be represented in a ***differentiable*** *matrix form* rather than its *static* graph formulation
  - $Q_{ij} = L + \sum_{j' \in C_i \backslash j} R_{j'i} \equiv L + \sum_{j'} R_{j'i} H_{j'i} - R_{ji}$
  - $R_{ji} = 2\text{arctanh}\left(\prod_{i' \in V_j \backslash i} \tanh\left(\frac{Q_{i'j}}{2}\right)\right)$

    $= 2\text{arctanh}\left(\prod_{i'}\left(\tanh\left(\frac{Q_{i'j}H_{ji'}}{2}\right) + (1 - H_{ji'})\right) / \left(\tanh\left(\frac{Q_{ij}}{2}\right) + \epsilon\right)\right)$

  - $(1 - H) \in \{0,1\}^{(n-k)\times n}$ represents the identity element of multiplication

- BP remains differentiable with respect to $\boldsymbol{H}$ as a composition of differentiable functions

Algorithm 1: Tensor Belief Propagation

```
function BP(llr, H, iters, eps=1e-7)
    // llr is the batched LLR matrix, (B, n)
    // H is the binary parity-check matrix, (n-k,n)
    // iters is the number of BP iterations
    H = H.unsqueeze(dim=0).T
    C = llr.unsqueeze(dim=-1)
    for t in range(iters) do
        Q = C if t == 0 else C + sum(R*H,dim=-1).unsqueeze(dim=-1) - R
        tmp = tanh(0.5*Q)
        R = 2*atanh( prod(tmp*H+(1-H),dim=1)/(tmp+eps) )
    return C.squeeze()+sum(R*H,dim=-1)
```

Belief Propagation Codes Optimization

1. Highly non-differentiable optimization
2. $\cancel{x = G(H)m = Gm}$
3. ~~Non-differentiable modulation $\phi(\cdot)$~~
4. ~~BP assumes a fixed code~~

- The tensor reformulation solves the major *graph learning challenge* (challenge 4).
- For any given $\boldsymbol{H}$ the conditional independence of error probability under symmetry [1] is satisfied for message passing algorithms
  - It is enough to optimize the ***zero codeword only*** i.e., c = Gm = 0 (challenge 2)
  - As a by-product, the optimization problem is ***invariant to the choice of modulation*** (challenge 3)

- To optimize $\boldsymbol{H}$ (challenge 1) we relax the NP-hard binary programming problem to an unconstrained parameterized problem
  - Defining $\Omega \in \mathbb{R}^{(n-k)\times n}$ we have $H := H(\Omega) = \text{bin}(\Omega)$
  - Implemented via shifted *straight-through estimator* (**STE**)

$$\text{bin}(u) = (1 - \text{sign}(u))/2\,, \quad \partial\,\text{bin}(u)/\partial u = -0.5\mathbb{1}_{|u|\leq 1}$$

  - The final objective is given by the empirical risk objective (with $\boldsymbol{c_s} = \boldsymbol{\phi}(\boldsymbol{c_0})$ denotes the *modulated* **zero** codeword)

$$\boxed{\mathcal{L}(\Omega) = \sum_{t=1}^{T}\sum_{i=1}^{n} \text{BCE}\Big(f_{\text{bin}(\Omega),t}\big(c_s + \varepsilon_i\big), c\Big) + \mathcal{R}(\text{bin}(\Omega))}$$

- While *highly non-convex*, the objective is (sub)differentiable and thus optimizable via classical ***first-order** methods*
  - Since $\boldsymbol{H}$ is binary, only changes in the **sign** of $\boldsymbol{\Omega}$ are relevant for the optimization.
  - Given the gradient $\boldsymbol{\nabla_\Omega \mathcal{L}}$ computed on sufficient statistic we propose a binary programming aware **line-search procedure** limiting the number of steps to up to $\boldsymbol{n(n-k)} \ll \boldsymbol{2^{n(n-k)}}$

$$\lambda^* = \arg\min_{\lambda\in\mathcal{I}_\Omega} \mathcal{L}(\Omega - \lambda\nabla_\Omega\mathcal{L}), \qquad \mathcal{I}_\Omega = \{s_i = \left(\frac{\Omega}{\nabla_\Omega\mathcal{L}}\right)_i | s_i > 0\}$$

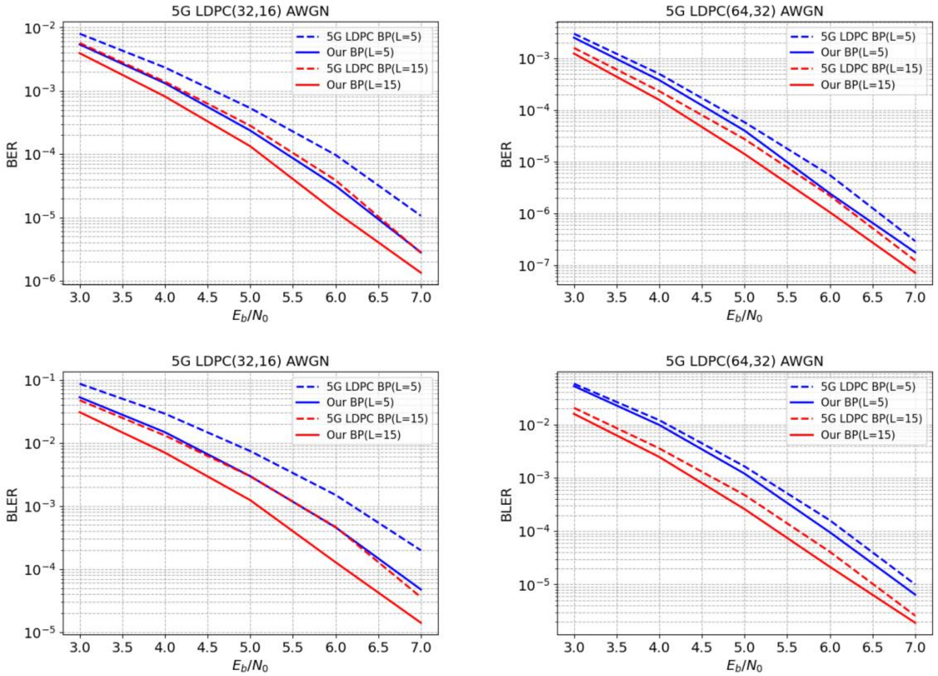
[1] Thomas J Richardson and Rüdiger L Urbanke. The capacity of low-density parity-check codes under message-passing decoding. IEEE Transactions on information theory, 2001.

Experiments

- Our method improves by *large margins* **all** code families on the three different **channel noise** scenarios and with various number of **decoding iterations** (L=5,15; first and second row).

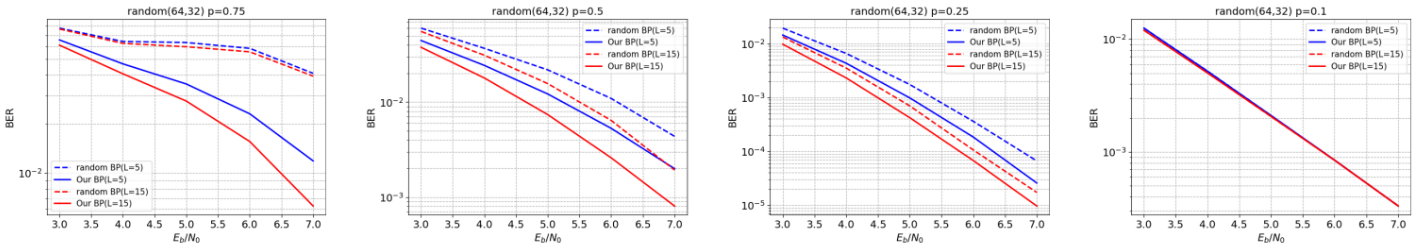
Channel	AWGN						Fading						Bursting					
Method	BP			Our			BP			Our			BP			Our		
E_b/N_0	4	5	6	4	5	6	4	5	6	4	5	6	4	5	6	4	5	6
BCH(63,45)	4.06	4.91	6.04	**5.44**	**6.93**	**8.60**	3.09	3.46	3.90	**3.96**	**4.58**	**5.27**	3.60	4.32	5.19	**4.05**	**5.07**	**6.27**
	4.21	5.24	6.59	**5.70**	**7.35**	**9.16**	3.13	3.55	4.04	**4.10**	**4.80**	**5.56**	3.67	4.52	5.59	**4.21**	**5.40**	**6.85**
CCSDS(128,64)	6.46	9.61	13.99	**7.34**	**10.48**	**14.37**	5.72	7.42	9.47	**6.73**	**8.45**	**10.45**	5.29	7.81	11.25	**6.23**	**8.80**	**11.90**
	7.32	10.83	15.43	**8.61**	**12.26**	**16.00**	6.43	8.29	10.28	**8.05**	**10.07**	**12.37**	5.98	8.85	12.53	**7.39**	**10.43**	**13.28**
LDPC(121,60)	4.81	7.17	10.75	**7.70**	**10.87**	**14.25**	4.10	5.23	6.68	**6.68**	**8.47**	**10.50**	3.97	5.75	8.40	**6.23**	**8.89**	**11.98**
	5.31	7.96	11.85	**8.86**	**11.91**	**14.41**	4.42	5.61	7.04	**7.71**	**9.67**	**11.76**	4.31	6.37	9.25	**7.26**	**10.03**	**12.88**
LDPC(121,80)	6.59	9.68	13.43	**7.77**	**11.21**	**15.06**	4.60	5.80	7.22	**5.55**	**6.90**	**8.36**	5.30	7.60	10.66	**6.23**	**8.87**	**12.19**
	7.35	10.94	15.46	**8.75**	**12.45**	**15.67**	4.97	6.29	7.82	**6.25**	**7.80**	**9.47**	5.81	8.50	12.15	**6.99**	**10.09**	**13.74**
LDPC(128,64)	3.66	4.65	5.80	**5.54**	**7.37**	**9.44**	3.22	3.80	4.44	**4.86**	**5.94**	**7.15**	3.23	4.08	5.09	**3.72**	**5.00**	**6.54**
	4.00	5.16	6.42	**6.56**	**8.70**	**10.81**	3.51	4.18	4.84	**5.64**	**6.85**	**8.14**	3.48	4.51	5.66	**4.13**	**5.72**	**7.66**
LDPC(32,16)	4.36	5.59	7.18	**5.48**	**7.02**	**8.92**	4.03	4.70	5.47	**5.26**	**6.02**	**6.82**	3.88	4.89	6.18	**4.77**	**6.02**	**7.52**
	4.64	6.07	7.94	**5.76**	**7.44**	**9.41**	4.29	5.06	5.90	**5.43**	**6.23**	**6.97**	4.09	5.26	6.76	**5.01**	**6.35**	**7.96**
LDPC(96,48)	6.73	9.48	12.98	**7.22**	**9.96**	**13.37**	3.83	4.57	5.35	**5.37**	**6.51**	**7.71**	5.68	7.94	10.90	**5.90**	**8.19**	**10.91**
	7.50	10.61	**14.26**	**8.29**	**11.12**	14.06	4.17	4.94	5.73	**6.14**	**7.38**	**8.65**	6.33	8.91	**11.99**	**6.71**	**9.28**	11.75
LTE(132,40)	2.94	3.32	3.57	**3.25**	**3.71**	**4.04**	3.17	3.45	3.67	**4.49**	**4.99**	**5.47**	2.75	3.17	3.47	**2.99**	**3.44**	**3.78**
	3.37	3.79	4.09	**3.93**	**4.49**	**4.89**	3.60	3.82	4.01	**5.32**	**5.81**	**6.31**	3.17	3.62	3.96	**3.53**	**4.03**	**4.41**
MACKAY(96,48)	6.75	9.45	**12.85**	**7.03**	**9.63**	12.78	6.28	7.86	9.55	**6.53**	**8.06**	**9.77**	5.72	7.97	10.81	**5.95**	**8.23**	**10.91**
	7.59	10.52	**14.09**	**7.99**	**10.97**	14.05	7.04	8.76	10.64	**7.47**	**9.32**	**11.19**	6.39	8.90	11.91	**6.82**	**9.41**	**12.71**
POLAR(128,86)	3.76	4.17	4.58	**4.83**	**5.87**	**6.58**	3.15	3.53	3.91	**3.64**	**4.28**	**4.94**	3.48	3.96	4.37	**3.69**	**4.51**	**5.18**
	4.02	4.67	5.38	**5.37**	**6.88**	**8.10**	3.28	3.73	4.18	**3.92**	**4.70**	**5.52**	3.65	4.31	4.97	**3.87**	**4.91**	**5.91**
RS(60,52)	4.41	5.32	6.41	**5.02**	**6.38**	**7.99**	3.11	3.41	3.77	**3.37**	**3.73**	**4.12**	3.85	4.58	5.44	**4.17**	**5.18**	**6.40**
	4.54	5.52	6.64	**5.07**	**6.47**	**8.12**	3.13	3.43	3.81	**3.38**	**3.75**	**4.15**	3.91	4.72	5.67	**4.21**	**5.27**	**6.56**
LDPC PGE2(64,32)	4.38	5.12	6.04	**4.45**	**5.19**	**6.10**	4.08	4.44	4.81	**4.10**	**4.46**	**4.85**	4.07	4.69	5.43	**4.07**	**4.70**	**5.43**
	4.38	5.13	6.04	**4.44**	**5.19**	**6.10**	4.08	4.44	4.81	**4.10**	**4.47**	**4.85**	4.06	4.69	5.43	**4.06**	**4.69**	**5.44**
LDPC PGE5(64,32)	6.02	8.20	10.95	**6.53**	**8.73**	**11.56**	5.63	6.86	8.31	**6.22**	**7.48**	**8.82**	5.18	6.97	9.34	**5.59**	**7.41**	**9.51**
	6.63	9.06	**12.30**	**7.13**	**9.48**	12.20	6.19	7.52	9.02	**6.96**	**8.34**	**9.85**	5.68	7.75	**10.19**	**6.12**	**8.06**	10.13
LDPC PGE10(64,32)	3.98	5.17	6.70	**5.56**	**7.22**	**9.13**	3.52	4.18	4.95	**5.02**	**6.00**	**7.11**	3.48	4.47	5.75	**4.26**	**5.50**	**7.01**
	4.27	5.77	7.67	**6.25**	**8.28**	**10.59**	3.71	4.47	5.30	**5.60**	**6.72**	**7.90**	3.67	4.90	6.46	**4.73**	**6.22**	**8.01**

Method	SCL			BP (Polar)			Our (Polar)			BP (5G LDPC)			Our (5G LDPC)		
E_b/N_0	4	5	6	4	5	6	4	5	6	4	5	6	4	5	6
(32,16)	6.22	8.06	10.28	4.36	5.59	7.18	5.48	7.02	8.92	6.06	7.53	9.23	**6.63**	**8.53**	**10.36**
	6.45	8.37	10.60	4.64	6.07	7.94	5.76	7.44	9.41	6.57	8.17	10.16	**7.11**	**8.92**	**11.31**
(64,32)	7.36	9.82	12.98	-	-	-	-	-	-	7.59	9.75	12.10	**7.87**	**10.12**	**12.92**
	8.10	10.73	14.00							8.36	10.50	13.02	**8.75**	**11.17**	**13.76**
(128,64)	8.49	11.46	16.16	-	-	-	-	-	-	9.90	13.20	16.73	**9.98**	**13.27**	**17.02**
	9.59	13.12	17.48							12.31	15.98	18.06	**12.04**	**16.18**	**18.66**

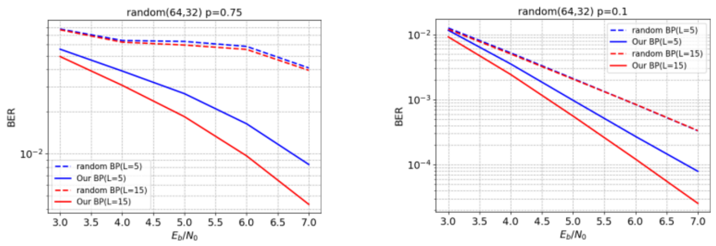


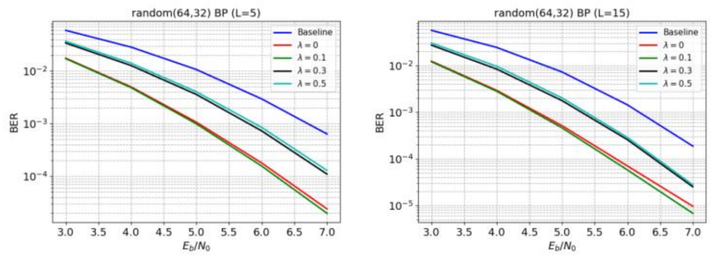
Analysis

- **Impact of initialization**
    - Initialization has a **large impact** the performance (local convergence)



- **Learned regularization**
    - Regularization enforces structure and can improve performance. **Sparsity** constraint has **low influence** since it coincides with *BP's inductive bias*.





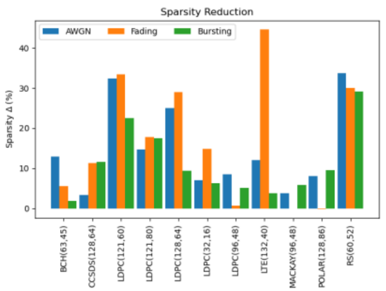
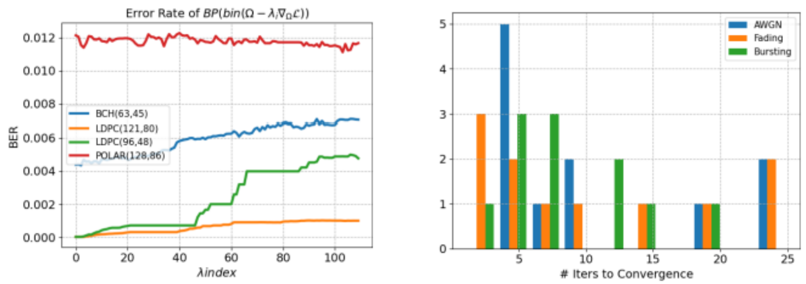
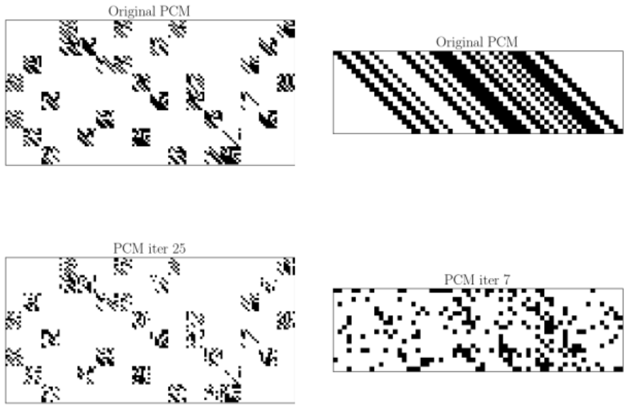


Figure 5: Sparsity reduction of the proposed codes.

- **Convergence**
    - Optimal step sizes are close to the *working point vicinity*.





Links

- **Papers:**
  - https://yonilc.github.io/

- **Code:**
  - https://github.com/yoniLc

- **Correspondence:**
  - choukroun.yoni@gmail.com