Evaluating interpretations of deep-learned error-correcting codes

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

2022 58th Annual Allerton Conference on Communication, Control, and Computing (Allerton) | 979-8-3503-9998-1/22/$31.00 ©2022 IEEE | DOI: 10.1109/ALLERTON49937.2022.9929417

***Abstract*—Deep-learned error-correcting codes have recently been shown to match and sometimes improve upon analytically derived codes in certain regimes. One such code, TurboAE, mimics the structure of a Turbo code. In a recent paper, we interpreted the encoder of TurboAE and matched it with a BCJR decoder, creating a complete interpretation of the TurboAE code. In this paper, we expand on that work and study various combinations of deep-learned and interpretable encoders and decoders with the goal of evaluating the interpretations and identifying features of the code that benefit from deep learning. In doing so we pursue a novel direction in Explainable AI (XAI) research, whereby different components of a deep-learned system can be replaced with their interpretable counterparts to better understand aspects of the original system. The initial observations based on a single code suggest that further similar studies for other codes might be of interest. Our experiments on the deep-learned TurboAE suggest the following: the nonlinear nature of two out of three TurboAE constituent encoders does not seem to play a role, and neither do distinct encoding functions used for the boundary bits. We also observe that TurboAE performs poorly if it is converted to a recursive systematic code, to the point that it trails a randomly generated convolutional code. Further, while TurboAE trained on the AWGN channel performs remarkably well on the Additive T-noise channel, it is outperformed when traditional codes use the correct channel model for decoding. A discussion that attempts to provide some insight into these results is included.**

I. Introduction

A new path has recently surfaced in the development of error correcting codes: use machine learning to "learn" the encoders and/or decoders of error correcting codes [1]–[9]. While deep-learned error-correcting codes (DL-ECCs) are fully specified by the parameters of the neural network models, because of the large number of parameters, these deep-learned codes are often considered black boxes in the sense that it is not "understood" how/when they perform well or whether/if they relate to known codes.

In our prior work [10] we considered interpreting TurboAE-binary [1], one of the first complete DL-ECCs. We developed post-hoc interpretability techniques to analyze the deep-learned *encoders* of TurboAE codes, using influence heatmaps, mixed integer linear programming (MILP), Fourier analysis, and property testing. We were able to derive both exact nonlinear and approximate affine representations of the Boolean encoding functions. We compared the learned, interpretable encoders combined with BCJR decoders to the original black-box code. Our focus was quantitative in nature, looking at developing faithful interpretations. However, we did little to evaluate our interpretations in the bigger picture of evaluating the performance of such interpretations, which we initially attempt here.

The problem studied in [10] and in this paper can be viewed as a case study for Explainable AI (XAI)[1], the field dealing with understanding learned black-box models like deep neural networks, for applications in science and engineering. Motivations for obtaining such an understanding of DL-ECCs are discussed in some detail in [10].

In order to understand a black box model, the first questions are what notion of explanation to use, and how to find the required kind of explanation. An important next question is what are relevant criteria for evaluating the explanations obtained and how to evaluate the explanations according to those criteria, see, e.g., [11]. Providing formal definitions of these criteria, and experimental and theoretical work on them is an active area of current research.

Explanations can either be local (explaining the output on a fixed input) or global (explaining the function computed by the learned model). Evaluation criteria have mostly been studied for local explanations. Some of the basic criteria are faithfulness of the explanations with respect to the model and robustness with respect to perturbations of the input.

In the context of DL-ECCs, one is mainly interested in global explanations. In [10] we found global explanations of TurboAE encoder in the form of modified Turbo codes with general (nonlinear) and affine constituent codes.

One can consider a channel code as a system with two components (the encoder and the decoder) in an environment (the channel), solving a task (communication). This is a special

This work was supported by NSF under awards 1705058 and 1934915, and the European Union project RRF-2.3.1-21-2022-00004 within the framework of the Artificial Intelligence National Laboratory (MILAB), Hungary. Computing resources for the experiments were provided in part by the NSF award 1828265 (COMPaaS DLV). This work was done in part while Gy. Turán was visiting the Simons Institute for the Theory of Computing.

The first 2 authors contributed equally.

[1]Previous papers on DL-ECC used the term interpretability and we follow this usage in most of the paper. In the XAI literature, it is more common to talk about explanations instead. In the brief discussion of XAI below we use such common terms as local and global explanations, but the remainder of the paper will generally use "interpretations."

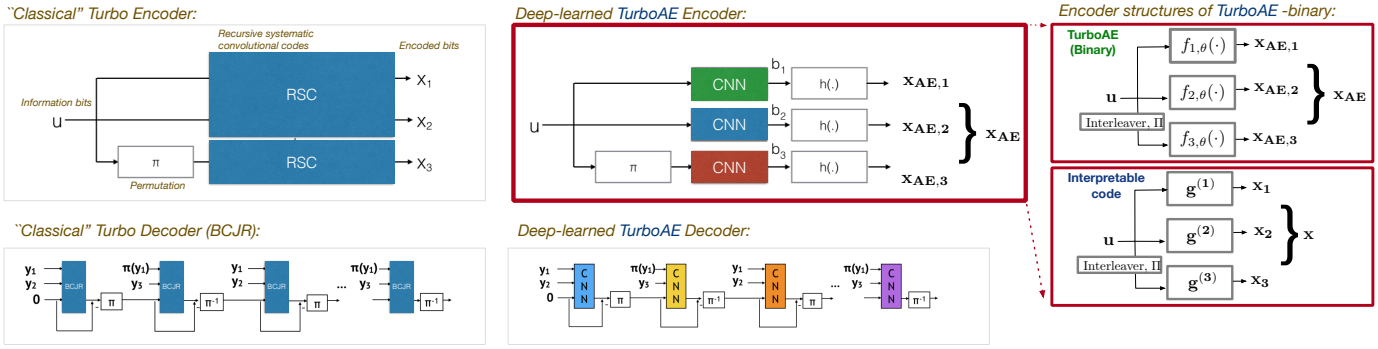

Fig. 1: The rate $R = \frac{1}{3}$ ($\mathbf{u} \in \{0,1\}^{100}, \mathbf{x}_{AE,j} \in \{\pm 1\}^{100}$) classical Turbo and TurboAE encoder and decoder structures. The recursive systematic (RCS) convolutional codes and the BCJR decoding blocks are both replaced by convolutional neural network (CNN) blocks. Functions $f_{j,\theta}(\cdot)$ are the constituent codes implemented as CNNs. Our interpretable codes either provide compact Boolean representations for the learned functions $f_{j,\theta}$ or approximate them with modified convolutional codes $\mathbf{g}^{(j)}$. Some images adapted from [1].

case of a system with black boxes for some components, which can then be explained by replacing (some of) those with interpretable components.

Evaluating the quality of those explanations goes hand in hand with trying to understand the reasons for the good performance of the black-box system. Thus evaluating explanations has the potential to be useful for the original discipline. For DL-ECCs, exploring the role of nonlinearity, trying to clarify the gains obtained from learning the encoder, resp., the decoder, and studying robustness with respect to the environment (i.e., channel noise) are all questions which have been studied in coding theory.

In this paper we describe experiments which pair our interpretable TurboAE encoders with BCJR decoders, as initially started in our prior work [10], or with new fine-tuned (extra training starting from TurboAE's original parameters) deep-learned decoders. The observations include:

- Boundary terms do not meaningfully affect the performance.
- Non-linearities of the encoding function do not appear to matter much for TurboAE's performance.
- Non-systematic, non-recursive codes are learned by TurboAE, as dictated by the CNN architecture. However, systematic, recursive forms of the exact or approximated TurboAE encoders paired with a matching BCJR decoder outperform TurboAE.
- Robustness against changes in channel: in [1] it was shown that the CNN decoder performs better than the BCJR decoder on the additive T-noise channel when the CNN is trained on the AWGN channel, and the BCJR decoder (incorrectly) assumes AWGN statistics. However, we observe that if we *do* know the channel statistics, e.g., that the channel is additive T-noise rather than AWGN, we can modify BCJR accordingly and significantly improve performance; as a result this interpretable decoder outperforms TurboAE.

These tentative conclusions could be studied in other contexts. The questions raised provide examples of the general types of insights XAI could offer when integrating deep-learning based results into the discipline where they are applied.

II. Background

In this section, we briefly review relevant material from our prior work [10].

A. TurboAE encoder

The architecture of the TurboAE encoder network [1] is based upon a classical rate $1/3$ Turbo code, with the three constituent codes being replaced by CNN blocks, as in Fig. 1. Similarly, the TurboAE decoder architecture replaces the iterations of the BCJR decoder by CNNs. The network is trained in an end-to-end fashion to obtain the network parameters of the encoder and decoder CNNs jointly. The network has two versions, TurboAE (with real-valued encoder outputs) and TurboAE-binary (with Boolean encoder outputs). In this paper, we discuss the latter, and for simplicity we drop the "binary" suffix.

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 $\mathbf{x}_{AE,j} \in \{\pm 1\}^{100}$. For simplicity, we omit details of power control modules. In [10] we also ignored the "boundary" effects, i.e. the treatment of the first 2 and last 2 bits of each block. These are discussed in detail in Section II-B.

The structure of the CNN blocks (see Fig. 2) implies that the constituent codes of block j implement a Boolean function $f_{j,\theta} : \{0,1\}^9 \to \pm 1$ of window 9 applied to bits $i-4 : i+4$ of u to produce the bit i of $\mathbf{x}_{AE,j}$.

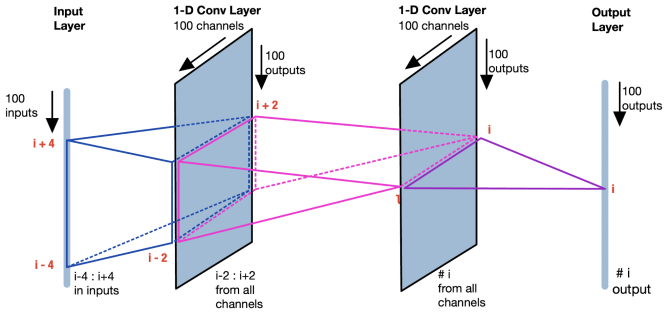

Fig. 2: Computation of i'th output of TurboAE Encoder Blocks: output i depends only on inputs $i-4 : i+4$. Figure taken from [10]

B. Approximate and exact TurboAE encoding functions

In [10] we found functions $\mathbf{g}^{(j)}, j \in \{1,2,3\}$ that either exactly or approximately represent the TurboAE constituent codes $f_{j,\theta}$. Our analysis showed that the encoder functions depend on 5 variables only (the encoder has memory 4 plus the 1 input bit). We first determined their best (in terms of the Hamming distance) affine approximations. In block-3 there are four best approximations (Table I). Replacing the constituent codes in each block with an affine approximation one gets a *modified Turbo code*, containing *modified convolutional* blocks. A *modified convolutional code* is *nonsystematic*, *nonrecursive*, involving *affine* functions instead of linear ones, and also including a *shift*, i.e., an output bit may depend on future input bits. See [10] for more details on these terms.

Block #	Approximate expressions for output #i
1	$1 \oplus u_1 \oplus u_2 \oplus u_3 \oplus u_4 \oplus u_5$
2	$u_1 \oplus u_3 \oplus u_4 \oplus u_5$
3	Solution 1: $u_1 \oplus u_2 \oplus u_4$ Solution 2: $1 \oplus u_1 \oplus u_2 \oplus u_3 \oplus u_4$ Solution 3: $1 \oplus u_1 \oplus u_2 \oplus u_4 \oplus u_5$ Solution 4: $1 \oplus u_1 \oplus u_2 \oplus u_3 \oplus u_4 \oplus u_5$
where	$u_1 = x_{i+2}$, $u_2 = x_{i+1}$, $u_3 = x_j$, $u_4 = x_{i-1}$, $u_5 = x_{i-2}$, $x =$ inputs

TABLE I: Best affine approximations for TurboAE's encoder. Block 3 has four equally good approximations.

We also determined the exact $\mathbb{F}_2$-polynomial representations of the encoder functions $f_{1,\theta}, f_{2,\theta}, f_{3,\theta}$. These are given in Table II. The polynomials are written in a non-standard *unate* form [10], allowing for negated variables, as this more general representation turns out to be natural in this context. Eliminating negations shows that solution 4 in Table I is the affine part of the exact representation for block 3.

Block #	Expression for output #$i \in \{3,4,\cdots,98\}$
1	$1 \oplus u_1 \oplus \bar{u}_2 \oplus u_3 \oplus \bar{u}_4 \oplus u_5$ $\oplus\ \bar{u}_2 u_3 \bar{u}_4 \oplus u_1 \bar{u}_2 u_3 \bar{u}_4 u_5$
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	$u_1 = x_{i+2}$, $u_2 = x_{i+1}$, $u_3 = x_i$, $u_4 = x_{i-1}$, $u_5 = x_{i-2}$, $x =$ inputs

TABLE II: Exact expressions for TurboAE Encoder's non-boundary bits.

The results above as first obtained in [10], apply to bits 3-98. The *boundary bits* have not been considered in [10]. In a regular *non-recursive* convolutional code block $B : \mathbb{F}_2^L \longrightarrow \mathbb{F}_2^L$ of length L and memory M, every output bit i implements some function $g_i : \mathbb{F}_2^M \to \mathbb{F}_2$ of the inputs $u^{(i)}$ in its fan-in, and if some part of $u^{(i)}$ falls outside of the input block range $1:L$, then that part can be set to constant 0 while evaluating g_i, e.g. for the bits at the boundaries of the block. All g_i's are the same and called the generator function $\mathbf{g}$ for the block. This allows the use of the function $\mathbf{g}$ as a convolutional filter to compute the output bits by appropriately *0-padding* the input block to a final length of $L + M - 1$.

Exact expressions for output #$i \in \{1,2,99,100\}$				
Block	i=1	i=2	i=99	i=100
1	$1 \oplus u_3$	$1 \oplus u_3$	$1 \oplus u_3$	$1 \oplus u_3 \oplus u_3 u_4 \bar{u}_5$
2	$1 \oplus u_1 \oplus u_3$	$1 \oplus u_1 \oplus u_3$	$1 \oplus u_3 \oplus u_5$	$1 \oplus u_3 \oplus u_5$
3	$u_1 u_3$	$u_1 \oplus u_2 \oplus u_3 \oplus u_4 \oplus u_1 u_2 \bar{u}_3 \bar{u}_4 \oplus \bar{u}_1 \bar{u}_2 \bar{u}_3 \bar{u}_4$	$1 \oplus u_3$	$1 \oplus u_3$
where	$u_1 = x_{i+2}$, $u_2 = x_{i+1}$, $u_3 = x_i$, $u_4 = x_{i-1}$, $u_5 = x_{i-2}$, $x =$ inputs			

TABLE III: Exact expressions for TurboAE encoder's boundary bits.

In TurboAE however, *0-padding* is not done just at the inputs, but at the first hidden layer as well. Because of the *0-padding* at the hidden layer, the g_i's are not all same. Only $g_3, g_4 \cdots g_{98}$ are the same, say g, because the *0-padding* in the hidden layer does not appear in their fan-in. $g_1, g_2, g_{99}, g_{100}$ are different from the rest of the bits and are referred to as *boundary bits*. We determined these boundary bit functions to be those in Table III. Hence, there is no single generator function that can be used as a convolutional filter for TurboAE block. When g is used as the generator with appropriate *0-padding* just at the inputs, the boundary bits are not the same as in the output of TurboAE.

Another minor technical difference between a regular convolutional code block, and TurboAE encoder's blocks is the layout of *0-padding*. In a convolutional code block, all the *0-padding* of width $M-1$ (= 4 here) would be done at the start of the input block, whereas in TurboAE 2-bit wide *0-padding* is done at both ends of the input. When viewed as a convolutional code, this is equivalent to running the encoder for 2 steps before we start transmitting, and padding 0's once we run out of input bits. Thus TurboAE encoder's output has a shift of 2-bits as compared to a regular convolutional code block. So any generic decoding algorithms like BCJR, when paired with TurboAE's encoder must adapt for this shift.

C. TurboAE's decoder

The decoder of TurboAE can be viewed as a single, large convolutional neural network with some internal structure incorporated to mimic iterative decoders for Turbo codes (Fig. 1). The convolutional neural network decoder structure pairs well with the non-recursive nature of the encoder, capturing the inductive bias that input bits should be decoded from local information in the received transmission.

In [10], and as briefly outlined above, we obtained exact non-linear and approximate affine representations / explanations of the TurboAE's constituent encoder functions, yielding explainable encoders. However, to obtain an interpretable code, one needs both an interpretable encoder and an interpretable decoder. To do so, we paired our exact and approximate encoders with BCJR decoders whose trellises

are provided by the exact or approximate encoding functions. Note that because of the different boundary functions and shift on the encoder (see Section II-B, [10]), the BCJR decoding algorithm needs slight modifications. Boundary functions can be accommodated by modifying the encoder's trellis structure for bits 1, 2, 99, and 100. We also can modify BCJR to handle a shifted encoder by adjusting its prior on the initial and final states of the encoder. We consider BCJR an interpretable decoder as it is based on a well-studied algorithm [12].

III. Empirical Results

We now examine a series of questions that help evaluate our interpretations of TurboAE[2].

To compare our codes with TurboAE, we estimate BERs on various channels w.r.t. uniformly chosen binary blocks of length 100, and channel noise. For all encoders, we use TurboAE's 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 that are 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 consider a series of questions by comparing a number of empirical results between the original TurboAE encoder-decoder pair with various combinations which pair an encoder or a decoder as follows:

- Encoder:
  - TurboAE-original: the original CNN-based encoder and decoder of [1]
  - TurboAE-exact: our exact representation of the Boolean function as in Table II (with the boundary functions of Table III or usually without)
  - Affine: the best found affine approximations of the encoder described in Section II-B and Table I
  - TurboAE-exact or Affine turned into Recursive Systematic form (RSC), as described in Section III-C
- Decoder:
  - CNN: original TurboAE-decoder from [1] fine-tuned (extra training starting from the original TurboAE CNN's parameters) to match the chosen encoder.
  - BCJR decoder with a trellis matched to the encoder and using a specific noise model (AWGN or T-noise).

A. Effect of boundary bits

While evaluating the interpreted codes we have a choice to use an encoder *with* special boundary bits, or *without* special boundary bits by extending the non-boundary functions over the boundary bits as well. In this section, we evaluate both variants and compare their performance. We pick the TurboAE's exact encoder *with* and *without* special boundary bits, and pair them with BCJR or TurboAE's CNN decoder. Since the CNN decoder was tuned to the exact encoder *with* special boundary bits by the training process, we fine-tuned a different CNN decoder to the exact encoder *without* boundary functions to study their impact on a CNN decoder.

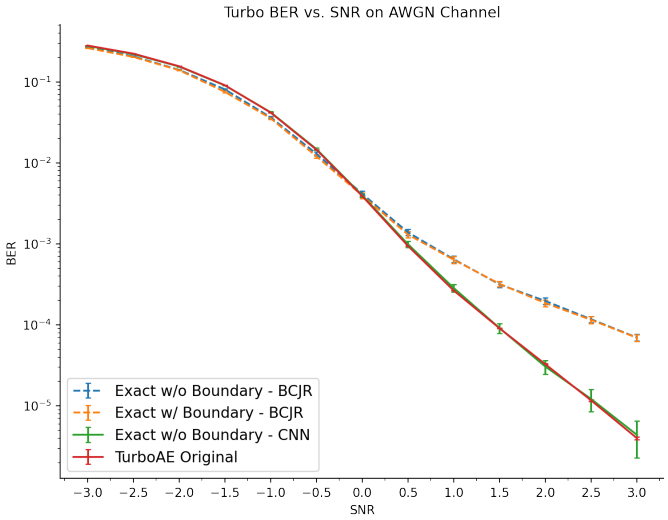

Fig. 3: Performance comparison of special *boundary* bits in the TurboAE's exact encoder. Solid curves for CNN decoder, dashed curves for BCJR decoder. For both decoders there is minimal difference between *with* and *without boundary* bits in the encoder.

We see in Figure 3 that removing the special boundary bits from the encoder led to a negligible change in BER with both the decoders. Thus having special boundary bits seems of limited utility, and it appears to be the result of an inductive bias due to the *0-padding* of the hidden layer in the encoder (see Section II-B).

From here on, since the boundary functions did not have a significant impact, we ignore them when looking at further questions: all new decoders are based on encoders *without* boundary functions. That is, we look at performance using the exact encoder from table II or the approximations from table I. All CNN-based decoders are initialized with the original TurboAE's CNN decoder and fine-tuned to the interpretable encoders. All BCJR decoders *do not* use shift so we have comparable performance with the CNN-based decoders. Although this produces subtle differences in the encoders, the change is justified because the properties of the encoder we want to study in the later sections are not tied to shift, unlike our specific boundary functions.

B. Impact of non-linearities and choice of linear approximation

We look at a) the effect of the non-linearities in TurboAE's encoding function which is coupled with b) the impact of the choice of the affine approximations to TurboAE's encoder block 3 on performance, and c) the impact of re-training the learned decoder for the different approximate solutions.

[2]Code for experiments at https://github.com/tripods-xai/allerton-2022. Experiments were written in Python [13] and made use of TensorFlow [14], PyTorch [15], and CommPy [16].

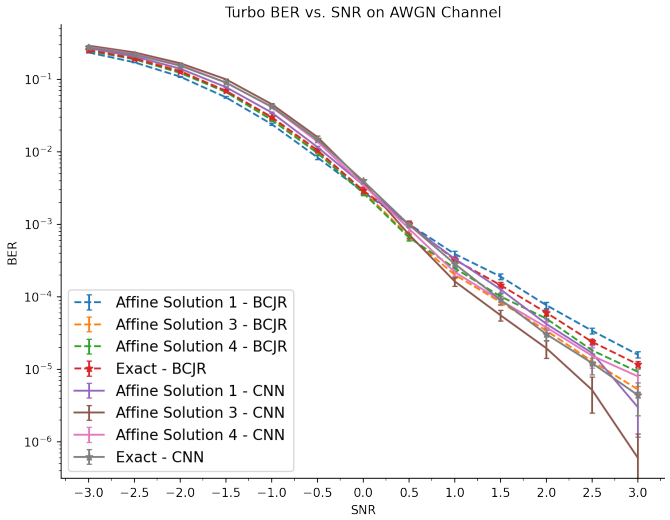

Fig. 4: Comparison of TurboAE's nonlinear encoder with its affine approximations. Solution 3 is the best-performing approximation and solution 1 the worst. Each is paired with either a BCJR decoder (dashed line) or a CNN decoder (solid line). Exact encoders' BER points are marked with stars. Solution 1 consistently underperforms TurboAE, while Solution 3 consistently outperforms it. CNN decoder consistently outperforms the BCJR decoder above SNR 0.5 when paired with the same encoder.

Figure 4 contrasts the BER of the TurboAE exact encoder with 3 of the 4 possible affine approximations when paired with either a CNN or BCJR decoder. We observe that the CNN decoder consistently outperforms the BCJR decoder above SNR 0.5 when paired with the same encoder. We also observe that one of the best affine approximations (solution 3) of the TurboAE encoder outperforms the original TurboAE encoder, both when paired with the CNN or the BCJR decoders [3]. On the other hand, another best affine approximation (solution 1) performs worse than the original TurboAE encoder.

The results suggest that the nonlinearities are not the main reason for the performance gains of the TurboAE code. This also suggests that TurboAE did not find a global optimum, and could be further improved by new training techniques. Additional evidence that this may be the case comes from observing that the truth table for the encoder function of block 3 is actually not balanced: it has 15 ones and 17 zeros. Finally, without further investigation we can not say whether the code that is further optimized will turn out to be affine or nonlinear.

C. Conversion to RSC

We saw that non-linearities do not appear to play a significant role in the performance of TurboAE. While the architecture of TurboAE allows for learning non-systematic non-recursive convolutional codes and prohibits the learning of recursive convolutional-like codes, one question to ask is whether, as one might generally expect given the literature [12], turning our exact or affine TurboAE encoder interpretations into recursive systematic forms, paired with a BCJR decoder, leads to better performance.

[3]One might expect that solution 4, the affine part of the exact expression, would perform best, but this turns out not to be the case.

We first address how to convert our non-systematic non-recursive exact and approximate TurboAE encoder interpretations into recursive systematic code (RSC) format. We do this by presenting a generalized conversion from the nonrecursive, nonsystematic version of TurboAE's exact encoder to a recursive, systematic code with the same set of codewords. First, we clarify what we mean by a generalized (potentially non-linear) convolutional code.

Definition 1. *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 the memory,* $h_1, \ldots, h_n : \mathbb{F}_2 \times \mathbb{F}_2^M \to \mathbb{F}_2$ *are the encoders, and* $g : \mathbb{F}_2 \times \mathbb{F}_2^M \to \mathbb{F}_2$ *is the 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 nonrecursive convolutional code to an RSC:

Theorem 2. *Let* $C = (h_1, h_2, (u,s) \mapsto u, M)$ *be a nonrecursive rate* $1/2$ *convolutional code 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.* $C' = ((u,s) \mapsto u, h_2, g, M)$ *has the same set of codewords as* C.

This result generalizes the standard construction (see, e.g., [12]); the proof is omitted due to space constraints.

As somewhat expected, Figure 5 shows that converting nonrecursive encoders to recursive ones shows dramatic improvement when paired with a BCJR decoder. Note, however, that this trend is not ubiquitous. Recall that in Section III-B we saw that some affine solutions for block 3 worsen performance. Here we see the same effect. The recursive variant of solution 1 shows even poorer performance than its nonsystematic counterpart.

In addition, we have compared our tested recursive codes against two benchmark recursive codes: Turbo-155-7 and TurboLTE described by:

- 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.

At SNR values above 1.5, both the recursive variants of the TurboAE exact encoder and affine solution 4 paired with BCJR encoder significantly outperform the benchmarks.

D. Comparison with Randomly Sampled Turbo Codes

In the previous sections, we saw that approximations of TurboAE's encoder with affine encoders can produce an improvement over the original code. However, is this improvement tied to the original encoder, or are these encoders' performance typical of other affine encoders of the same memory? To answer this question, we randomly select five affine turbo codes with memory 4 and compare their performance with our exact and approximated encoders when paired with BCJR. In

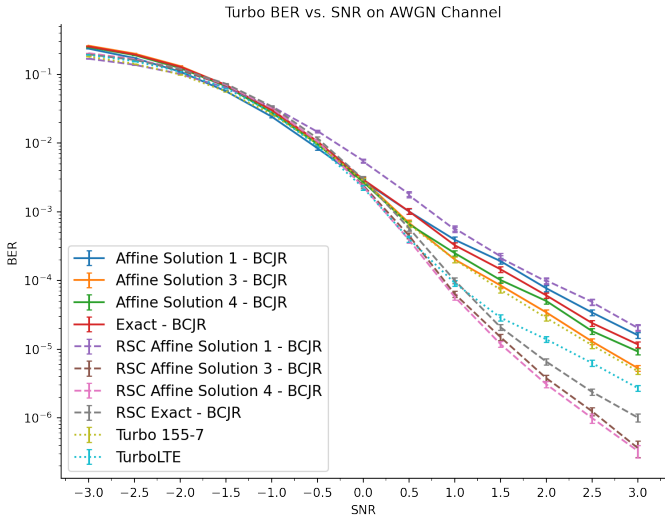

Fig. 5: Comparison of nonrecursive encoders (solid lines) to their RSC variants (dashed lines) and to benchmark codes (dotted lines). All RSC variants show improvement over nonsystematic counterparts except for affine solution 1. Both the RSC variants of the exact encoder and affine solutions 3, 4 show improvement over benchmarks at SNRs above 1.5.

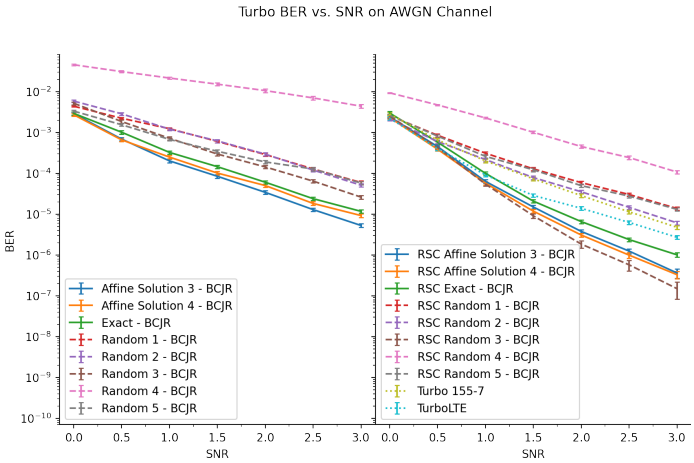

Fig. 6: *Left*: Comparison of 5 random affine Turbo codes (dotted) with our exact encoder and the best affine solutions 3, 4 (solid). Exact and affine encoders outperform all random codes. *Right*: Comparison of random codes converted to RSC (dotted) with exact's and affine solutions' RSC variants (solid) and benchmarks (dotted). Our encoders are no longer the best; random code 3 shows significant improvement over both our RSCs and benchmarks.

Figure 6, we compare the random codes' original nonsystematic forms against our TurboAE-derived encoders. Then we convert them to RSC using the transformation described in Section III-C and compare against our TurboAE-derived RSC encoders.

The results in Figure 6 highlight two important observations: (1) the nonsystematic exact and approximated affine encoders outperform all 5 of the randomly sampled encoders (2) while the RSC variant of our encoder performs well, we were still able to find a better performing random code. It is not surprising that TurboAE's encoder outperforms the nonsystematic random encoders; TurboAE was trained as a nonsystematic, nonrecursive code, so we expect its encoder to perform better than average. On the other hand, converting a code to RSC changes the structure of its trellis, so it does not necessarily follow that a good nonsystematic, nonrecursive code would be a good recursive code. We already saw in figures 4 and 5 that solution 3 was best out of the nonsystematic approximations while solution 4 became best when converted to recursive codes. Comparing with the benchmarks, these results suggest that the improvement of our RSCs over the benchmarks may be a result of increased memory; Turbo-155-7 uses memory 2 and TurboLTE uses memory 3. On the other hand, comparing with the random RSCs, the results also suggest that we may be able to learn better Turbo codes by training an RSC-like code instead. We leave this for future work.

E. Robustness to Channel

We now investigate the observation made in [1] that the learned encoder-decoder TurboAE pair appears more robust to incorrect channel statistics. That is, TurboAE trained in AWGN appears to outperform existing codes employing AWGN statistics on other channels. We explore this by comparing the performance of various codes over the Additive-T-noise channel first presented in [1].

We first reproduce the results presented in [1], where all BCJR decoded codes assume (incorrectly) that the channel is AWGN when computing symbol transition probabilities. We also benchmark TurboAE without any fine-tuning to the particular channel used. As seen in Figure 7, even without fine-tuning TurboAE shows significant improvement over BCJR. Since TurboAE's decoder also performed competitively when compared against BCJR in Figure 4, as far as we can tell, TurboAE is *not* sacrificing AWGN channel performance for robustness.

On the other hand, we also compare with a BCJR decoder that correctly assumes the channel is Additive-T when computing the symbol transition probabilities. In Figure 8 we see, as expected, that correctly tuned BCJR outperforms TurboAE. In other words, TurboAE is more robust than the BCJR decoder if one assumes that the decoder can not be adapted to the underlying channel statistics. However, given the information about channel statistics (see also [17]), the BCJR decoder can directly use it, and as our plots show, will significantly outperform the CNN decoder (even if the latter is more robust). Note that the CNN decoder needs to be retrained if the channel model changes, so the channel information, when available, cannot be as readily used to improve its performance.

IV. Discussion

Our experiments elucidate various subtleties regarding the question of whether the encoder or the decoder is responsible for the performance. On the one hand, simply replacing the CNN decoder with a BCJR decoder as in Fig. 3 and Fig. 4 demonstrates that fine-tuned learned decoders outperforms their BCJR counterpart pairwise for the exact same encoders. However, our affine interpretations of the encoder

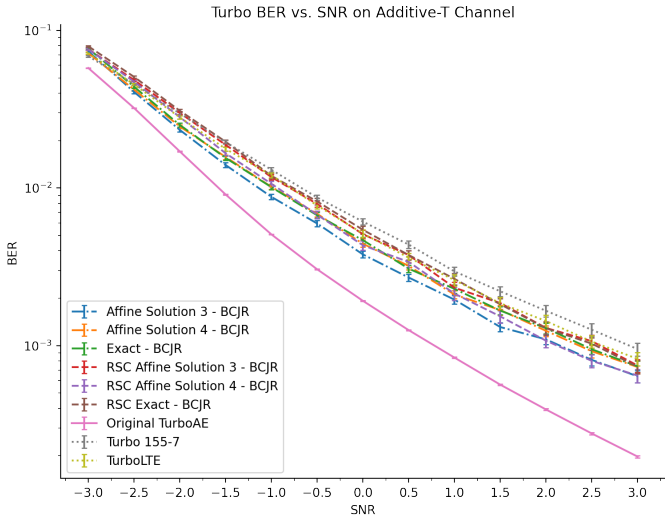

Fig. 7: Comparison of codes transmitted over the ATN channel with *(1)* a BCJR decoder using (incorrect) AWGN statistics (dashed lines), *(2)* TurboAE's original CNN encoder and decoder, no fine-tuning (solid line) and *(3)* two benchmarks using AWGN statistics (dotted). TurboAE shows significant robustness over all BCJR codes that use AWGN statistics.

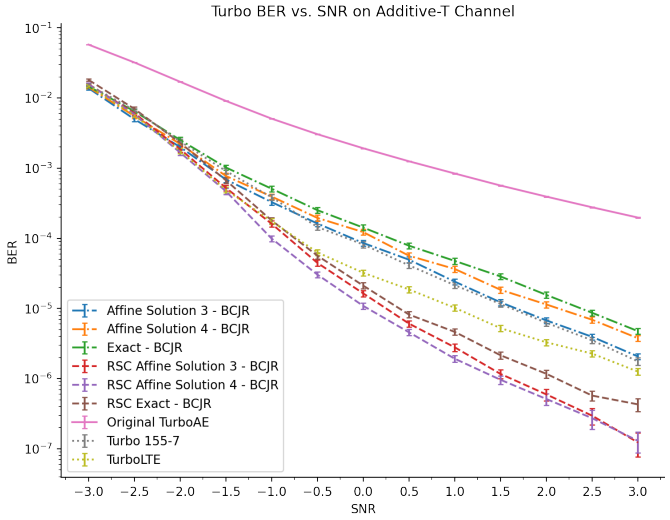

Fig. 8: Comparison of codes transmitted over the ATN channel with *(1)* TurboAE's original CNN encoder and decoder, no fine-tuning (solid line), *(2)* a BCJR decoder using correct Additive-T statistics (dashed lines) and *(3)* two benchmarks using ATN statistics (dotted). There is still a significant gap between TurboAE and BCJR when BCJR uses correct ATN statistics.

have remarkably good performance: these non-systematic non-recursive codes (affine solution 3 paired with BCJR) perform similar to the benchmark RSC Turbo 155-7 (see Fig. 5), with the RSC version of our affine approximate solution 4 outperforming all benchmarks. An interesting further experiment would be to compare the RSC affine Solution 4 encoder paired with the BCJR decoder versus with a retrained CNN decoder. The results suggest that TurboAE's encoder (including its non-linearities) is not responsible for TurboAE's performance, and that its decoder is making up for the non-systematic non-recursive nature of the encoder, which is a result of the choice of the learned architecture.

Another important question is whether simultaneously optimizing the encoder and the decoder leads to a better performing code than when the two are designed independently. Most DL-ECCs are trained by optimizing the cross-entropy (CE) with the idea that this will also minimize the BER[4]. We thus next try to gain further insight into the question above by focusing on the CE.

Suppose we start with a random variable $U \in \mathbb{F}_2^k$ sampled uniformly that is encoded via $f : \mathbb{F}_2^k \to \mathbb{R}^n$. The receiver receives a random variable $Y \in \mathbb{R}^n$ (for $n = k/R$ for rate R) after channel noise has been applied. Our goal is to find our encoder f paired with a *soft* decoder $g : \mathbb{R}^n \to [0,1]^k$ that minimizes the average binary cross entropy of the decoded bits. That is, we want to minimize $BCE_f(g) = -\frac{1}{k}\sum_{i=1}^k E[1_{U_i=1}\log g(Y)_i + 1_{U_i=0}\log(1-g(Y)_i)]$. The binary cross-entropy admits a decomposition in terms of KL-divergence and binary (conditional) entropy (e.g. [19]), as

$$BCE_f(g) = \frac{1}{k}\sum_{i=1}^k E[D_{KL}(\mathbb{P}[U_i|Y]||g(Y)_i)] + \frac{1}{k}\sum_{i=1}^k \mathbb{H}(U_i|Y),$$

where $\mathbb{H}(\cdot|\cdot)$ is the conditional entropy function, and $D_{KL}(\cdot||\cdot)$ is the Kullback-Leibler Divergence. Using Gibb's Inequality, we see the first term is minimized when $g(Y)_i = \mathbb{P}[U_i = 1|Y]$. The second term does not depend on g. Indeed, in a typical, supervised learning binary classification problem, we would treat $\mathbb{H}(U_i|Y)$ as a constant. However, in our case, Y depends on the encoder f. Since for any encoder, we can express its optimal soft decoder as $\mathbb{P}[U_i = 1|Y]$ (and the minimal KL divergence is 0), this tells us the optimal (from a machine-learning perspective minimizing CE) encoder-decoder pair is really determined by an encoder that minimizes $\mathbb{H}(U_i|Y)$.

In the training of TurboAE [1] the authors note that alternating the training of the encoder for fixed decoder and vice versa helps the deep-learning algorithm converge. This can be related to the above decomposition in the following sense: perhaps it is "easier" for the optimization algorithm which minimizes the sum of two components to minimize one component at a time, and alternate between them. This is what holding the encoder (or decoder) fixed and optimizing / learning the parameters of the others does.

This decomposition further suggests that the design of the encoder and decoder, in a deep-learning setting under CE, can be decoupled provided a maximum likelihood (ML) decoder can be designed, echoing somewhat what we know from coding theory (but cannot always implement as it is known that finding the ML decoder is in general NP-hard). One can further speculate that deep-learned decoders minimizing the CE try to find the ML solution, which is backed up by recent results such as [20], where Gaussian-like constellations based on the Golden ratio, random Gaussian codes, and a deep-learning decoder was learned and shown empirically to

[4]There have been attempts to show this formally [18] but it appears that this is not true without additional assumptions.

yield near-maximum-likelihood (ML) performance for small constellations and blocklengths. This suggests that a worthwhile further direction would be to explore optimizing the encoder separately to minimize this entropy, and the decoder separately once the encoder has settled and investigate whether this achieves close to ML performance.

V. Conclusions

We presented numerous refinements of our initial interpretations in [10]. Many interesting observations can be made, among others that boundary terms appear to not impact performance, non-linearity does not appear to play a role in performance of TurboAE, TurboAE seems to have learned a good non-systematic code but recursive systematic codes with BCJR decoders outperform the original TurboAE (binary version), and any robustness against channel statistics should also consider the possibility of adapting the decoder if that information is known.

We have studied a single DL-ECC, and similar experiments for other codes and communication scenarios would also be of interest. The general questions involved, such as the role of non-linearity and robustness with respect to channel noise, are important questions for coding theory and communication. Thus findings along these lines could lead "back to science" from interpretability, as discussed briefly in the introduction.

Combining channels and deep-learned encoders and decoders has been discussed earlier [1]. In this paper we used this approach in a *cut-and-paste* manner with interpretable encoders and decoders to get information about both deep learning and coding theory. A similar approach could be used to try to understand systems with interpretable components. This would provide one possible direction to use interpretability for addressing the challenges posed by the black box nature of machine learning methods.

References

[1] Y. Jiang *et al.*, "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.

[2] H. Kim, S. Oh, and P. Viswanath, "Physical Layer Communication via Deep Learning," *IEEE Journal on Selected Areas in Information Theory*, vol. 1, no. 1, pp. 5–18, 2020.

[3] 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.

[4] 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.

[5] Y. Jiang *et al.*, "MIND: Model Independent Neural Decoder," in *2019 IEEE 20th International Workshop on Signal Processing Advances in Wireless Communications (SPAWC)*, Jul. 2019, pp. 1–5.

[6] J. Whang *et al.*, "Neural Distributed Source Coding," May 2022. [Online]. Available: http://arxiv.org/abs/2106.02797

[7] R. K. Mishra *et al.*, "Distributed Interference Alignment for K-user Interference Channels via Deep Learning," in *2021 IEEE International Symposium on Information Theory (ISIT)*, Jul. 2021, pp. 2614–2619.

[8] 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.

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

[10] N. Devroye *et al.*, "Interpreting Deep-Learned Error-Correcting Codes," in *2022 IEEE International Symposium on Information Theory (ISIT)*, Jun. 2022, pp. 2457–2462.

[11] G. Vilone and L. Longo, "Notions of explainability and evaluation approaches for explainable artificial intelligence," *Information Fusion*, vol. 76, pp. 89–106, Dec. 2021.

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

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

[14] M. Abadi *et al.*, "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015. [Online]. Available: https://www.tensorflow.org/

[15] A. Paszke *et al.*, "PyTorch: An Imperative Style, High-Performance Deep Learning Library," in *Advances in Neural Information Processing Systems*, 2019, vol. 32. [Online]. Available: https://proceedings.neurips.cc/paper/2019/hash/bdbca288fee7f92f2bfa9f7012727740-Abstract.html

[16] V. Taranalli, B. Trotobas, and contributors, "CommPy: Digital communication with Python." [Online]. Available: https://github.com/veeresht/CommPy

[17] W.-C. Tsai *et al.*, "Neural Network-Aided BCJR Algorithm for Joint Symbol Detection and Channel Decoding," in *2020 IEEE Workshop on Signal Processing Systems (SiPS)*, Oct. 2020, pp. 1–6.

[18] E. Balevi and J. G. Andrews, "Autoencoder-Based Error Correction Coding for One-Bit Quantization," *IEEE Transactions on Communications*, vol. 68, no. 6, pp. 3440–3451, Jun. 2020.

[19] D. Ramos *et al.*, "Deconstructing Cross-Entropy for Probabilistic Binary Classifiers," *Entropy*, vol. 20, no. 3, p. 208, Mar. 2018.

[20] B. He, Z. Wu, and F. Wang, "Rethinking: Deep-learning-based demodulation and decoding," 2022. [Online]. Available: https://arxiv.org/abs/2206.06025