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

2022 IEEE International Symposium on Information Theory (ISIT) | 978-1-6654-2159-1/22/$31.00 ©2022 IEEE | DOI: 10.1109/ISIT50566.2022.9834599

***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].

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.

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 the TurboAE and TurboAE-binary models of [16], [17]. . 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[1].

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 [18], [19] – are replaced by CNN blocks, as in Fig. 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

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

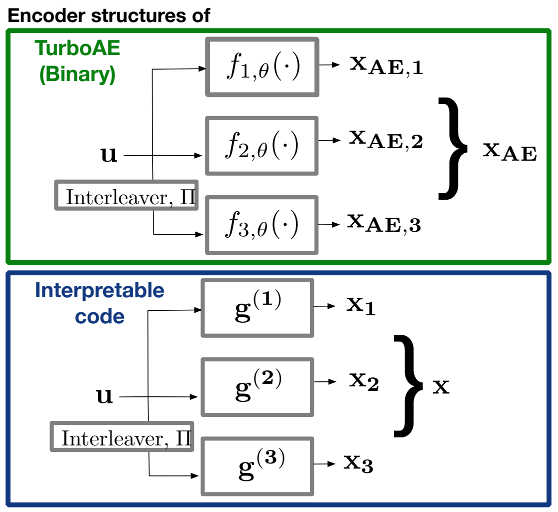

Fig. 1: The rate $R = \frac{1}{3}$ ($\mathbf{u} \in \{0,1\}^{100}, \mathbf{x}_{AE,j} \in \{\pm 1\}^{100}$) TurboAE-binary encoder structure. Functions $f_{j,\theta}(\cdot)$ are the constituent codes implemented as CNNs. Our interpretable codes either find compact Boolean representations for the learned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

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 [20], [21]. 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. 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, we interpret the encoder only) and the network is trained in an end-to-end fashion to obtain network parameters θ.

A. Modified Turbo codes

To develop the interpretation of TurboAE-binary we 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. 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 may be a general function $f_{j,\theta}^{block} : \{0,1\}^k \to \pm 1^k$. We look for the best modified convolutional code approximation to this arbitrary black box. 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:

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

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

A MILP can be obtained from (2) using a reformulation of binary arithmetic (see, e.g., [22]); 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 [23]. 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%, 1.5% and 24.3% on average respectively, including edge effects. Multiple equivalent solutions for block 3 exist and are related to the Fourier coefficients (see Section VI).

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 variable [24], which is a natural importance measure in our context (other measures used in XAI are described in [25]).

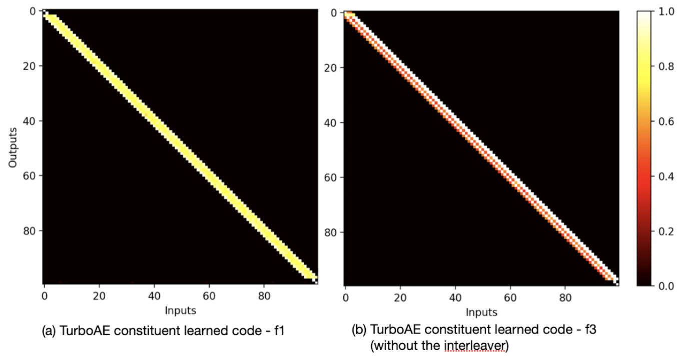

Fig. 2: TurboAE Constituent Code Influence heatmaps of (a) Block 1, (b) Block 3 without the interleaver.

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 $\{\pm1\}^n$) 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 (shuffled if interleaved). 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 $\chi_S = \prod_{i\in S} x_i$ [24]. 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 [26] 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

In *property testing* [27], the objective is to decide if an unknown black-box function has a property or if it is far from having the property. Property testing is intended to provide a preliminary, but very efficient test for the black box. 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. 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* [28] if for some $\epsilon' < \epsilon$ functions having distance at most ϵ' from the property are accepted with probability at least 2/3 as well. Tolerant property testing seems suitable for our context as an exploratory tool.

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 [29]. 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.

A *cyclically shifted multi-affine (CSMA)* function is a CSMP function, except parity functions are replaced by affine ones. As an affine function is either linear or its complement is, Theorem 1 can be extended to this case (see Appendix of [29]). Property testing is efficient but it does not provide the approximating parity or affine function (that requires further testing using the self-correction property).

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 encoding functions can be determined exactly by brute force, 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 2, now we make use of their values as well. The nonzero influence values in each row for block 1 (read right to left) are $(\frac{15}{16}, \frac{13}{16}, \frac{13}{16}, \frac{13}{16}, \frac{15}{16})$, for block 2 they are $(1, 0, 1, 1, 1)$, and for block 3 they are $(1, 1, \frac{1}{2}, 1, \frac{1}{2})$. 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 [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 [29]. The proof of part *a)* uses the edge isoperimetric inequality for the hypercube [33]. As Theorem 2 and Table I[2] show, TurboAE-binary's constituent codes are *nonsystematic* and *nonrecursive*. Blocks 1 and 3 are *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

[2]Linear part of Block 3 in Table I is one of the MILP solutions - 110100.

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	$u_1 \oplus u_2 \oplus u_4 \oplus \bar{u}_3\bar{u}_5$
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.

$\mathbb{P}(u_i = 1|\mathbf{y})$ for received message $\mathbf{y}$ [34] [35], whereas the 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 (NNCC) into recursive SCCs (RSCC) as in [37] since block 2 of both our exact and MILP approximated representations are parities. A rigorous formulation is in the Appendix of [29].

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 (unlike [9]), 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 Figure 3, we get a fully interpretable Turbo code (TurboAE Exact NNCC) that performs better than TurboAE-binary below SNR 0.5 by simply replacing the black-box decoder. However, TurboAE-binary's decoder outperforms BCJR above SNR 0.5. BCJR is not guaranteed to be optimal for Turbo codes, and these results suggest that TurboAE-binary's neural decoder has learned a better decoder for higher

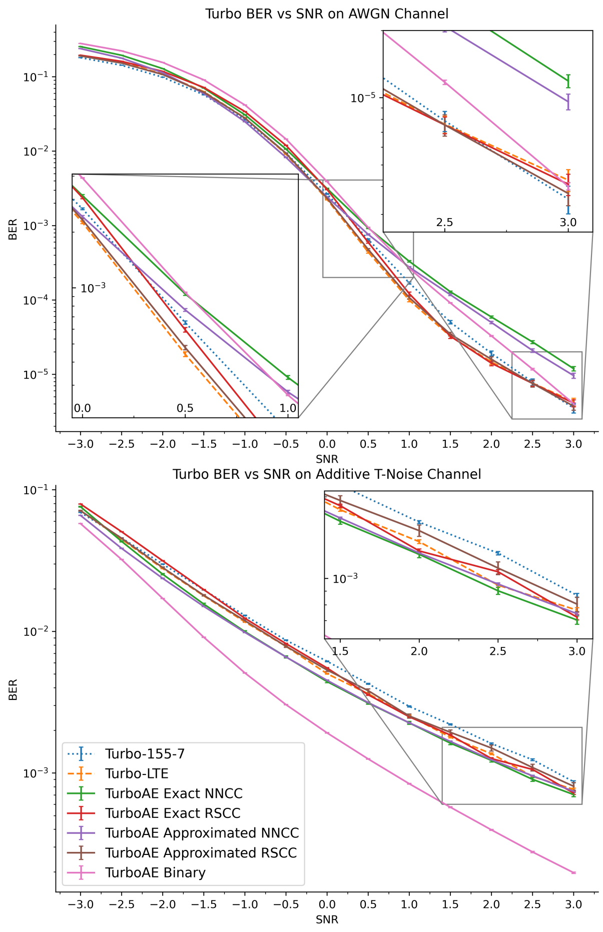

Fig. 3: Performance of TurboAE-binary, exact and MILP approximations of TurboAE-binary, and benchmark Turbo codes. Error bars are 2 standard deviations on the estimated mean BER. For TurboAE-binary, we used the weights provided by the authors of [9].

SNRs. On the ATN channel, TurboAE-binary significantly outperforms the other tested codes. TurboAE-binary's decoder appears unusually robust compared to BCJR, suggesting that it is not simply "approximating" BCJR. Although TurboAE's decoder has a similar message passing structure to iterative BCJR, each iteration involves a different neural network and it passes more features between iterations. It also uses information from encoded stream 1, unlike the BCJR decoding architecture for NCCs from [36].

For the AWGN channel the linear approximated representation performs just as well as the exact expression (and better on some SNRs). In future work, it would interesting to explore whether fine tuning TurboAE's decoder with the approximated representation improves performance of TurboAE.

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

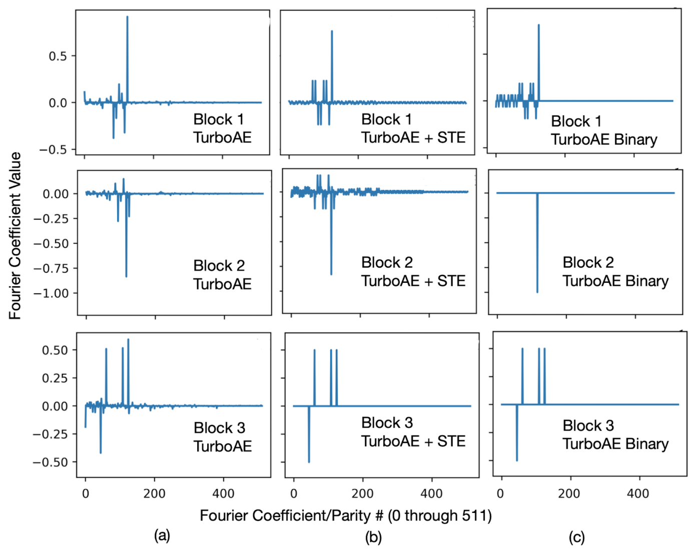

Fig. 4: Fourier Coefficients of Block Encoder functions at different stages of training. (a) Trained TurboAE, (b) Trained TurboAE with STE module, (c) Trained TurboAE Binary

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.

VII. Conclusion

We conclude our initial study of the interpretability of the TurboAE-binary with:

- TurboAE-binary is a nonlinear modified Turbo code with few nonlinear terms, approximated by a modified Turbo code can be found by MILP.
- Influence heatmaps provide valuable information.
- 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 serve as a stand-in for TurboAE's neural decoder.
- 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 investigation of the learned decoder and its apparent improvements on BCJR. 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] S. Lin and D. J. Costello, *Error Control Coding*. Pearson, 2005.

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

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

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

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

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

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

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

[26] 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.

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

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

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

[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.