Decomposing the Training of Deep Learned Turbo codes via a Feasible MAP Decoder

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

***Abstract*—Most deep-learned error-correcting codes (DL-ECCs) use binary cross-entropy (BCE) between the true input bits and the soft decoded outputs of the learned encoder/decoder pair as a loss function during training. It is known that this decomposes into two terms which in the case of a DL-ECC correspond to: a Kullback-Leibler (KL) divergence between the true and estimated posteriors on the input bits given the noisy channel outputs, and the conditional entropy (CE) of the input bits given the channel outputs. We use this decomposition to explore the training process of one particular DL-ECC termed TurboAE, which replaces constituent codes in the encoders/decoders of a Turbo code with learned convolutional neural networks. Evaluating each term in the BCE decomposition of TurboAE is facilitated through the junction tree algorithm for exact inference on graphs, yielding a MAP decoding of TurboAE-like codes of up to 40 input, 120 output bits (vs. the original 100-bit inputs). Plots of this decomposition over training offer insight into the alternating training process used in TurboAE and other DL-ECCs. We add to the growing body of work on interpreting DL-ECCs by providing a new lens through which to view any DL-ECC training process that uses BCE as a loss function.**

I. Introduction

Applications of neural networks to error correction, while partially explored decades ago [1], have seen substantial recent developments [2]–[4]. Literature on deep-learned error-correcting codes (DL-ECCs) falls into two categories: (1) development of techniques to "learn" the encoders, decoders, or both, of error-control schemes, and (2) interpretation of said techniques through theoretical and empirical lenses. This work falls in (2), as we will present new tools for interpreting the *training process* of one DL-ECC termed TurboAE [3].

TurboAE [3] is a rate $1/3$ Turbo-like architecture in which the three constituent codes are replaced by learned convolutional neural network (CNN) based encoding blocks, and the BCJR decoder is replaced by a somewhat iterative learned CNN decoder. All parameters are learned in an auto-encoder-like framework with binary cross-entropy (BCE) as a loss function, and with noise in the middle to simulate the noisy channel. The performance of TurboAE mimics that of good Turbo codes at low SNRs on AWGN but exhibits a degree of robustness to the knowledge of the noise statistics.

This work was supported by NSF under awards 1934915 and 2217023, and the NRDI Office within the framework of the Artificial Intelligence National Laboratory program (MILAB), Hungary.

On the interpretation side, the literature is sparser. [5] explores both the principles guiding approaches for learning error-correcting codes as well as some analysis of the results of such methods. [6] explored methods for interpreting TurboAE through exact (non-linear) and approximate (linear convolutional codes) representations of the constituent learned encoders, which can be paired with BCJR decoders to look at the benefit of the learned (versus classical) encoder and/or decoder. This line was further studied in [7] along with more technical details of TurboAE, like the irregularities imposed on the encoder via zero padding. [8] instead looked at the training dynamics through the Fourier lens and proposed the Goldreich-Levin algorithm to quickly determine the dominant encoder Fourier coefficients.

A. Contributions

The authors in [8] re-examined the decomposition of the binary cross-entropy (BCE) loss function, using it as inspiration for an alternative (non-neural) optimization algorithm for learning ECCs. This work further explores potential uses of this decomposition, this time as a lens to understand the optimization dynamics between neural encoders and decoders:
(1) We first present a BER-optimal algorithm to decode rate $R = \frac{k}{n} = \frac{1}{3}$ Turbo codes for input length $k = 40$ (versus the original $k = 100$) using the **Junction Tree algorithm** [9]. While discussed before [10], [11], ours appears to be the first practical implementation for decoding Turbo codes.
(2) The well-known decomposition of BCE into two terms provides insight into the training process used by TurboAE (and any learned DL-ECC using BCE as a loss): one term depends on both the learned encoder and decoder (Kullback-Leibler divergence (KL) between the true posteriors and the estimated posteriors on the input bits given the channel outputs), while the other only depends on the learned encoder (the conditional entropy (CE) of the input bits given the channel outputs). We explore the interplay between these two terms during the neural network training process, in the flavor of [12]. By plotting the two terms in the BCE, we are able to see how the optimization is dominated by a particular stage during different broader phases of the alternating training (see Section II-A) used. Evaluating these two terms for a learned decoder is enabled through the junction tree algorithm.

II. Preliminaries

A. TurboAE

The DL-ECC called "TurboAE" [3] has an architecture that resembles a rate $R = \frac{k}{n} = \frac{1}{3}$ Turbo code at the encoder: it has three "constituent codes" replaced by CNN blocks $f_b^{(\theta)}(\cdot)$ as in Fig. 1 (from [8]) taking a sequence $\mathbf{u}$ of $k = 100$ bits, and outputting three 100-bit sequences $\mathbf{x}_{AE,b} \in \{\pm 1\}^{100}$ for each block $b \in [3]$. These three blocks are then concatenated to produce codewords of length $n = 300$. Here θ represents the CNN weights or parameters. The network has two versions, TurboAE-Cont (with real-valued encoder outputs, performing coding and modulation tasks jointly) and TurboAE-Binary (with Boolean encoder outputs). The power control modules and edge effects discussed in [7] are omitted. The TurboAE decoder architecture replaces the (in this case 6) iterations of the BCJR decoder by soft decoder CNNs $g_{t,1}^{(\gamma)}, g_{t,2}^{(\gamma)}$ for iteration $t \in [6]$ as in Fig. 1. Note that $\mathbf{y_b} = \mathbf{x_{AE,b}} + \mathbf{z_b}$, for $\mathbf{z_b}$ i.i.d. Gaussian noise of zero mean and variance σ, for each stream $b \in [3]$. The CNN parameters γ are obtained through an *alternating training procedure*: the decoder is held fixed while the encoder is trained for a number of gradient descent steps, then the encoder is fixed while the decoder is trained for a number of steps; this process is repeated until convergence.

The TurboAE-Cont encoder is a binary input, real-output sliding window code of *window* length 9 (memory 8), meaning each output depends on 9 consecutive binary inputs (ignoring edge effects [7]), and *delay* 4, meaning the window for each output contains 4 future input bits. TurboAE-Binary has similar input dependence but outputs binary values instead of real values. In [6] it was found that TurboAE-Binary's encoders actually only depend on 5 inputs, instead of 9. This property was used to obtain simpler non-linear and approximate (affine, or parity) representations of the CNN encoders [8, Table I, II, Appendix]. Here we further this investigation of TurboAE by looking at the binary cross-entropy (BCE) loss and how it evolves over the alternating training procedure.

B. Decomposition of BCE

Consider optimizing an encoder function $f^{(\theta)} : \mathbb{F}_2^k \to \mathbb{R}^n$, where $f^{(\theta)} = \{f_b^{(\theta)}\}_{b\in[3]}$, parameterized by $\theta \in \Theta \subset \mathbb{R}^m$ for some $m \in \mathbb{N}$. We take $U \sim \text{Unif}[\mathbb{F}_2^k]$ and $Y \in \mathbb{R}^n$ to be random variables representing the input and received (channel output) sequence, respectively. Note that Y depends on θ. We can similarly parameterize a soft decoder $g^{(\gamma)} : \mathbb{R}^n \to [0,1]^k$, where $g^{(\gamma)}$ represents the iteration of functions $\{g_{t,s}^{(\gamma)}\}_{t\in[6],s\in[2]}$ described in Fig. 1, for parameters $\gamma \in \Gamma \subset \mathbb{R}^l$ for some $l \in \mathbb{N}$. These encoder and decoder parameters are then ideally found as solutions to the optimization problem:

Problem 1. *Find $\theta \in \Theta, \gamma \in \Gamma$ so $f^{(\theta)}, g^{(\gamma)}$ minimize the expected BCE, $\mathbb{C}(f^{(\theta)}, g^{(\gamma)})$, where*

$$\mathbb{C}(f^{(\theta)}, g^{(\gamma)}) = \mathbb{E}\Big[\frac{1}{k}\sum_{i=1}^{k}[-U_i \log g_i^{(\gamma)}(Y^{(\theta)})$$

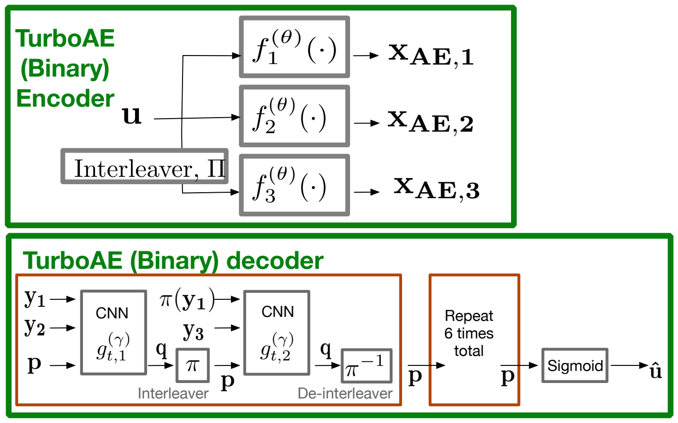

Fig. 1: Rate $R = \frac{1}{3}$ ($\mathbf{u} \in \mathbb{F}_2^{100}, \mathbf{x}_{AE,b} \in \{\pm 1\}^{100}$) TurboAE-Binary encoder and decoder structures. The parameters of the CNNs $f_b^{(\theta)}, g_{t,s}^{(\gamma)}$ are trained. The noisy channel outputs of the three encoded streams $\mathbf{x}_{AE,1}, \mathbf{x}_{AE,2}, \mathbf{x}_{AE,3}$ are given by $\mathbf{y_1}, \mathbf{y_2}, \mathbf{y_3}$. The decoder produces probabilities $\hat{\mathbf{u}}$ that each input bit is 0 or 1. Here $\mathbf{p}$ represents prior information and $\mathbf{q}$ represents posterior information from each CNN iteration.

$$- (1 - U_i)\log(1 - g_i^{(\gamma)}(Y^{(\theta)}))]\Big] \quad (1)$$

$$= \frac{1}{k}\sum_{i=1}^{k}\mathbb{E}[D_{KL}(\mathbb{P}[U_i = 1|Y^{(\theta)}]||g_i^{(\gamma)}(Y^{(\theta)}))] + \mathbb{H}(U_i|Y^{(\theta)}).$$

The decomposition in the last line is well-known, but it has extra pertinence to optimizing DL-ECCs as noted in [5], [8]. For a given encoder $f^{(\theta)}$, its bitwise MAP decoder achieves the minimum 0 of $\mathbb{E}[D_{KL}(\mathbb{P}[U_i = 1|Y^{(\theta)}]||g_i^{(\gamma)}(Y^{(\theta)}))]$. Furthermore $\mathbb{H}(U_i|Y^{(\theta)})$ only depends on $f^{(\theta)}$, not on $g^{(\gamma)}$. Thus, we can think of BCE optimization as the management of two components: (1) optimality of the decoder for a fixed encoder (KL term), and (2) quality of the encoder for the channel (CE term).

Using bit error rate (BER) as a loss function might be more desirable, but the differentiability of BCE allows for gradient methods to optimize the DL-ECC parameters. Unfortunately, [8] showed that a BCE-optimal DL-ECC is not necessarily a BER-optimal DL-ECC, though tight connections exist [8].

The BCE and its decomposition (1) have additional value beyond their connection to BER. The decomposition allows us to study the dynamics of the alternating training scheme in [3]. Unfortunately, even though BCE is easy to estimate via Monte-Carlo methods, both $\mathbb{E}[D_{KL}(\mathbb{P}[U_i = 1|Y^{(\theta)}]||g_i^{(\gamma)}(Y^{(\theta)}))]$ and $\mathbb{H}(U_i|Y^{(\theta)})$ (equivalently, mutual information) are not. There is a significant line of work investigating both parametric and non-parametric approaches to this estimation problem [13]. We present a more direct approach to computing the decomposition terms. Leveraging the fact that MAP probabilities for Turbo codes can be computed via inference over a factor graph, we can apply the junction tree algorithm [9] to explicitly compute the optimal soft decoder. If $g_*^{(\theta)}$ is the optimal decoder, $(g_{*,i}^{(\theta)}(Y^{(\theta)}) := \mathbb{P}[U_i = 1|Y^{(\theta)}])$, then $\mathbb{E}[D_{KL}(\mathbb{P}[U_i = 1|Y^{(\theta)}]||g_{*,i}^{(\theta)}(Y^{(\theta)}))] = 0$ for all $i \in [k]$ and

$$\mathbb{C}(f^{(\theta)}, g_*^{(\theta)}) = \frac{1}{k}\sum_{i=1}^{k}\mathbb{H}(U_i|Y^{(\theta)}).$$

From this, we can then infer both terms in decomposition (1).

III. Junction Tree Inference Algorithm

We propose to use the junction tree inference algorithm [9] to obtain a MAP decoder for a given Turbo code. Specifically, we follow the relevant algorithms in [9, Ch. 9, 10].

Nonrecursive Turbo codes determine a factor graph over our input bits U_i for $i \in [k]$. Specifically, we are interested in computing $\mathbb{P}[Y^{(\theta)}, U_i = u]$ for $u \in \mathbb{F}_2$, from which we can obtain the desired $\mathbb{P}[U_i = 1|Y^{(\theta)}]$. For a nonrecursive convolutional code with window w and delay d, this probability factors into the sum-product $\sum_{U\in\mathbb{F}_2^k, U_i=u}\prod_{j=1}^{k}\mathbb{P}[Y_j^{(\theta)}, U_{\texttt{wind}}]$ for memoryless channels, where $\texttt{wind} := j - w + 1 + d : j + d$. Here we take $U_l := 0$ for $l < 1$ and $l > k$. For a Turbo code, the interleaver π introduces additional factors. The new factorization is then,

$$\sum_{U\in\mathbb{F}_2^k, U_i=u}\prod_{j=1}^{k}\mathbb{P}[\widetilde{Y}_j^{(\theta)}, U_{\texttt{wind}}]\prod_{j=1}^{k}\mathbb{P}[\widetilde{Y'}_j^{(\theta)}, U_{\pi^{-1}(\texttt{wind})}], \quad (2)$$

where $\widetilde{Y}^{(\theta)}$ denotes our noninterleaved received data and $\widetilde{Y'}^{(\theta)}$ our interleaved received data (taking $\pi(l) := l$ for $l < 1, l > k$). This factorization defines a factor graph, where nodes represent each U_j for $j \in [k]$, and two nodes have an edge if and only if they appear in the same factor. Naively evaluating equation (2) would result in $O(k2^k)$ operations for a fixed rate, infeasible for $k \gtrapprox 26$. However, carefully applying the variable elimination algorithm to break the sum up and marginalize over each U_j one at a time can drastically decrease the complexity [9]. This observation forms the crux of the junction tree algorithm. Note that as we marginalize over each U_j, we will need to keep track of intermediate factors that may grow to include variables linked to U_j. In a convolutional code, intermediate factors do not grow by more than one variable if we eliminate variables sequentially, explaining why BCJR (with convolutional codes) has a runtime of $O(k2^w)$. However, with an interleaver, more than one variable is usually added to our growing factors with each elimination.

The runtime of the variable elimination algorithm depends on the largest factor. Finding the optimal elimination order is equivalent to finding a minimal chordal completion of the factor graph, an NP-complete problem [14]. However, [9] describes several greedy heuristics that can be used. Fig 2 shows how the maximum factor size grows with the input block length (k) using these heuristics, averaged over many interleavers. Additional encoder-dependent tricks can be used to lower the maximum factor size; we are able to optimally decode nonrecursive window 3 turbo codes up to $k = 80$.

If we apply variable elimination to compute equation (2), we only compute $\mathbb{P}[Y^{(\theta)}, U_i]$ for a given $i \in [k]$. To compute for all $i \in [k]$ we would need to rerun the elimination k times. The junction tree algorithm is an extension of variable elimination that allows us to compute $\mathbb{P}[Y^{(\theta)}, U_i]$ for each $i \in [k]$ without having to redo intermediate computations. The trick is to note that an elimination order also induces a junction tree (also known as clique tree) [9], where nodes are clusters of variables U_i so that each cluster does not split a factor, and if two clusters, C and C', share a variable, then the variable is in all clusters along the unique path between C and C'. With this tree, we can run Pearl's Belief Propagation algorithm [15] to compute joint posterior probabilities on all the variables involved in a cluster. Then we marginalize to get the posterior on the variable of interest. Since each cluster only contains a subset of the variables, we will marginalize over a much smaller set of variables compared to the naive approach described earlier. A single variable may be in multiple clusters, in which case marginalizing over any cluster will produce the same result.

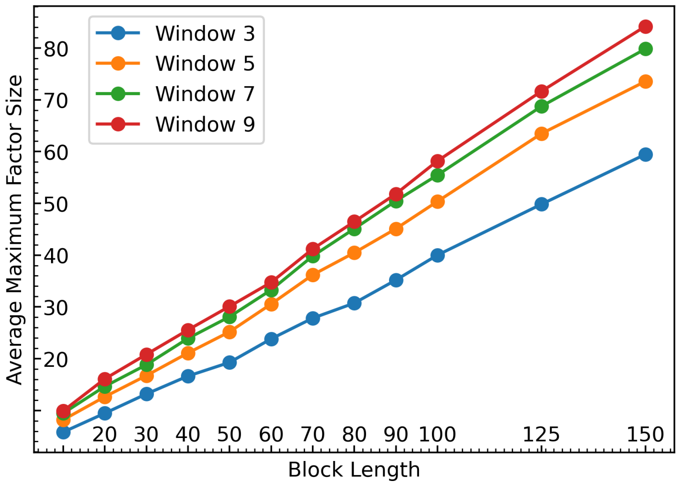

Fig. 2: Maximum factor size in the junction tree vs. input block length k and window w. The average was taken over 10 different interleavers. Junction tree decoding is exponential in runtime and memory in the maximum factor size.

Fig 3 shows different examples of the resulting junction trees, coloring the nodes based on the number of variables involved. For the interleaver of Sec. IV, the junction tree is a single line, but in general more interesting junction trees can emerge, as shown. Larger clusters tend to be in the center.

While authors have noted that this approach can be used [10], in particular for block codes [11], [16], this is the first implementation for optimal turbo decoding we are aware of.

IV. Training Dynamics via BCE decomposition

In end-to-end learned encoder/decoder pairs it is important to understand the impact of the encoder versus the decoder, and whether these have been successfully trained. We propose to use the two terms in the BCE decomposition to elucidate the training: 1) the KL between the true and estimated posteriors on the input bits given the channel outputs (encoder and decoder dependent), and 2) the CE of the input bits given the channel output (encoder dependent). This direction is reminiscent of [12]'s "Information Bottleneck for Deep-Learning" where the training process of a deep neural network (DNN) which classifies inputs X as Y is analyzed by looking at two

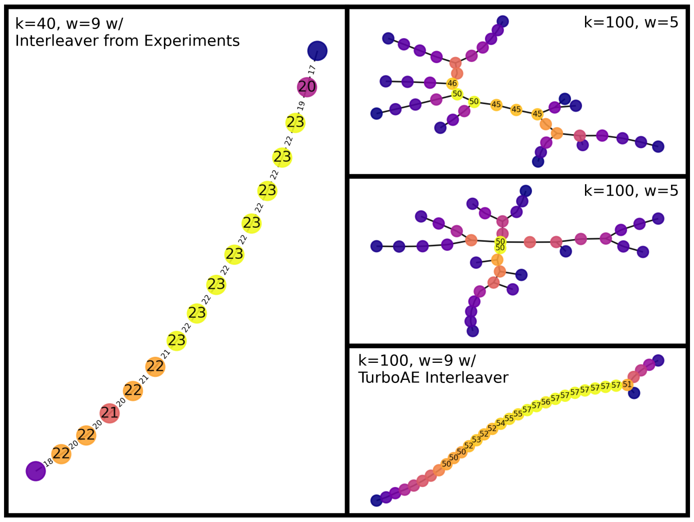

Fig. 3: Left: Junction Tree from the interleaver used for training in Fig [4]. Node labels are cluster sizes, and edge labels are the sizes of intersections between the two nodes. Lighter colors indicate larger numbers. Right: Top to bottom, (1) and (2) are two junction trees made for interleavers on a $k = 100$, window 5 code. (3) is a junction tree for the original TurboAE codes, with $k = 100$ and window 9.

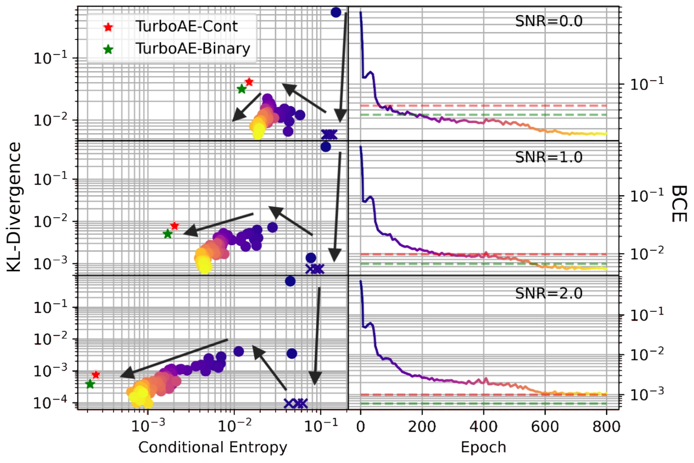

Fig. 4: Left: two terms in BCE decomposition ([1]) during training at SNRs 0, 1, and 2 of the $k = 40$ version of the newly trained TurboAE[1]; dark colors are the start, yellow is the end of training. Also plotted are the final trained points of the original TurboAE-Cont and TurboAE-Binary from [3], with decoders fine-tuned for $k = 40$. Right: total BCE during training, same colors.

quantities as the network is trained: the mutual information between the input and hidden layers, and the mutual information between these hidden layers and the output (classification). They interpret the phases of the training using these metrics. While both the setup and metrics here are different, this idea offers a glimpse into the training dynamics, particularly of the alternating training proposed in [3]. We believe this decomposition can provide insight *in general* into the training process under BCE for DL-ECCs.

[1]The "X" points have estimated KL-Divergence negative because it appears that the junction tree decoder was not optimal for those epochs. This is likely due to numerical imprecision. Further investigation is needed to confirm this. The points should be treated as $D_{KL} \equiv 0$.

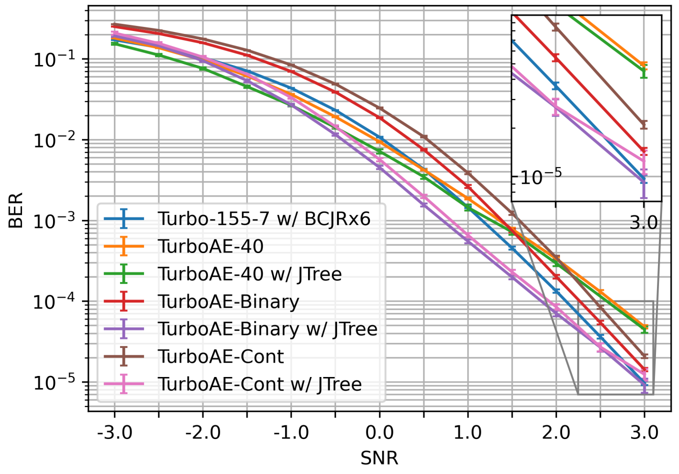

Fig. 5: BER comparison of rate 1/3 codes: benchmark Turbo 155-7 code (blue), a learned $k = 40$ code with the optimal junction tree decoder (green) and with a learned decoder (yellow), original TurboAE-Binary (red) and TurboAE-Cont (brown) from [3], with decoder fine-tuned for $k = 40$, and their encoders paired with optimal junction tree decoders (purple, pink).

We study a variant of TurboAE-Cont with input length $k = 40$ rather than 100, due to the intractability of using the junction tree inference algorithm at higher input lengths. As noted in [6], TurboAE-Cont is a window $w = 9$ nonsystematic convolutional code with delay $d = 4$. While much of the details remain the same as the original TurboAE-Cont [3], we make a few minor changes to otherwise improve the training and interpretability: (1) we perform padding on the input to eliminate edge effects [7], (2) we pad with -1 rather than 0 since TurboAE-Cont takes inputs in $\{\pm 1\}$, and (3) we directly normalize the output table of TurboAE, rather than using batch-level statistics, for power normalization. Using this TurboAE variant, we train at $k = 40$ for 800 epochs. Each epoch consists of training the encoder for 25 steps and the decoder for 125 steps (the alternating training procedure) over input batches sampled IID from Unif$[\{\pm 1\}^k]$. Like in [3], we train the encoder at SNR 2.0, and the decoder at a range of SNRs from -1.5 to 2.0. We train with batch sizes of 500, then 1000 for 100 epochs each, then a batch size of 2000 for the remaining epochs[2].

Both the cross-entropy and decomposition metrics of the trained model can be viewed in Fig [4]. Purple designates earlier epochs of the training, and yellow later epochs. Each dot on the left panel of Fig [4] corresponds to a measurement *after* 8 epochs, with the top-rightmost dot corresponding to the initialization. On the right panel are the corresponding cross-entropies during training. We produce measurements at SNRs 0, 1, and 2. Note the zig-zag pattern in the evolution on the left panels. During training, there is a back-and-forth between improvement of the decoder, and improvement of the encoder, sometimes at the cost of additional KL. This "back-and-forth" is **not** the same as the alternating training process. Here the "back-and-forth" evolution is happening on the scale

[2]All code has been made publicly available at https://github.com/tripods-xai/istc-2023 for reproducibility.

of *hundreds* of epochs. A single alternation in the alternating training process is *one* epoch.

The right-hand panels tell a complementary picture. Notice how the training evolves in ”spurts”. Optimization appears to be stuck in a saddle point for sometimes hundreds of epochs, until it suddenly improves fairly rapidly, only to get stuck in another critical point. Aligning these sudden improvements with the evolution on the left-hand side, we see they correspond to sudden improvements in the *encoder*, as evidenced by movement leftward along the CE (x-)axis. Following improvement in the encoder, there is a quick fine-tuning of the *decoder*, as evidenced by the downward movement along the KL (y-)axis. From this it follows, the more difficult part of the optimization process is optimizing the encoder, as improvement in the encoder is what it takes for the DL optimization to move out of its saddle point.

We also added the decomposition and BCE of both TurboAE-Cont and Binary, with the decoder fine-tuned for $k = 40$ to Fig 4. These codes tell a slightly different story. They have much lower CE, but their KL is significantly higher than our trained code. These codes were originally trained at $k = 100$ in [3], which may explain their lower CE.

Finally, to situate our learned code amongst the other benchmarks, we estimate its BER at various SNRs, shown in Fig 5, along with those of our benchmarks. The neural benchmarks are the original TurboAE codes in [3] with the fine-tuned decoder described above. Our learned code is significantly better than the original neural codes below SNR 1.5. Fig 4 explains this nicely. The original codes have better encoders, but suboptimal decoders compared to our trained code. At lower SNRs, the difference in decoders brings our trained code ahead. However, as the SNR increases, both decoders start to approach MAP, and the quality of the encoder dominates. Despite the differences, the neural codes were trained similarly. This kind of variability seems common with neural codes. In fact, an earlier attempt produced a worse code, and the key distinguishing factor between it and the code in this paper was the CE of the encoders. This suggests the random seed used for training has a major impact on the encoder learned. We also included BER measurements of our trained CNN encoder with a junction tree decoder. As noted, the CNN decoder was already close to MAP, so we do not see much improvement. Outside of neural benchmarks, we compared against a memory 2 recursive systematic (RSC) code (Turbo-155-7). The RSC variant outperforms all neural codes. This is likely because iterative BCJR has better error-correcting capabilities when one of the streams is systematic. However, upon replacing the CNN decoder of TurboAE-Binary with a junction tree decoder, we see they outperform the RSC. The dramatic improvement is explained by the suboptimality of the fine-tuned decoders as shown in Fig 4.

V. Conclusion

In this work, we presented a feasible MAP decoder for Turbo codes and used it to unravel the roles played by the neural decoder and encoder during the optimization of DL-ECCs. In equation (1), we decomposed BCE into an encoder-specific term (CE) and a term that measures how close the decoder is to MAP (KL). At input length $k = 40$, we applied our MAP decoder to observe the evolution of this decomposition during the training process. We discovered a ”zig-zag” pattern in the evolution of this decomposition: learning happens in ”spurts”, with sudden improvements in the encoder, followed by quick adjustments of the decoder to match the encoder. This work opens further questions regarding why the alternating training scheme of [3] may be better for optimization than other schemes (e.g. the decoupled scheme from [8]). In particular, when the neural encoder is stuck in a saddle point, what properties do the nearby better encoders have relative to the current saddle point? Work from [17] suggests that the place to look may be in changes in Fourier coefficients during these jumps. This decomposition may also be useful in the future comparison of learned error-correcting codes trained with BCE loss.

References

[1] W. Caid and R. Means, “Neural network error correcting decoders for block and convolutional codes,” in *[Proceedings] GLOBECOM ’90*. San Diego, CA, USA: IEEE, 1990, p. 1028–1031.

[2] T. O’Shea and J. Hoydis, “An Introduction to Deep Learning for the Physical Layer,” *IEEE Trans. on Cogn. Commun. and Netw.*, 2017.

[3] Y. Jiang *et al.*, “Turbo autoencoder: Deep learning based channel codes for point-to-point communication channels,” in *NIPS*, Dec. 2019, pp. 2758–2768.

[4] Y. C. Eldar *et al.*, *Machine learning and wireless communications*. Cambridge University Press, 2022.

[5] S. Cammerer *et al.*, “Trainable Communication Systems: Concepts and Prototype,” *IEEE Transactions on Communications*, vol. 68, no. 9, pp. 5489–5503, Sep. 2020.

[6] N. Devroye *et al.*, “Interpreting Deep-Learned Error-Correcting Codes,” in *ISIT*, Jun. 2022, pp. 2457–2462.

[7] ——, “Evaluating interpretations of deep-learned error-correcting codes,” in *Allerton*, Sep. 2022.

[8] ——, “Interpreting Training Aspects of Deep-Learned Error-Correcting Codes – extended ArXiv version,” Jun. 2023. [Online]. Available: https://arxiv.org/abs/2305.04347

[9] D. Koller and N. Friedman, *Probabilistic Graphical Models: Principles and Techniques - Adaptive Computation and Machine Learning*. The MIT Press, 2009.

[10] R. McEliece, D. MacKay, and Jung-Fu Cheng, “Turbo decoding as an instance of Pearl’s ”belief propagation” algorithm,” *IEEE Journal on Sel. Areas in Commun.*, vol. 16, no. 2, pp. 140–152, Feb. 1998.

[11] M. Xu, “Iterative decoding and graphical code representations,” Ph.D. dissertation, California Institute of Technology, Mar 2008.

[12] R. Shwartz-Ziv and N. Tishby, “Opening the black box of deep neural networks via information,” 2017. [Online]. Available: http://arxiv.org/abs/1703.00810

[13] J. Walters-Williams and Y. Li, *Estimation of Mutual Information: A Survey*, ser. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, vol. 5589, p. 389–396.

[14] Y. Li, L. Allison, and K. Korb, “Proving the NP-completeness of optimal moral graph triangulation,” Mar. 2019. [Online]. Available: https://arxiv.org/abs/1903.02201

[15] J. Pearl, *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. San Mateo, CA: Morgan Kaufmann, 1988.

[16] L. P. Natarajan, K. P. Srinath, and B. S. Rajan, “Generalized distributive law for ML decoding of STBCs,” in *ITW*. Paraty, Brazil: IEEE, Oct. 2011, pp. 10–14.

[17] E. Abbe, E. B. Adsera, and T. Misiakiewicz, “The merged-staircase property: a necessary and nearly sufficient condition for sgd learning of sparse functions on two-layer neural networks,” in *Proceedings of Thirty Fifth Conference on Learning Theory*. PMLR, Jun 2022, p. 4782–4887.