Hardware Implementation Aspects of a Syndrome-based Neural Network Decoder for BCH Codes

E. Kavvousanos and V. Paliouras
Electrical and Computer Engineering Department
University of Patras, Greece

***Abstract*—Deep-Learning-based Decoders have been recently introduced for use with short-length codes. They have been found to act as a Soft-Decision-Decoding method achieving near Maximum-Likelihood error correcting capability. However, Deep-Learning decoding methods are hard to implement as they normally require millions of operations for inference. In order for Deep-Learning decoding to be a competitive candidate for practical applications, research effort is required to reduce the computational complexity and storage requirements of the Neural Networks involved. In this work, a structured flow is presented that significantly compresses a trained Syndrome-Based Neural Network Decoder by pruning up to 80% of the network's weights and quantizing them to 8-bit fixed-point representation, with no loss in its BER performance. The attained compressed Neural Network can then be used for inference, by designing specific hardware or by using a generic Deep-Learning hardware accelerator that exploits the compressed structure of the network. The deployment of the DL Decoder in an embedded application is showcased, using the AI Edge platform by Xilinx. To accomplish this, a simple method to obtain a computationally equivalent convolutional layer from a fully-connected one is introduced. Implementation results are provided for the compressed DL Decoder, regarding throughput rate and BER performance. To our knowledge, this is the first DL decoder in hardware reported.**

I. Introduction

In recent years, Deep Learning has gained paramount attention as it is a powerful problem-solving method in extremely diverse fields. Different types of Neural Networks can be trained to perform a task with high accuracy. The effectiveness of such networks can surpass even humans in computer vision and language processing problems. Beyond the typical aforementioned applications, Deep Learning proliferates in numerous diverse fields.

Several works have investigated the adoption of Deep Learning techniques in Telecommunications, including the area of Error Correction. Error Correcting Codes are used to achieve reliable communication over unreliable channels. The transmitter encodes an information word into a codeword and sends it to the receiver via the noisy channel. The receiver attempts to decode the received word using a Decoding Algorithm that exploits the structure of the code used. The idea in DL decoders is to train a Neural Network to decode the received word. Interestingly, such decoding networks can be trained for different codes and noise characteristics, potentially leading towards a universal decoding framework, allowing for a single architecture to support several evolving and diverse standards existing in the contemporary telecom landscape and envisioned for the future.

Nachmani *et al.* [1] borrow the idea from the Belief Propagation (BP) Algorithm, used for decoding LDPC codes, to construct a Neural Network that is the unfolded and weighted equivalent of the algorithm's execution on the code's Tanner Graph. The weights of the Neural Network are obtained by training, therefore the graph can be optimized for other classes of codes such as BCH. In this work, regular and recurrent Neural Network architectures are demonstrated. While the approach by Nachmani *et al.* [1] shows better results than the original BP Algorithm used with BCH codes, it highly constraints the design of the Neural Network.

Another approach, introduced by Bennatan *et al.* [2], uses a Neural Network framework in accordance with modern Deep Learning techniques. The framework consists of three parts: a pre-prepocessing stage, a noise-estimation stage and a post-processing stage. The pre-processing stage calculates the input to the Neural Network, which consists of the absolute value of the received sequence reliabilities along with the calculated syndrome. The noise estimation stage, which is implemented by the Neural Network, identifies the erroneous bits in the sequence. Lastly, the post-processing stage estimates the transmitted codeword. The advantage of this approach is that the noise estimation Neural Network can be designed freely. Thus, different architectures can be implemented, from plain Fully Connected Networks to Recurrent Neural Networks, trading computational complexity and accuracy.

The disadvantage of these methods is that they are only applicable for short codes with length of approx. 100 bits. As the code length increases, larger and more complex Neural Network architectures are needed along with larger training datasets, which render the procedure of training and inference, computationally inefficient. This issue is known as the curse of dimensionality.

In general, Neural Networks are computationally intensive

978-1-7281-2769-9/19/$31.00 ©2019 IEEE

algorithms, with hundreds of thousands to millions of parameters which translate to an equal amount of multiply-accumulate operations. In order to deploy Deep Learning models to embedded and mobile devices, researchers have been working on methods to compress the size of Neural Networks in terms of storage and required computations. Such methods are parameter pruning and quantization.

Pruning is the process of eliminating unnecessary parameters the contribution of which is insignificant to the function of the network. Magnitude-based pruning methods are mostly used as they are easily implemented in comparison with other methods that require complex mathematical computations [3]. In magnitude-based pruning, the network parameters with the lowest magnitude are set to zero, removing completely their contribution to the output of the network. The pruning algorithm is usually iterative, pruning away low magnitude weights and retraining the rest of the weights at each step [4], [5], [6]. Reasonably, there is a loss in the accuracy of the network when pruned.

Neural Network quantization reduces the number of bits required to store each weight. The training process is usually performed in single or double floating-point precision. In most common cases, the trained network parameters are quantized to 8 bits, achieving 4–8× compression of the network.

Having compressed the Neural Network by pruning and quantization, special hardware accelerators can be designed that take advantage of these models. Such accelerators [7], [8] outperform CPU and GPU implementations in both runtime and energy efficiency.

In this paper, we extend the work in [9] by training a Fully-Connected Neural Network Syndrome-Based Decoder [2] for BCH(63,45) code. Then, systematic pruning and quantization is applied to achieve 80% weight sparsity and 4× model compression with minimal loss in accuracy. The introduced procedure allows the implementation of the Neural Network decoder in special hardware designed for Deep-Learning inference. In this case, the compressed model is deployed using the DNNDK Framework by Xilinx [10], setting an initial attempt to adopt Deep Learning Decoders in embedded environments. To the best of our knowledge, this paper describes the first DL-based error-correction decoder in hardware.

The remainder of this paper is organized as follows: Section II details the proposed workflow for the compression of a Syndrome-Based Neural Network Decoder. Section III describes the implementation of that Neural Network Decoder in Deep-Learning hardware accelerators. Section IV presents the implementation results, in terms of throughput rate and BER performance. Finally, in Section V, our findings are discussed and the paper is concluded.

II. Overview of the Syndrome-Based NN Decoder

A. Communication model

In our analysis, we assume communication over an Additive White Gaussian Noise (AWGN) Channel, using a binary, systematic BCH(63,45) code and BPSK modulation. Let $\boldsymbol{x}$ be the transmitted codeword and $\boldsymbol{y}$ be the received sequence (channel reliabilities) at the receiver, both in bipolar form. It holds that $\boldsymbol{y} = \boldsymbol{x} + \tilde{\boldsymbol{n}}$, where $\tilde{\boldsymbol{n}} \sim \mathcal{N}(0, \sigma^2)$ and σ^2 is the channel noise variance. We denote the binary hard decision on the received sequence as $\boldsymbol{y}_b$.

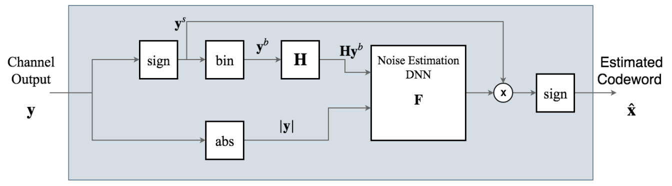

Fig. 1. Syndrome-Based NN Decoder Organization.

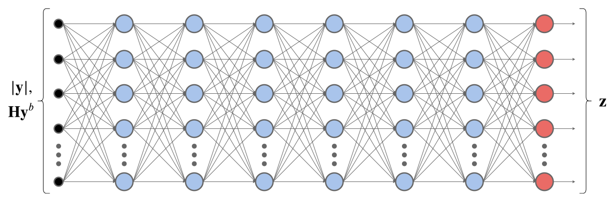

Fig. 2. Neural Network Decoder Structure.
Black: Input Values, Blue: Hidden Layer, Red: Output Layer

B. Neural Network Decoder

Our main focus is the optimization for hardware implementation of a Syndrome-Based Neural Network Decoder [2], trained for a binary BCH(63,45) code. The input of the Neural Network consists of the absolute value of the channel reliabilities $|\boldsymbol{y}|$ and the syndrome of the transmitted codeword $\boldsymbol{s} = \boldsymbol{H}\boldsymbol{y}_b$, where $\boldsymbol{H}$ is the parity check matrix of the evaluated block code. For a (N, K) block code the size of the input vector is $2N - K$, where K is the size of the information word and N is the size of the codeword. In our case, we have an input vector with 81 values, *i.e.*, 18 syndrome values and 63 absolute values of the channel reliabilities. The organization of the decoder is shown in Fig. 1.

The proposed Neural Network consists of seven Fully Connected Layers. The first six layers comprise 300 units each, using a Rectified Linear Unit (ReLu) as activation function. The output layer has 63 units, employing hyperbolic tangent for activation function. Each unit of the output layer corresponds to the respective bit in the codeword (*i.e.*, the first unit corresponds to the first bit, the second unit to the second bit, *etc.*) and acts as a binary classifier, which classifies the corresponding bit as either correct or erroneous. Thus, the output of the network can be interpreted as an estimated error pattern for the received sequence. As $\tanh(x) : \mathbb{R} \rightarrow [-1, 1]$, we conventionally assign the value 1 to represent a correct bit and the value -1 to represent a bit in error. The structure of the proposed Neural Network is visualized in Fig. 2.

C. Training Configuration

Regarding the training of the Neural Network, the transmission of the all-zeros codeword through an AWGN channel was simulated. A dataset of 100 million transmitted codewords was used along with their respective error patterns, out of which 90 million were used for training and 10 million for validation. The level of the simulated noise was chosen to be fixed at $E_b/N_0 = 4$ dB. The noise level during training should be chosen considering the error correcting capability of the code and the average bit error rate at that level. In our case, the BCH(63,45) code is capable of correcting up to three errors and the average bit error rate per transmitted codeword at $E_b/N_0 = 4$ dB is 1.8 bits. This way, the neural network encounters sufficient error patterns, the majority of which can be corrected by the code being learnt.

The network parameters are initialized according to a normal distribution. The Adam optimizer [11] was used for their optimization, using sample batches of size 2048. The learning rate for the gradient propagation is initialized to 10^{-3} and is reduced by a factor of 10^{-1} when the validation loss stops reducing for 5 epochs. In order to assist the process of parameter quantization later in our workflow, maximum absolute value constraint was applied to the parameters, so they lie in the range $[-1, 1]$, limiting their dynamic range. It was observed that this constraint had a negligible impact on the network performance. The binary cross-entropy was used as the loss function L of the network, *i.e.*,

$$L = -\frac{1}{N}\sum_{n=1}^{N}\left\{\tilde{t}_n \log\left(\tilde{z}_n\right) + \left(1-\tilde{t}_n\right)\log\left(1-\tilde{z}_n\right)\right\}, \quad (1)$$

where t_n is the *desired (target)* n-th output of the network corresponding to the n-th bit of the codeword, z_n is the actual n-th network output. The tilde symbol is used to emphasize that these values are mapped from $[-1, 1]$ to $[0, 1]$.

The training of the Neural Network is accomplished using Keras [12] along with TensorFlow [13]. In order to quantify the performance of our decoder during training, we define its accuracy A as

$$A = \frac{1}{N}\sum_{n=1}^{N}\text{equal}\left\{t_n, \text{sign}(z_n)\right\}, \quad (2)$$

where the equal function is defined as

$$\text{equal}(a,b) = \begin{cases} 0, \text{if } a \neq b \\ 1, \text{if } a = b \end{cases} \quad (3)$$

and $\text{sign}(x)$ returns the sign of its argument.

D. Weights Pruning

Following the training of the proposed baseline model, systematic pruning is applied using the magnitude-based method presented in [6] and which is implemented in Tensorflow Optimization Toolkit.

The pruning algorithm [6] iteratively prunes low magnitude weights in each layer until a targeted sparsity rate is reached. In between pruning steps, the model is retrained so that the remaining connections are adapted to the loss of the pruned ones. By trial and error, we set our targeted sparsity to 80% and retrain the model for three epochs with the half of the training dataset after each pruning step (constraining again the weights to be in $[-1, 1]$). At epoch no. 30 the desired sparsity is reached and we continue to train the network in order to fine-tune its accuracy by reducing the learning rate gradually.

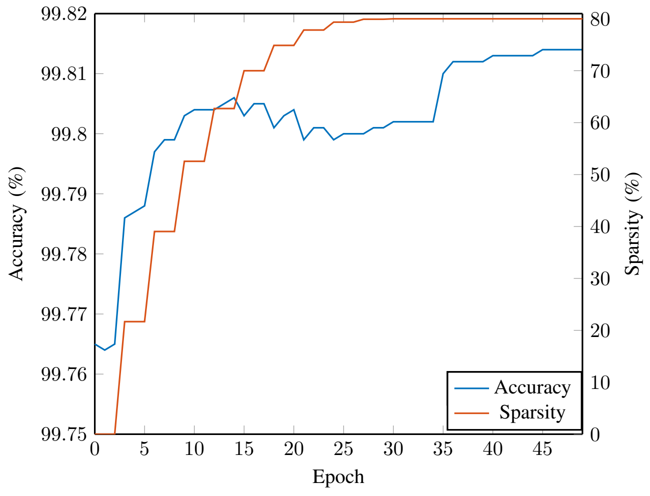

Fig. 3. Neural Network's accuracy and sparsity during the pruning process.

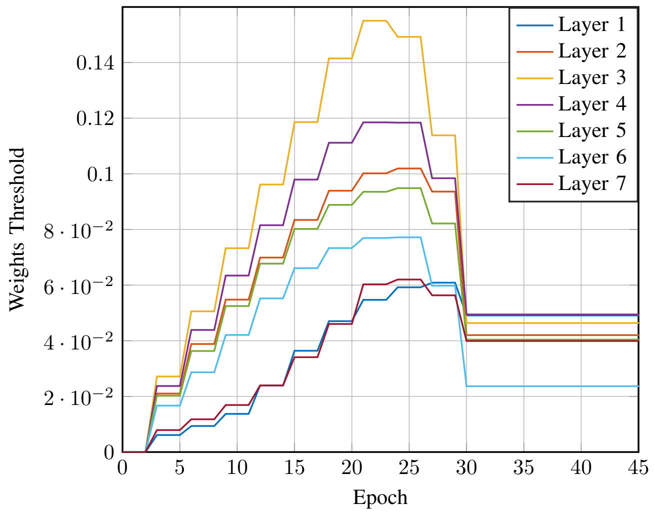

Fig. 4. Thresholds applied to the weights of each layer during pruning.

Interestingly, the proposed Neural Network Decoder can operate with just 20% of the initial weights with negligible loss in its accuracy. This translates to $98,640$ weights for the sparse model versus $493,200$ for the dense one (biases are excluded from the pruning process). Moreover, by eliminating weights with low magnitude, we further assist the procedure of quantization, as detailed in the next subsection.

In Fig. 3, the accuracy and sparsity of the neural network are illustrated, during pruning. Also, in Fig. 4, we can observe the magnitude threshold applied to the weights during the operation.

E. Quantization of the proposed network

Having compressed the proposed network in terms of the number of its non-zero parameters, we subsequently quantize them, in order to reduce the required bits and, therefore, storage requirements and computation cost. At this stage, because of the training and pruning steps, the remaining weights of the network have two properties:

1) Their dynamic range is $[-1, 1]$; and
2) Their magnitude is greater than $2 \cdot 10^{-2}$ as it can be seen in Fig. 4

Property 2 holds at least for the vast majority of the weights, since the training after pruning makes an insignificant number of weights to have magnitude less than that. That is, for an arbitary weight w it holds

$$2 \cdot 10^{-2} \leq |w| \leq 1. \tag{4}$$

Thus, the use of 1 bit for the integral part and 7 bits for the fractional part are sufficient to cover the dynamic range and precision of the weights in the particular application.

To perform the quantization, we use the DECENT tool, from the Xilinx DNNDK Framework. As the framework is designed for use with Convolutional Neural Networks and does not support pure fully-connected networks, we slightly modified the network architecture. As it is known, a fully-connected layer is essentially a matrix-vector multiplication (layer weights), a vector addition (layer biases) and the computation of the activation function. Assume an input vector of size m to a fully-connected layer with n neurons. This translates to a weight matrix $\boldsymbol{W}$ of dimensions $n \times m$, where each row consists of the weights of the respective neuron, *i.e.*,

$$\boldsymbol{W} = \begin{bmatrix} w_{1,1} & \cdots & w_{1,m} \\ \vdots & \ddots & \vdots \\ w_{n,1} & \cdots & w_{n,m} \end{bmatrix} \tag{5}$$

The bias vector is of size n, the same as the number of neurons. To obtain a computationally equivalent convolutional layer, each neuron of the fully-connected layer becomes a filter kernel $\boldsymbol{f}_i$ of the convolutional layer with dimensions $m_1 \times m_2$, where $m_1 \cdot m_2 = m$. The parameters of each filter are the reshaped version of the corresponding row of the weight matrix. That is,

$$\boldsymbol{f}_i = \begin{bmatrix} w_{i,1} & \cdots & w_{i,m_2} \\ \vdots & \ddots & \vdots \\ w_{i,(m_1-1)\cdot m_2+1} & \cdots & w_{i,m} \end{bmatrix}, \; i = 1, 2, \ldots, n \tag{6}$$

Also, the input vector is reshaped accordingly, to have the same dimensions as the filters, *i.e.*, $m_1 \times m_2$. The bias vector remains as is. Effectively, the computations of the resulting convolutional layer are exactly the same as the ones of the fully-connected layer and the output of each filter, which is a single value, coincides with the output of the respective neuron. In our case, we replaced the first layer of our network with an equivalent convolutional layer and reshaped the inputs from a vector of size 81, to a 9×9 2–D array. The particular modification is important because it enables the use of the DNNDK framework for our application.

We provide the tool with a calibration dataset consisting of 1 million samples, in order to determine the representation for the activation of each layer. The tool supports conversion

TABLE I
QUANTIZATION DETAILS OF NEURAL NETWORK DECODER

	Fixed-point Representation
Input	8.5
Layer 1	8.6
Layer 2	8.5
Layer 3	8.3
Layer 4	8.2
Layer 5	8.1
Layer 6	8.2
Layer 7	8.2
Weights	8.7

from 32-bit floating point to 8-bit fixed point values. In our context, we use the notation (word length).(fractional length) to describe a fixed point representation. As expected, the tool uses 8.7 representation for the parameters of the network. The representation for each activation is detailed in Table I.

III. Hardware Implementation

The proposed compressed Neural Network Decoder is implemented using the Xilinx DNNDK Framework [10]. Our targeted board is a Xilinx ZCU104, which accommodates a Zynq UltraScale+ XCZU7EV-2FFVC1156 MPSoC device. To our knowledge, this is the first reported implementation of a DL decoder in hardware.

A. Architectural Details

Xilinx's DNNDK Framework is an SDK targeted for deep-learning inference in the edge. The inference of the deep-learning networks is assisted by the Deep Processing Units (DPUs). The DPUs are application specific processors, which accelerate the computations involved in a deep-learning network by taking advantage of the FPGA architecture.

The Deep Neural Network Compiler (DNNC), takes the quantized model from the DECENT tool and generates DPU-specific instructions (Kernel) for the execution of the network. Although our model is pruned significantly, the release of DNNDK that was used does not exploit the sparsity of the weights in our model, hence all the computations are executed.

The Deep-Learning application is heterogeneous, with different parts running on an ARM CPU and on the DPU cores on the Programmable Logic (PL). The main program running on the CPU, manages the pre-processing of the Neural Network inputs and the post-processing of its outputs. In our system, it is possible to instantiate two DPU cores. The block diagram of the inference system is visualized in Fig. 5.

Having compiled the network to DPU-compatible instructions, we develop an application in C++, using the DNNDK API. The application simulates the transmission of random generated bits, grouped as BCH(63,45) codewords through an AWGN Channel at different E_b/N_0 levels. Then from the received sequence, the inputs to the Neural Network Decoder are calculated and sent to the DPU for inference. Then, the output vector of the Neural Network is returned to the CPU program for the decoding post-processing. By using multiple threads inside the main program, it is possible to take

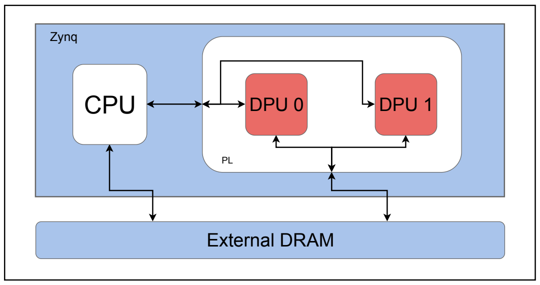

Fig. 5. Xilinx DNNDK Computation Platform, employed for the proposed DL decoder.

advantage of the two DPUs in our system to run two copies of the Neural Network in parallel, as each DPU execute a single Task at a time. A DPU Task is defined as a DPU specific thread that is designed to run a Neural Network (DPU Kernel).

IV. Results

Considering the runtime of the deployed NN Decoder, it was measured to be 500 μs. The fraction of time that is associated with the DPU is the dominant factor of this latency, hence the CPU time is negligible. This latency translates to a decoding throughput of 90 kbps. By utilizing both DPUs to run two Tasks in parallel we gain a throughput rate improvement of approximately 1.85×, reaching 165 kbps. Of course, a proportionally faster throughput would be achieved by instantiating more DPU cores. We should also note again, that the version of DNNDK used for the developed prototype does not take advantage of the high sparsity present in the NN weights. If that were the case, we would expect up to 5× faster inference.

The Bit Error Rate results of our implemented decoder are presented in Fig. 6. As we can observe, the BER performance of the DPU deployed Kernel is very close to that of the baseline non-compressed model and even surpass it slightly at high E_b/N_0 levels and converges to the order-2 OSD decoder [14].

V. Conclusion

In this work, we showcase the hardware implementation of a Syndrome-Based Neural Network Decoder, using a generic Deep-Learning hardware acceleration framework by Xilinx. A step-by-step procedure is elaborated for the conversion of a proposed Neural Network to a sparse and quantized equivalent. Furthermore, as the DNNDK framework strictly requires the use of at least one Convolutional layer, a simple method for the conversion of the first fully-connected layer to a Convolutional one is shown. This method is particularly useful as it enables, except from Convolutional Neural Networks, the use of Fully-Connected ones with that framework. It is shown that a Fully-Connected Neural Network used in the noise-estimation stage of the decoder can function without any loss in its error-correcting performance by pruning up to 80% of its weights and quantizing them from a 32-bit floating-point to an 8-bit

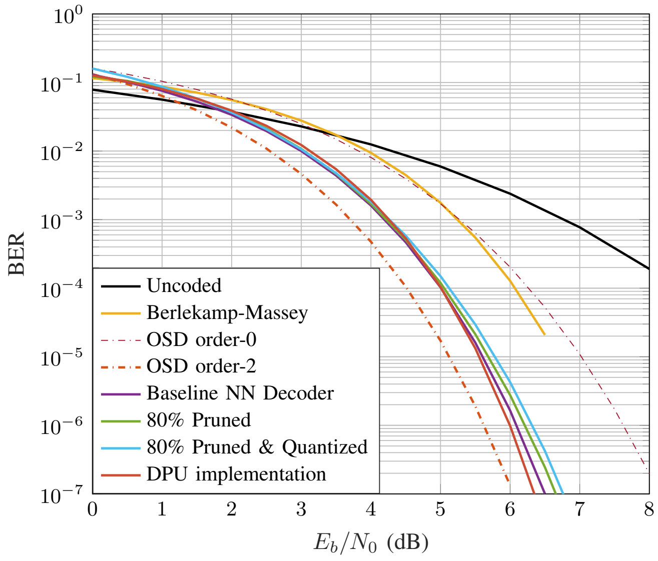

Fig. 6. BER vs. noise level comparison for various compression stages and decoding algorithms.

fixed-point representation, therefore allowing its implementation in relatively low-precision hardware accelerators.

However, the throughput of the developed prototype DL Decoder reported here is quite low, achieving much less than 1 Mbps, currently limiting its usage to relatively low-rate applications. As the majority of the Deep-Learning acceleration frameworks focus on Convolutional Neural Networks for Computer Vision applications, there is further need for dedicated hardware architectures targeting Neural Networks such as the proposed one.

Overall, the idea of the Deep-Learning decoders seems promising, since they can be implemented with low-precision as found in this paper. DL decoders can be investigated and evaluated for use with any code and with any channel noise model, towards the objective of a universal method for decoding. Additional research is needed, in order to attain DL networks which can scale to long-length codes and computationally-efficient hardware architectures for the implementation of these decoders.

References

[1] E. Nachmani, E. Marciano, L. Lugosch, W. J. Gross, D. Burshtein, and Y. Beery, "Deep learning methods for improved decoding of linear codes," *IEEE Journal of Selected Topics in Signal Processing*, vol. 12, no. 1, pp. 119–131, Feb 2018.

[2] A. Bennatan, Y. Choukroun, and P. Kisilev, "Deep learning for decoding of linear codes - a syndrome-based approach," *CoRR*, vol. abs/1802.04741, 2018.

[3] Y. LeCun, J. S. Denker, and S. A. Solla, "Optimal brain damage," in *Advances in Neural Information Processing Systems 2*, D. S. Touretzky, Ed. Morgan-Kaufmann, 1990, pp. 598–605.

[4] S. Han, H. Mao, and W. J. Dally, "Deep compression: Compressing deep neural network with pruning, trained quantization and huffman coding," *CoRR*, vol. abs/1510.00149, 2015.

[5] S. Han, J. Pool, J. Tran, and W. Dally, "Learning both weights and connections for efficient neural network," pp. 1135–1143, 2015.

[6] M. Zhu and S. Gupta, "To prune, or not to prune: Exploring the efficacy of pruning for model compression," *ArXiv*, vol. abs/1710.01878, 2018.

[7] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, and W. J. Dally, “Eie: Efficient inference engine on compressed deep neural network,” in *Proceedings of the 43rd International Symposium on Computer Architecture*, ser. ISCA ’16. Piscataway, NJ, USA: IEEE Press, 2016, pp. 243–254.

[8] S. Han, J. Kang, H. Mao, Y. Hu, X. Li, Y. Li, D. Xie, H. Luo, S. Yao, Y. Wang, H. Yang, and W. B. J. Dally, “Ese: Efficient speech recognition engine with sparse lstm on fpga,” in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA ’17. New York, NY, USA: ACM, 2017, pp. 75–84.

[9] E. Kavvousanos, V. Paliouras, and I. Kouretas, “Simplified deep-learning-based decoders for linear block codes,” in *2018 25th IEEE International Conference on Electronics, Circuits and Systems (ICECS)*, Dec 2018, pp. 769–772.

[10] “Xilinx Edge AI platform.” [Online]. Available: https://www.xilinx.com/products/design-tools/ai-inference/edge-ai-platform.html

[11] D. P. Kingma and J. Ba, “Adam: A method for stochastic optimization,” *CoRR*, vol. abs/1412.6980, 2014. [Online]. Available: http://arxiv.org/abs/1412.6980

[12] F. Chollet *et al.*, “Keras,” https://keras.io, 2015.

[13] M. Abadi *et al.*, “Tensorflow: A system for large-scale machine learning,” in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, 2016, pp. 265–283.

[14] M. P. C. Fossorier and S. Lin, “Soft-decision decoding of linear block codes based on ordered statistics,” *IEEE Transactions on Information Theory*, vol. 41, no. 5, pp. 1379–1396, Sep 1995.