Using Deep Neural Networks to Predict and Improve the Performance of Polar Codes

Mathieu Léonardon and Vincent Gripon
IMT Atlantique, Lab-STICC, UMR CNRS 6285, F-29238 Brest, France

***Abstract*—Polar codes can theoretically achieve very competitive Frame Error Rates. In practice, their performance may depend on the chosen decoding procedure, as well as other parameters of the communication system they are deployed upon. As a consequence, designing efficient polar codes for a specific context can quickly become challenging. In this paper, we introduce a methodology that consists in training deep neural networks to predict the frame error rate of polar codes based on their frozen bit construction sequence. We introduce an algorithm based on Projected Gradient Descent that leverages the gradient of the neural network function to generate promising frozen bit sequences. We showcase on generated datasets the ability of the proposed methodology to produce codes more efficient than those used to train the neural networks, even when the latter are selected among the most efficient ones.**

I. Introduction

Polar Codes are a family of Error Correcting Codes that are used in the 5G NR standard. In order to construct these codes, traditional methods consist in estimating the bit error probabilities with mathematical models. Usually these approaches consider the Successive Cancellation (SC) decoding algorithm as it allows such mathematical formulations. In practice the Successive Cancellation List (SCL) decoder allows to achieve much better error rates. However predicting the bit error probabilities for SCL is not as easy as for SC. Finding good code constructions is therefore challenging, and many works have been published with that aim [1]–[3]. Most of these methods require estimating bit/frame error rates (BER/FER) using Monte Carlo simulations, and training models to lower them. Ideally, a well trained machine learning algorithm could lead to finding error rates lower than those encountered on its training data.

In this paper, we aim at using neural networks to predict the FER of a polar code from its construction parameters. Once such a neural network is trained, it is possible to use it to generate competitive codes. To this end we propose to use an adaptation of a classical algorithm typically meant to generate adversarial inputs, i.e. inputs that are specifically designed to fool the neural network decision. In our case, these frozen bit sequences inputs will be generated with the aim at lowering the error rate. We propose two datasets made of bit frozen sequences and associated FERs, obtained using Gaussian Approximation or Density Evolution. For each one, we show that it is possible to train neural networks able to predict the FER with high confidence, resulting in an error inflation of about 5% in average on previously unseen inputs. Using these neural networks, we propose new polar codes achieving lower FERs than the ones used during training. We release both the datasets and the code used in this paper at the following address: https://github.com/brain-bzh/PolarCodesDNN.

II. Constructing Polar Codes

A. Conventions

In this paper we consider (N, K) polar codes, where N is the codeblock length and K the length of the information sequence. We are more precisely interested in polar codes with Arıkan kernels, as defined in [4]. Apart from both N and K, such a code is defined thanks to the frozen bit sequence $\boldsymbol{f} \in \{0,1\}^N$. Namely, the 1s in $\boldsymbol{f} = \{f_i\}_{0 \leq i < N}$ correspond to the indices of the frozen bits.

B. Mathematical models

Constructing polar codes refers to the way to generate a frozen bits sequence $\boldsymbol{f}$, given N, K, and a channel model. In [4], construction was only defined for Binary Erasure Channels. Methods to construct polar codes for Additive White Gaussian Noise (AWGN) channels were later given in [5], [6]. All three methods sort the positions of frozen bits, from the most to the least reliable, considering SC decoding. However, freezing the least reliable bits is not guaranteed to offer the best performance, expressed as the Frame Error Rate (FER) for a targeted SNR in more complex situations. As a matter of fact, finding the best frozen bit sequence (i.e. that yielding the lower Frame Error Rate (FER)), for a given channel, target SNR and decoder, is a difficult challenge that has to be addressed specifically, using complex models or empirical approaches.

C. Learning Approaches

Some learning-based methods have been recently proposed to further improve the error correcting performance of polar codes. They are based on the mathematical constructions mentioned above and particularize this construction to the specific channel and decoding conditions. In [1], a genetic algorithm is used to modify the frozen bits set in order to improve the performance of polar codes with different channel models (AWGN, Rayleigh) and different decoding algorithms e.g. Belief Propagation (BP), SCL, etc. Reinforcement Learning (RL) has also been used to address the same problematic [3]. The polar code construction is formulated as a maze-traversing game, which is solved using RL methods. This allows to

improve the performance of polar codes constructed with [5] in some of the studied cases.

Deep Learning Techniques have also been used to improve existing codes [2]. It is mentioned that neural networks are used to predict performance of a polar code under certain channel and decoding conditions, and some results are provided. However, the methods used to construct and train the network are not described. In this paper, we propose to further investigate the capabilities of neural networks to model and construct polar codes. We give hints about how to construct these neural networks, with detailed experiments and results. Finally, we propose methods that allow to use neural networks to improve the construction of polar codes for a given channel and decoding algorithm.

III. Polar Codes Design with Neural Networks

A. Dataset Generation

We propose to train a neural network to predict the FER performance of a polar code. The neural network is trained on a dataset consisting of pairs of frozen bits sets $\boldsymbol{f}$ associated with their corresponding FERs. The FERs are obtained by Monte Carlo simulations, using the AFF3CT toolbox as a library [7]. The frozen bits set is the only parameter that change in the communication chain accross the simulations. All other parameters are constant, e.g. N, K, E_b/N_0. The frozen bits sets space is a high dimensional space $\binom{N}{K}$ in which most of its elements yield bad FERs. Directly training on poorly performing codes would inevitably lead the neural network to focus predicting high FERs.

In order to generate a relevant subset, we first use the Gaussian Approximation (GA) method to generate a list of the frozen bits positions $\boldsymbol{p} \in \{0, 1, \dots N-1\}^{N-1}$, that are sorted according to the reliability that GA associates with each of them. Usually, this sorted positions are directly used to generate the frozen bits set $\boldsymbol{f}$, where

$$f_{p_i} = \begin{cases} 1 & \text{if } i < K \\ 0 & \text{if } i \geq K \end{cases}, \forall 0 \leq i < N \tag{1}$$

To generate our dataset, a subset of the values in $\boldsymbol{p}$ are randomly shuffled to get a new vector of frozen bits positions $\boldsymbol{p^d}$:

$$p_i^d = \pi^d(p_i), \forall K - r \leq i < K + r \tag{2}$$

where $\{\pi^d\}_{0 \leq d < D}$ is a set of uniformly distributed random permutations. The shuffling range r is chosen empirically. According to our experiments, the best results are obtained when r is chosen so that there is a one to ten ratio between the worst and the best measured FER in the generated dataset. In order to get a dataset of size D, the shuffling operation is performed D times, to get D variants of the frozen bits positions $\boldsymbol{p^d}$ and the corresponding frozen bits sets $\boldsymbol{f^d}$.

B. Neural network

Let us first recall that a neural network can be modelled as a mathematical function that is obtained by assembling elementary subfunctions called *layers*. A typical layer is a function of the form $F^\ell : \boldsymbol{x} \mapsto \sigma(\boldsymbol{W}^\ell \boldsymbol{x} + \boldsymbol{b}^\ell)$, where $\boldsymbol{W}^\ell$ and $\boldsymbol{b}^\ell$, respectively a matrix and a vector, are made of values that are trainable parameters, and σ is a nonlinear predetermined function which acts on each component of its input independently. The number of dimensions in the input $\boldsymbol{x}$ is denoted d_ℓ^{input} and the number of dimensions in the output $F_\ell(\boldsymbol{x})$ is denoted d_ℓ^{output}. Assembling layers can be performed by performing composition, additions, concatenations, etc.

In our work, we consider very simple architectures that are defined using three hyperparameters:

- The *number of layers* L. We denote each associated layer function $(F_\ell)_{1 \leq \ell \leq L}$. We use Rectified Linear Units (ReLU) as nonlinear functions in all layer functions, except the last for which we do not use a nonlinear function.
- The *number of neurons in hidden layers*, denoted H. We then fix the dimensions: $d_1^{input} = N$, $(d_\ell^{input} = H, \forall \ell > 1)$, $(d_\ell^{output} = H, \forall \ell < L)$, $d_L^{output} = 1$.
- The *shortcut gap* G defined in analogy to celebrated ResNet architectures [8] (see details below).

Once all parameters L, H and G are fixed, we can define the neural network function F. Denote $\boldsymbol{f}$ an input sequence of bits. We compute the following sequence:

$$\begin{cases} \boldsymbol{f}^0 &= \boldsymbol{f} \\ \boldsymbol{f}^{\ell+1} &= \begin{cases} F_{\ell+1}(\boldsymbol{f}^\ell) + \boldsymbol{f}^{\ell+1-G} & \text{if } \exists k \in \mathbb{N}^*, \ell = kG \\ & \text{and } \ell + 1 < L \\ F_{\ell+1}(\boldsymbol{f}^\ell) & \text{otherwise} \end{cases} \end{cases} \tag{3}$$

Then, we define $F(\boldsymbol{f}) = \boldsymbol{f}^L, \forall \boldsymbol{f}$.

In particular, when G is no lesser than $L-1$, this sequence boils down to composing $F = F^L \circ F^{L-1} \circ \cdots \circ F^1$. A toy depiction of this architecture is shown in Figure 2.

To train F to associate an input bit sequence $\boldsymbol{f}$ with the corresponding FER denoted $\text{FER}_{\boldsymbol{f}}$, we use a variant of the Stochastic Gradient Descent algorithm named Adam [9], which is meant to minimize the Mean Square Error between $F(\boldsymbol{f})$ and $\text{FER}_{\boldsymbol{f}}$ in logscale. In other words, we aim at approximately solving:

$$\arg \min_{\{\boldsymbol{W}^\ell, \boldsymbol{b}^\ell\}_{1 \leq \ell \leq L}} \mathbb{E}[(F(\boldsymbol{f}) - \log(\text{FER}_{\boldsymbol{f}}))^2]. \tag{4}$$

We use a logscale to better encompass for the range of FERs we expect to observe in applications.

C. Inflation Of Error (IOE)

To train a neural network, we make use of a training set T and a validation set V, both made of pairs $(\boldsymbol{f}, \text{FER}_{\boldsymbol{f}})$. The idea is to define these sets so that $V \cap T = \emptyset$. Once a neural network has been trained using T as a proxy to the expectation of Equation (4), we evaluate on V its ability to generalize predictions to previously unseen inputs.

To properly reflect the exponential scale of FER, we define the Inflation Of Error (IOE) of $(\boldsymbol{f}, \text{FER}_{\boldsymbol{f}}) \in V$ for the neural network function F as:

$$\text{IOE}(\boldsymbol{f}, F) = \max \left\{ \frac{\text{FER}_{\boldsymbol{f}}}{\exp(F(\boldsymbol{f}))}, \frac{\exp(F(\boldsymbol{f}))}{\text{FER}_{\boldsymbol{f}}} \right\} - 1. \tag{5}$$

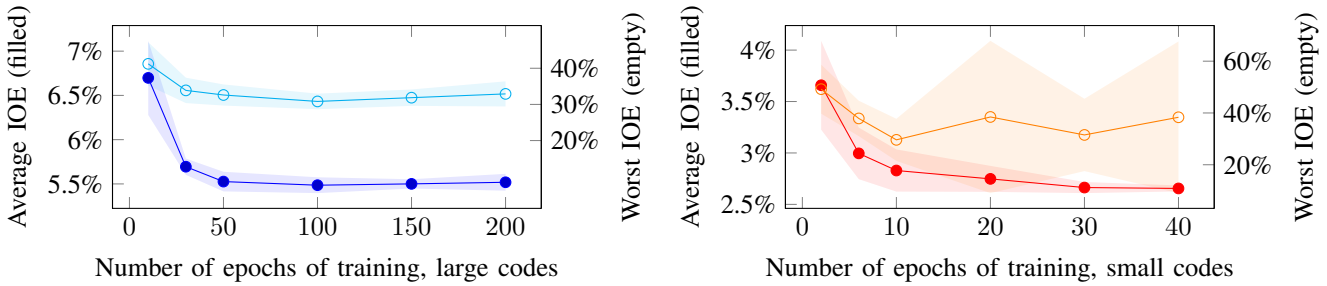

Fig. 1. Evolution of validation Inflation Of Errors depending on the number of epochs used for training, in both the case of large codes (left) and small codes (right). Standard deviation obtained on 10 runs is also shown.

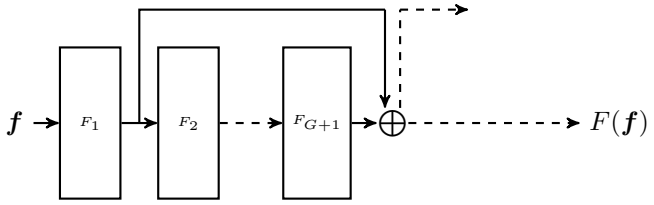

Fig. 2. Depiction of the architectures used to predict the FER based on input frozen bit sequences. The direct flow of data from input to output is periodically combined with shortcuts of length G.

As an example, an IOE of 100% means that the FER predicted using the neural network is half or twice the actual FER of the considered validation sample.

In our experiments, we are typically interested in measuring the average IOE or the worst (i.e. maximal) IOE on the validation set.

D. Constructing codes using neural networks

Once neural networks have been trained to predict FER based on input frozen bit sequences, we can leverage them to propose new efficient codes.

We adapt the methodology described in [10], called Projected Gradient Descent (PGD). PGD has been widely used in the context of adversarial attacks, where the aim is to generate inputs meant to fool the neural network predictions. The idea is to fix the neural network parameters, and use gradient descent to update inputs so that it translates the corresponding output towards new decisions. In our case, the goal is to use the prediction of the neural network as the function to minimize, since it means lowering the FER.

In more details, we implement a straight-through [11] procedure, in which we manipulate inputs that are not necessarily binary: such inputs can be seen as a relaxation of bit sequences into real-valued ones. When we estimate the FER using the neural network, we manipulate a quantized version of the input, where all values below the median are set to one bit and all others the other one. We compute the gradient with respect to this input, but apply it to its real-valued version. More details are available in Algorithm 1. It comes with two parameters: a number of iterations I and a gradient step μ.

Algorithm 1: Algorithm used to generate low FER polar codes.

$\boldsymbol{f} \leftarrow$ random binary initialization
for *a fixed number of iterations* I **do**
  $\tilde{\boldsymbol{f}} \leftarrow$ quantized version of $\boldsymbol{f}$
  $y \leftarrow F(\tilde{\boldsymbol{f}})$
  for $0 \leq i < N$ **do**
    $f_i \leftarrow f_i - \mu \frac{\partial y}{\partial f_i}$
Return quantized version of $\boldsymbol{f}$

IV. Experiments

Throughout this section, we consider two types of polar codes, namely $(256, 128)$, termed *small* and $(1024, 512)$, termed *large*. In both cases, an SCL decoder is used, with a list depth of respectively $\mathcal{L} = 4$ and $\mathcal{L} = 32$. The SNR values used for any construction method are respectively $E_b/N_0 = 3.2dB$ and $E_b/N_0 = 2.7dB$. For the small codes, we generated a total of $77'466$ frozen bit sequences $\mathbf{f}$, split arbitrarily in 80% used for training and 20% for validation. For the large codes, we generated a total of $15'862$ frozen bit sequences, with the same proportions for training and validation.

We use classical data standardization techniques, where we center and reduce both input and output components. On the input, this has the effect of replacing 0s with -1s. We also remove coordinates in the inputs that are constant on our datasets. We end up using 36 out of the 256 input dimensions for small codes and 112 out of the 1024 input dimensions for large codes. The minimal FER in the training and validation sets are 1.67e-4 for small codes and 5.75e-5 for large ones. As a reference for upcoming experiments, we also computed the average IOE and worst IOE obtained when using a constant predictor (predicting the average of outputs). We obtained: average IOE: 86.03% and worst IOE: 1541.32% for small codes and average IOE: 47.90% and worst IOE: 4716.20% for large ones. These quantities can be thought of as "chance levels". For all reported results, we compute at least an average over 10 runs, where each run is obtained with different randomly drawn initial weights for the layer parameters, and different order of presenting data during training.

A. Effect of hyperparameters on IOE

As a first series of experiments, we wanted to empirically study the impact of hyperparameters in our architectures on the average and worst IOEs. We first randomly searched for an efficient starting set of hyperparameters, and then looked at the influence of varying a single parameter, while leaving the others constant. We ended up using 20 epochs for training for small codes and 100 epochs for training for large ones. In both cases, we used 320 neurons in the hidden layers, a network depth of 6 and a shortcut gap of 3.

Prediction performance of neural networks:

On small codes, we obtain after 10 runs the following performance: average IOE: 2.65% $\pm$ 0.06%, worst IOE: 29.36% $\pm$ 5.63% for small codes (confidence intervals obtained at 95%). For the large codes, we obtain: average IOE: 5.46% $\pm$ 0.04%, worst IOE: 31.28% $\pm$ 0.04%. In other words, the ratio between predicted FER and actual one is no more than 6% in average for our datasets, which is way smaller than the chance level we estimated. Also, in the worst case, the actual FER is about 30% larger or smaller than the actual one, which demonstrates the outstanding generalization ability of the trained neural networks. It is also worth pointing out that the estimation of FER using Monte Carlo simulations is subject to a small error, and that this error could be significant with respect to the average IOE measured in our experiments.

Effect of the number of epochs in training:

In Figure III-A, we report the average and worst IOEs obtained depending on the number of epochs used to train the neural network architectures. As expected the small codes require less epochs to reach convergence, as the dataset is way larger than in the case of large codes. Interestingly, we observe a small increase of IOEs when training large codes for too many epochs, which can indicate a small risk of overfitting.

Effect of the number of neurons in hidden layers:

Next, we wanted to study the impact of the number of neurons in hidden layers H in the architecture. In Figure IV-A, we depict the evolution of IOEs as a function of H. Except for the small outlier obtained with the blue curves – which is probably due to the limited number of runs –, we observe an expected global decrease of IOE as the number of neurons grow. A plateau is soon reached around $H = 300$. Interestingly, the worst IOE for small codes seems to increase with the largest values of H, making one suspect about potential overfitting.

Effect of the depth of the architecture:

In Figure IV-A, we vary the depth of trained architectures and report the corresponding IOEs. The effect of the depth seems to play a role analogous to that of the number of neurons in hidden layers, where too small or too large values can cost higher IOEs. The best obtained average IOEs are for depth 3 and 9 respectively, with a pretty insignificant effect between these values. On the contrary, increasing the depth seems to cause way larger worst IOEs for small codes.

Effect of the shortcut gap length:

Next, we tested the effect of the shortcut gap G on the IOE of trained architectures. Obtained results are presented in Figure IV-A. Of all tested parameters, the shortcut gap length seems to be the one having the smallest influence on obtained IOEs. It is worth noting that a depth of 6 is equivalent to not using shortcuts in our architectures. When zooming, we still observe a significant improvement when using small gaps in both the case of small and large codes.

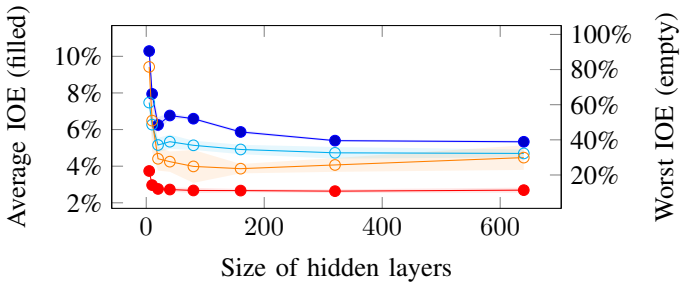

Fig. 3. Evolution of validation Inflation Of Errors depending on the size of the hidden layers in the trained architectures, for large codes (blue and cyan) and small codes (red and orange). Standard deviation obtained on 10 runs is also shown.

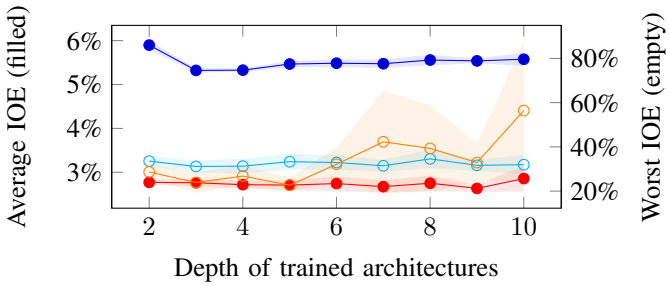

Fig. 4. Evolution of validation Inflation Of Errors depending on the depth of the trained architectures, for large codes (blue and cyan) and small codes (red and orange). Standard deviation obtained on 10 runs is also shown.

Using standard methods to increase predicted IOEs:

Finally, we implemented standard techniques from the field of Deep Learning meant to either prevent overfitting or improve prediction performance. We implemented DropOut [12], including on the input [13], where we randomly mask coordi-

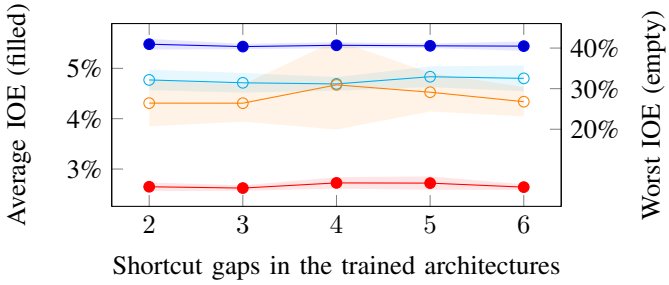

Fig. 5. Evolution of validation Inflation Of Errors depending on the shortcut gaps in the trained architectures, for large codes (blue and cyan) and small codes (red and orange). Standard deviation obtained on 10 runs is also shown.

TABLE I
INFLUENCE OF COMMON TECHNIQUES IN DEEP LEARNING ON THE AVERAGE IOE (CONFIDENCE INTERVALS AT 95% ARE INDICATED): DROPOUT, BATCH-NORMS (BNS) AND MIXUP.

DropOut	BNs	Mixup	IOE (Large)	IOE (Small)
			5.46% ($\pm$ 0.04%)	**2.65%** ($\pm$ 0.06%)
		✓	7.69% ($\pm$ 0.06%)	3.08% ($\pm$ 0.17%)
	✓		8.95% ($\pm$ 0.37%)	6.42% ($\pm$ 0.80%)
	✓	✓	9.14% ($\pm$ 0.50%)	8.05% ($\pm$ 0.11%)
✓			36.78% ($\pm$ 0.28%)	58.92% ($\pm$ 0.37%)
✓		✓	37.43% ($\pm$ 0.44%)	60.78% ($\pm$ 0.69%)
✓	✓		32.77% ($\pm$ 0.23%)	52.96% ($\pm$ 0.21%)
✓	✓	✓	32.89% ($\pm$ 0.33%)	53.31% ($\pm$ 0.18%)

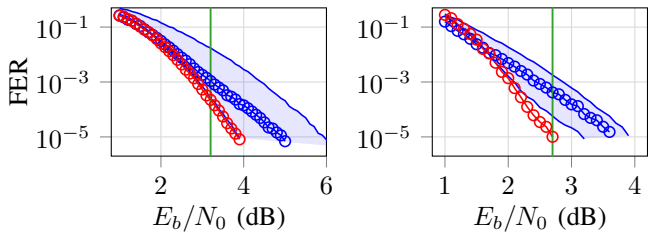

Fig. 6. Frame Error Rate performance of small (left) and large (right) codes, with the GA construction (blue circles), the generated datasets (blue areas), and our proposed construction (red circles). Green vertical lines correspond to the target SNR.

nates of input or output vectors computed throughout the architecture during training, BatchNorms [14], which center and reduce each dimension of computed vectors during training, and learn corresponding coefficients to be used on validation data, and Mixup [15], a simple data-augmentation procedure in which training inputs are linearly interpolated and trained to associate the corresponding linear interpolation of the outputs. Results are presented in Table I. Interestingly, none of the tested methods resulted in improved IOEs on our tests. We suspect that it is because these techniques are mostly meant to be applied when dealing with raw input signals, such as in the case of vision or audio signals. In our case, their discrete and combinatorial nature might make these techniques unsuitable.

B. Constructing Codes

After having explored the impact of hyperparameters on the IOEs, we fixed parameters for constructing interesting codes. We chose: $(L, H, G) = (3, 640, 3)$ trained for 100 epochs for large codes (i.e. not using shortcuts) and $(L, H, G) = (5, 320, 3)$ trained for 40 epochs for small codes. When using Algorithm 1, we used a maximum of 5000 iterations and a gradient step $\mu = 0.1$.

With large codes, we obtained a frozen bit sequence reaching as low as 1.01e-5 as FER, validated using Monte Carlo simulations, which is way lower than the minimum observed on the training set (5.75e-5). With small codes, we obtained several interesting candidates that reached competitive FERs, but not surpassing by a large margin the examples of the training set. This was expected due to the relative small size of these codes and the abundance of already very optimized bit sequences in the training set. The corresponding FERs of best candidates are depicted in Figure 6.

V. CONCLUSION

In this paper, we investigated how to build neural networks capable of predicting the error correction performance of polar codes. We proposed an algorithm to extract competitive codes from trained neural networks. Two datasets of frozen bits sets associated with their corresponding FER values, for a fixed SNR, were generated for two different code lengths, under SCL decoding. These datasets were used to train neural networks to predict the FER of any frozen bits set. We obtained in average a ratio of errors of less than 1.06 for the (1024,512) codes and 1.02 for (256,128) codes. The codes we generated using trained neural networks were shown to perform better than those of the training datasets. Finally, the source code used in this article as well as the generated datasets are published for reproducibility and reuse.

REFERENCES

[1] A. Elkelesh, M. Ebada, S. Cammerer, and S. t. Brink, "Decoder-tailored polar code design using the genetic algorithm," *IEEE Transactions on Communications*, vol. 67, no. 7, pp. 4521–4534, July 2019.

[2] L. Huang, H. Zhang, R. Li, Y. Ge, and J. Wang, "Ai coding: Learning to construct error correction codes," *IEEE Transactions on Communications*, vol. 68, no. 1, pp. 26–39, 2019.

[3] Y. Liao, S. A. Hashemi, J. Cioffi, and A. Goldsmith, "Construction of polar codes with reinforcement learning," *arXiv preprint arXiv:2009.09277*, 2020.

[4] E. Arikan, "Channel polarization: A method for constructing capacity-achieving codes for symmetric binary-input memoryless channels," *IEEE Transactions on Information Theory (TIT)*, vol. 55, no. 7, pp. 3051–3073, Jul. 2009.

[5] I. Tal and A. Vardy, "List decoding of polar codes," in *International Symposium on Information Theory (ISIT)*. IEEE, Jul. 2011, pp. 1–5.

[6] P. Trifonov, "Efficient design and decoding of polar codes," *IEEE Transactions on Communications (TCOM)*, vol. 60, no. 11, pp. 3221–3227, Nov. 2012.

[7] A. Cassagne, O. Hartmann, M. Léonardon, K. He, C. Leroux, R. Tajan, O. Aumage, D. Barthou, T. Tonnellier, V. Pignoly, B. Le Gal, and C. Jégo, "Aff3ct: A fast forward error correction toolbox!" *Elsevier SoftwareX*, vol. 10, p. 100345, Oct. 2019. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S2352711019300457

[8] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770–778.

[9] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.

[10] A. Madry, A. Makelov, L. Schmidt, D. Tsipras, and A. Vladu, "Towards deep learning models resistant to adversarial attacks," *arXiv preprint arXiv:1706.06083*, 2017.

[11] Y. Bengio, N. Léonard, and A. Courville, "Estimating or propagating gradients through stochastic neurons for conditional computation," *arXiv preprint arXiv:1308.3432*, 2013.

[12] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: a simple way to prevent neural networks from overfitting," *The journal of machine learning research*, vol. 15, no. 1, pp. 1929–1958, 2014.

[13] T. DeVries and G. W. Taylor, "Improved regularization of convolutional neural networks with cutout," *arXiv preprint arXiv:1708.04552*, 2017.

[14] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," in *International conference on machine learning*. PMLR, 2015, pp. 448–456.

[15] H. Zhang, M. Cisse, Y. N. Dauphin, and D. Lopez-Paz, "mixup: Beyond empirical risk minimization," *arXiv preprint arXiv:1710.09412*, 2017.