NEURAL NETWORK AND POLAR CODE DESIGN

Batsari Vasiliki
Milasheusi Usevalad
Rezaei Kiarash
Ruan Francesco

Outline:

- Brief Explanation of Polar Codes
- Constraints of the project
- Implementation Phases
  - **Phase I:** Dataset Generation
  - **Phase II:** Data Pre-processing
  - **Phase III:** Neural Network Design
  - Performance Metric
  - Evaluation over network parameters
  - **Phase IV:** FER prediction
- Results
- Conclusion

Purpose of the project

- Improve the efficiency of Polar Codes for SCL decoder
- Using the Neural Network to find the best position of frozen bits for a specific SNR
- Analyze the behavior of the new polar code with different SNR
- Evaluate the performance of the generated frozen bit set

Polar Code

- Linear Block error-correcting code
- Implemented using a generator matrix in a recursive way

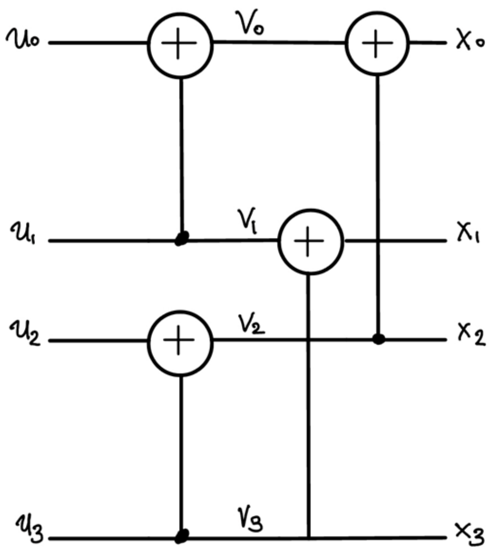
POLAR CODE PC(N,F) CONSTRUCTION

Choose length $N = 2^n$ for some $n \in \mathbb{N}$ with a set F of frozen bits.

$$G_2 = \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix} \qquad G_{2^n} = G_2^{\otimes n}$$

PC(N,F)=$\{PC(N,F) = uG_2^{\otimes n} : u_F = 0,\ u_{F^c} \in \{0,1\}^{|F^c|}\}$

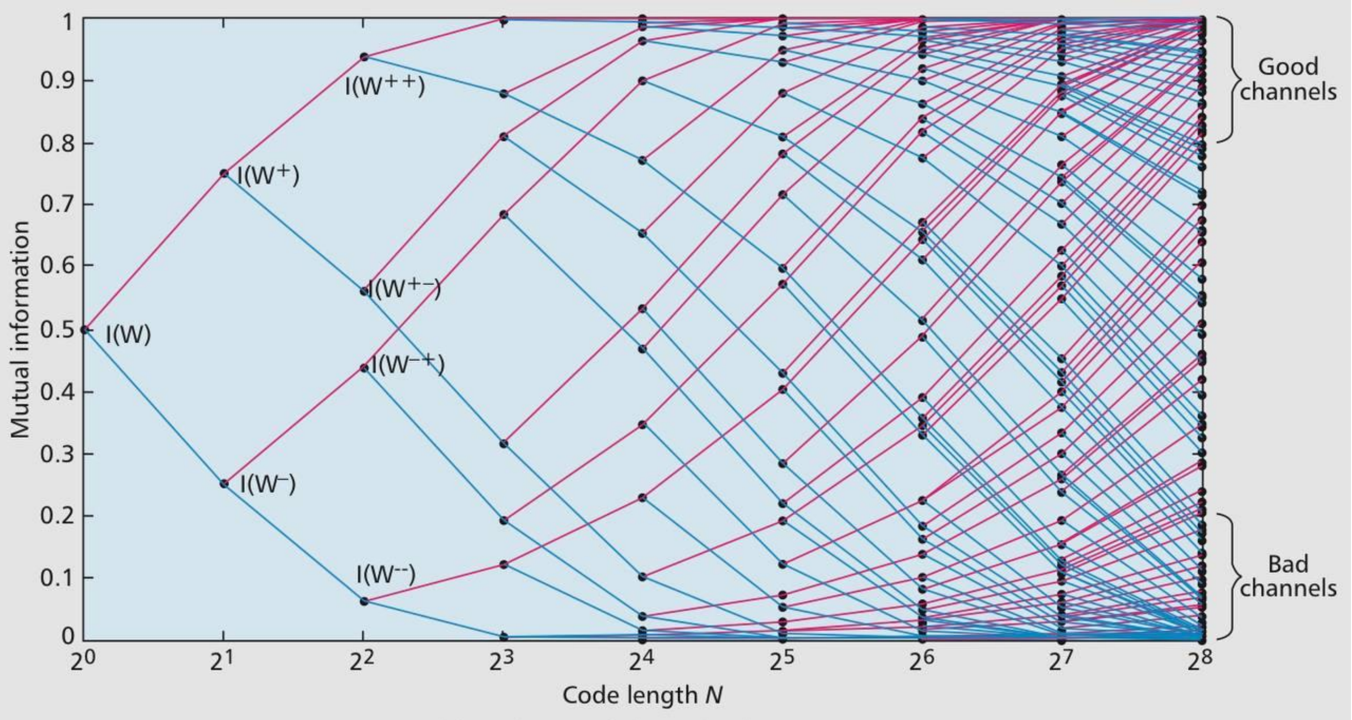
Computational complexity : $O(N\log_2 N)$



Code length N=4 : $x_0^3 = u_0^3\, G_4 = u_0^3 \begin{bmatrix} G_2 & 0 \\ G_2 & G_2 \end{bmatrix} = u_0^3 \begin{bmatrix} 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 1 & 1 & 1 & 1 \end{bmatrix}$

POLAR CODE

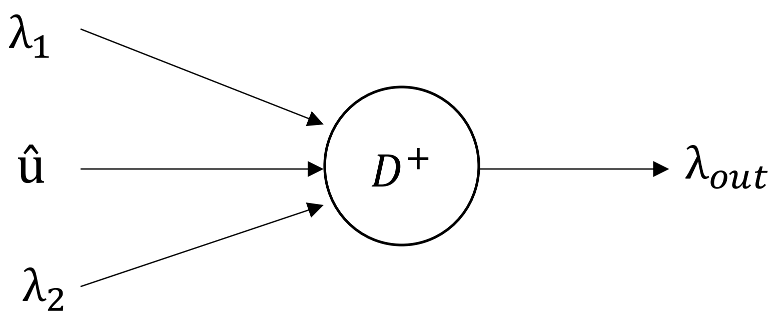
- N-implementation of the polar transformation can polarize the channel, splitting it in virtual channels.
- Reliable channels will improve constantly
- Bad channels will tend to have worst reliability



(d) Evolution of channel polarization

SC Decoder

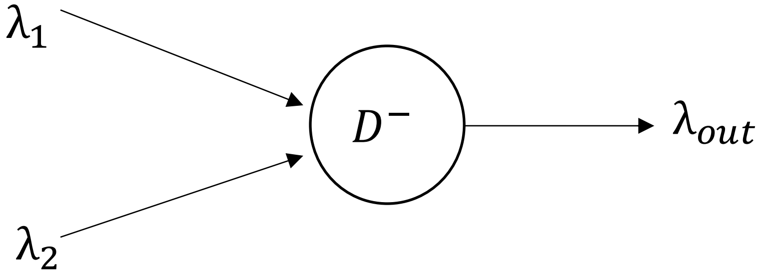
- Decoder D^+:



λ_1 and $\lambda_2$are two log-likelihood ratio and on bit $\hat{u} \in (0,1)$.

$\lambda_{out} = (-1)^{\hat{u}}\,\lambda_1 + \lambda_2$

-Decoder D^-:



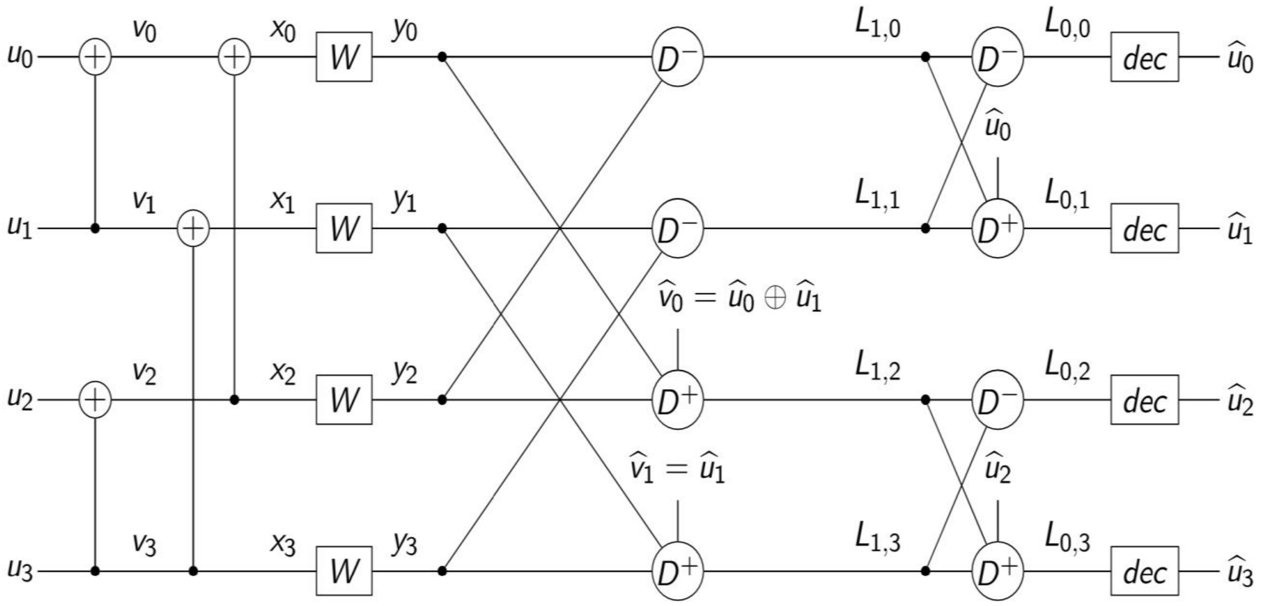
$$\lambda_{out} = 2\tanh^{-1}\left(\tanh\left(\frac{\lambda_1}{2}\right)\tanh\left(\frac{\lambda_2}{2}\right)\right)$$

SC Decoder

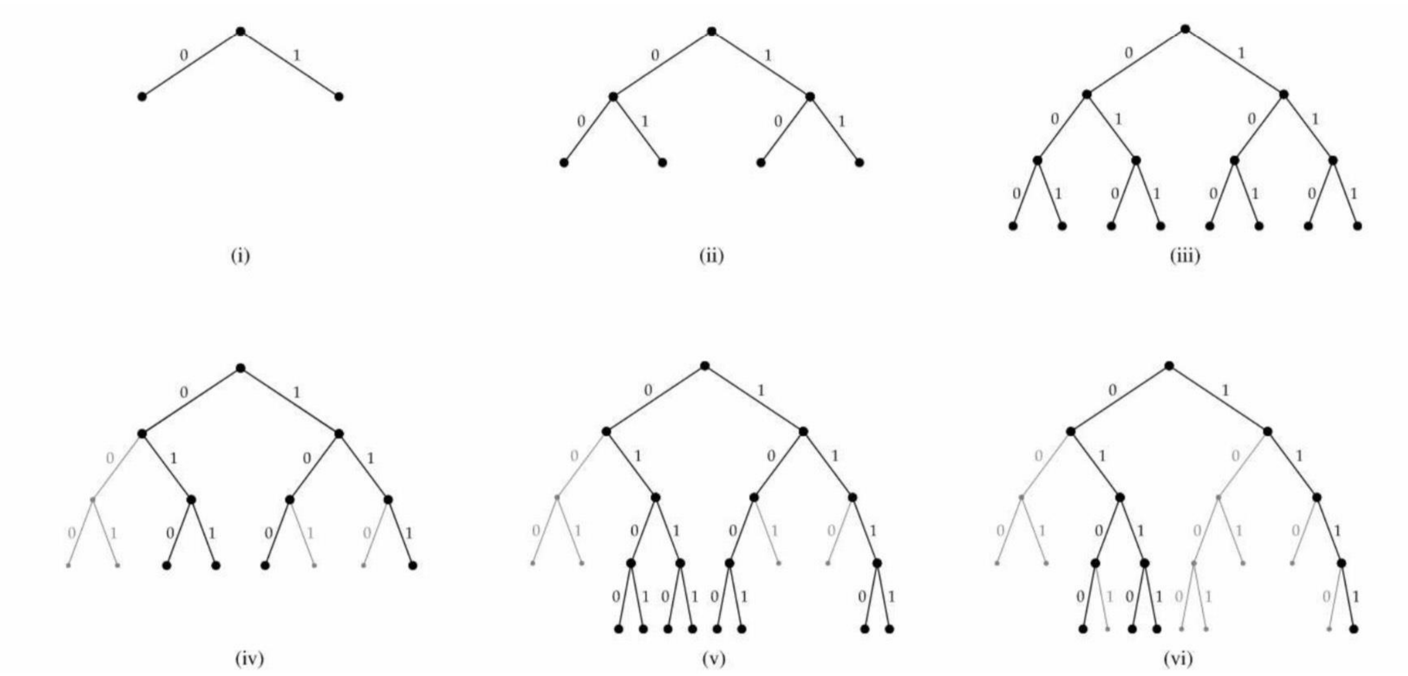
- The process of decoding each u is not run in parallel but it has a specific order.

- Main idea:

$$\hat{u}_0 = \hat{u}_0(Y_0^3)$$
$$\hat{u}_1 = \hat{u}_1(Y_0^3, \hat{u}_0)$$
$$\hat{u}_2 = (Y_0^3, \hat{u}_0^1)$$
$$\hat{u}_3 = (Y_0^3, \hat{u}_0^2)$$



Successive Cancellation List(SCL)



- It takes in consideration both option of $\hat{u}_i$, either it is 0 or 1.
- A path is set of decision on past bits $\hat{u}_i^{i-1}$ for $0 \leq i < N$.
- For each path we associate a specific path metric
- We have a finite list where we collect L path and discard path with the largest metrics.
- The output of the decoder is a the codeword with the smallest path metric

Project Constraints

- Payload (# of information bits) = 64 Bytes
- Rate = 1/2
- Target SNR = 2.7 dB
- Dataset generation method : Gaussian Approximation's permutations
- FER computation is based on Monte Carlo simulation
- The decoder is Successive Cancellation List (SCL) with L parameter equal to 32

** Although we could not provide results for other large codes (Payload = 256, 1K bytes), due to hardware resource limitations, the approach is the same*

Implementation Phases:

Phase I: Dataset Generation

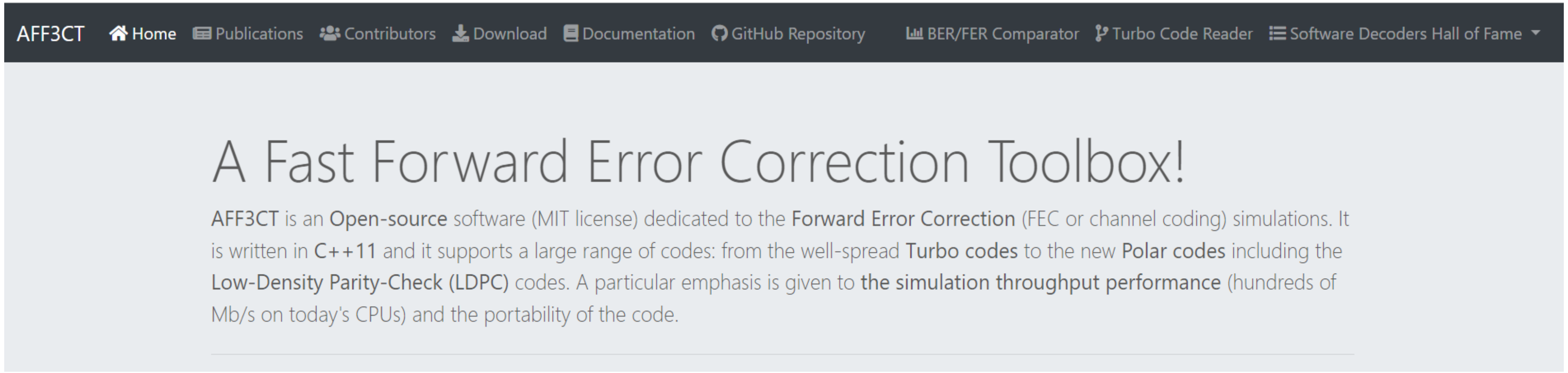
Phase II: Data Pre-processing

Phase III: Neural Network Design

Phase IV: FER Prediction

Phase I: Dataset Generation

- Aff3ct is a Fast Forward Error Correction simulator
- It implements Monte Carlo simulation
- Given a sequence of frozen and information bits' positions as input, it provides us with its FER and BER for a specific SNR or a range of SNR



Phase I: Dataset Generation

- Starting from a frozen bitset for Gaussian approximation (GA), using Python, we applied several random bit permutations in order to get the required number of samples.
- We created ~15K sequences of 1024 bits each, as inputs for the toolbox (payload K = 64 bytes).
- By using the AFF3CT toolbox and some bash scripting, we obtained the corresponding FERs, according to the given parameters.

```
vicky@vicky-VirtualBox:~/Desktop/aff3ct_master_linux_gcc_x64_avx2_8fa65a3ca/build_linux_gcc_x64_avx2/bin$ ./aff3ct -C POLAR -N 1024 -K 51
SCL -L 32 --enc-fb-gen-method FILE --enc-fb-awgn-path ~/Desktop/my_aff3ct/sample.txt
# ----------------------------------------------------
# ---- A FAST FORWARD ERROR CORRECTION TOOLBOX >> ----
# ----------------------------------------------------
# Parameters:
# * Simulation ------------------------------------
#    ** Type                     = BFER
#    ** Type of bits             = int32
#    ** Type of reals            = float32
#    ** Date (UTC)               = 2022-07-11 13:42:38
#    ** Git version              = v3.0.2
#    ** Code type (C)            = POLAR
#    ** Noise range              = 2.7 -> 2.7 dB
#    ** Noise type (E)           = EBN0
#    ** Seed                     = 0
#    ** Statistics               = off
#    ** Debug mode               = off
#    ** Inter frame level        = 1
#    ** Multi-threading (t)      = 4 thread(s)
#    ** Coset approach (c)       = no
#    ** Coded monitoring         = no
#    ** Bad frames tracking      = off
#    ** Bad frames replay        = off
#    ** Bit rate                 = 0.5 (1/2)
# * Source ----------------------------------------
#    ** Type                     = RAND
#    ** Implementation           = STD
#    ** Info. bits (K_info)      = 512
# * Codec -----------------------------------------
#    ** Type                     = POLAR
#    ** Info. bits (K)           = 512
#    ** Codeword size (N_cw)     = 1024
#    ** Frame size (N)           = 1024
#    ** Code rate                = 0.5 (1/2)
# * Encoder ---------------------------------------
#    ** Type                     = POLAR
#    ** Systematic               = yes
```

```
#    ** Num. of lists (L)      = 32
#    ** Polar node types       = {R0,R0L,R1,REP,REPL,SPC}
# * Modem --------------------------------------------
#    ** Type                   = BPSK
#    ** Implementation         = STD
#    ** Bits per symbol        = 1
#    ** Sigma square           = on
# * Channel ------------------------------------------
#    ** Type                   = AWGN
#    ** Implementation         = STD
#    ** Complex                = off
#    ** Add users              = off
# * Monitor ------------------------------------------
#    ** Lazy reduction         = off
#    ** Frame error count (e) = 100
#    ** Compute mutual info    = no
# * Terminal -----------------------------------------
#    ** Show Sigma             = off
#    ** Enabled                = yes
#    ** Frequency (ms)         = 500
#
# The simulation is running...
# ---------------------||------------------------------------------------------||---------------------
#  Signal Noise Ratio  ||   Bit Error Rate (BER) and Frame Error Rate (FER)    ||  Global throughput
#         (SNR)        ||                                                      ||  and elapsed time
# ---------------------||------------------------------------------------------||---------------------
# ----------|----------||----------|----------|----------|----------|----------||----------|----------
#     Es/N0 |    Eb/N0 ||      FRA |       BE |       FE |      BER |      FER ||  SIM_THR |    ET/RT
#      (dB) |     (dB) ||          |          |          |          |          ||   (Mb/s) | (hhmmss)
# ----------|----------||----------|----------|----------|----------|----------||----------|----------
      -0.31 |     2.70 ||    90352 |      486 |       55 | 1.05e-05 | 6.09e-04 ||    0.566 | 00h01'21  x16
# End of the simulation.
```

Bash Scripting...

```
for i in {1..100}
do
    read -r x
    echo "$x" > input"$i".txt
done < InitialInput.txt
```

```
for i in {1..100}
do
 sed -i '1, 3s/^/1024\nawgn\n0.437899\n/' input"$i".txt
done
```

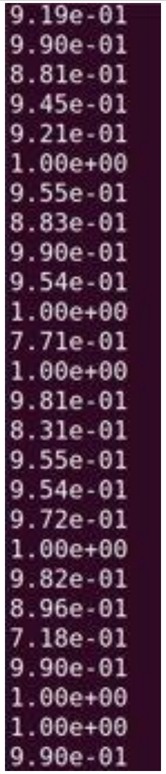
```
for i in {1..100}
do
 ./aff3ct -C POLAR -N 1024 -K 512 -R 2.7 --enc-type POLAR -D SCL --enc-fb-gen-method FILE --enc-fb-awgn-path ~/Desktop/aff3ct_master_linux_gc
c_x64_avx2_8fa65a3ca/build_linux_gcc_x64_avx2/bin/input"$i".txt > output"$i"
done
```

```
for i in {1..100}
do
 awk 'NR==75 {print substr($0,71,8)}' output"$i" > output"$i".txt
done
```

```
cat output{1..100}.txt >> the_real_output.txt
```

Phase I: Dataset Generation

- The results had really high FER (~ 0.9), even if we tried to generate the input sequences with many different ways.
- So, we could not use them to feed the neural network, as the latter would not be able to generalize and give us an optimal output.
- Taking, however, as inputs, the sequences of the paper provided to us, we extracted the same results as it did, which could serve as an input in our neural network.
- Thus, we replicated the paper's results, taking the same samples of bit-sequences and corresponding FERs.



Phase II: Data Pre-processing

- We have separated the input dataset into **bit sequences dataset** and corresponding **FER dataset**
- Each bit with value 1 is a frozen bit and each one with value 0 is an information bit
- The Mean value and standard deviation (STD) for each position of bits is computed.
- We call the positions with STD not equal to zero, **varying bit** positions (for each sample)
- Standardization function: we standardize varying bits [(Bit – Mean(Bit))/ STD] and use them as the new training set
- Now for the new training and validation sets, **info bit** values are equal to **–1** and frozen bit values are equal to 1 and the dimension reduced to the number of position of varying bits

Phase II: Data Pre-processing

REASON: to bring down all the features to a common scale without distorting the differences in the range of the values. Also, reducing the dimension and use the features(varying bits) that affect the performance in order not to feed the model with useless data in training process

Phase II: Data Pre-processing

PAYLOAD = 64 Bytes

- Size of training set: 12689 samples
- Size of validation set: 3173 samples
- Number of varying bit positions: 112

Phase III: Neural Network Design

- **Training:**
  - We used **MSE** as the loss function and **Adam optimizer** for learning rate optimization.
  - The approach is to predict FERs for **each batch** (batch_size= 32, to reduce the computation time) of input data. Computing the loss function, updating the gradients and learning rate afterwards in an iterative process

Performance Metric

In order to evaluate the performance of the neural network, *inflation of error (IOE)* was used as a metric, which indicates how inaccurate the output is with respect to the true value

$$\mathrm{IOE}(f, F) = \max\left\{\frac{\mathrm{FER}_f}{\exp(F(f))}, \frac{\exp(F(f))}{\mathrm{FER}_f}\right\} - 1.$$

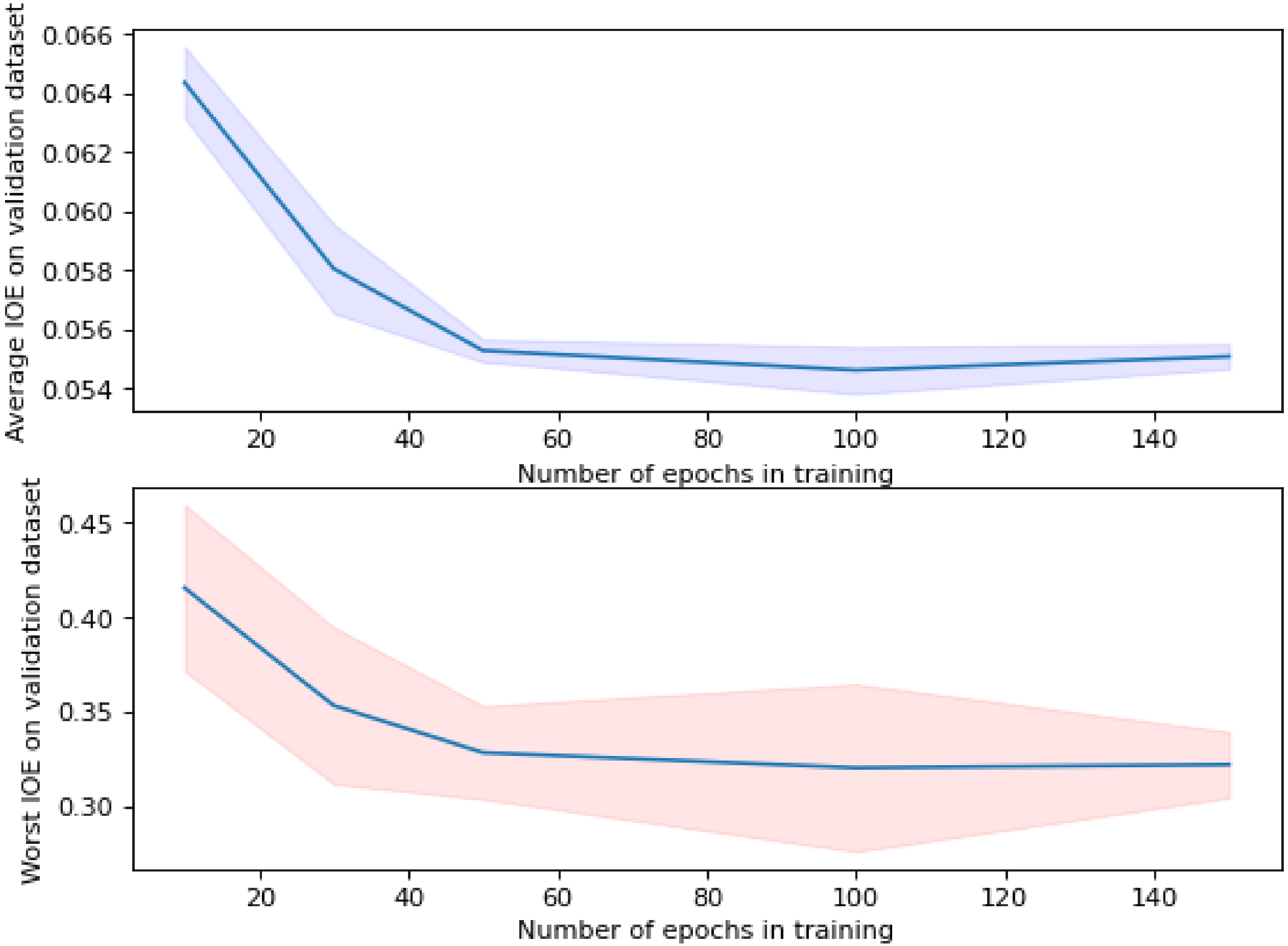
Ex. IOE is 5%, which means our model undershoots / overshoots by 5 percent wrt. the ground truth

Evaluation over the Network Parameters

We have investigated the effect of changing the following parameters on the performance of our network to optimize its performance. Thus, we test the network based on them.

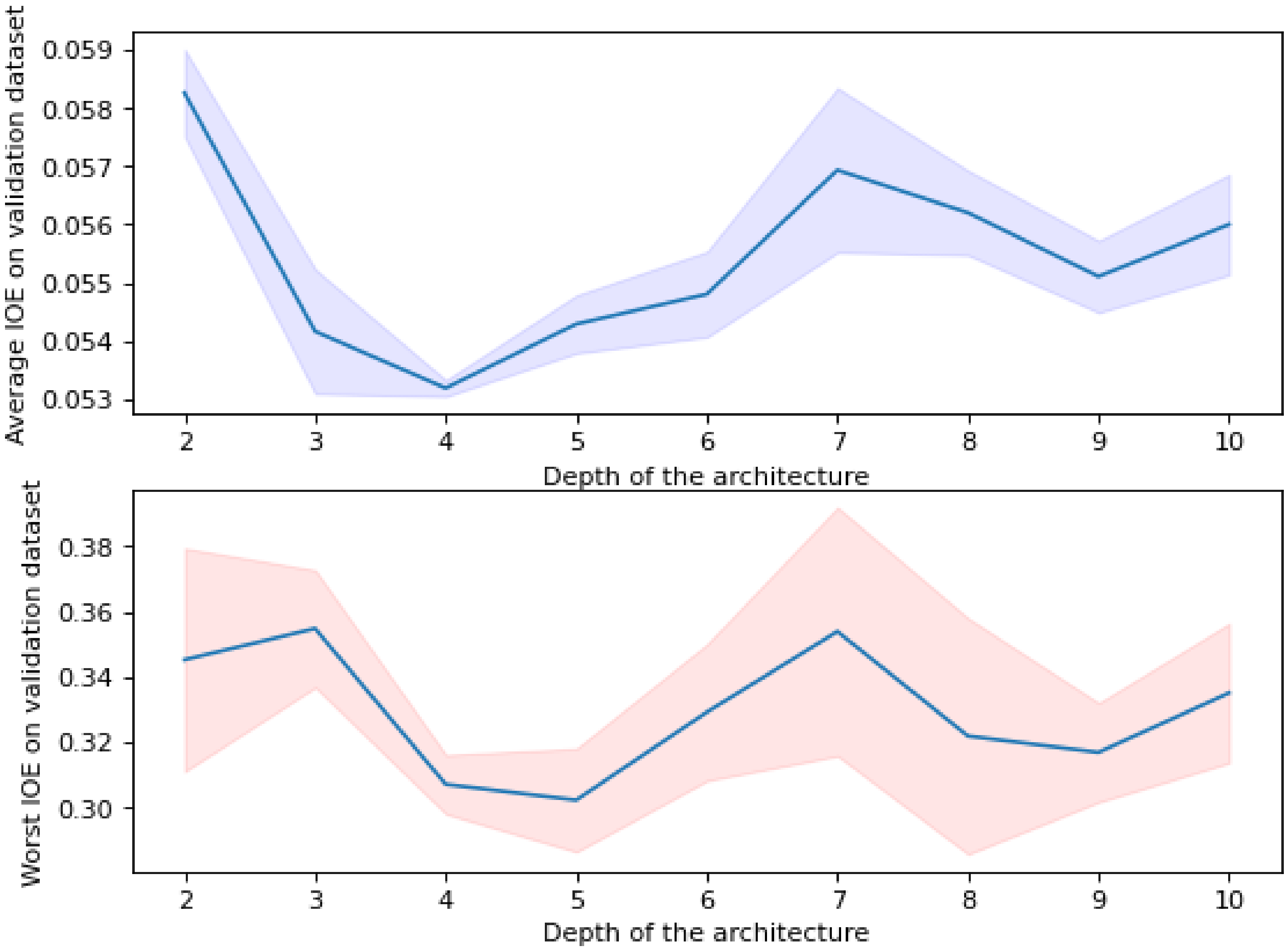
1. The number of Epochs
2. The number of layers (Depth)
3. The number of Skip connections
4. The number of neurons in hidden layers

Evaluation of IOE over the number of epochs:



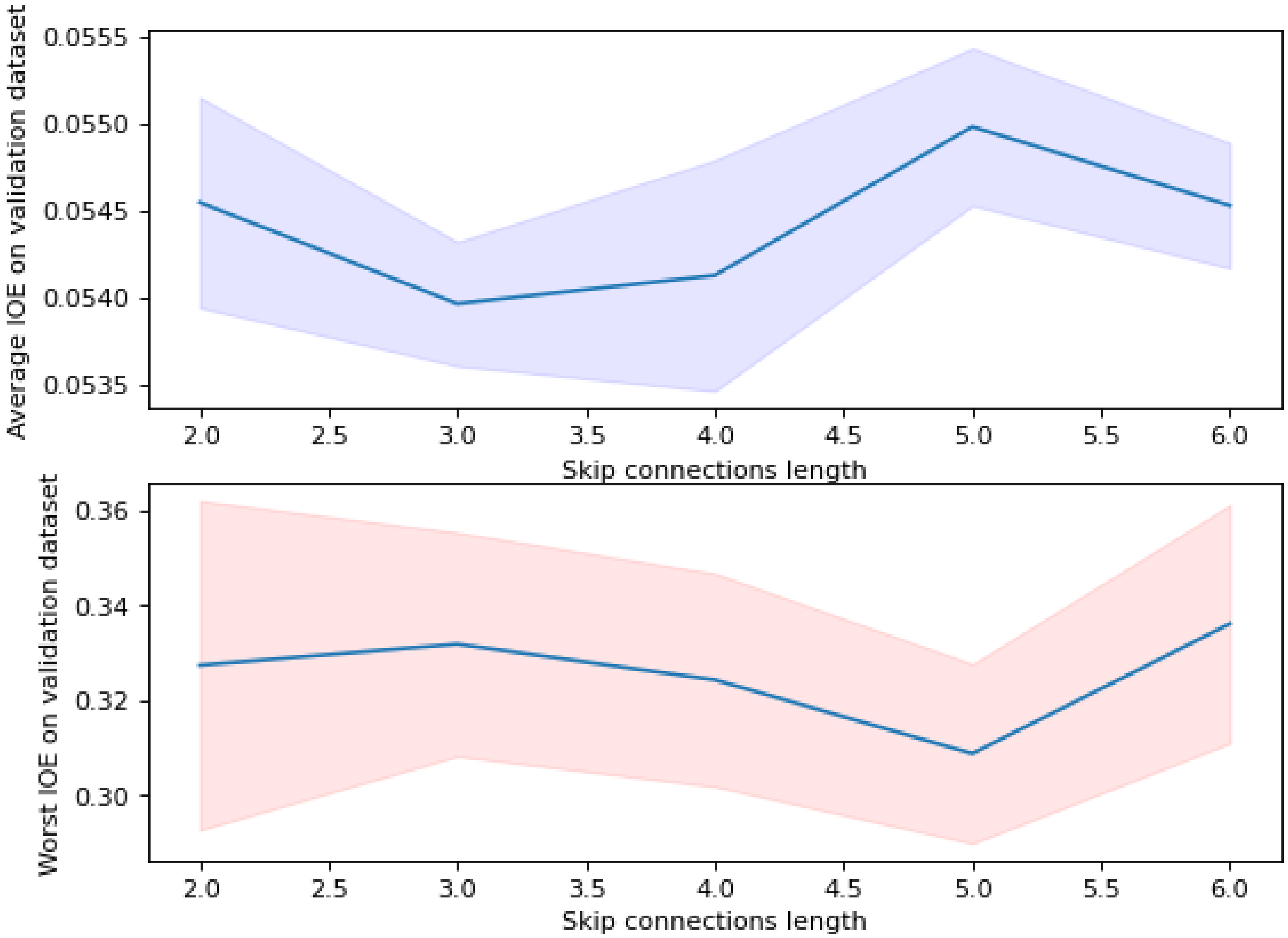
* Confidence bound over 5 runs

Evaluation of IOE over the number of layers(depth):



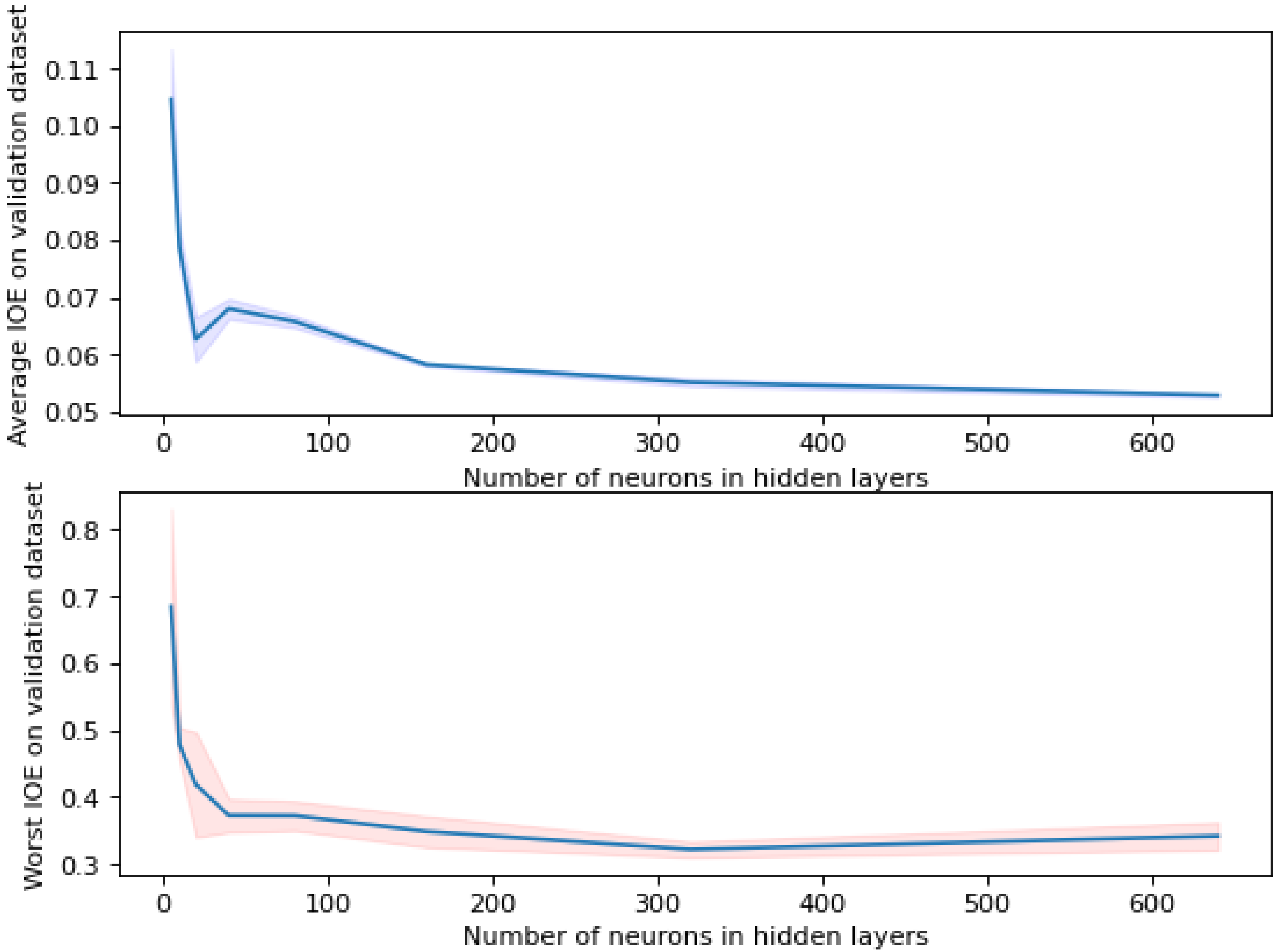
* Confidence bound over 5 runs

Evaluation of IOE over the length of skip connections:



* Confidence bound over 5 runs

Evaluation of IOE over the number of neurons in hidden layers:



* Confidence bound over 5 runs

Phase IV: FER Prediction

In order to predict an optimal frozen bit set two following methods were used:

"Projected Gradient Descent " and *"Random Search"* algorithms. Abstract explanation for both of them is to find the input such that the output of the function (NN in our case) is maximized.

Phase IV: FER Prediction

- **Random Search:** using an ensemble of several pretrained networks, for a given amount of time we input 1000 random string of bits (maintaining R = ½) and acquire a set of FERs corresponding to the input. By taking the average by an ensemble , we store the minimal FER predicted. This process is repeated for a fixed amount of time.

Phase IV: FER Prediction

- **Projected Gradient Descent(PGD).** Previously described method is very straight forward and does not use any heuristics whatsoever. In this method, having frozen the parameters of the network, we treat the input as a parameter which we update using SGD in order to find the global minimum. The parameters inputted are the number of iterations to be performed and the learning rate.
- Generally, we generate new varying bits and use them as the input of the trained network in order to fool it

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**
    $f_i \leftarrow f_i - \mu \frac{\partial y}{\partial \tilde{f}_i}$
Return quantized version of $\boldsymbol{f}$

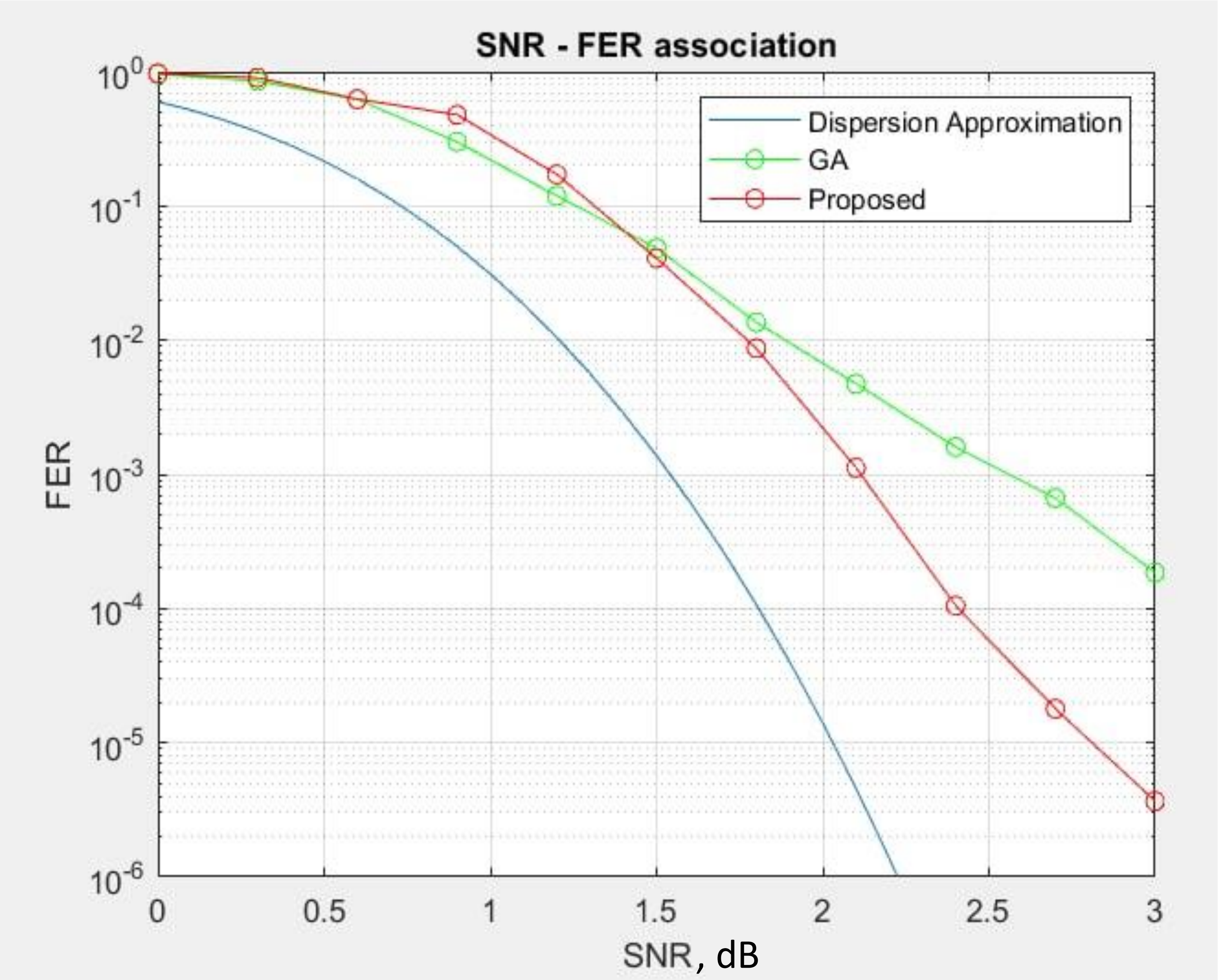
Phase IV: FER Prediction

The parameters of the model used for frozen bitset generation are:

- Depth: 3
- Hidden layer size: 640 neurons
- Skip gaps: 3
- Trained on 100 epochs
- Lamda = 0.1
- # iterations = 2000

Results:

- The proposed sequence has an FER equal to 1.79e-05, for SNR = 2.7 dB



Conclusion

- The proposed frozen bit set which is driven for target SNR = 2.7 dB, can give us reliable results even for other values of SNR. It is closer to the lower limit and has a better performance than our initial assumption on GA, since it requires smaller SNR for the same value of FER.