Interpreting Deep-Learned Error-Correcting Codes

N. Devroye[1], N. Mohammadi[1], A. Mulgund[1], H. Naik[1], R. Shekhar[1], Gy. Turán[1,2], Y. Wei[1] and M. Žefran[1]
[1]University of Illinois at Chicago, Chicago, IL, USA
[2]MTA-SZTE Research Group on Artificial Intelligence, ELRN, Szeged, Hungary
{devroye, nmoham24, mulgund2, hnaik2, rshekh3, gyt, ywei30, mzefran}@uic.edu

***Abstract*—Deep learning has been used recently to learn error-correcting encoders and decoders which may improve upon previously known codes in certain regimes. The encoders and decoders are learned "black-boxes", and interpreting their behavior is of interest both for further applications and for incorporating this work into coding theory. Understanding these codes provides a compelling case study for Explainable Artificial Intelligence (XAI): since coding theory is a well-developed and quantitative field, the interpretability problems that arise differ from those traditionally considered. We develop post-hoc interpretability techniques to analyze the deep-learned, autoencoder-based encoders of TurboAE-binary codes, using influence heatmaps, mixed integer linear programming (MILP), Fourier analysis, and property testing. We compare the learned, interpretable encoders combined with BCJR decoders to the original black-box code.**

I. Introduction

Recently, a new path emerged in the development of error correcting codes: "learn" the encoders and/or decoders of error correcting codes using deep learning in an end-to-end fashion [1]–[9]. The results are mixed: while some learned codes significantly outperform known codes, generally on channels for which error-correcting codes have not been studied at length [7], in others [2], [10], the general purpose neural networks-based code designs achieve bit error rates (BERs) comparable to convolutional codes, below those of near-optimal codes. While deep-learned codes are explicitly given by specific neural networks, those can be considered black boxes in the sense that it is not "understood" how/when they perform well or whether/if they relate to known codes.

Deep learning has been enormously successful in improving the prediction capabilities of machine learning (ML) algorithms and extending their applicability to new domains. The interpretability of learned models is a fundamental requirement, important in itself, but also for achieving other objectives, such as trust. It has mostly been discussed for perception tasks such as image understanding and societal applications such as loan approval. There are other domains where it is equally important but has a different nature. In scientific applications, the lack of interpretability of predictions obtained through deep learning hinders the incorporation of new findings into current scientific knowledge [11]. Compared with societal applications, scientists have more precisely defined notions of an interpretation. The question whether it can be achieved in such contexts is also of interest for understanding the nature of scientific research using ML [12].

Thus the study of interpretability of deep-learned error-correcting codes is motivated by information theory and XAI. We present initial approaches for one of the simplest examples of end-to-end learned codes, termed Turbo Autoencoder (TurboAE and TurboAE-binary, focusing on the latter) [9], which are learned using convolutional neural networks (CNN). These are among the first end-to-end learned channel codes with reliability comparable to modern codes such as Turbo codes on Additive White Gaussian Noise (AWGN) channels for moderate block lengths (a few hundred) and signal to noise ratios (SNRs) below around 1dB (low SNR) [9, Figure 1].

We focus on *post-hoc* interpretability, i.e., on interpreting the output of the learned model. By interpretability we mean comprehensibility for the information and communication theory research community, which is consistent with the context-dependence of the notion. Thus Turbo codes and the BCJR decoder are considered interpretable. The iterative Turbo decoder is complex [13] and may be interpreted itself [14], [15], but arguably its opacity is of a different degree than that of a neural network. Even though in this work network structure and size allow brute-force examination, our objective is to develop techniques that may be applicable in general, such as influence heatmaps, mixed integer linear programming (MILP), Fourier analysis, and property testing.

Outline. In Section II we describe the encoder, and define modified Turbo codes used in the interpretation of TurboAE[1]. Sections III, IV and V discuss approximate and exact polynomial representations of the encoding functions and BER performance, coupled with BCJR decoders. Observations on the training dynamics are given in Section VI. Section VII summarizes and formulates open problems. Details may be found in the Appendix of [18][2].

II. TurboAE-binary and modified Turbo codes

The TurboAE encoder architecture [9] resembles a classical rate $1/3$ Turbo code, where the three constituent codes – generally recursive convolutional codes for classical Turbo codes [19], [20] – are replaced by CNN blocks, as in Fig.

This work was supported by NSF under awards 1705058 and 1934915, and the Ministry of Innovation and Technology NRDI Office within the framework of the Artificial Intelligence National Laboratory program (MILAB), Hungary. The authors are in alphabetic order.

[1]In particular the TurboAE and TurboAE-binary models at [16], [17] respectively. Note that these models do not reproduce [9, Fig.1] exactly as they are missing additional fine-tuning we were unable to reproduce.

[2]Code for experiments at https://github.com/tripods-xai/isit-2022

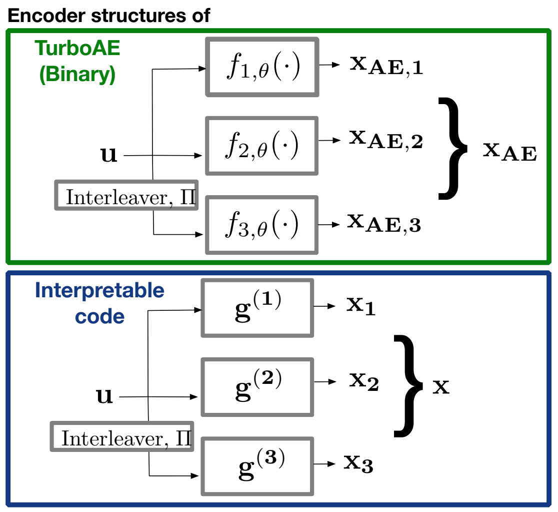


Fig. 1: The rate $R = \frac{1}{3}$ ($\mathbf{u} \in \{0,1\}^{100}, \mathbf{x}_{AE,j} \in \{\pm 1\}^{100}$) TurboAE-binary encoder structure. Functions $f_{j,\theta}(\cdot)$ are the constituent codes implemented as CNNs. Our interpretable codes either find compact Boolean representations for the learned function $f_{j,\theta}$ or approximate it with a modified convolutional code $\mathbf{g}^{(j)}$.

1. The input to the network is a sequence $\mathbf{u}$ of 100 bits, and the output of each block $j \in \{1,2,3\}$ is a sequence $\mathbf{x}_{AE,j}$ of equal length of either real numbers (for Turbo-AE) or $\{\pm 1\}$ (for binarized version Turbo-AE-binary). The focus of this paper is TurboAE-binary [9, Section 3.2], a modification of the TurboAE architecture where the functions $f_{j,\theta}$ are binarized using a sign function and Straight-Through-Estimator [21], [22]. Therefore, in the rest of the paper, and in Fig. 1 we assume that encoding functions $f_{j,\theta} : \{0,1\}^{100} \to \{\pm 1\}$, and we drop the "binary" suffix. Note that TurboAE also has power control modules and zero-padding – we refer the readers to [9] for the details omitted to keep things simple.

A closer look at the CNN blocks reveals that the constituent codes of block j implements a real-valued Boolean function $f_{j,\theta} : \{0,1\}^9 \to \mathbf{R}$ (Turbo-AE) or Boolean function $f_{j,\theta} : \{0,1\}^9 \to \pm 1$ (Turbo-AE-binary) of memory 9 applied to bits $\ell - 4 : \ell + 4$ of u to produce the bit ℓ of $\mathbf{x}_{AE,j}$. The encoder CNN is paired with a decoder CNN function Φ_θ (omitted as we focus on interpreting the encoder only) and the network is trained in an end-to-end fashion to obtain network parameters θ. Details are found in the Appendix of [18].

A. Modified Turbo codes

To develop the interpretation of TurboAE-binary we need to consider *modified Turbo codes*. Here, the constituent codes are *modified convolutional codes*: *nonsystematic*, *nonrecursive*, involve *affine* functions instead of linear ones (as a Boolean function and its complement are equivalent for the neural network so it may converge to either), and may also include a *delay (shift)*, i.e., an output bit may depend on future input bits (up to a lookahead horizon L). Thus, x_ℓ, the ℓth bit of the modified convolutional encoding output, equals:

$$x_\ell = \bigoplus_{i=1}^{M} g_i \, u_{\ell+L-i+1} \oplus g_{M+1}, \tag{1}$$

where $\oplus$ is binary mod 2 addition, $g_i \in \{0,1\}, i \in 1:(M+1)$ are the code parameters, and M is the memory length.

III. Approximate Turbo encoding

We now explore several alternatives for how to find the best affine approximations of the constituent encoders. Schematically, the approximation problem is depicted in Fig. 1.

A. Mixed integer linear programming (MILP)

The first approach is to treat the encoder entirely as a black box, with no assumption on its architecture, and formulate the approximation as a MILP problem. In particular, we do not assume that each encoding block consists of sequentially applying the same function to a sliding window of the input bits as in a convolutional code. Instead, each learned encoder block is considered as a general function $f_{j,\theta}^{block} : \{0,1\}^k \to \pm 1^k$. We look for the best approximation to this arbitrary black box that is of the form of a modified convolutional code. This is a reasonable first approximation given that TurboAE is meant to "mimic" Turbo codes (for which constituent codes are convolutional codes), and since convolutional codes are such a well-studied class of codes with relatively few parameters.

Let Γ_{conv} be the set of all modified convolutional codes. The code $g_{\text{conv}}^{(j)} \in \Gamma_{\text{conv}}$ closest to the TurboAE encoder block $f_{j,\theta}^{block}$ minimizes the expected Hamming distance between the corresponding encoder outputs (codewords) produced by the two codes. Given $g_{\text{conv}} \in \Gamma_{\text{conv}}$, let $\mathbf{x}_{g_{\text{conv}}}(\mathbf{u})$ and $\mathbf{x}_{AE,j}(\mathbf{u})$ be the output strings obtained by encoding the input string $\mathbf{u}$ with g_{conv} (applied repeatedly as in a standard convolutional code) and $f_{j,\theta}^{block}$, respectively. Then:

$$g_{\text{conv}}^{(j)} = \underset{g_{\text{conv}} \in \Gamma_{\text{conv}}}{\arg\min} \, E_{\mathbf{u} \in \{0,1\}^k} [d_H(\mathbf{x}_{g_{\text{conv}}}(\mathbf{u}), \mathbf{x}_{AE,j}(\mathbf{u}))] \tag{2}$$

where $d_H(\mathbf{a}, \mathbf{b}) = \frac{1}{k} \sum_{\ell=1}^{k} a_\ell \oplus b_\ell$ is the Hamming distance. We can parameterize each modified convolutional code $g_{\text{conv}}^{(j)}$ by a binary vector $\mathbf{g}^{(j)}$ of length $M+1$ according to Eq. (1).

A MILP can be obtained from (2) using a reformulation of binary arithmetic (see, e.g., [23]); linearity follows from the linearity of (1) in parameters g_i (and could be extended to non-linear functions in input $\mathbf{u}$) and is readily solved using available solvers; we use the Gurobi solver [24]. While it may not be true in general, in our case Gurobi's solution is optimal as confirmed by brute-force search. The expected value in (2) is approximated through random sampling. For sufficiently large shift L and memory length M, the optimal generators are $\mathbf{g}^{(1)} = 111111$, $\mathbf{g}^{(2)} = 101110$, and $\mathbf{g}^{(3)} = 111101$, where the last bit is the parity bit. These produce encoded bits which differ from those of TurboAE-binary for about 10%, 2% and 25% on average respectively, including edge effects. Multiple solutions for block 3 exist and are related to the Fourier coefficients (see Section VI, Appendix of [18]).

B. Influence and Fourier representation

In this section we consider another approach that takes some information available about the network – that it is a CNN – into account. We use the notion of the influence of a

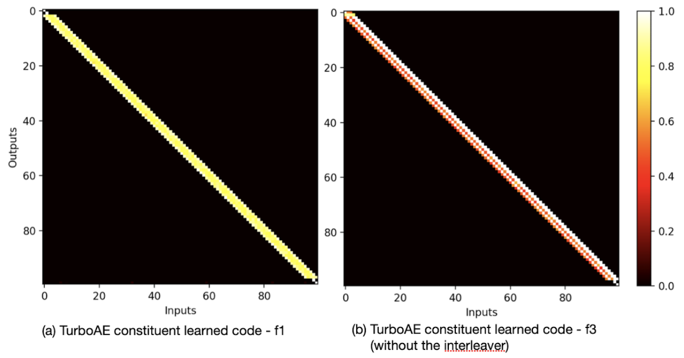


Fig. 2: TurboAE Constituent Code Influence heatmaps of (a) Block 1, (b) Block 3 without the interleaver.

variable [25], which is a natural importance measure in our context (other measures used in XAI are described in [26]).

The *influence* of the input variable x_i for a single-output Boolean function $f : \{0,1\}^k \to \{0,1\}$ is defined as

$$\operatorname*{Inf}_i(f) = \frac{1}{2^k} \sum_{x \in \{0,1\}^k} |f(x) - f(x^{(i)})| = Pr(f(x) \neq f(x^{(i)}), \tag{3}$$

where $x^{(i)}$ is x with the ith coordinate flipped. The influence of a variable is 0 iff the function does not depend on that variable. For an affine function $a \oplus \bigoplus_{i \in I} x_i$, with $I \subseteq \{1, \ldots, n\}$ the influence of variable x_i is 1 if $i \in I$ and 0 otherwise. For an (n,k) code viewed as a multi-output Boolean function $f : \{0,1\}^k \to \{0,1\}^n$ (or $\{\pm 1\}^n$) the matrix of influences $\mathrm{Inf}_i(f)$ can be visualized as a heatmap.

The heatmap for a nonrecursive convolutional code shows a staircase pattern (which gets shuffled if interleaving is applied). Influence can be computed exhaustively, or estimated by random sampling. Influences for the encoder functions $f_{1,\theta}$ and $f_{3,\theta}$ for TurboAE are shown in Fig. 2; $f_{2,\theta}$'s heatmap is that of a parity. For ℓth output, inputs only in window $\ell - 2 : \ell + 2$ have non-zero influences (for each block). The architecture would allow for a non-zero influence window of length 9, so it is interesting that only a length 5 emerged from training. Consistent with the structure of a CNN, the influence pattern is the same for outputs $3 : 98$ of each block.

The CNN architecture also implies that the output bits in a block compute the *same* function of their input bits. This function can be studied, in particular for finding a best affine approximation, using Fourier analysis. Switching to the domain $\{1, -1\}$, Boolean functions have a unique *Fourier representation* as a multilinear polynomial

$$f(x) = \sum_{S \subseteq [n]} \hat{f}(S) \chi_S,$$

where $\chi_S = \prod_{i \in S} x_i$ [25]. The Fourier coefficients are $\hat{f}(S) = \langle f, \chi_S \rangle$, for inner product $\langle f, g \rangle = E_x(f(x)g(x))$. Let $d(f,g) = Pr(f(x) \neq g(x))$ be the distance of f and g. Then $\langle f, \chi_S \rangle = 1 - 2d(f, \chi_S)$, and so the best parity approximation of f corresponds to the largest Fourier coefficient.

The Fourier coefficients of the three functions computed by the three blocks of TurboAE-binary are shown in Figure 4(c) and are consistent with those obtained in the previous section. The multiple optimal approximations (4 large Fourier coefficients) in block 3 are also visible. The computation is done by brute force based on the influence information providing the number of relevant variables, or memory size.

The Goldreich-Levin algorithm [27] is a randomized learning algorithm which computes large Fourier coefficients with high probability in polynomial time. It requires query access to the function, which is available in our setup. This algorithm could be used without any information about memory size.

C. Property testing

We now consider another approach to study affine approximations. An affine function is either linear or its complement is, so we can restrict to linearity. In the framework of *property testing* [28], one must decide if an unknown black-box function has a property. The black box is queried by an input, and the function value at that input is returned. A notion of *distance* of a function from the property is assumed. For a given ϵ, the function is accepted with probability 1 if it has the property, and is rejected with probability at least 2/3 if its distance from the property is at least ϵ. A testing algorithm is *tolerant* [29] if for some $\epsilon' < \epsilon$ functions having distance at most ϵ' from the property are accepted with probability at least 2/3 as well.

We consider property testing for multi-output Boolean functions $f = (f_1, \ldots, f_n) : \{0,1\}^n \to \{0,1\}^m$. The distance of two such functions f, g is $d(f,g) = Pr_{i,x}(f_i(x) \neq g_i(x))$. The distance of f from a property is the minimum of $d(f,g)$ over functions g having the property.

A *multi-parity* function is of the form $g = (h, \ldots, h) : \{0,1\}^n \to \{0,1\}^n$, where h is a parity function. For $x = (x_1, \ldots, x_n)$ let $s(x) = (x_2, \ldots, x_n, x_1)$ be the cyclic shift of x. A *cyclically shifted multi-parity (CSMP)* function is of the form $f(x) = (h(x), h(s(x)), \ldots, h(s^{(n-1)}(x)) : \{0,1\}^n \to \{0,1\}^n$, where h is a parity. A CSMP function is similar to a convolutional code with the exception of the wraparound.

Theorem 1. *There is a tolerant testing algorithm (with $\epsilon' = \epsilon/18$) for CSMP using $O(1/\epsilon)$ queries.*

The proof is given in the Appendix of [18]. Testing a single output Boolean function f for being a parity function is based on testing $f(x) \oplus f(y) = f(x \oplus y)$ for randomly chosen x and y, and repeating this test $O(1/\epsilon)$ times [30]. In the multi-input case one can select random vectors x, y *and random indices* i, j, k, and test $f_i(x) \oplus f_j(y) = f_k(x \oplus y)$. The analysis uses the Fourier approach of [31] for linearity testing. Note that property testing is very efficient but it does not provide the approximating parity function (that requires further testing using the self-correction property). It can be viewed as a preliminary test, which is suitable for our context.

IV. Nonlinear Turbo encoding

We now look not at approximating the constituent encoders, but at describing them *exactly.* The truth tables of the 5-variable (5 obtained from influence) encoding functions can be determined exactly by brute force as well and, in order to understand the nonlinearity of the functions, one can turn these functions into their unique representation as a multilinear polynomial over $\mathbb{F}_2$.

We consider an extended $\mathbb{F}_2$-polynomial representation, which we refer to as a *unate* multilinear polynomial. In unate form, variables can also have negative polarity, i.e., have all their occurrences negated. A polynomial obtained from another polynomial by replacing all occurences of a subset of the variables by their negations, and/or by negating the function, is a *unate variant*. For example, $x_1 \oplus \bar{x}_2 \oplus x_1\bar{x}_2$ is a unate variant of $x_1 \oplus x_2 \oplus x_1x_2$. The use of unate polynomials allows for more compact representation, appealing for interpretability.

The simplest unate polynomials for the encoding functions of TurboAE-binary are in Table I. These are moderately nonlinear syntactically, having 3 nonlinear terms altogether. The function in block 1 is also moderately nonlinear semantically (differs from parity at only 3 points), but the function in block 3 is semantically further from linear (differs at 8 points).

It turns out that there is a more direct way to obtain these polynomials. Going beyond the nonzero pattern of influences in the heatmap, now we make use of their values as well. The nonzero influence values in each row for block 1 are $(\frac{15}{16}, \frac{13}{16}, \frac{13}{16}, \frac{13}{16}, \frac{15}{16})$, for block 2 they are $(1, 1, 1, 0, 1)$, and for block 3 they are $(\frac{1}{2}, 1, \frac{1}{2}, 1, 1)$. These particular influence values determine unique functions up to unate variants. This is a very special case of the *inverse influence problem* and such a result cannot be expected in general, but it suggests that it may be of interest to study conclusions that can be drawn from influences. Influences are also called the Banzhaf index, and related questions have been studied in [32].

Theorem 2. a) *(Block 1) Let $f : \{0,1\}^5 \to \{0,1\}$ have variable influences $\frac{15}{16}, \frac{13}{16}, \frac{13}{16}, \frac{13}{16}, \frac{15}{16}$. Then*

$$f(u) = u_1 \oplus u_2 \oplus u_3 \oplus u_4 \oplus u_5 \oplus (1 \oplus u_1u_5)u_2u_3u_4$$

or a unate variant.

b) *(Block 2) Let $f : \{0,1\}^5 \to \{0,1\}$ have variable influences $1, 0, 1, 1, 1$, then $f(u) = u_1 \oplus u_3 \oplus u_4 \oplus u_5$, or a unate variant.*

c) *(Block 3) Let $f : \{0,1\}^5 \to \{0,1\}$ have variable influences $1, 1, \frac{1}{2}, 1, \frac{1}{2}$, then $f(u) = u_1 \oplus u_2 \oplus u_4 \oplus u_3u_5$, or a unate variant.*

The proof is in the Appendix of [18]. The proof of part *a)* uses the edge isoperimetric inequality for the hypercube [33]. As Theorem 2 and Table I show, TurboAE-binary's constituent codes are *nonsystematic* and *nonrecursive*. Blocks 1 and 3 and *nonlinear*, while block 2 is *affine*.

V. Decoding

So far we have considered interpreting the encoder, leaving the decoder untouched. We investigate how the modified Turbo code found using MILP and the exact nonlinear Turbo code perform when coupled with an iterative BCJR decoder. Iterative BCJR attempts to calculate the posterior probabilities $\mathbb{P}(u_i = 1|\mathbf{y})$ for received message $\mathbf{y}$ [34] [35], whereas the

Block #	Expression for output #j
1	$1 \oplus u_1 \oplus \bar{u}_2 \oplus u_3 \oplus \bar{u}_4 \oplus u_5$ $\oplus\ \bar{u}_2u_3\bar{u}_4 \oplus u_1\bar{u}_2u_3\bar{u}_4u_5$
2	$u_1 \oplus u_3 \oplus u_4 \oplus u_5$
3	$1 \oplus u_1 \oplus u_2 \oplus u_4 \oplus \bar{u}_3\bar{u}_5$
where	$u_1 = x_{j+2}$, $u_2 = x_{j+1}$, $u_3 = x_j$, $u_4 = x_{j-1}$, $u_5 = x_{j-2}$, $x =$ inputs

TABLE I: Exact expressions for TurboAE-Binary encoder

outputs of the black-box CNN decoder of TurboAE-binary may or may not correspond to true probabilities.

Although coupling systematic convolutional codes (SCCs) with a BCJR decoder can be done as in [35], our codes are nonsystematic convolutional codes (NCCs). To allow for NCCs, we used the decoding architecture from [36]. Alternatively, we can turn our nonrecursive NCCs into recursive SCCs as in [37] since block 2 of both our exact and MILP approximated representations are parities. A rigorous formulation of the conversion can be found in the Appendix of [18].

To compare our codes with TurboAE-binary, we estimate BERs on various channels w.r.t. uniformly chosen binary blocks of length 100, uniformly chosen interleaver permutations, and channel noise. For estimating TurboAE-binary's expected BER, we use the original training interleaver only. For input message $\mathbf{x} \in \mathbb{F}_2^m$ of length m, the channel outputs $\mathbf{y} = \mathbf{x} + \mathbf{z}$, and the channel noise vector $\mathbf{z}$ is independent and identically distributed (iid) according to one of the distributions below, and parameterized by a signal-to-noise ratio $SNR(\sigma^2) = -10\log_{10}\sigma^2$ for noise variance $\sigma^2 \in \mathbb{R}$:

- *AWGN*: z_i is iid $\sim \mathcal{N}(0, \sigma^2)$;
- *Additive T-distribution Noise (ATN)*: z_i is iid $\sim T(3, \sigma^2)$; $T(\nu, \sigma^2)$ denotes the T-distribution with distribution parameter ν and scaled to have variance σ^2.

We also benchmark against the following two Turbo codes:

- code rate $R = 1/3$ with generating function $\left(1, \frac{1+x^2}{1+x+x^2}\right)$, which is denoted Turbo-155-7.
- code rate $R = 1/3$ with generating function $\left(1, \frac{1+x^2+x^3}{1+x+x^3}\right)$, which is denoted Turbo-LTE.

In all experiments we use only 6 decoding iterations to remain consistent with the benchmarking in [9]. All decoders are unchanged for experiments on channels other than the AWGN channel. That is, TurboAE-Binary is not fine-tuned to the new channels, and BCJR (incorrectly) assumes the channel is AWGN in its calculations. The expected BERs of our different codes are shown in Figure 3. All experiments are implemented using Python and Tensorflow [38], [39].

Examining the performance of the exact representation of TurboAE-binary BER plot for the AWGN channel in Figure 3, observe that by replacing the black-box decoder of TurboAE-binary we get a fully interpretable Turbo code that performs better than TurboAE-binary on every SNR. Although this does not tell us whether or not the black-box decoder was approximating BCJR, we do see that the black-box decoder is suboptimal. On the other hand, the BER plot for the ATN

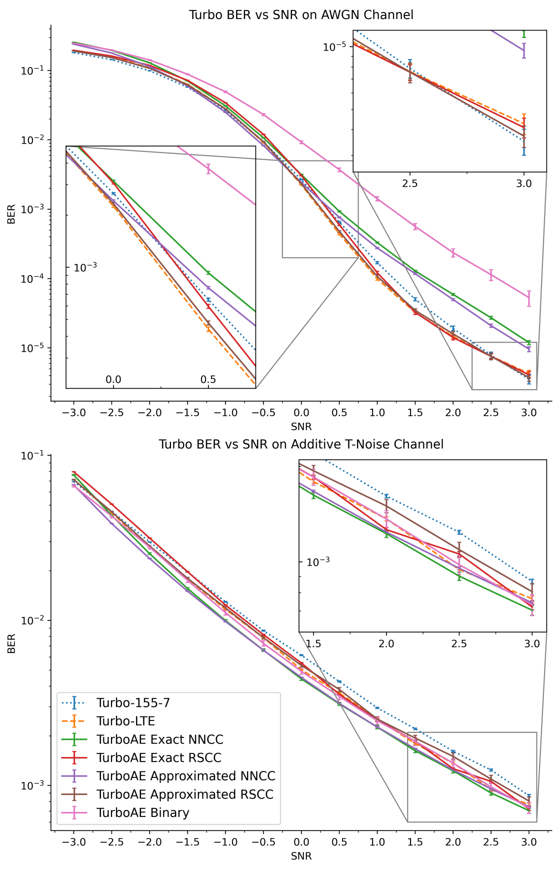


Fig. 3: Performance of TurboAE-binary, exact and MILP approximations of TurboAE-binary, and benchmark Turbo codes. Error bars are 2 standard deviations on the estimated mean BER. For TurboAE-binary, we used the weights provided by the authors of [9]; these BER performances do not exactly reproduce figures in [9].

channel shows TurboAE-binary performing competitively with the other tested codes. TurboAE-binary's decoder appears unusually robust compared to BCJR suggesting that it is finding some tradeoff between performance on AWGN vs. robustness on other channels, echoing the findings in [9].

For the AWGN channel the approximated representation performs just as well as the exact expression (and better on some SNRs). These BER plots suggest that the non-linearity in TurboAE-binary may be a "bug". That is, the CNN encoder behind TurboAE-binary may have been approaching an affine encoding function during training, but, due to complex training dynamics, was not able to converge. Fourier analysis (see Fig. 4) may help explain this phenomenon.

VI. Learning

Up to now we have discussed the input-output behavior of TurboAE-binary. In this section we give some remarks on the *training dynamics*, that is, on the evolution of the output during the learning process. The encoding function learned by TurboAE is a *real-valued Boolean function* $f : \{0,1\}^n \to \mathbf{R}$.

Figure 4 shows snapshots of the 32 Fourier coefficients for the encoding functions after the three stages of training TurboAE-binary: after training the real-valued-Boolean TurboAE (a), after taking the sign function (b), and the final result after re-training using the Straight-Through Estimator (STE) (c). Each coefficient corresponds to a parity function. The largest coefficient are dominating in each snapshot. For each block, the snapshots after each stage are similar, but the dynamics during the first stage and the more subtle changes after that require further study. The similarity of the Fourier representations before and after applying the sign functions is related to Plancherel's theorem.

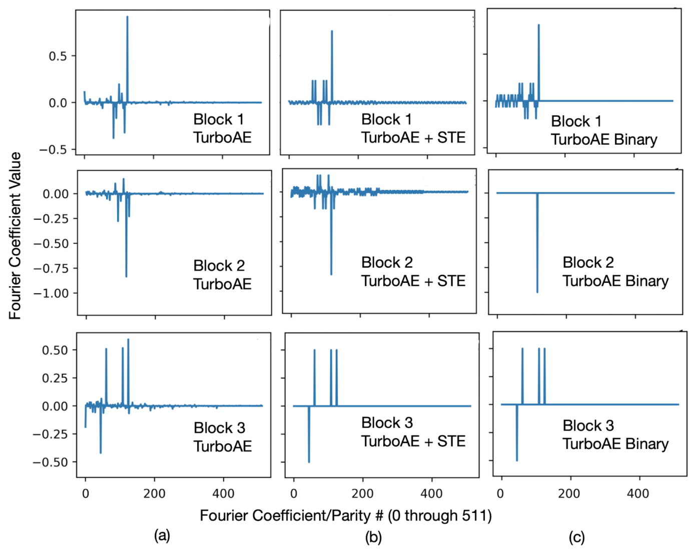


Fig. 4: Fourier Coefficients of Block Encoder functions at different stages of training. (a) Trained TurboAE, (b) Trained TurboAE with STE module, (c) Trained TurboAE Binary

VII. Conclusion

We studied the interpretability of the TurboAE-binary deep-learned error-correcting code. We conclude:

- TurboAE-binary is a nonlinear modified Turbo code with few nonlinear terms, well approximated by a modified Turbo code, that can be found by MILP.
- Influence heatmaps provide valuable information about the encoding functions computed by the network.
- Interpretable representations (e.g. modified CC) need to be flexible to accommodate approximations.
- Multi-output property testing and MILP are potential techniques for further exploring interpretability.
- Using more interpretable modules, e.g. the iterated BCJR decoder instead of learned decoders, can be advantageous.
- There lies potential in applying neural networks to search the space of non-linear Turbo codes.
- The Fourier representation of Boolean functions can be a useful tool for exploring the training dynamics.

Several related aspects need further study. These include robustness of the properties found for other learned models and robustness of the learned decoder (compared to BCJR) for non-AWGN noise models. The approaches developed could also be used for the hidden layers, to understand both the final representation and the training dynamics.

References

[1] H. Kim, S. Oh, and P. Viswanath, “Physical layer communication via deep learning,” *IEEE Journal on Sel. Areas in Inf. Theory*, vol. 1, no. 1, pp. 5–18, 2020.

[2] Y. Jiang *et al.*, “Learn codes: Inventing low-latency codes via recurrent neural networks,” *IEEE Journal on Selected Areas in Information Theory*, vol. 1, no. 1, pp. 207–216, 2020.

[3] T. J. O’Shea, K. Karra, and T. C. Clancy, “Learning to communicate: Channel auto-encoders, domain specific regularizers, and attention,” in *2016 IEEE International Symposium on Signal Processing and Information Technology (ISSPIT)*, 2016, pp. 223–228.

[4] Y. Jiang *et al.*, “Mind: Model independent neural decoder,” in *2019 IEEE 20th International Workshop on Signal Processing Advances in Wireless Communications (SPAWC)*, 2019, pp. 1–5.

[5] J. Whang *et al.*, “Neural distributed source coding,” *CoRR*, vol. abs/2106.02797, 2021. [Online]. Available: https://arxiv.org/abs/2106.02797

[6] R. K. Mishra *et al.*, “Distributed Interference Alignment for K -user Interference Channels via Deep Learning,” in *International Symposium on Information Theory (ISIT)*, 2021.

[7] H. Kim *et al.*, “Deepcode: Feedback codes via deep learning,” *IEEE Journal on Sel. Areas in Inf. Theory*, vol. 1, no. 1, pp. 194–206, 2020.

[8] Y. Jiang *et al.*, “Joint channel coding and modulation via deep learning,” in *2020 IEEE SPAWC*, 2020, pp. 1–5.

[9] ——, “Turbo autoencoder: Deep learning based channel codes for point-to-point communication channels,” in *Proceedings of the 33rd International Conference on Neural Information Processing Systems*, Dec. 2019, pp. 2758–2768.

[10] H. Ye, L. Liang, and G. Y. Li, “Circular convolutional auto-encoder for channel coding,” in *2019 IEEE 20th International Workshop on Signal Processing Advances in Wireless Communications (SPAWC)*, 2019, pp. 1–5.

[11] T. Ching *et al.*, “Opportunities and obstacles for deep learning in biology and medicine,” *Journal of The Royal Society Interface*, vol. 15, no. 141, p. 20170387, Apr. 2018.

[12] H. Naik and G. Turán, “Explanation from Specification,” in *Explainable Agency in AI Workshop, 35th AAAI Conference*, vol. abs/2012.07179, 2021. [Online]. Available: https://arxiv.org/abs/2012.07179

[13] J. M. Walsh, P. A. Regalia, and C. R. Johnson, “Turbo decoding as iterative constrained maximum-likelihood sequence detection,” *IEEE Transactions on Information Theory*, vol. 52, no. 12, pp. 5426–5437, 2006.

[14] J. Walsh, C. Johnson, and P. Regalia, “A refined information geometric interpretation of turbo decoding,” in *Proceedings. (ICASSP ’05). IEEE International Conference on Acoustics, Speech, and Signal Processing, 2005.*, vol. 3, 2005, pp. iii/481–iii/484 Vol. 3.

[15] B. Muquet, P. Duhamel, and A. de Courville, “Geometrical interpretation of iterative turbo decoding,” in *Proceedings IEEE International Symposium on Information Theory,*, 2002, pp. 142–.

[16] “TurboAE github for TurboAE,” 2020. [Online]. Available: https://github.com/yihanjiang/turboae/blob/master/models/dta_cont_cnn2_cnn5_enctrain2_dectrainneg15_2.pt

[17] “TurboAE github for TurboAE-binary,” 2020. [Online]. Available: https://github.com/yihanjiang/turboae/blob/master/models/dta_steq2_cnn2_cnn5_enctrain2_dectrainneg15_2.pt

[18] N. Devroye *et al.*, “Interpreting deep-learned error-correcting codes,” Jan. 2022. [Online]. Available: https://devroye.lab.uic.edu/research-2/publications/

[19] S. Lin and D. J. Costello, *Error Control Coding*. Pearson, 2005.

[20] T. Richardson and R. Urbanke, *Modern Coding Theory*. Cambridge: Cambridge University Press, 2008.

[21] Y. Bengio, N. Léonard, and A. C. Courville, “Estimating or propagating gradients through stochastic neurons for conditional computation,” *ArXiv*, vol. abs/1308.3432, 2013.

[22] I. Hubara *et al.*, “Binarized neural networks,” in *Proceedings of the 30th International Conference on Neural Information Processing Systems*, Dec. 2016, pp. 4114–4122.

[23] F. Gurski, “Efficient binary linear programming formulations for boolean functions,” *Statistics, Optimization & Information Computing*, vol. 2, no. 4, pp. 274–279, Nov. 2014.

[24] Gurobi Optimization, LLC, “Gurobi Optimizer Reference Manual,” 2021. [Online]. Available: https://www.gurobi.com

[25] R. O’Donnell, *Analysis of Boolean functions*. Cambridge University Press, 2014.

[26] C. Molnar, *Interpretable Machine Learning: A Guide for Making Black Box Models Interpretable*. Leanpub, 2019.

[27] O. Goldreich and L. A. Levin, “A hard-core predicate for all one-way functions,” in *Proceedings of the 21st Annual ACM Symposium on Theory of Computing*, 1989, pp. 25–32.

[28] O. Goldreich, Ed., *Introduction to Property Testing*. Cambridge University Press, 2017.

[29] M. Parnas, D. Ron, and R. Rubinfeld, “Tolerant property testing and distance approximation,” *J. Comput. Syst. Sci.*, vol. 72, pp. 1012–1042, 2006.

[30] M. Blum, M. Luby, and R. Rubinfeld, “Self-testing/correcting with applications to numerical problems,” *J. Comput. Syst. Sci.*, vol. 47, pp. 549–595, 1993.

[31] M. Bellare *et al.*, “Linearity testing in characteristic two,” *IEEE Trans. Inf. Theory*, vol. 42, no. 6, pp. 1781–1795, 1996.

[32] N. Alon and P. H. Edelman, “The inverse Banzhaf problem,” *Social Choice and Welfare*, vol. 34, pp. 371–377, 2010.

[33] S. Hart, “A note on the edges of the n-cube,” *Discr. Math.*, vol. 14, pp. 157–163, 1976.

[34] L. Bahl *et al.*, “Optimal decoding of linear codes for minimizing symbol error rate (corresp.),” *IEEE Transactions on Information Theory*, vol. 20, no. 2, pp. 284–287, 1974.

[35] C. Berrou and A. Glavieux, “Near optimum error correcting coding and decoding: Turbo-codes,” *IEEE Transactions on Communications*, vol. 44, no. 10, pp. 1261–1271, Oct. 1996.

[36] O. Y. Takeshita, O. M. Collins, and D. J. Costello Jr, “Turbo codes with non-systematic constituent codes,” in *9th NASA Symposium on VLSI Design*, 2000.

[37] D. J. C. MacKay, *Information Theory, Inference, and Learning Algorithms*. Cambridge, UK: Cambridge University Press, 2003.

[38] G. Van Rossum and F. L. Drake, *Python 3 Reference Manual*. Scotts Valley, CA: CreateSpace, 2009.

[39] M. Abadi *et al.*, “TensorFlow: Large-scale machine learning on heterogeneous systems,” 2015, software available from tensorflow.org. [Online]. Available: https://www.tensorflow.org/

[40] S. Arora and B. Barak, *Computational Complexity: A Modern Approach*. Cambridge Univ. Press, 2009.

Appendix A
Relationship between architecture and influence memory length, extension of Section II

Architecture dictates memory 9. The first 1-D conv layer has 100 channels, where each channel has 100 outputs. For each channel c, all its outputs are evaluated via the same convolutional kernel/filter $K_{1,c}$ operating on a window of 5 inputs. Hence each output j of layer 1 depends on window $j-2:j+2$ of inputs. The second 1-D conv layer also has 100 channels, where each channel has 100 outputs. Again for each channel c, all its outputs are evaluated via the same convolutional kernel/filter $K_{2,c}$ operating on a window of 5×100 outputs from layer 1. Hence each output j of layer 2 depends on window $j-2:j+2$ across all the channels of layer 1. Finally the last layer is a single channel of 100 outputs, where each output j is evaluated using a linear layer K_3 operating on output j from all the channels of layer 2. Thus fan-in of each output j when resolved up to the input layer contains the inputs in window $j-4:j+4$, which is of size 9 as shown in Fig 5.

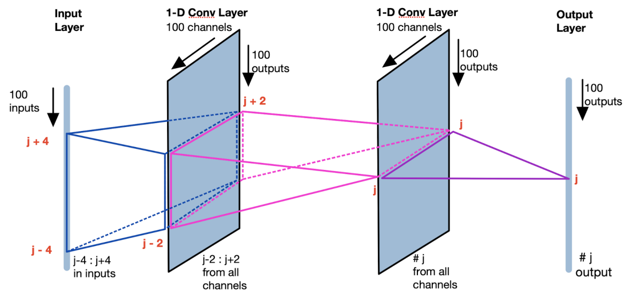


Fig. 5: Fan-in of *j*'th output of TurboAE Encoder Blocks. *j*'th output depends only on inputs *j-4:j+4*

Appendix B
Formulation of approximating code as an MILP, extension of Section III-A

We next show that the approximation problem given by Eq. (2) can be formulated as a Mixed-Integer Linear Program (MILP). Recall that any modified convolutional code with memory M can be parameterized with a binary vector $\mathbf{g} \in \{0,1\}^{M+1}$ according to (1). Accordingly, we can write $\mathbf{x}_{\mathbf{g}}(\mathbf{u})$ rather than $\mathbf{x}_{g_{\text{conv}}}(\mathbf{u})$. If we approximate the expected value in (2) with a random sampling over $\{0,1\}^k$, problem (2) can then be rewritten as:

$$\mathbf{g}^{(j)} = \underset{\mathbf{g}\in\{0,1\}^{M+1}}{\arg\min} \frac{1}{N}\sum_{i=1}^{N} d_H(\mathbf{x}_{\mathbf{g}}(\mathbf{u}^i), \mathbf{x}_{AE,j}(\mathbf{u}^i)), \tag{4}$$

where $\mathbf{u}^i$ is the ith random sample of $\mathbf{u}$ and N is the number of random samples. To determine N in practice we repeatedly solve the problem, each time increasing N by a constant factor, until the solution stabilizes.

Next, we use the fact that $\oplus$ can be expressed as a system of linear inequalities:

Proposition 3 (Gurski [23]). *Given $a, b, c \in \{0,1\}$, let $\Gamma(a,b,c) = (a+b+c-2, a-b-c, -a+b-c, -a-b+c)$ and $\mathbf{0}$ be the all zero vector. Then $a = b \oplus c$ if and only if $\Gamma(a,b,c) \leq \mathbf{0}$.*

This result is easy to generalize to expressions like Eq. (1):

Proposition 4. *Given $a \in \{0,1\}$ and $\mathbf{b} = (b_1, \dots, b_m) \in \{0,1\}^m, m \geq 3$, then $a = \bigoplus_{i=1}^m b_i$ iff the following has a solution:*

$$\begin{gathered} c_m = a \\ \Gamma(c_2, b_1, b_2) \leq \mathbf{0} \\ \Gamma(c_i, c_{i-1}, b_i) \leq \mathbf{0},\ i = 3, \dots, m. \end{gathered} \tag{5}$$

It is assumed that in each instance, the variables c_i are different.

Proof: (By induction.) We start with the base case $m = 3$, where $a = b_1 \oplus b_2 \oplus b_3$. Let

$$c_3 = a, \tag{6}$$

and

$$c_2 = b_1 \oplus b_2, \tag{7}$$

which means that

$$c_3 = c_2 \oplus b_3 \tag{8}$$

But according to Proposition 3, (7) is equivalent to $\Gamma(c_2, b_1, b_2) \leq \mathbf{0}$ while (8) is equivalent to $\Gamma(c_3, c_2, b_3) \leq \mathbf{0}$. These two inequalities, together with (6) exactly correspond to (5).

Now let's assume that the proposition holds for $m = n-1$. We will then show that it also holds for $m = n$. For $m = n$, we have $a = \bigoplus_{i=1}^n b_i$. Let

$$c_n = a, \tag{9}$$

and

$$c_{n-1} = b_1 \oplus \dots \oplus b_{n-1}. \tag{10}$$

According to the inductive assumption, (10) is equivalent to inequalities

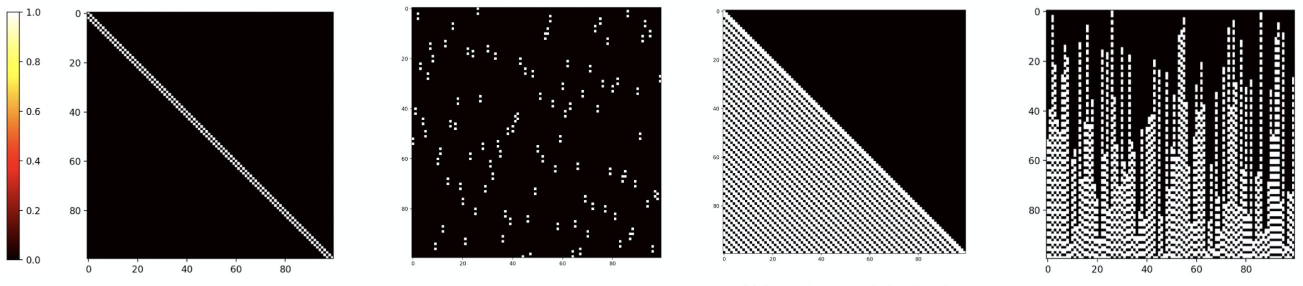
$$\begin{gathered} \Gamma(c_2, b_1, b_2) \leq \mathbf{0} \\ \Gamma(c_i, c_{i-1}, b_i) \leq \mathbf{0},\ i = 3, \dots, n-1. \end{gathered} \tag{11}$$

But $c_n = c_{n-1} \oplus b_n$ which is equivalent to $\Gamma(c_n, c_{n-1}, b_n)$. The last inequality, together with (9) and(11) exactly corresponds to (5) for $m = n$. ■

Denote the system of inequalities (5) by $\text{INEQ}(a, \mathbf{b})$. We next formulate Problem (4) of finding the best approximation with a modified convolutional code described by Eq. (1) as an MILP. Let x_ℓ^i and $x_{AE,j\,\ell}^i$ denote the ℓth component of $\mathbf{x}_{\mathbf{g}}(\mathbf{u}^i)$ and $\mathbf{x}_{AE,j}(\mathbf{u}^i)$, respectively.

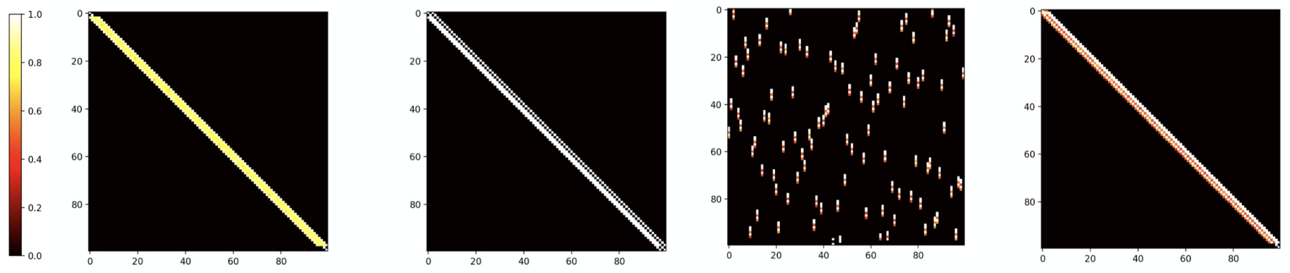
Theorem 5. *Problem (4) is equivalent to the following Mixed-Integer Programming Problem:*

$$\mathbf{g}^{(j)} = \underset{\mathbf{g}\in\{0,1\}^{M+1}}{\arg\min} \frac{1}{N}\sum_{i=1}^{N}\sum_{\ell=1}^{k} z_\ell^i,$$



(a) Non-recusrive convolutional code generator 101
(b) Interleaved non-recusrive convolutional code generator 101
(c) Recursive convolutional code generator 101/111
(d) Interleaved recursive convolutional code generator 101/111

Fig. 6: Convolutional Encoder Influence heatmaps. The forward generator in (a)-(d) is [101]. The feedback polynomial for the recursive convolutional encoders in (c) and (d) is [111].



(a) TurboAE constituent learned code f1
(b) TurboAE constituent learned code f2
(c) TurboAE constituent learned code f3 Together with interleaving (third branch)
(d) TurboAE constituent learned code f3 AFTER de-interleaving

Fig. 7: TurboAE Constituent Learned Codes Influence heatmaps.

M	(N, # input bits)	$\mathbf{g}^{(1)}$	HD_1	$\mathbf{g}^{(2)}$	HD_2	$\mathbf{g}^{(3)}$	HD_3
5		000100	41.67%	111001	39.58%	111101	41.67%
10	(1, 100)	00**11111**0001	5.21%	00**10111**0000	2.08%	00**11010**0000	18.75%
15		00**11111**000000001	5.21%	00**10001**000000000	2.08%	00**11010**000000000	18.75%
5		001001	44.06%	000001	47.29%	001100	47.19%
10	(10, 1000)	00**11111**001	9.69%	00**10111**0000	2.08%	00**11110**0001	24.38%
15		00**11111**000000001	9.69%	00**10111**000000000	2.08%	00**11110**000000001	24.38%
5		001001	46.71%	101101	49.26%	011000	47.19%
10	(100, 10000)	00**11111**0001	10.61%	00**10111**0000	2.08%	00**11110**0001	25.22%
15		00**11111**000000001	10.61%	00**10111**000000000	2.08%	00**11110**000000001	25.22%

TABLE II: MILP experimental results for the modified convolutional codes $\mathbf{g}^{(i)}$ that best approximate the constituent learned codes $f_{i,\theta}$ of TurboAE-binary in the sense of (4). In bold is the length-5 $\mathbf{g}^{(i)}$ save for the $M+1$-st bit, shown in red.

such that for every $i = 1, \dots, N$, $\ell = 1, \dots, k$

$$\Gamma(z_\ell^i, x_\ell^i, x_{AE,j_\ell}^i) \leq \mathbf{0} \tag{12}$$

and

$$\mathrm{INEQ}(x_\ell^i, [g_1\, u_{\ell+L}^i, g_2\, u_{\ell+L-1}^i, \dots, g_M\, u_{\ell+L-M+1}^i, g_{M+1}]). \tag{13}$$

Proof: We have:

$$\begin{aligned}
\mathbf{g}^{(j)} &\overset{(1)}{=} \operatorname*{arg\,min}_{\mathbf{g}\in\{0,1\}^{M+1}} \frac{1}{N}\sum_{i=1}^{N} d_H(\mathbf{x}_{\mathbf{g}}(\mathbf{u}^i), \mathbf{x}_{AE,j}(\mathbf{u}^i)) \\
&\overset{(2)}{=} \operatorname*{arg\,min}_{\mathbf{g}\in\{0,1\}^{M+1}} \frac{1}{N}\sum_{i=1}^{N}\sum_{\ell=1}^{k} x_\ell^i \oplus x_{AE,j_\ell}^i \\
&\overset{(3)}{=} \operatorname*{arg\,min}_{\mathbf{g}\in\{0,1\}^{M+1}} \frac{1}{N}\sum_{i=1}^{N}\sum_{\ell=1}^{k} z_\ell^i
\end{aligned}$$

where equality (1) is Eq. (4), equality (2) holds by definition of Hamming distance and equality (3) holds due to Proposition 3 and Eq. (12). Finally, x_ℓ^i is given by Eq. (1), which is equivalent to Eq. (13). ■

The results of the MILP approximation are shown in Table II. Three parameters need to be chosen to find the MILP approximation: (a) M, the memory order of the modified convolutional codes; (b) L, the lookahead horizon; and (c) N, the number of input samples. The lookahead horizon L needs to be sufficiently large to capture the dependency of the current output bit on the future input bits. But increasing L increases M (since at the least $M \geq L$), which in turn increases the computation time. In our case L was chosen to be $L = 4$.

We see that N needs to be sufficiently large to find the correct solution (consistent with that produced by the influence analysis). In other words, the space of inputs needs to be properly sampled. In practice, N can be successively increased by some factor $\mu > 1$ and the computations repeated until the solution stabilizes.

Appendix C
Influence and Property Testing, Extensions of Sections III-B and III-C

In this section we give more details about influence heatmaps and present a proof of Theorem 1.

Influence calculation details. Recall the definition of influence of (3). Given an (n, k) linear code with a matrix $G = (g_{ij})$, consider the corresponding multi-output Boolean function $f : \{0,1\}^k \to \{0,1\}^n$ with $f = (f_1, \dots, f_n)$. Then

$$\operatorname*{Inf}_{f_j}(i) = \begin{cases} 1 & \text{if } g_{ij} = 1 \\ 0 & \text{if } g_{ij} = 0 \end{cases}.$$

More generally, every (n, k) code can be represented by a multi-output Boolean function $f : \{0,1\}^k \to \{0,1\}^n$. One can then consider the matrix of influences $\text{Inf}_{f_j}(i)$, and visualize the influences as a heatmap.

The heatmap for nonrecursive convolutional codes shows the familiar staircase pattern shown in Fig. 6(a) – if the input is interleaved before being sent to a non-recursive convolutional encoder we obtain the heatmap in 6(b). Fig. 6(c) and (d) show heatmaps for recursive convolutional codes without and with interleaving before the convolution. These heatmaps are black and white.

We sample the influences of the TurboAE-binary model in the form of a 100×300 matrix. The 100×100 matrices for the three blocks are shown on Fig. 7(a)-(d), where (c) and (d) show the influences when the interleaver is kept, and when it is removed. The heatmaps show that the influence structure of TurboAE-binary is similar to that of a Turbo code. The colors indicate that the influences are different from 0 and 1 to varying degrees. The Turbo code is *nonsystematic*, i.e., there is no block corresponding to a single diagonal. It is *nonrecursive*, as the blocks show the staircase pattern. For the third block this pattern is obtained after de-interleaving. Furthermore, the convolutional codes involved have memory 5 in each case. Interestingly, and we believe coincidentally, 5 is also the size of the kernel of the CNN.

Proof of Theorem 1: (There is a tolerant testing algorithm with $\epsilon' = \epsilon/18$ for CSMP using $O(1/\epsilon)$ queries.)

The proof is a reduction to multi-parity. For $g = (g_1, \dots, g_n) : \{0,1\}^n \to \{0,1\}^n$ let $g^* = (g_1*, \dots, g_n^*)$, where $g_i^*(x) = g_i(s^{-(i-1)}(x))$. Then g is CSMP iff g^* is multi-parity. The $*$ operator is distance-preserving, i.e., $d(f, g) = d(f^*, g^*)$ for every f, g. By the definition of distance, $d(f, g) = E_i(d(f_i, g_i))$ and $d(f^*, g^*) = E_i(d(f_i^*, g_i^*))$. But $d(f_i, g_i) = d(f_i^*, g_i^*)$, as the shift is a bijection between the two domains. Hence Theorem 1 follows from the following result.

Lemma 6. *There is a tolerant testing algorithm (with $\epsilon' = \epsilon/18$) for multi-parity using $O(1/\epsilon)$ queries.*

Proof: is an adaptation of the Fourier analysis of linearity testing [31]. Switching to truth values ± 1, consider a function $f = (f_1, \dots, f_n) : \{\pm 1\}^n \to \{\pm 1\}^n$. The following test modifies the standard linearity test [30] by picking the function indices i, j, k randomly as well. The componentwise product of x and y is denoted by xy.

1) Pick indices i, j, k and vectors x, y randomly.
2) Accept if $f_i(x)f_j(y) = f_k(xy)$.

The test rejection rate is $P_{i,j,k,x,y}(f_i(x)f_j(y) \neq f_k(xy))$. The distance of f from multi-parity functions is $\min_S P_{i,x}(f_i(x) \neq \chi_S(x))$. The following lemma is a reformulation of the statement that the test rejection rate is at least the distance.

Lemma 7. *It holds that*

$$P_{i,j,k,x,y}(f_i(x)f_j(y) = f_k(xy)) \leq \max_S P_{i,x}(f_i(x) = \chi_S(x)).$$

Proof: First note that $f_i(x)f_j(y) = f_k(xy)$ iff $f_i(x)f_j(y)f_k(xy) = 1$. Fix i, j, k. Then it follows as in [40], p. 479, that

$$E_{x,y}(f_i(x)f_j(y)f_k(xy)) = \sum_S \hat{f}_i(S) \cdot \hat{f}_j(S) \cdot \hat{f}_k(S).$$

Taking expectations w.r.t. i, j, k we get

$$\begin{aligned}
&E_{i,j,k,x,y}(f_i(x)f_j(y)f_k(xy)) = \\
&E_{i,j,k}\left(\sum_S \hat{f}_i(S) \cdot \hat{f}_j(S) \cdot \hat{f}_k(S)\right) = \\
&\sum_S E_{i,j,k}\left(\hat{f}_i(S) \cdot \hat{f}_j(S) \cdot \hat{f}_k(S)\right) = \\
&\sum_S (E_i(\hat{f}_i(S)))^3 \leq \\
&\left(\max_S E_i(\hat{f}_i(S))\right) \cdot \sum_S (E_i(\hat{f}_i(S)))^2 \leq \\
&\left(\max_S E_i(\hat{f}_i(S))\right) \cdot \sum_S E_i((\hat{f}_i(S))^2) = \\
&\left(\max_S E_i(\hat{f}_i(S))\right) \cdot E_i\left(\sum_S (\hat{f}_i(S))^2\right) = \\
&\max_S E_i(\hat{f}_i(S)) = \max_S E_i(f_i(x)\chi_S(x)).
\end{aligned}$$

If $P_{i,j,k,x,y}(f_i(x)f_j(y) = f_k(xy)) = 1 - \alpha$ then it holds that $E_{i,j,k,x,y}(f_i(x)f_j(y)f_k(xy)) = 1 - 2\alpha$. Thus $E_{i,x}(f_i(x)\chi_S(x)) \geq 1 - 2\alpha$ for the maximal S above, and $P_{i,x}(f_i(x) = \chi_S(x)) \geq 1 - \alpha$. ■

Now the property testing algorithm is the following: *repeat the test $2/\epsilon$ times and accept iff each round accepts.*

Assume that the distance of f from multi-parity functions is at least ϵ. Then by Lemma 7 the the acceptance probability of the test is at most $(1-\epsilon)^{2/\epsilon} < 1/e^2 < 1/3$, and so f is rejected with probability at least $2/3$.

Assume that the distance of f from multi-parity functions is at most $\epsilon/18$. The rejection rate of the testing algorithm is upper-bounded by $d \cdot t \cdot q$, where d is the distance, t is the number of repetitions and q is the number of points queried in one round [29]. This follows from the fact the probability of ever drawing a point from the error region of the closest multi-parity function is at most $d \cdot t \cdot q$ by the union bound. This implies that f is rejected with probability at most $1/3$. This completes the proof of Lemma 6, and thus of Theorem 1 as well. ■

Self-correction for linearity testing allows to determine the target function value for *every* input with high probability [30]. In the multi-parity case computing $\chi_S(x)$ for any fixed x can be achieved by computing

$$f_j(y) f_k(xy)$$

for random j, k, y. If the distance of f from multi-parity functions is at most ϵ then $P_{j,y}(f_j(y) \neq \chi_S(y)) \leq \epsilon$ and $P_{k,y}(f_k(xy) \neq \chi_S(xy)) \leq \epsilon$. Thus with probability at least $1-2\epsilon$ we compute $\chi_S(y) \cdot \chi_S(xy) = \chi_S(x)$.

Appendix D
Proof of Theorem 2 in Section IV

Part a): Block 1

Let f be a 5-variable Boolean function with influences $Inf_1(f) = Inf_5(f) = 15/16$ and $Inf_2(f) = Inf_3(f) = Inf_4(f) = 13/16$. Consider the function $g = f \oplus \bigoplus_{i=1}^5 u_i$. It is sufficient to determine the possible functions g as $f = g \oplus \bigoplus_{i=1}^5 u_i$.

For every variable i it holds that $Inf_i(f) + Inf_i(g) = 1$, as $f(u) \neq f(u^{(i)})$ iff $g(u) \neq g(u^{(i)})$ for every edge $(u, u^{(i)})$. Thus $Inf_1(g) = Inf_5(g) = 1/16$ and $Inf_2(g) = Inf_3(g) = Inf_4(g) = 3/16$. Note that influence $Inf_i(g)$ is the fraction of edges along dimension i where the function values are different. Thus, for example, the number of edges along dimension 2 is 3.

Let $T(g) = \{u : g(u) = 1\}$, E be the set of hypercube edges in $T(g)$ and *edge boundary size* $\sigma(g)$ be the number of edges between $T(g)$ and its complement. It holds that $\sigma(g) = 5|T(g)| - 2|E|$. It follows from the influences that $\sigma(g) = 11$. Assume w.l.o.g. $|T(g)| \leq 16$ (otherwise consider the negation of g).

We claim that $T(g) = \{a, a^{(1)}, a^{(5)}\}$ for some $a \in \{0,1\}^5$.

First we show that $|T(g)| = 3$. If $|T(g)| \leq 2$ then $\sigma(g) \leq 10$. If $4 \leq |T(g)| \leq 16$ then $\sigma(g) \geq 12$ by the exact formula in the edge-isoperimetric theorem [33]. We note that for $|T(g)| = 4$ this follows directly: the maximal number of edges in $T(g)$ is 4, corresponding to a 4-cycle (5 edges would have to contain a triangle), and thus $\sigma(g) \geq 20 - 8 = 12$. This is not sufficient for the general claim as the smallest edge boundary size is not a monotone function of set size.

For $|T(g)| = 3$ we need $|E| = 2$ to get $\sigma(g) = 11$. This implies that E is a path of two edges, thus $T(g) = \{a, a^{(i)}, a^{(j)}\}$. In order to match the specifications, it has to be the case that $\{i, j\} = \{1, 5\}$.

If $a = 01110$ then $T(g) = \{01110, 11110, 01111\}$. Thus $g = u_2u_3u_4\bar{u}_5 \vee \bar{u}_1u_2u_3u_4 = (\bar{u}_1 \vee \bar{u}_5)u_2u_3u_4$. Using $y \vee z = 1 \oplus \bar{y}\bar{z}$ we get $g = (1 \oplus u_1u_5)u_2u_3u_4$.

For other values of a one gets the unate versions of this polynomial. Considering negations of these polynomials one gets the other unate versions.

Part b): Block 2

Assume that for a function f it holds that $Inf_1(f) = 1$. Consider $u = (u_1, \ldots, u_n)$ and let $u' = (u_2, \ldots, u_n)$. The identity $f(u) = f(0, u') \oplus u_1(f(0, u') \oplus f(1, u'))$ implies that $f(u) = f(0, u') \oplus u_1$. Influences of the subfunction $f_{u_1=0} = f(0, u')$ are unchanged, i.e., $Inf_i(f_{u_1=0}) = Inf_i(f)$ for $i > 1$.

For the function f in block 2, $Inf_2(f) = 0$ implies that it does not depend on u_2, and the argument above implies that it is a parity of the other 4 variables or their negation.

Part c): Block 3

The claim in Part b) implies that $f(u)$ is of the form $a \oplus u_1 \oplus u_2 \oplus u_4 \oplus g(u_3, u_5)$, where both variables of g have influence $1/2$. Here g can be a conjunction or a unate variant (these are all 2-variable functions excluding constants, variables, negated variables and the two affine functions). ■

Appendix E
Extension of Section V

Here we establish the generalized conversion between non-recursive nonsystematic convolutional codes (NNCCs) and recursive systematic convolutional codes (RSCCs). First some preliminaries:

Definition 8. *A rate 1/n **generalized convolutional code** is a tuple $C = (h_1, h_2, \ldots, h_n, g, M)$. $M \in \mathbb{N}$ is our memory, $h_1, \ldots, h_n : \mathbb{F}_2 \times \mathbb{F}_2^M \to \mathbb{F}_2$ are our encoders, and $g : \mathbb{F}_2 \times \mathbb{F}_2^M \to \mathbb{F}_2$ is our feedback. A bit $u \in \mathbb{F}_2$ and state $s \in \mathbb{F}_2^M$ is encoded as $(h_1(u, s), h_2(g(u, s), s), \ldots, h_n(g(u, s), s))$, and the next state is set to $(g(u, s), s_1, \ldots, s_{M-1})$. The state is always initialized as 0.*

The following theorem establishes sufficient conditions for converting a generalized NNCC to an RSCC:

Theorem 9. *Let $C = (h_1, h_2, (u, s) \mapsto u, M)$ be a nonrecursive rate $1/2$ NCC of memory M. If $\forall s \in \mathbb{F}_2^M$ $\forall u \in \mathbb{F}_2$ we have that $h_1(u, s) \neq h_1(\neg u, s)$, then there exists a feedback function g s.t. $C' = ((u, s) \mapsto u, h_2, g)$ has the same set of codewords as C.*

Proof: We will use h_1 to directly construct g. Consider the function $H : \mathbb{F}_2 \times \mathbb{F}_2^M \to \mathbb{F}_2 \times \mathbb{F}_2^M$ defined as $(u, s) \mapsto (h_1(u, s), s)$.

Note that H is injective. To see this, let $(u, s), (u', s') \in \mathbb{F}_2 \times \mathbb{F}_2^M$. Then if $s \neq s'$, $H(u, s) \neq H(u', s')$. On the other hand, if $s = s'$ and $u \neq u'$, then because $h_1(u, s) \neq h_1(\neg u, s)$, we know $H(u, s) \neq H(u', s')$. Thus $H(u, s) = H(u', s')$ iff $(u, s) = (u', s')$ and H is injective. Since H is injective and because its domain and codomain are finite with equal cardinality, we know H is surjective and thus a bijection.

Define g to be $(u, s) \mapsto \pi_1(H^{-1}(u, s))$, where π_1 is the first coordinate projection. That is, $C' = ((u, s) \mapsto u, h_2, g, M)$. Let $\mathcal{C}(C')$ denote the set of codewords of C'. Now it remains to show $\mathcal{C}(C') = \mathcal{C}(C)$. Suppose we are given input message is $\mathbf{u} \in \mathbb{F}_2^k$ for message length k, and, after applying C, we get encoded streams $\mathbf{x}^{(1)}, \mathbf{x}^{(2)} \in \mathbb{F}_2^k$. Let $\mathbf{u}' = \mathbf{x}^{(1)}$ be the input to C' and let $\mathbf{x}^{(1)'}, \mathbf{x}^{(2)'} \in \mathbb{F}_2^k$ be the encoded streams produced. By construction, $\mathbf{x}^{(1)} = \mathbf{x}^{(1)'}$. To see the remaining encoded streams are the same, denote $r_t = (u_t, s_t)$ and $r'_t = (g(u'_t, s'_t), s'_t)$ the inputs to the second encoder for C and C' respectively at timestep $t \in [k]$. To show they are the same, we induct on $t \in [k]$. At $t = 0$, we have $r_0 = (u_0, 0) = (\pi_1(H^{-1} \circ H(u_0, 0)), 0), 0) = (g(h(u_0, 0), 0), 0) = (g(u'_0, 0), 0) = r'_0$. Suppose $r_{t-1} = r'_{t-1}$. Then at timestep t, C receives u_t, and $r_t = (u_t, s_t)$. Since $r_{t-1} = r'_{t-1}$, we know $s_t = s'_t$ (C and C' are convolutional codes). Furthermore, $u_t = \pi_1(H^{-1} \circ H(u_t, s_t)) = g(h(u_t, s_t), s_t) = g(u'_t, s'_t)$. Thus $r_t = (u_t, s_t) = (g(u'_t, s'_t), s'_t) = r'_t$ and $h_2(r_t) = h_2(r'_t)$ $\forall t \in [k]$. Therefore $\mathbf{x}^{(i)} = \mathbf{x}^{(i)'}$ $\forall i \in [2]$ and $\mathcal{C}(C) \subseteq \mathcal{C}(C')$.

To get the opposite direction inclusion, we can apply a similar argument. Let $\mathbf{u}' \in \mathbb{F}_2^k$ be the input to C' and define $\mathbf{u} \in \mathbb{F}_2^k$, the input to C, so that $u_t = g(u'_t, s'_t)$ for $t \in [k]$. We apply induction on $r_t, r'_t \in \mathbb{F}_2 \times \mathbb{F}_2^M$, the inputs to the encoder functions for C and C' respectively for $t \in [k]$. At $t = 0$, we have $r_0 = (u_0, 0) = (g(u'_0, 0), 0) = r'_0$. Suppose $r_{t-1} = r'_{t-1}$. Then at timestep t, C' receives u'_t, and $r'_t = (g(u'_t, s'_t), s'_t)$. Since $r_{t-1} = r'_{t-1}$, we know $s_t = s'_t$ (C and C' are convolutional codes). Furthermore, $u'_t = g(u'_t, s'_t) = u_t$. Thus $r_t = (u_t, s_t) = (g(u'_t, s'_t), s'_t) = r'_t$, so we have that $r_t = r'_t$ $\forall t \in [k]$. To see stream 1 is the same, observe that $\forall t \in [k]$ we have that $x_t^{(1)} = h(u_t, s_t) = h(g(u'_t, s_t), s_t) = \pi_1(H \circ H^{-1}(u'_t, s_t)) = \pi_1(u'_t, s_t) = u'_t = x_t^{(1)'}$. Since $r_t = r'_t$ $\forall t \in [k]$, we also see that $\mathbf{x}^{(2)} = \mathbf{x}^{(2)'}$. Therefore $\mathcal{C}(C') \subseteq \mathcal{C}(C)$, and we have that $\mathcal{C}(C') = \mathcal{C}(C)$. ■

Since for an NNCC, $(h_1, h_2, (u, s) \mapsto u, M)$ has the same set of codewords as $(h_2, h_1, (u, s) \mapsto u, M)$, the theorem also holds if any other encoding function has the desired property.