Cyclically Equivariant Neural Decoders for Cyclic Codes

Xiangyu Chen [1] **Min Ye** [1]

Abstract

Neural decoders were introduced as a generalization of the classic Belief Propagation (BP) decoding algorithms, where the Trellis graph in the BP algorithm is viewed as a neural network, and the weights in the Trellis graph are optimized by training the neural network. In this work, we propose a novel neural decoder for cyclic codes by exploiting their cyclically invariant property. More precisely, we impose a shift invariant structure on the weights of our neural decoder so that any cyclic shift of inputs results in the same cyclic shift of outputs. Extensive simulations with BCH codes and punctured Reed-Muller (RM) codes show that our new decoder consistently outperforms previous neural decoders when decoding cyclic codes. Finally, we propose a list decoding procedure that can significantly reduce the decoding error probability for BCH codes and punctured RM codes. For certain high-rate codes, the gap between our list decoder and the Maximum Likelihood decoder is less than 0.1dB. Code available at github.com/cyclicallyneuraldecoder

1. Introduction

In recent years, machine learning methods have been successfully applied to the area of decoding error-correcting codes. The usage of neural networks (Nachmani et al., 2016; Gruber et al., 2017; Cammerer et al., 2017; Nachmani et al., 2018; Kim et al., 2018a;b; Vasić et al., 2018; Teng et al., 2019; Buchberger et al., 2020), autoencoders (Jiang et al., 2019), graph neural networks (Nachmani & Wolf, 2019) and reinforcement learning (Carpi et al., 2019; Habib et al., 2020) have demonstrated improvements over classical algorithms in decoding various families of error-correcting codes with short to moderate block length, including BCH codes, polar codes, LDPC codes and Reed-Muller (RM) codes.

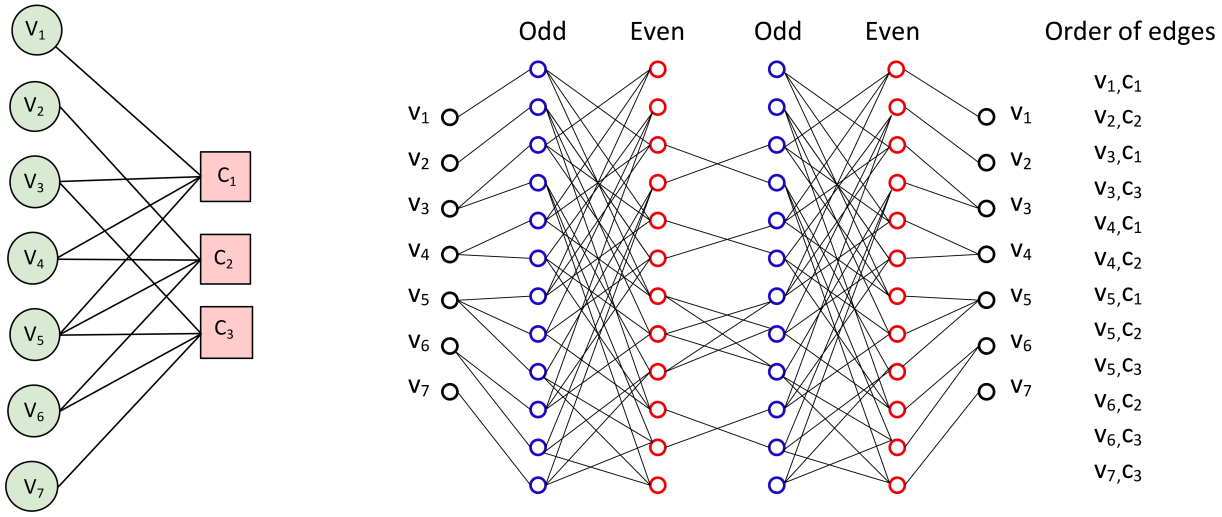
In particular, one line of research pioneered by (Nachmani et al., 2016; 2018) introduced neural decoders as a generalization of the classic Belief Propagation (BP) decoding algorithm. More precisely, the Trellis graph in the BP algorithm is viewed as a fully connected neural network (Nachmani et al., 2016), and the weights in the Trellis graph are optimized by training the neural network. Later, (Nachmani et al., 2018) suggested to replace the fully connected neural networks with recurrent neural networks (RNNs). More recently, the tools of graph neural networks were also introduced to this setting (Nachmani & Wolf, 2019). While these neural BP decoders improve upon the vanilla BP decoder for a wide range of error-correcting codes, their designs rarely utilize the algebraic properties of any particular code family. On the other hand, certain algebraic properties of some code families have proven to be the key to their good performance in error correction, e.g., the recursive structure of polar codes (Arikan, 2009) and the doubly transitive property of BCH codes and RM codes (Kudekar et al., 2017).

In this paper we design a novel neural decoder for an important class of codes called cyclic codes, including two extensively studied and widely applied code families—BCH codes and punctured RM codes. As suggested by their name, cyclic codes are invariant to cyclic shifts, and this property is fully exploited in the design of our new decoder. Inspired by the fact that the Maximum Likelihood (ML) decoder of any cyclic code is equivariant to cyclic shifts, we impose a shift invariant structure on the weights of our neural decoder so that it shares the equivariant property of the ML decoder, i.e., any cyclic shift of inputs results in the same cyclic shift of the decoding outputs.

We carry out extensive simulations to test the performance of our new decoder on BCH codes and punctured RM codes with short to moderate code length. Simulation results show that our decoder consistently outperforms previous neural decoders. In particular, we observe a consistent 0.7dB improvement over the neural decoder proposed in (Nachmani et al., 2018) for various choices of code parameters. Our decoder also demonstrates a 0.3dB improvement over the hyper-graph-network decoder proposed in (Nachmani & Wolf, 2019) with 300 times smaller training time. As a con-

[1]Data Science and Information Technology Research Center, Tsinghua-Berkeley Shenzhen Institute, Tsinghua Shenzhen International Graduate School, Shenzhen, China. Correspondence to: Min Ye <yeemmi@gmail.com>.

Proceedings of the 38^{th} International Conference on Machine Learning, PMLR 139, 2021.



(a) (b)

Figure 1. (a) The Tanner graph corresponding to the parity matrix in (1). The variable node v_j corresponds to the jth column of the matrix, and the check node c_i corresponds to the ith row. (b) The corresponding Trellis graph with two iterations. The 12 nodes in each of the 4 middle layers(columns) represent the 12 edges in the Tanner graph, whose order is listed in the rightmost column. Node e in the sth middle layer is connected to node e' in the $(s-1)$th layer if $x^{[s-1]}(e')$ is involved in the calculation of $x^{[s]}(e)$; see (2)–(4).

crete example, for BCH codes with length 63 and dimension 45, it only takes 10 minutes to train our decoder while the training of the hyper-graph-network decoder (Nachmani & Wolf, 2019) takes more than 2 days on the same platform.

Finally, we propose a list decoding procedure that can significantly reduce the decoding error probability for BCH codes and punctured RM codes. Our list decoding procedure exploits the rich automorphism groups of extended BCH codes and RM codes. More precisely, extended BCH codes and RM codes are obtained by adding an overall parity bit to the BCH codes and punctured RM codes, respectively, and these two extended code families are invariant to a large automorphism group of permutations. In order to make use of this property, we add a dummy symbol to the received noisy codeword and then apply the permutations in the automorphism group to the extended noisy codeword. For each permutation, our neural decoder gives us an intermediate decoding result, and the final decoding result of the list decoding procedure is given by a ML decoding among all the intermediate decoding results. Extensive simulations show that the list decoding method provides up to 3dB gain. As a final remark, we note that this list decoding method can be coupled with any decoding algorithm for BCH codes and punctured RM codes, not just our neural decoder.

To conclude this section, we summarize the **main contributions** of this paper:

- We propose a novel neural decoder that is provably equivariant to cyclic shifts.
- Extensive simulations with BCH codes and punctured RM codes show that our neural decoder consistently improves upon (Nachmani et al., 2018) by 0.7dB, and it also improves upon the hyper-graph-network decoder (Nachmani & Wolf, 2019) by 0.3dB with 300 times smaller training time.
- We propose a list decoding procedure that provides up to 3dB gain for BCH codes and punctured RM codes. For certain high-rate codes, our list decoder with list size $n+1$ has almost the same performance as the ML decoder, where n is the code length.

2. Background and Previous Neural Decoders

A linear code with code length n and code dimension k is a k-dimensional subspace of $\mathbb{F}_2^n$, where $\mathbb{F}_2 = \{0, 1\}$ is the binary field. It can be defined in two equivalent ways—either by a binary generator matrix G of size $k \times n$ or by a binary parity check matrix H of size $(n-k) \times n$.

Each parity check matrix H entails a Tanner graph, which is a bipartite graph with n variable nodes labelled as $v_1, v_2, \ldots, v_n$ on the left side and $n-k$ check nodes labelled as $c_1, c_2, \ldots, c_{n-k}$ on the right side. An edge is connected between v_j and c_i in the Tanner graph if and only if $H_{ij} = 1$. As a concrete example, a parity check matrix of the $(n = 7, k = 4)$ Hamming code is

$$\begin{matrix} 1 & 0 & 1 & 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 & 1 \end{matrix}, \tag{1}$$

and the corresponding Tanner graph is given in Fig. 1a. In

BP algorithms, messages propagate back and forth through the edges in the Tanner graph, and this message passing procedure is best depicted by the Trellis graph, obtained from unrolling the Tanner graph; see Fig. 1b. A Trellis graph with t iterations of message passing consists of one input layer, one output layer, and $2t$ middle layers. Both the input and output layers consist of n variable nodes while the nodes in each middle layer correspond to the edges in the Tanner graph.

In BP algorithms, messages propagate through the Trellis graph from left to right. The n variable nodes in the input layer hold the log likelihood ratios (LLR) of the n input bits:

$$L_j = \log \frac{\mathbb{P}(y_j|C_j = 0)}{\mathbb{P}(y_j|C_j = 1)} \quad \text{for } j \in [n],$$

where $(C_1, \dots, C_n)$ is a randomly chosen codeword, and $(y_1, \dots, y_n)$ is the channel output after transmitting $(C_1, \dots, C_n)$ through n independent copies of some noisy channel. BP algorithm aims to recover the codeword from the channel output, or equivalently, from the LLRs. Let E be the set consisting of all the edges in the Tanner graph, and we use $e = (v_j, c_i)$ and $e = (c_i, v_j)$ interchangeably to denote the same edge. Let the vector $x^{[s]} = (x^{[s]}(e), e \in E)$ be the message vector held by the sth middle layer in the Trellis graph. Suppose that the BP algorithm has t iterations. For each $s = 1, 2, \dots, 2t$, the message vector $x^{[s]}$ is calculated recursively from $x^{[s-1]}$ and the LLR vector $(L_1, \dots, L_n)$, where the initialization $x^{[0]}$ is the all zero vector. More precisely, for odd s and an edge $e = (c_i, v_j) \in E$, the message $x^{[s]}(e)$ is given by

$$\begin{aligned} &x^{[s]}(e) = x^{[s]}((c_i, v_j)) \\ &= \tanh\Big(\frac{1}{2}\Big(L_j + \sum_{e' \in N(v_j)\setminus\{e\}} x^{[s-1]}(e')\Big)\Big), \end{aligned} \tag{2}$$

where $N(v_j) \subseteq E$ is the set of all the edges containing v_j as an endpoint in the Tanner graph. For even s and an edge $e = (c_i, v_j) \in E$, the message $x^{[s]}(e)$ is given by

$$x^{[s]}(e) = 2\tanh^{-1}\Big(\prod_{e' \in N(c_i)\setminus\{e\}} x^{[s-1]}(e')\Big), \tag{3}$$

where $N(c_i) \subseteq E$ is the set of all the edges containing c_i as an endpoint in the Tanner graph. Finally, the output of the classic BP algorithm is

$$o_j = L_j + \sum_{e \in N(v_j)} x^{[2t]}(e) \quad \text{for } j \in [n]. \tag{4}$$

In (Nachmani et al., 2016; 2018), a set of learnable weights are added into the calculations of odd and output layers while the calculations of even layers, i.e., (3), remain unchanged. More precisely, (2) and (4) are replaced by

$$\begin{aligned} x^{[s]}(e) = x^{[s]}((c_i, v_j)) = \tanh\Big(\frac{1}{2}\Big(w^{[s]}(j, e)L_j \\ + \sum_{e' \in N(v_j)\setminus\{e\}} w^{[s]}(e', e)\, x^{[s-1]}(e')\Big)\Big), \end{aligned} \tag{5}$$

$$\text{and } o_j = L_j + \sum_{e \in N(v_j)} w^{\text{out}}(e, j)\, x^{[2t]}(e), \tag{6}$$

respectively.

3. Cyclic Codes and Our New Decoder

A cyclic code $\mathcal{C}$ is a linear code satisfying the following property: If $(C_1, C_2, \dots, C_n) \in \mathcal{C}$ is a codeword, then its cyclic shift $(C_n, C_1, C_2, \dots, C_{n-1})$ is also a codeword in $\mathcal{C}$. This implies that cyclic codes are invariant under all cyclic shifts. More precisely, for $b \in [n]$, let us define the cyclic shift π_b on the set $[n]$ as

$$\begin{aligned} &\pi_b(i) = i + b - 1 \quad \text{for } 1 \le i \le n - b + 1, \\ &\pi_b(i) = i + b - 1 - n \quad \text{for } n - b + 2 \le i \le n. \end{aligned} \tag{7}$$

Then for every cyclic code $\mathcal{C}$ and every $b \in [n]$,

$$\begin{aligned} &\{(C_{\pi_b(1)}, C_{\pi_b(2)}, \dots, C_{\pi_b(n)}) : (C_1, C_2, \dots, C_n) \in \mathcal{C}\} \\ &= \mathcal{C}. \end{aligned} \tag{8}$$

Now consider the Maximum Likelihood (ML) decoder of a cyclic code $\mathcal{C}$. By definition, given the channel outputs $y = (y_1, \dots, y_n)$, the ML decoding result $\hat{C}^{\text{ML}}(y) = (\hat{C}_1, \dots, \hat{C}_n)$ satisfies that

$$\prod_{j=1}^{n} \mathbb{P}(y_j|\hat{C}_j) \ge \prod_{j=1}^{n} \mathbb{P}(y_j|C_j) \text{ for all } (C_1, \dots, C_n) \in \mathcal{C}.$$

Therefore, for every $b \in [n]$,

$$\begin{aligned} \prod_{j=1}^{n} \mathbb{P}(y_{\pi_b(j)}|\hat{C}_{\pi_b(j)}) \ge \prod_{j=1}^{n} \mathbb{P}(y_{\pi_b(j)}|C_{\pi_b(j)}) \\ \text{for all } (C_1, \dots, C_n) \in \mathcal{C}. \end{aligned}$$

The cyclically invariant property (8) further implies that

$$\begin{aligned} \prod_{j=1}^{n} \mathbb{P}(y_{\pi_b(j)}|\hat{C}_{\pi_b(j)}) \ge \prod_{j=1}^{n} \mathbb{P}(y_{\pi_b(j)}|C_j) \\ \text{for all } (C_1, \dots, C_n) \in \mathcal{C}. \end{aligned}$$

Thus the ML decoding result of $(y_{\pi_b(1)}, \dots, y_{\pi_b(n)})$ is

$$\hat{C}^{\text{ML}}\big((y_{\pi_b(1)}, \dots, y_{\pi_b(n)})\big) = \big(\hat{C}_{\pi_b(1)}, \dots, \hat{C}_{\pi_b(n)}\big)$$

for all $b \in [n]$. This proves that **the ML decoder of a cyclic code is equivariant to all cyclic shifts**.

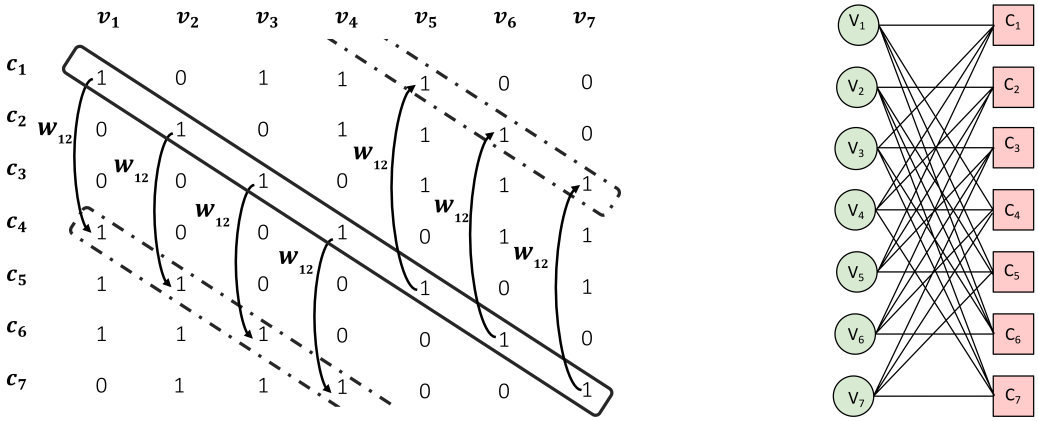


Figure 2. The parity check matrix of the $(7, 4)$ Hamming code used in our decoder (left) and the corresponding Tanner graph (right). Each column of the matrix is obtained by one cyclic downward shift of its previous column. The weight $w^{[s]}((c_{\pi_j(1)}, v_j), (c_{\pi_j(4)}, v_j))$ in (5) can be viewed as a measure of "importance" of $x^{[s-1]}((c_{\pi_j(1)}, v_j))$ in the calculation of $x^{[s]}((c_{\pi_j(4)}, v_j))$, where π_j is defined in (7). The cyclically invariant structure implies that the relation between the edges $(c_{\pi_j(1)}, v_j)$ and $(c_{\pi_j(4)}, v_j)$ is the same for all $j \in [n]$, so in our decoder we set $w^{[s]}((c_{\pi_j(1)}, v_j), (c_{\pi_j(4)}, v_j)) = w_{1,2}^{(s)}$ for all $j \in [n]$. The superscript of $w_{1,2}^{(s)}$ is omitted in the figure.

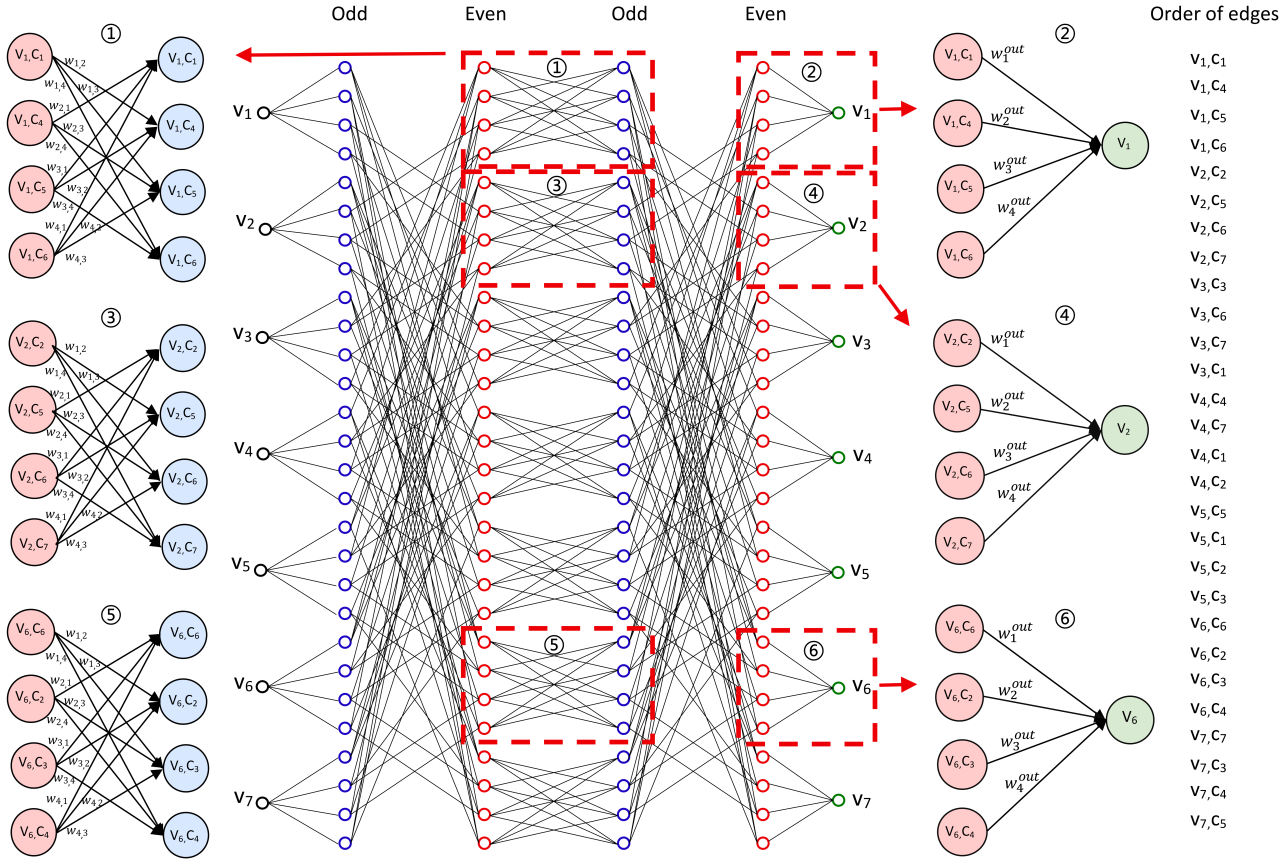


Figure 3. The Trellis graph corresponding to the matrix in Fig. 2. The order of edges in the middle layers is listed in the rightmost column. Due to the cyclically invariant property, each $\left(x^{[s]}((c_{\pi_j(1)}, v_j)), x^{[s]}((c_{\pi_j(4)}, v_j)), x^{[s]}((c_{\pi_j(5)}, v_j)), x^{[s]}((c_{\pi_j(6)}, v_j))\right)$ is the **same** linear function of $\left(x^{[s-1]}((c_{\pi_j(1)}, v_j)), x^{[s-1]}((c_{\pi_j(4)}, v_j)), x^{[s-1]}((c_{\pi_j(5)}, v_j)), x^{[s-1]}((c_{\pi_j(6)}, v_j))\right)$ and L_j for all $j \in [7]$, meaning that the **same** set of weights is shared among all $j \in [7]$ in (5). This is illustrated by the small figures ①,③,⑤. (Weight of L_j is omitted.) Similarly, in the output layer, the **same** set of weights $w_1^{\text{out}}, w_2^{\text{out}}, w_3^{\text{out}}, w_4^{\text{out}}$ is shared among all the 7 linear mappings from $\left(x^{[2t]}((c_{\pi_j(1)}, v_j)), x^{[2t]}((c_{\pi_j(4)}, v_j)), x^{[2t]}((c_{\pi_j(5)}, v_j)), x^{[2t]}((c_{\pi_j(6)}, v_j))\right)$ to o_j for all $j \in [7]$, as illustrated by figures ②,④,⑥.

In light of this, we use a parity check matrix of size $n \times n$ instead of $(n-k) \times n$, and we impose a shift-invariant structure on the weights of our new decoder so that it shares the equivariant property of the ML decoder.

The first new ingredient of our decoder is the choice of the parity check matrix. A careful inspection of (1) indicates that the rows of this parity check matrix are cyclic shifts of each other. This is in fact not a coincidence. According to Chapter 7.4 of (MacWilliams & Sloane, 1977), every cyclic code $\mathcal{C}$ with length n and dimension k possesses an $(n-k) \times n$ parity check matrix of the form

$$\begin{array}{ccccccccc} h_k & \dots & h_2 & h_1 & h_0 & 0 & 0 & \dots & 0 \\ 0 & h_k & \dots & h_2 & h_1 & h_0 & 0 & \dots & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & \dots & 0 & h_k & \dots & h_2 & h_1 & h_0 \end{array}, \tag{9}$$

where $(h_k, \dots, h_1, h_0)$ is a binary vector of length $k+1$. For the $(n=7, k=4)$ Hamming code, the parity check matrix in (1) corresponds to $(h_4, h_3, h_2, h_1, h_0) = (1, 0, 1, 1, 1)$. It is well known that the dual code of a cyclic code is also cyclic; see Chapter 7.4 of (MacWilliams & Sloane, 1977). As a consequence, all the cyclic shifts of the rows in (9) are parity checks of the code $\mathcal{C}$. In our new decoder, we use an $n \times n$ parity check matrix consisting of all the n cyclic shifts of the first row of matrix (9), as opposed to matrix (9) itself, which only contains the first $(n-k)$ cyclic shifts of its first row. Note that matrix (9) is widely used in previous neural decoders when decoding cyclic codes such as BCH codes (Nachmani et al., 2016; 2018).

As a concrete example, for the $(n=7, k=4)$ Hamming code, the matrix in Fig. 2 is used in our new decoder. Each row in this matrix is obtained by one cyclic right shift of its previous row. As a consequence, each column is also obtained by one cyclic downward shift of its previous column. According to (5), only the edges sharing the same variable node are involved in the calculations of odd layers. More specifically, for the Tanner graph in Fig. 2 and for odd s, the calculations of $\big(x^{[s]}((c_1,v_1)), x^{[s]}((c_4,v_1)), x^{[s]}((c_5,v_1)), x^{[s]}((c_6,v_1))\big)$ only involve $\big(x^{[s-1]}((c_1,v_1)), x^{[s-1]}((c_4,v_1)), x^{[s-1]}((c_5,v_1)), x^{[s-1]}((c_6,v_1))\big)$ and L_1; calculations of $\big(x^{[s]}((c_2,v_2)), x^{[s]}((c_5,v_2)), x^{[s]}((c_6,v_2)), x^{[s]}((c_7,v_2))\big)$ only involve $\big(x^{[s-1]}((c_2,v_2)), x^{[s-1]}((c_5,v_2)), x^{[s-1]}((c_6,v_2)), x^{[s-1]}((c_7,v_2))\big)$ and L_2; and so on. The weight $w^{[s]}((c_1,v_1),(c_4,v_1))$ in (5) can be viewed as a measure of "importance" of $x^{[s-1]}((c_1,v_1))$ in the calculation of $x^{[s]}((c_4,v_1))$. Similarly, $w^{[s]}((c_2,v_2),(c_5,v_2))$ measures the "importance" of $x^{[s-1]}((c_2,v_2))$ in the calculation of $x^{[s]}((c_5,v_2))$. An important observation is that due to the cyclically invariant structure of the code and the parity matrix, the relation between the edges (c_1,v_1) and (c_4,v_1) is the same as the relation between (c_2,v_2) and (c_5,v_2). Therefore, in our decoder, we enforce the weights $w^{[s]}((c_1,v_1),(c_4,v_1))$ and $w^{[s]}((c_2,v_2),(c_5,v_2))$ to take the same value. More generally, we set

$$w^{[s]}((c_{\pi_j(1)},v_j),(c_{\pi_j(4)},v_j)) = w^{[s]}((c_1,v_1),(c_4,v_1)) \tag{10}$$

for all $j \in [n]$, as illustrated in Fig. 2, where the cyclic shift π_j is defined in (7). In the calculations of $\big(x^{[s]}((c_1,v_1)), x^{[s]}((c_4,v_1)), x^{[s]}((c_5,v_1)), x^{[s]}((c_6,v_1))\big)$, there are $4 \times 3 = 12$ weights multiplied with $\big(x^{[s-1]}((c_1,v_1)), x^{[s-1]}((c_4,v_1)), x^{[s-1]}((c_5,v_1)), x^{[s-1]}((c_6,v_1))\big)$. As a natural generalization of (10), in our decoder these 12 weights are shared in the calculations of $\big(x^{[s]}((c_{\pi_j(1)},v_j)), x^{[s]}((c_{\pi_j(4)},v_j)), x^{[s]}((c_{\pi_j(5)},v_j)), x^{[s]}((c_{\pi_j(6)},v_j))\big)$ for every $j \in [n]$. In other words, we set

$$w^{[s]}((c_{\pi_j(i_1)},v_j),(c_{\pi_j(i_2)},v_j)) = w^{[s]}((c_{i_1},v_1),(c_{i_2},v_1))$$

for all $j \in [n]$ and all $i_1, i_2 \in \{1,4,5,6\}$ such that $i_1 \neq i_2$; see Fig. 3 for an illustration.

In addition to the changes of $w^{[s]}(e', e)$ in (5), we also make changes to $w^{[s]}(j, e)$, the weight of L_j. Again by the cyclically invariant structure of the code and the matrix, the role of L_1 in the calculation of $x^{[s]}((c_1,v_1))$ is the same as the role of L_2 in the calculation of $x^{[s]}((c_2,v_2))$. Therefore, in our decoder, we enforce the weights $w^{[s]}(1,(c_1,v_1))$ and $w^{[s]}(2,(c_2,v_2))$ to take the same value. More generally, we set $w^{[s]}(j,(c_{\pi_j(i)},v_j)) = w^{[s]}(1,(c_i,v_1))$ for all $j \in [n]$ and all $i \in \{1,4,5,6\}$. Finally, in the calculations of the output layer, we also impose a similar shift-invariant structure on the weights $w^{\text{out}}(e,j)$ in (6). More precisely, we set $w^{\text{out}}((c_{\pi_j(i)},v_j),j) = w^{\text{out}}((c_i,v_1),1)$ for all $j \in [n]$ and all $i \in \{1,4,5,6\}$. see Fig. 3 for an illustration.

Now we have finished the description of our new decoder for the special case of $(7,4)$ Hamming code. For a general cyclic code, the first step of our decoding algorithm is to extend the $(n-k) \times n$ parity check matrix in (9) to obtain an $n \times n$ parity check matrix H of the form in Fig. 2. Suppose that the number of 1's in each column of H is u, and let $\{i_1, i_2, \dots, i_u\} \subseteq [n]$ be the set satisfying $H_{i_1,1} = H_{i_2,1} = \cdots = H_{i_u,1} = 1$. Then $(c_{\pi_j(i_1)},v_j), (c_{\pi_j(i_2)},v_j), \dots, (c_{\pi_j(i_u)},v_j)$ are the u edges that contain v_j as an endpoint in the Tanner graph. In the calculations of each odd layer, we use the following u^2 weights: $\{w^{[s]}_{b,b'} : b, b' \in [u], b \neq b'\}$ and $\{w^{[s]}_b : b \in [u]\}$. More precisely, for odd s and an edge $e = (c_{\pi_j(i_b)}, v_j)$ with $b \in [u]$, the message $x^{[s]}(e)$ is given by

$$x^{[s]}(e) = x^{[s]}((c_{\pi_j(i_b)},v_j)) = \tanh\Big(\frac{1}{2}\Big(w^{[s]}_b L_j + \sum_{b' \in [u]\setminus\{b\}} w^{[s]}_{b',b}\, x^{[s-1]}((c_{\pi_j(i_{b'})},v_j))\Big)\Big). \tag{11}$$

In the calculations of the output layer, we use the following u weights: $\{w_b^{\text{out}} : b \in [u]\}$. The jth output is given by

$$o_j = L_j + \sum_{b\in[u]} w_b^{\text{out}}\, x^{[2t]}((c_{\pi_j(i_b)}, v_j)) \qquad (12)$$

for $j \in [n]$, where t is the total number of iterations. As for the calculations of even layers, we still use the same formula (3) as in the vanilla BP algorithm. In the supplementary material, we prove that **our new decoder is equivariant to all cyclic shifts**.

4. List Decoding Procedure

BCH codes and punctured RM codes are two families of extensively studied and widely applied cyclic codes. In this section we propose a list decoding procedure for these two code families. Let $(C_1, C_2, \ldots, C_n)$ be a codeword of an (n, k) BCH code. If we prepend an overall parity bit $C_0 = C_1 + C_2 + \cdots + C_n$ to this codeword (addition is over binary field), then the resulting vector $(C_0, C_1, C_2, \ldots, C_n)$ is a codeword of an $(n+1, k)$ extended BCH code. Similarly, prepending an overall parity bit to an (n, k) punctured RM code results in an $(n + 1, k)$ RM code. Both RM codes and extended BCH codes are invariant to the affine group (Kasami et al., 1967). In the list decoding procedure, we use the permutations from a subset $\mathcal{S}$ of the affine group. The details about how to choose this subset $\mathcal{S}$ are elaborated in the supplementary material. For now we only need to know that the size of $\mathcal{S}$ is equal to the length of the corresponding RM or extended BCH code.

We will describe our list decoding algorithm for BCH codes, and the algorithm for punctured RM codes is exactly the same. Let $(L_1, L_2, \ldots, L_n)$ be the LLR vector of the channel outputs after transmitting a randomly chosen codeword $(C_1, \ldots, C_n)$ from a BCH code with code length n. Then the size of $\mathcal{S}$ is $|\mathcal{S}| = n + 1$. Choose the list size in the list decoding algorithm to be $\ell \le n + 1$, and the algorithm works as follows:

Step 1: Prepend a dummy symbol $L_0 = 0$ to the LLR vector. As mentioned above, $(C_0, C_1, \ldots, C_n)$ is a codeword from an extended BCH code, where C_0 is the overall parity. The dummy symbol $L_0 = 0$ is the LLR of C_0, indicating that the probability of $C_0 = 0$ is the same as the probability of $C_0 = 1$ because we do not have any direct information about C_0.

Step 2: Apply ℓ permutations to $(L_0, L_1, \ldots, L_n)$. We pick ℓ permutations from the set $\mathcal{S}$. Denote them as $\sigma_1, \ldots, \sigma_\ell$. Apply these permutations to $(L_0, L_1, \ldots, L_n)$ and we obtain ℓ vectors $(L_0^{(i)}, L_1^{(i)}, \ldots, L_n^{(i)}) = (L_{\sigma_i(0)}, L_{\sigma_i(1)}, \ldots, L_{\sigma_i(n)})$ for $i \in [\ell]$.

Step 3: Decode $(L_1^{(i)}, L_2^{(i)}, \ldots, L_n^{(i)})$ using our neural decoder. For each $i \in [\ell]$, decode $(L_1^{(i)}, L_2^{(i)}, \ldots, L_n^{(i)})$ using our decoder proposed in Section 3 and denote the result as $(\bar{C}_1^{(i)}, \bar{C}_2^{(i)}, \ldots, \bar{C}_n^{(i)})$.

Step 4: Check whether $(\bar{C}_1^{(i)}, \bar{C}_2^{(i)}, \ldots, \bar{C}_n^{(i)})$ is a codeword or not. Multiply $(\bar{C}_1^{(i)}, \bar{C}_2^{(i)}, \ldots, \bar{C}_n^{(i)})$ with the parity check matrix of the BCH code. If the result is 0 (meaning that this is a codeword) then we do nothing; otherwise we set $(\bar{C}_1^{(i)}, \bar{C}_2^{(i)}, \ldots, \bar{C}_n^{(i)})$ to be the all zero vector.

Step 5: Prepend an overall parity to $(\bar{C}_1^{(i)}, \ldots, \bar{C}_n^{(i)})$. Prepend an overall parity $\bar{C}_0^{(i)} = \bar{C}_1^{(i)} + \bar{C}_2^{(i)} + \cdots + \bar{C}_n^{(i)}$ to obtain $(\bar{C}_0^{(i)}, \bar{C}_1^{(i)}, \ldots, \bar{C}_n^{(i)})$.

Step 6: Apply the inverse permutation σ_i^{-1} to $(\bar{C}_0^{(i)}, \bar{C}_1^{(i)}, \ldots, \bar{C}_n^{(i)})$. Apply the inverse permutation σ_i^{-1} to obtain $(\hat{C}_0^{(i)}, \hat{C}_1^{(i)}, \ldots, \hat{C}_n^{(i)}) = (\bar{C}^{(i)}_{\sigma_i^{-1}(0)}, \bar{C}^{(i)}_{\sigma_i^{-1}(1)}, \ldots, \bar{C}^{(i)}_{\sigma_i^{-1}(n)})$ for all $i \in [\ell]$.

Step 7: ML decoding among the ℓ candidates $(\hat{C}_0^{(i)}, \hat{C}_1^{(i)}, \ldots, \hat{C}_n^{(i)}), i \in [\ell]$. ML decoding amounts to finding $i \in [\ell]$ that minimizes $L_0\hat{C}_0^{(i)} + L_1\hat{C}_1^{(i)} + \cdots + L_n\hat{C}_n^{(i)}$. Denote the minimizer as i^* and set $(\hat{C}_0, \hat{C}_1, \ldots, \hat{C}_n) = (\hat{C}_0^{(i^*)}, \hat{C}_1^{(i^*)}, \ldots, \hat{C}_n^{(i^*)})$.

Step 8: Discard the first bit $\hat{C}_0$. The final decoding result is $(\hat{C}_1, \hat{C}_2, \ldots, \hat{C}_n)$.

Note that in Step 3, our neural decoder can be replaced by any decoder of BCH codes. As a final remark, Step 4 is very important for reducing the decoding error probability because without this step, some non-codeword might be the minimizer in Step 7, which results in a decoding error.

5. Simulation Results

Similarly to the neural decoders proposed in (Nachmani et al., 2016; 2018; Nachmani & Wolf, 2019), it is easy to verify that our new decoder also satisfies the message passing symmetry conditions in Definition 4.81 of (Richardson & Urbanke, 2008). Hence, by Lemma 4.90 of (Richardson & Urbanke, 2008), the decoding error probability is independent of the transmitted codeword. A direct implication is that we can train our decoder solely using the all-zero codeword, instead of using randomly chosen codewords.

In the experiments, we find that a simple **boosting method** can effectively reduce the decoding error probability. More precisely, the output vector of neural BP decoders (both ours and (Nachmani et al., 2016; 2018)) is still an LLR vector (see (6) and (12)), so we can feed this output LLR vector as an input to the neural decoder again, and we can repeat this procedure many times.

We compare the performance of our neural decoder with the feed-forward (FF) neural decoder proposed in Section III of (Nachmani et al., 2018) and the hyper graph decoder in (Nachmani & Wolf, 2019). In (Nachmani et al., 2018),

Table 1. The negative natural logarithm of BER for three SNR values and different decoders. Higher is better. N18 refers to (Nachmani et al., 2018), and B is the number of boosting used in the decoder. For all codes, we train with a batch size of 160 samples, among which we produce 20 samples from each of the following 8 SNR values: 1dB, 2dB, ..., 8dB. We use 10^5 samples for testing.

Decoder	BP			N18, $B=0$			N18, $B=2$			Ours, $B=0$			Ours, $B=2$		
Code/SNR	4	5	6	4	5	6	4	5	6	4	5	6	4	5	6
BCH(63,24)	3.18	4.07	5.18	3.24	4.36	5.80	3.43	4.62	6.16	3.78	5.11	7.07	3.98	5.42	7.34
BCH(63,36)	3.83	4.69	5.91	3.97	5.27	7.05	4.00	5.35	7.43	4.63	6.48	8.86	4.75	6.40	10.02
BCH(63,45)	3.94	4.84	6.30	4.37	5.71	7.45	4.42	5.83	7.50	5.12	6.97	9.46	5.39	7.45	10.45
BCH(127,36)	2.21	2.71	3.52	2.23	2.76	3.72	2.31	3.06	4.38	2.29	2.96	4.24	2.44	3.36	4.98
BCH(127,64)	2.99	3.63	4.33	3.00	3.82	5.01	3.04	3.91	5.29	3.17	4.24	6.05	3.23	4.43	6.42
BCH(127,99)	3.67	4.38	5.87	4.08	5.34	7.12	4.07	5.42	7.29	4.51	6.30	8.86	4.63	6.59	9.61
PRM(63,22)	2.83	3.55	4.46	2.85	3.76	4.98	3.07	3.99	5.33	3.32	4.52	6.23	3.70	5.13	7.16
PRM(63,42)	4.61	6.00	7.79	4.81	6.47	8.87	4.85	6.47	9.11	5.92	8.26	10.85	6.27	8.81	11.55
PRM(127,64)	2.95	3.53	4.12	2.89	3.55	4.90	2.94	3.69	5.00	3.14	4.17	5.93	3.22	4.37	6.35
PRM(127,99)	4.44	5.82	7.65	4.56	6.30	8.35	4.58	6.33	8.43	5.69	8.19	11.15	5.97	8.83	14.33

Table 2. The negative natural logarithm of FER of the list decoding algorithm. >11.5 means FER$<10^{-5}$ since we only use 10^5 samples for testing. For all list sizes, our decoder is boosted 20 times for codes with length 63 and 50 times for codes with length 127.

List size	$\ell=1$			$\ell=2$			$\ell=4$			$\ell=8$			$\ell=n+1$		
Code/SNR	4	5	6	4	5	6	4	5	6	4	5	6	4	5	6
BCH(63,24)	2.51	4.02	6.21	3.30	5.08	7.60	4.40	6.21	>11.5	5.52	9.21	>11.5	9.21	>11.5	>11.5
BCH(63,36)	2.89	4.68	7.49	3.53	5.65	8.87	4.23	7.26	10.13	5.04	8.52	>11.5	9.21	>11.5	>11.5
BCH(63,45)	3.01	5.07	8.15	3.48	5.76	8.42	4.01	6.68	9.90	4.65	7.49	10.41	6.06	9.57	>11.5
BCH(127,36)	0.62	1.50	3.13	0.94	2.12	4.26	1.35	2.92	5.71	1.92	3.95	7.26	5.10	9.21	>11.5
BCH(127,64)	0.79	2.01	4.00	1.10	2.61	5.08	1.51	3.41	6.34	2.02	4.17	7.52	4.66	7.82	>11.5
BCH(127,99)	1.63	3.55	7.13	1.91	4.12	7.82	2.22	4.77	>11.5	2.60	5.45	>11.5	4.29	9.21	>11.5
PRM(63,22)	2.03	3.45	5.43	2.70	4.40	7.13	3.61	5.68	8.11	4.79	7.82	>11.5	9.21	>11.5	>11.5
PRM(63,42)	4.06	6.57	9.90	4.55	7.39	>11.5	5.18	8.62	>11.5	5.91	9.21	>11.5	6.21	9.72	>11.5
PRM(127,64)	0.69	1.83	3.82	0.95	2.38	4.84	1.33	3.17	6.07	1.75	3.95	7.13	4.34	7.13	>11.5
PRM(127,99)	3.13	6.15	>11.5	3.61	6.95	>11.5	4.06	7.82	>11.5	4.41	8.42	>11.5	5.50	>11.5	>11.5

Table 3. Ablation analysis. The numbers are negative natural logarithm of BER. N18 refers to (Nachmani et al., 2018).

Code	BCH(63,24)			BCH(63,36)			BCH(63,45)			PRM(63,22)		
Decoder/SNR	4	5	6	4	5	6	4	5	6	4	5	6
N18, $(n-k)\times n$ parity matrix	3.24	4.36	5.80	3.97	5.27	7.05	4.37	5.71	7.45	2.85	3.76	4.98
N18, randomly extended $n\times n$ matrix	3.27	4.38	5.78	3.97	5.28	7.07	4.49	5.81	7.47	2.90	3.86	5.17
N18, $n\times n$ cyclic matrix	3.67	5.08	6.78	4.42	5.93	7.84	4.87	6.19	7.76	3.15	4.33	5.86
Ours, $n\times n$ cyclic matrix	3.78	5.11	7.07	4.63	6.48	8.86	5.12	6.97	9.46	3.32	4.52	6.23

Table 4. Decoding time of N18 (Nachmani et al., 2018) and our decoder

Code	N18	Ours	N18/Ours
BCH(63,36)	4.09ms	4.68ms	0.87
BCH(63,45)	3.85ms	4.94ms	0.78
BCH(127,64)	8.60ms	8.63ms	0.99
BCH(127,99)	4.05ms	16.7ms	0.24
PRM(127,99)	3.97ms	7.45ms	0.53

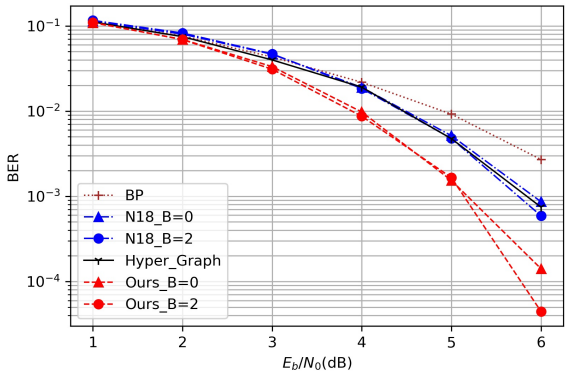
several variations of the FF neural decoder were also proposed, such as RNN and min-sum decoders. Yet in many cases the FF neural decoder gives better performance than its variations in terms of decoding error probability, so we only compare with the FF decoder. For the hyper graph decoder, we are only able to obtain results for BCH$(63, 36)$ and BCH$(63, 45)$ due to the high complexity of its training procedure; see Fig. 4a–4b. We also test the performance of the list decoding algorithm with various list sizes. Following the common practice in the literature, we use Bit Error Rate (BER) to measure the decoding error probability for neural BP algorithms, and we use Frame Error Rate (FER) to measure the decoding error probability for list decoding algorithms. In our simulations, the number of BP iterations is 5 for all the methods.

The simulation results are given in Tables 1–2 and Fig. 4, and explanations of these results are given in the captions. Here we only discuss the comparison between our list decoder and the ML decoder: For certain high-rate codes (rate ≥ 0.7), our list decoder with list size $\ell = n + 1$ provides almost the same performance as the ML decoder. More precisely, simulation results in Fig. 4(d)–(f) show that the gap between our list decoder with $\ell = n + 1$ and the ML decoder is no larger than 0.1dB for these high-rate codes. However, for codes with rate ≤ 0.5, the gap between our decoder with list size $\ell = n + 1$ and the ML lower bound can be larger than 1dB. It remains unclear to us why there is such a distinction between high-rate codes and medium-to-low-rate codes. As a final remark, we note that efficient decoders with almost ML performance were already proposed for low-rate RM codes; see for example (Dumer & Shabunov, 2006; Ye & Abbe, 2020).

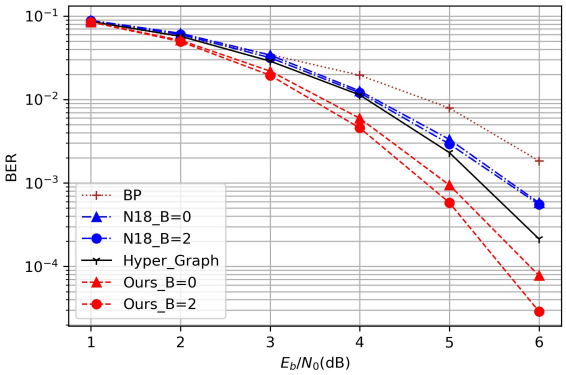
Next we compare the complexity between our decoder and (Nachmani et al., 2018). The memory requirements of our decoder is much smaller than (Nachmani et al., 2018) while the time complexity of our decoder is larger than (Nachmani et al., 2018). In both our decoder and (Nachmani et al., 2018), the memory is used to store the trained weights, so the memory requirement depends on the number of weights used in the decoder. In our decoder, we reuse the same set of weights n times in each odd layer due to the shift-invariant structure. This effectively reduces the number of weights compared to (Nachmani et al., 2018). More precisely, according to the discussion above (11), the number of weights in our decoder is u^2t, where u is the number of 1's in each row of the parity check matrix, and t is the number of iterations in the BP algorithm. A simple analysis shows that the number of weights in (Nachmani et al., 2018) is at least $\frac{(n-k)^2}{n}u^2t$. Therefore, except for extremely high-rate codes where $n-k \leq \sqrt{n}$, the memory requirement of our decoder is typically much smaller than (Nachmani et al., 2018).

As for the time complexity, we note that the number of additions and multiplications are both proportional to the number of edges in the Tanner graph. The number of edges in the Tanner graph of (Nachmani et al., 2018) is $u(n-k)$ while for our decoder this number is un. Therefore, if we only count the number of additions and multiplications, the ratio between (Nachmani et al., 2018) and our decoder is $\frac{n-k}{n}$. Although the actual ratio of running time is not exactly $\frac{n-k}{n}$, the above analysis tells us that the running time ratio between (Nachmani et al., 2018) and our decoder is smaller for high-rate codes and larger for low-rate codes. Table 4 compares the decoding time of the vanilla versions of (Nachmani et al., 2018) and our decoder, i.e., no boosting, no list decoding.

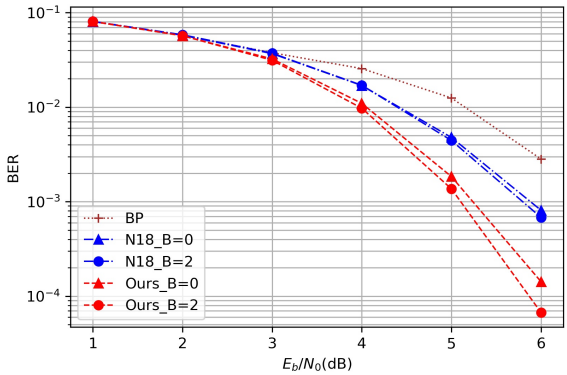
In Tables 1–2 and Fig. 4, we use the $(n-k) \times n$ parity matrix of the form (9) for the FF neural decoder and hyper graph decoder, and we use the $n \times n$ cyclic parity matrix for our decoder. To evaluate the contribution of the $n \times n$ cyclic matrix, we ran an ablation analysis, where we further consider another two cases: (i) we append k random parity rows to matrix (9) so that we obtain an randomly extended $n \times n$ matrix, and we use the FF neural decoder together with this matrix; (ii) we use the FF neural decoder together with the $n \times n$ cyclic matrix. Simulation results in Table 3 show that using randomly extended matrix has very little effect on the BER while using the $n \times n$ cyclic matrix for the FF decoder improves over the $(n-k) \times n$ matrix by 0.45dB, so what matters here is not the matrix size, but the cyclic structure. When both using the $n \times n$ cyclic matrix, our decoder still demonstrates a 0.25dB improvement over the FF decoder.



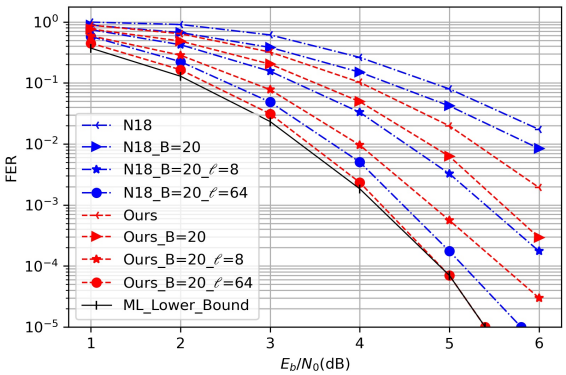
(a) BCH(63,36)



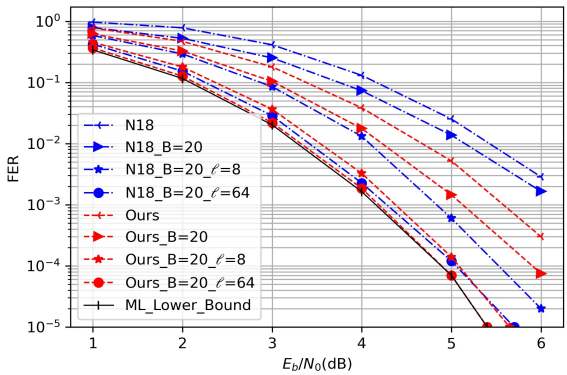
(b) BCH(63,45)



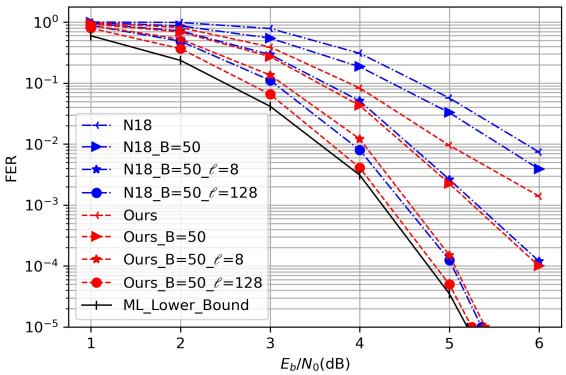
(c) BCH(127,99)



(d) BCH(63,45) List decoding



(e) Punctured RM(63,42) List decoding



(f) Punctured RM(127,99) List decoding

Figure 4. N18 refers to the neural decoder proposed in Section III of (Nachmani et al., 2018); B is the number of boosting used in the decoder; ℓ is the list size. If we do not specify B (respectively, ℓ), it means that $B = 0$ (respectively, $\ell = 1$). For the first three plots, we use BER (the fraction of incorrect bits in the decoding results) to measure the decoding error probability, and for the last three plots, we use FER (the fraction of incorrect codewords in the decoding results) because it involves list decoding. Some additional plots are provided in the supplementary material.

Without list decoding, our neural decoder consistently improves upon (Nachmani et al., 2018) by 0.7dB, and it also improves upon the hyper-graph-network decoder (Nachmani & Wolf, 2019) by 0.3dB. Moreover, the list decoding algorithm provides up to 3dB gain if we set the list size to be $n + 1$. Even for a small list size $\ell = 8$, it also gives $0.7 \sim 0.9$dB gain over the algorithm without list decoding. In (d)–(f), when the list size is $\ell = n + 1$, our list decoding algorithm has almost the same performance as the ML decoder.

References

Arikan, E. Channel polarization: A method for constructing capacity-achieving codes for symmetric binary-input memoryless channels. *IEEE Transactions on Information Theory*, 55(7):3051–3073, 2009.

Buchberger, A., Häger, C., Pfister, H. D., Schmalen, L., and i Amat, A. G. Pruning neural belief propagation decoders. In *2020 IEEE International Symposium on Information Theory (ISIT)*, pp. 338–342. IEEE, 2020.

Cammerer, S., Gruber, T., Hoydis, J., and Ten Brink, S. Scaling deep learning-based decoding of polar codes via partitioning. In *GLOBECOM 2017-2017 IEEE Global Communications Conference*, pp. 1–6. IEEE, 2017.

Carpi, F., Häger, C., Martalò, M., Raheli, R., and Pfister, H. D. Reinforcement learning for channel coding: Learned bit-flipping decoding. In *2019 57th Annual Allerton Conference on Communication, Control, and Computing (Allerton)*, pp. 922–929. IEEE, 2019.

Dumer, I. and Shabunov, K. Soft-decision decoding of Reed-Muller codes: Recursive lists. *IEEE Transactions on Information Theory*, 52(3):1260–1266, 2006. doi: 10.1109/TIT.2005.864443.

Gruber, T., Cammerer, S., Hoydis, J., and ten Brink, S. On deep learning-based channel decoding. In *2017 51st Annual Conference on Information Sciences and Systems (CISS)*, pp. 1–6. IEEE, 2017.

Habib, S., Beemer, A., and Kliewer, J. Learning to decode: Reinforcement learning for decoding of sparse graph-based channel codes. In *Advances in Neural Information Processing Systems*, 2020.

Jiang, Y., Kim, H., Asnani, H., Kannan, S., Oh, S., and Viswanath, P. Turbo autoencoder: Deep learning based channel codes for point-to-point communication channels. In *Advances in neural information processing systems*, pp. 2758–2768, 2019.

Kasami, T., Lin, S., and Peterson, W. W. Some results on cyclic codes which are invariant under the affine group and their applications. *Information and Control*, 11(5-6): 475–496, 1967.

Kim, H., Jiang, Y., Kannan, S., Oh, S., and Viswanath, P. Deepcode: Feedback codes via deep learning. In *Advances in Neural Information Processing Systems*, pp. 9436–9446, 2018a.

Kim, H., Jiang, Y., Rana, R., Kannan, S., Oh, S., and Viswanath, P. Communication algorithms via deep learning. In *6th International Conference on Learning Representations, ICLR 2018*, 2018b.

Kudekar, S., Kumar, S., Mondelli, M., Pfister, H. D., Şaşoğlu, E., and Urbanke, R. L. Reed–muller codes achieve capacity on erasure channels. *IEEE Transactions on Information Theory*, 63(7):4298–4316, 2017.

MacWilliams, F. J. and Sloane, N. J. A. *The theory of error correcting codes*, volume 16. Elsevier, 1977.

Nachmani, E. and Wolf, L. Hyper-graph-network decoders for block codes. In *Advances in Neural Information Processing Systems*, pp. 2329–2339, 2019.

Nachmani, E., Be'ery, Y., and Burshtein, D. Learning to decode linear codes using deep learning. In *2016 54th Annual Allerton Conference on Communication, Control, and Computing (Allerton)*, pp. 341–346, 2016.

Nachmani, E., Marciano, E., Lugosch, L., Gross, W. J., Burshtein, D., and Be'ery, Y. Deep learning methods for improved decoding of linear codes. *IEEE Journal of Selected Topics in Signal Processing*, 12(1):119–131, 2018.

Richardson, T. and Urbanke, R. *Modern coding theory*. Cambridge university press, 2008.

Teng, C.-F., Wu, C.-H. D., Ho, A. K.-S., and Wu, A.-Y. A. Low-complexity recurrent neural network-based polar decoder with weight quantization mechanism. In *ICASSP 2019-2019 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pp. 1413–1417. IEEE, 2019.

Vasić, B., Xiao, X., and Lin, S. Learning to decode LDPC codes with finite-alphabet message passing. In *2018 Information Theory and Applications Workshop (ITA)*, pp. 1–9. IEEE, 2018.

Ye, M. and Abbe, E. Recursive projection-aggregation decoding of Reed-Muller codes. *IEEE Transactions on Information Theory*, 66(8):4948–4965, 2020.