Neural Decoding Architecture for Parity Based Error Correction Codes

Andreea-Lavinia Diniș and Ștefan Malanik
Politehnica University of Timișoara

***Abstract*—The project targets the development of an error correction scheme for parity based codes using machine learning approaches. We aim for a custom deep neural network, based on the Tanner graph of the error correction code, using Min-Sum based algorithm in log domain. The off-line trained neural network will be implemented on a ZedBoard platform.**

I. Introduction

The new low redundancy and reduced overhead codes (LRRO) cover the same advantages as Bose - Chaudhuri - Hocquenghem (BCH) codes, but have a reduced overhead. The main focus of this project is to implement a more lightweight version of the Belief Propagation (BP) decoder using a deep neural network by adapting the offset min-sum (OMS) algorithm.

This adaptation of the offset min-sum algorithm can be easier implemented in hardware as it uses the min-sum approximation instead of repeated multiplications as in the BP decoder case, thus the former being a better option when considering practical decoder implementations.

During earlier development stages, the neural network was generalized to accept BCH codes as it was seen that the proposed LRRO code did not perform as expected for the adapted BP decoder..

II. Python Model

A. General description

The Python implementation of the neural network model makes use of the Keras + Tensorflow libraries and closely reproduces the described procedures presented in paper [1]. The model has multiple components:

- variable node (odd) layer, performing multiple additions akin to a multiplication by a matrix that only contains zeros and ones, not possessing any weights.
- check node (even) layer, computing the minimum absolute value of a set of edges which gets adjusted by a bias acting as a weight.
- output layer

Each of the two complementary layers operate on units representing Tanner Graph edges (Fig. 1).

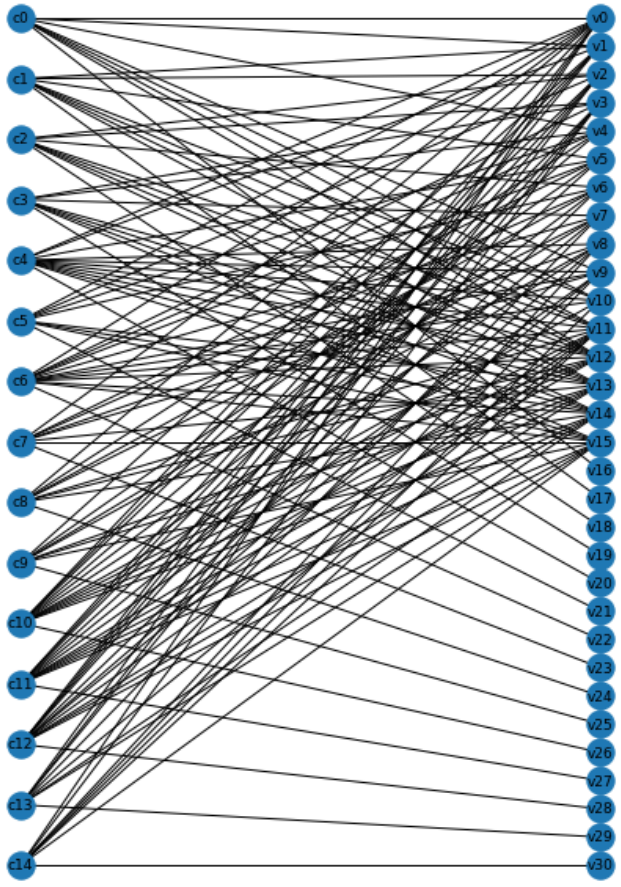

Fig.1 Tanner Graph of BCH(31, 16)

The first two components represent complementary layers that are repeated for a fixed number of times, generating separate weights for each iteration. The repetitive nature of the decoding algorithm is revealed through the structure of the variable node layer, since the output of the previous iteration's even layer becomes its input in the following iteration. In addition, the initial codeword is also received as an input on every iteration.

The biases of the check node layer are specific to each individual iteration, and they represent the only trainable weight contained by the model. Their purpose [1] is to minimize the influence of cycles in the Tanner Graph, which degrade the performance of a decoder for a given encoding scheme.

B. *Channel and codeword properties*

The channel model employed for testing and data generation is the binary symmetric channel. However, additive white Gaussian noise channels are the most common model used in the mentioned references.

For this project's purposes, there are two representations of any given codeword:

- hard decision, consisting of the initial 1's and 0's that are sent or received through the use of a binary symmetric channel. Each element of data has a size of exactly 1 bit.
- LLR (log-likelihood ratio), explained in detail in [2], consisting of an attribution of a value to each bit, accounting for the errors added by the channel if the noise characteristics are known. The use of a binary symmetric channel implies that there are only two possible values at its output. As a result, applying the LLR principle maps the 0's and 1's to two unique and opposite values, depending on the crossover probability. Each element of data will either be a float, or will have a size of a constant number of bytes, depending on the desired precision.

The atomic unit of data exchanged by layers is always in the LLR form within the model. As an abstract black box, however, the model expects the input data in the LLR form, and outputs a codeword as a hard decision.

C. *Integer conversion*

During training, the Python model uses floating point values throughout to properly perform gradient descent. The hardware model has to implement an integer-only decoder, however. As a result, the issue of conversion arises.

To solve this problem, a transformation from values in a floating point format to integers is performed, by multiplying the value with a power of two (indicating the decimal place), then rounding to integers.

A convenient size for the integer value was found to be 8 bits, with a decimal size of 4. This represents the minimum space occupied by a single LLR, without any significant increase of the BER(Bit Error Rate) or FER(Frame Error Rate) metrics.

D. *Mathematical model of error correction codes used*

As per [2], the min-sum approximation of BP can be used on the LDPC class of codes. The referenced article studies the behavior of BCH codes, and they were found to perform as mentioned.

Since this project's main focus is the hardware implementation of the BP approximation, an LDPC code with a sparse matrix is ideal for this particular use case. However, LRRO codes, presented in [3], had a very poor performance compared to BCH, as can be seen in Fig. 2.

E. *Model training and data export*

The model was trained with an Adam optimizer that had a learning rate of 0.01 and by batching the input data in chunks of 120 codewords.

The final adjacency matrices (containing just ones and zeros) specific to all layers were saved in an .npz format by using the Numpy library. Biases obtained after the model training were first passed through the integer converter and then also saved in a different .npy file. The generator matrix was saved as well to have a way to test the decoder by creating new random codewords.

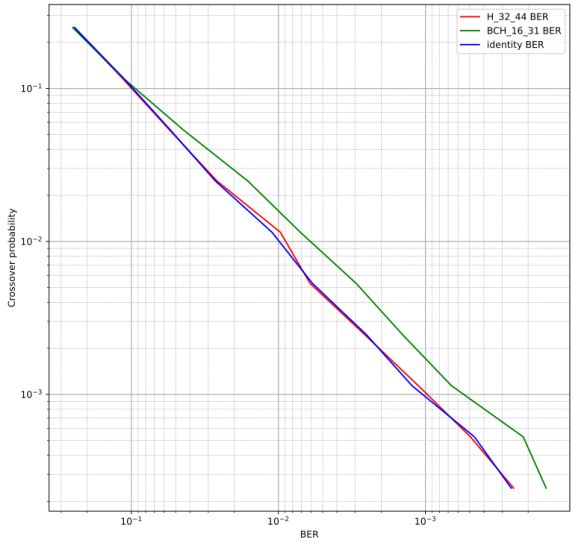

Fig. 2 Log Scale of Crossover Probability and BER, for BCH(BCH_16_31), LRRO(H_32_44) and Identity Matrix encoding

III. C Model & Testbench

A. *Top-level view of the testbench*

The testbench has a parallelized structure, as can be seen from Fig. 3. It consists of four tasks running in a join block, and three message boxes to synchronize the input and output codewords.

The DPI-C interface is also employed to access a custom C library. The decoding logic is implemented separately in C for testing purposes.

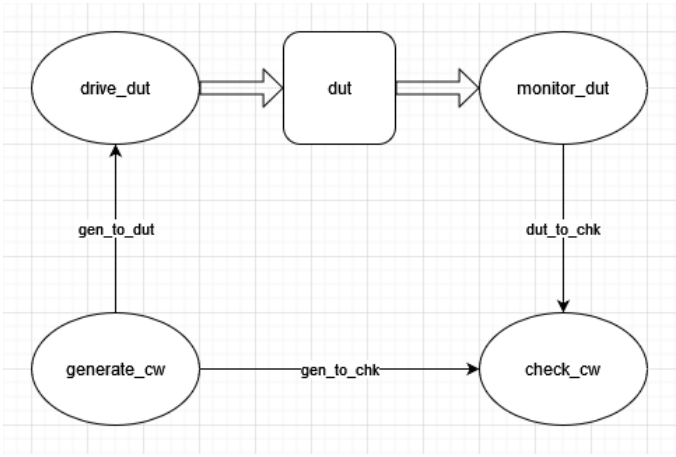

Fig. 3 Overview of testbench
Circles indicate parallel tasks. Full arrows indicate message boxes

B. Parallel tasks description & interfacing the DUT

The first task, generate_cw, calls the C interface to generate a new random codeword that has passed through a channel. Following that, it gets transmitted to the driver task, and it also gets processed by another C function that emulates the whole Python model and sent to the final checker task.

The tasks associated with the DUT, namely drive_dut and monitor_dut, handle the mapping of the full data stored together onto multiple chunks sent or received sequentially, based on the device interface.

The last task, check_cw, collects both the C interface and hardware decoder results, for comparison and output.

C. DPI-C library

The C code specific to the project has its own makefile and test script. It emulates various logical data elements (matrices) and layers through the use of structures.

All data specific to one code type (codeword size, number of edges), but also to design decisions (LLR size and the two possible values associated with 0's and 1's) is parametrized. As a result, the data obtained from the Python model and stored in the numpy specific format had to be converted into a C-friendly format. To that end, a Python script that generates an import C file was created. The generated C file is then integrated normally in the library.

To ensure that the model mirrored the hardware as accurately as possible, it only uses integer LLRs and implements a saturation behavior, described further in the HDL implementation.

IV. HDL Model

A. General description

The equivalent of the NOMS decoder was implemented in hardware using modules for the different types of layers as presented in the beginning, which were generated mainly based on adjacency matrices connecting different layers. A schematic of the top-level module can be seen in Fig. 5

B. Top module finite state machine

The logic of the top module (Fig. 4) was designed keeping in mind three main steps of the decoding part: the initial loading of the LLRs, the processing step and the offloading of the outputted and corrected codeword.

- loading part consists of sequential reading stages of segments of the entire input codeword
- processing part deals with passing the LLRs of the inputted codeword through the layers of the model, resulting a decoded and corrected codeword
- outputting part sequentially writes the output on the databus

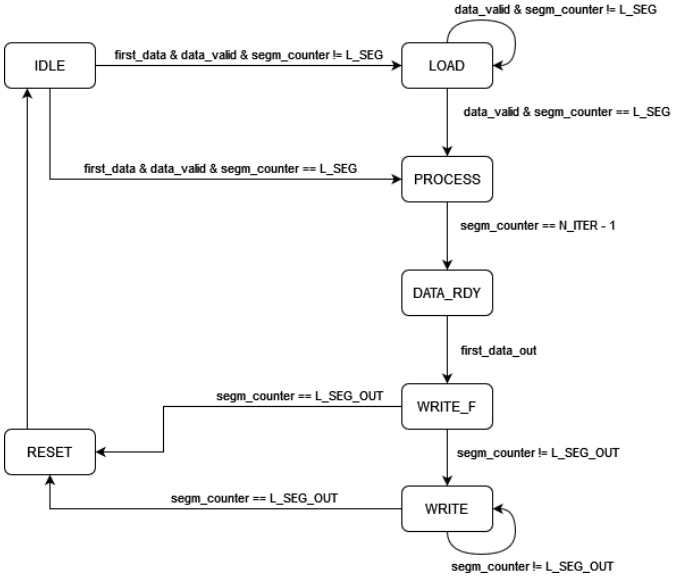

Fig. 4 Decoder State Diagram

C. Code generation

To ensure a versatile decoder that can work on any model trained by the Python neural network, the following key files had to be generated by using another Python script:

- variable node layer (variable_nodes.v)
- check node layer (check_nodes.v)
- output layer (out_layer.v)
- LUT for biases (lut_biases.v)

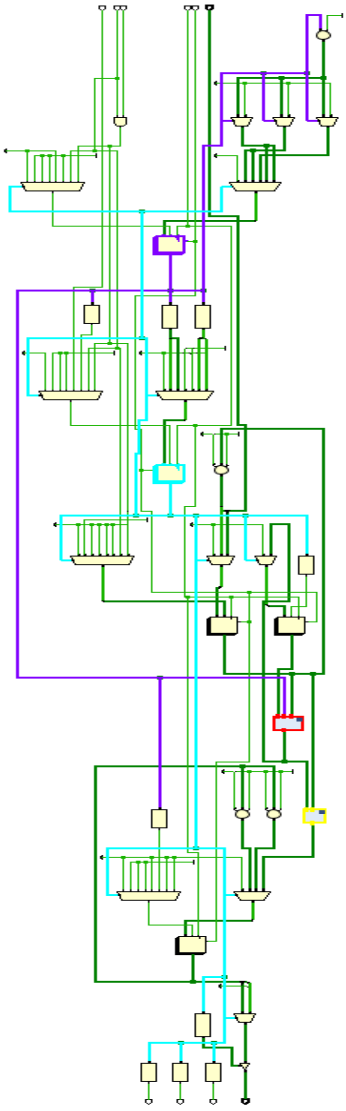

Fig. 5 RTL Model of Decoder
Cyan - State
Purple - Segment Counter
Red - Intermediate Layer
Yellow - Output Layer

All generated layers are designed to be combinational. Variable and check node layers are grouped in an intermediate layer module that solely handles the iterations of the approximated BP algorithm. The LUT layer is also part of the intermediate module, and stores all the relevant biases for each particular iteration.

D. Saturation

The saturation module is essential to ensure that no overflow occurs during the repeated additions in the variable node and output layers, as well as the bias addition in the check node layer. Initially, to fully prevent overflow, a number of extended bits was chosen (4 bits, which translate to 16 “safe” additions).

The saturation module establishes that the extra bits appended are not at all used to store data, and if they are, the final output will be the maximum (or minimum) value possible on the unextended number of bits.

V. Conclusions

➔ The LRRO codes have not worked as expected, even though they are designed particularly for hardware implementation due to their sparsity.

➔ Initially, Matlab was supposed to be the environment in which the neural network was created and trained. However, some obstacles were encountered regarding the application of gradient descent, and the logic was ported into Python.

➔ There are some discrepancies between the C model and the Python model results, which is to be expected as there are big differences between having a full-precision floating-point-valued model and an integer-based one.

➔ Unfortunately, the code was not uploaded on any hardware, even though the project synthesizes properly and the simulation indicates that it is working as expected

References

[1] E. Nachmani, E. Marciano, L. Lugosch, W. J. Gross, D. Burshtein and Y. Be’ery, "Deep Learning Methods for Improved Decoding of Linear Codes," in IEEE Journal of Selected Topics in Signal Processing, vol. 12, no. 1, pp. 119-131, Feb. 2018, doi: 10.1109/JSTSP.2017.2788405.

[2] L. Lugosch and W. J. Gross, "Neural offset min-sum decoding," *2017 IEEE International Symposium on Information Theory (ISIT)*, 2017, pp. 1361-1365, doi: 10.1109/ISIT.2017.8006751.

[3] Saiz-Adalid, L.-J.; Gracia-Morán, J.; Gil-Tomás, D.; Baraza-Calvo, J.-C.; Gil-Vicente, P.-J. Reducing the Overhead of BCH Codes: New Double Error Correction Codes. *Electronics* **2020**, *9*, 1897. https://doi.org/10.3390/electronics9111897