Functional Error Correction for Reliable Neural Networks

Kunping Huang
CSE Department
Texas A&M University
College Station, TX 77843
kun150kun@tamu.edu

Paul H. Siegel
ECE Department
University of California, San Diego
La Jolla, CA 92093
psiegel@ucsd.edu

Anxiao (Andrew) Jiang
CSE Department
Texas A&M University
College Station, TX 77843
ajiang@cse.tamu.edu

***Abstract*—When deep neural networks (DNNs) are implemented in hardware, their weights need to be stored in memory devices. As noise accumulates in the stored weights, the DNN's performance will degrade. This paper studies how to use error correcting codes (ECCs) to protect the weights. Different from classic error correction in data storage, the optimization objective is to optimize the DNN's performance after error correction, instead of minimizing the Uncorrectable Bit Error Rate in the protected bits. That is, by seeing the DNN as a function of its input, the error correction scheme is function-oriented. A main challenge is that a DNN often has millions to hundreds of millions of weights, causing a large redundancy overhead for ECCs, and the relationship between the weights and its DNN's performance can be highly complex. To address the challenge, we propose a Selective Protection (SP) scheme, which chooses only a subset of important bits for ECC protection. To find such bits and achieve an optimized tradeoff between ECC's redundancy and DNN's performance, we present an algorithm based on deep reinforcement learning. Experimental results verify that compared to the natural baseline scheme, the proposed algorithm achieves substantially better performance for the functional error correction task.**

I. Introduction

Deep learning has become a boosting force for AI with many applications. When a neural network is implemented in hardware, its weights need to be stored in memory devices. Noise in such devices will accumulate over time, degrading the neural network's performance [1], [4], [11], [15], [33]. It is important to protect neural networks using error correction schemes. In this work, we study how to use error correcting codes (ECCs) to protect the weights of neural networks.

The protection of neural networks has a different optimization objective from classic error correction in data storage systems. In classic error correction, the objective is to minimize the Uncorrectable Bit Error Rate (UBER) in the protected bits. For neural networks, however, the objective is to optimize its performance (e.g., classification accuracy). That is, by seeing the neural network as a function of its input, the error correction scheme is function-oriented.

Several challenges exist for the protection of neural networks. First of all, a deep neural network (DNN) often has many weights. For example, DNNs in computer vision often have millions to hundreds of millions of weights [13]. This can cause a very large redundancy overhead for ECCs. So it is important to design schemes that can reduce redundancy, and achieve an optimized redundancy-performance tradeoff.

Secondly, the relationship between a neural network's weights and its performance is highly complex. The current understanding on the relationship is very limited, and is an active topic of research in many areas [10], [18]. Therefore, it is challenging to design efficient algorithms that can identify weights that are most important for preserving the performance of neural networks.

In this paper, we propose a *Selective Protection* (SP) scheme, which chooses only a subset of important bits for ECC protection. Furthermore, for different layers of edges, the sets of protected bits for their weights are different. To optimize the scheme, we present an algorithm based on deep reinforcement learning (DRL). The key of the algorithm is to learn the complex relation between which bits to protect and the network's corresponding performance. (Note that the DRL algorithm can be run before the neural network is implemented in hardware devices. So the DRL algorithm itself is noiseless because it is run in a noiseless environment such as a high-end computer; but the neural network, after implemented in hardware such as edge devices, will become increasingly noisy over time in the harsher environments.)

Our algorithm can be evaluated based on the redundancy-performance tradeoff as follows. Let k_{total} denote the total number of bits used to represent the neural network's weights. Let k_{pro} denote the number of bits we protect with ECCs. Let the ECCs be (n, k) linear codes, where n denotes the codeword length and k denotes the number of information bits. Then the number of parity-check bits is $\frac{n-k}{k} \cdot k_{pro}$. We normalize it by k_{total}, and call it *redundancy* r, namely,

$$r = \frac{k_{pro}(n-k)}{k_{total}k}. \tag{1}$$

As for the performance of the neural network, for classification tasks (which this work focuses on), it usually refers to the classification accuracy, namely, the probability that the inputs are classified correctly.

We compare the performance of our algorithm to a natural baseline scheme, where all layers of the neural network receive the same level of protection from ECCs. Experimental results verify that our proposed algorithm achieves substantially better performance. For example, when the neural network is ResNet-18 and its weights are represented by bits using the IEEE-754 standard (i.e., the single-precision floating-point

format used in most hardware systems), and when the Bit Error Rate (BER) is 1% and BCH codes are used, the baseline scheme's classification accuracy drops very quickly once its redundancy r is below the threshold 0.0453. In comparison, our algorithm can decrease the corresponding threshold to 0.0388, which represents a reduction of 14.3% in the redundancy requirement. If the ECC approaches the Shannon capacity, this reduction can be further enlarged to 25.7%.

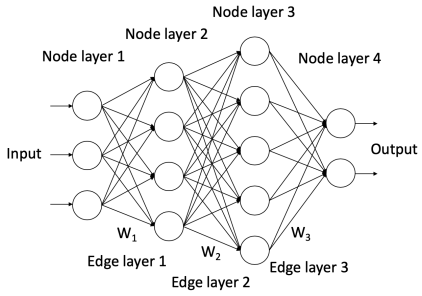

Fig. 1. A neural network with four node layers (an input layer, two hidden layers and an output layer) and three edge layers. Here W_1, W_2, W_3 are the set of weights in each edge layer.

The topic explored in this paper is related to several research areas. They include *robustness of neural networks against noise*, where researchers study the effect of noise on the performance of neural networks [2], [3], [5], [7], [8], [16], [22], [24], [25], [26], [29], [30], [32]. In [25], Qin *et al.* studied random bit errors for weights stored as bits, and developed an ECC with one parity bit to improve the network's performance and robustness. In [30], Upadhyaya *et al.* studied random noise for weights stored as analog numbers, and developed analog ECCs to correct the analog noise. Note that different from the above works, this paper proposes the Selective Protection scheme for the first time, which protects *different* sets of bits for different layers. Such a prioritized and non-uniform error correction scheme can further optimize performance. The spirit of this method has also been explored for a different topic [17], where system resources are allocated non-uniformly to noisy weak classifiers in a boosting classifier (e.g., AdaBoost) based on their varied importance.

In the area of *model compression*, a lot of works have focused on how to reduce the size of a neural network without affecting its performance [10], [12], [23], [31]. They use various techniques to either prune or quantize the weights in neural networks, including deep reinforcement learning techniques [12], [31], and the simplified networks need to be retrained. Note that in our work, we search for important bits and protect them, without the need to modify the weights or retrain the network.

In the area of *reliability of computational circuits*, researchers have studied the use of ECCs to ensure the correctness of circuits [6], [9], [28]. In comparison, our work focuses on the redundancy-performance tradeoff, where the neural network's performance does not have to be the same before and after ECC protection.

The rest of the paper is organized as follows. In Section II, we introduce the SP scheme, and present its deep reinforcement learning algorithm. In Section III, we evaluate the SP scheme by experiments, which verify that the scheme can substantially improve the redundancy-performance tradeoff for neural networks. The results also show a very interesting discovery that, depending on how weights are represented as bits, the bits that are most important to protect are not necessarily Most Significant Bits (MSBs) in the data representation. We present a detailed analysis for this interesting phenomenon. In Section IV, we present concluding remarks.

II. Selective Protection Scheme by Deep Reinforcement Learning

In this section, we present the Selective Protection (SP) scheme for functional error correction. It protects the most important bits in weights by ECC in order to achieve an optimized redundancy-performance tradeoff. We first introduce weight representation for neural networks, and define the Selective Protection scheme. We then present a deep reinforcement learning (DRL) algorithm for the SP scheme.

A. Weight Representation in Neural Networks

Neural networks have been used widely in deep learning. An example of a neural network is shown in Figure 1, which has four node layers and three edge layers between them. There are different ways to represent weights in neural networks as bits. We introduce two important weight representations (both will be used in experiments): 1) the *IEEE-754 Standard Floating-Point Representation*: IEEE-754 is an international standard very widely used in hardware. We adopt its 32-bit version. Given a weight $w \in \mathbb{R}$, let $B_w^{32} = (b_0, b_1, \cdots, b_{31})$ be its binary representation: $w = (-1)^{(b_0)_2} \times 2^{(b_1 b_2 \cdots b_8)_2 - 127} \times (1.b_9 b_{10} \cdots b_{31})_2$. Here b_0 is the *sign bit*, $b_1 b_2 \cdots b_8$ are the *exponent bits*, and $b_9 b_{10} \cdots b_{31}$ are the *fraction bits*. 2) the *Fixed-Point Representation*: in this representation, the weights in a range $[-c, c]$ are linearly quantized and represented as bits. (Such a representation has been used in neural networks before, including [31].) Consider its m-bit version. Let $s = c/(2^{m-1} - 1)$ be a scaling factor. Given a weight $w \in [-c, c]$, let $D_w^m = (b_0, b_1, \cdots, b_{m-1})$ be its binary representation: $w = (-1)^{(b_0)_2} \times (b_1 b_2 \cdots b_{m-1})_2 \times s$.

B. Selective Protection Scheme

We now present the Selective Protection (SP) scheme, which selects important bits and protects them from errors with ECCs. Consider a neural network with N edge layers. For $i = 1, 2, \cdots, N$, let L_i denote the ith edge layer, and let W_i denote the set of weights in L_i. Assume that every weight is represented by m bits. The SP scheme will select a *bit-mask vector*

$$M_i = (\mu_{i,0}, \mu_{i,1}, \cdots, \mu_{i,m-1}) \in \{0, 1\}^m$$

for each edge layer L_i. For each weight $w = (b_0, b_1, \cdots, b_{m-1}) \in W_i$, its jth bit b_j will be protected by ECC if $\mu_{i,j} = 1$. Naturally, we let $\mu_{i,j} = 1$ for the layer L_i if

its bits in the jth position are critical for the neural network's performance.

The neural network has $k_{total} = m\sum_{i=1}^{N}|W_i|$ bits in total. The number of bits protected by ECCs is $k_{pro} = \sum_{i=1}^{N}|W_i|\sum_{j=0}^{m-1}\mu_{i,j}$. When the ECCs are (n,k) linear codes, by Equation (1), the *redundancy* of the SP scheme is

$$r(M_1, M_2, \cdots, M_N) = \frac{(n-k)\sum_{i=1}^{N}|W_i|\sum_{j=0}^{m-1}\mu_{i,j}}{km\sum_{i=1}^{N}|W_i|}$$

Let $\mathcal{P}(M_1, M_2, \cdots, M_N)$ denote the performance of the neural network (e.g. classification accuracy). Let $\bar{r}$ be a target redundancy. The optimization objective of the SP scheme is to maximize $\mathcal{P}(M_1, M_2, \cdots, M_N)$ given that $r(M_1, M_2, \cdots, M_N) \leq \bar{r}$.

C. Deep Reinforcement Learning for Selective Protection

We now present a deep reinforcement learning algorithm for the SP scheme. We assume that the bits suffer from errors of a Binary Symmetric Channel (BSC) with Bit Error Rate (BER) p, and a suitable (n,k) linear ECC is used that can correct errors of BER p with a probability that approaches 1. Therefore, after error correction, only the bits not protected by ECC will have errors. Note that for a neural network, its performance is a highly complex function of its weights. The DRL algorithm will learn this complex function, and choose the important bits to protect accordingly.

In the following, we first present the essential components of the DRL algorithm: its *state space*, *action space*, *reward function*, and *policy of agents*. We then present the overall learning process of the DRL algorithm. We focus on Convolutional Neural Networks (CNNs), a very important and widely used family of DNNs. A CNN usually has two types of layers: convolutional layers and fully-connected layers. Note that a fully-connected layer can be seen as a special case of a convolutional layer, where its convolutional kernel has the same size as its input feature map. And the DRL algorithm can be easily extended if other types of layers are also considered.

1) State Space: For $i = 1, 2, \cdots, N$, let c_{in}^i (resp., c_{out}^i) be the number of input (resp., output) channels for the ith layer L_i (i.e., the number of input or output feature maps). Let s_{kernel}^i be its kernel size (i.e. the size of its filter for the convolution operation). Let s_{stride}^i be its stride for convolution. Let s_{feat}^i be the size of its input feature map (i.e., each input feature map is a two-dimentional array of size $s_{feat}^i \times s_{feat}^i$). Let $a_i \in \mathcal{A}$ be the most recent action taken by the agent for L_i, where $\mathcal{A}$ denotes the action space, whose details will be introduced later. Let $\alpha_i = (c_{in}^i, c_{out}^i, s_{kernel}^i, s_{stride}^i, s_{feat}^i, |W_i|, a_i)$ denote a state vector associated with L_i. Then, the *global state* θ is defined as $\theta = (\alpha_1, \alpha_2, \cdots, \alpha_N)$. To simplify the learning process, each layer L_i will use a *local state* π_i defined as: $\pi_i = (c_{in}^i, c_{out}^i, s_{kernel}^i, s_{stride}^i, s_{feat}^i, |W_i|, a_{i-1})$. When $i = 1$, the parameter $a_{i-1} = a_0$ can be a constant. Note that in π_i, only the action of its previous layer a_{i-1} is used, instead of the actions of all its previous layers $a_1, a_2, \cdots, a_{i-1}$.

2) Action Space: We now present the space of actions for the DRL algorithm. For $i = 1, 2, \cdots, N$, the action of the ith layer L_i is to choose a value $a_i \in \{0,1\}^m$ for its bit-mask vector $M_i = (\mu_{i,0}, \mu_{i,1}, \cdots, \mu_{i,m-1})$. The overall action is the sequence of actions $(a_1, a_2, \cdots, a_N)$. Note that in each iteration of the DRL algorithm, the actions $a_1, a_2, \cdots, a_N$ are chosen sequentially. When the layer L_i takes the action a_i, it chooses the value of a_i (i.e., sets its bit-mask vector M_i) based on its local state π_i and the reward function (to be introduced).

Let the above method be called the *BitMask* method. For comparison, we also study its simplified version: the *TopBits* method, where each layer always chooses the first few bits of its weights for ECC protection, although the number of bits chosen by different layers can still be different. Intuitively, *TopBits* is an excellent strategy because the MSBs of weights are usually considered more important than LSBs. However, our research will show the surprising result that it is not always the case.

3) Reward Function: We now present the reward function for the DRL algorithm. After each iteration of the DRL algorithm (where the N layers take their actions $(a_1, a_2, \cdots, a_N)$ and set their bit-mask vectors $(M_1, M_2, \cdots, M_N)$ accordingly), random bit errors of BER p are added to all bits in the N layers (but note that some of them are chosen to be protected by ECCs), and then the performance $\mathcal{P}$ (e.g., classification accuracy) of the neural network is measured. The reward function is a linear combination of the current performance $\mathcal{P}$ and redundancy r (or more specifically, how far r deviates from the target redundancy $\bar{r}$). Interested readers can find details of the reward function in our full paper [14].

4) Policy of Agents and the Learning Process: In the DRL algorithm, every layer L_i has an *agent* A_i that takes the action a_i based on the local state π_i and an estimated reward function $\hat{R}$. How the agent A_i chooses the action a_i based on the available information is called its *policy*. In this part, we present the policy of the N agents $A_1, A_2, \cdots, A_N$.

We build four deep neural networks: an *Actor Network*, a *Target Actor Network*, a *Critic Network*, and a *Target Critic Network*. The four networks are illustrated in Figure 2. They are all Multilayer Perceptron (MLP) neural networks of four node layers, where the two hidden layers have size 400 and 300, respectively. Additional information on their architectures is as follows:

- *Actor Network* and *Target Actor Network*: For both networks, the input is the local state π_i, and the output is the action a_i. The two networks have similar functions, but update their weights with different algorithms during training.
- *Critic Network* and *Target Critic Network*: For both networks, the input consists of the local state π_i and the action a_i, and the output is an estimated value for the summation of the current and the future rewards in the same iteration (where future rewards are discounted in certain ways). Specifically, let γ be a discount factor. Then for $t = 1, 2, \cdots, N$, the output of the two networks is the value of the following Q function: $Q(\pi_t, a_t) =$

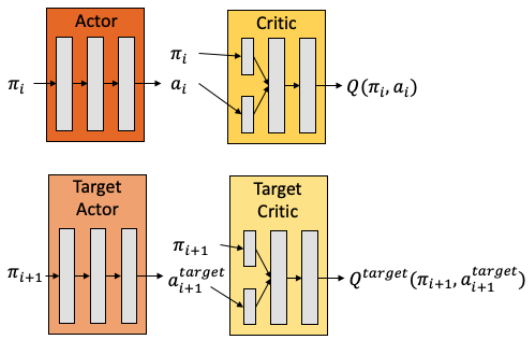

Fig. 2. The four neural networks used in the DRL algorithm: the Actor Network (top left), the Target Actor Network (bottom left), the Critic Network (top right) and the Target Critic Network (bottom right).

$\sum_{i=t}^{N} \gamma^{i-t} \hat{R}(\pi_i, a_i)$ where $\hat{R}(\pi_i, a_i)$ is an estimation of the real reward of this iteration. As before, the two networks also have similar functions, but update their weights differently during training.

The DRL algorithm keeps using the Actor Network to generate actions. In each iteration, the N agents $A_1, A_2, \cdots, A_N$ generate the actions $a_1, a_2, \cdots, a_N$ sequentially. That is, for $i = 1, 2, \cdots, N$, the Actor Network takes π_i as input, and outputs the action a_i. (Note that the Actor Network outputs real numbers, and we round them to the nearest integers to get the action a_i.) After an iteration, the N local states $(\pi_1, \pi_2, \cdots, \pi_N)$, the N actions $(a_1, a_2, \cdots, a_N)$ and the overall reward R of the iteration are stored in a buffer. The buffer has a fixed size. When new data come in, if the buffer is full, the oldest data will be removed. Therefore, the buffer always stores the most recent results.

After each iteration, a number of samples will be randomly chosen from the buffer to train the four networks. Each sample has the form of $(\pi_i, a_i, \pi_{i+1}, R)$. The four networks update their weights as follows, using the idea of the DDPG algorithm [21]:

- Step 1: *train the Critic Network*. As shown in Figure 2, the Critic Network takes π_i and a_i as input, and outputs a value $Q(\pi_i, a_i)$. We also concatenate the Target Actor Network and the Target Critic Network (as shown in Figure 2), and use π_{i+1} as input to generate the output $Q^{target}(\pi_{i+1}, a_{i+1}^{target})$. The loss function of the Critic Network is then set as $\mathcal{L}_{critic} = (Q(\pi_i, a_i) - \gamma Q^{target}(\pi_{i+1}, a_{i+1}^{target}) - (R - \mathcal{B}))^2$, where the baseline $\mathcal{B}$ is defined as an exponential moving average of all previous rewards in order to reduce the variance of gradient estimation. A small number of samples are used as a mini-batch, and their total loss is used to update the weights of the Critic Network via backpropagation.
- Step 2: *train the Actor Network*. We concatenate the Actor Network and the Critic Network (as shown in Figure 2), and use π_i to generate the output $Q(\pi_i, a_i)$. The loss function is then set as $\mathcal{L}_{actor} = -Q(\pi_i, a_i)$. Then the total loss of a mini-batch of such samples is used to update the weights of the Actor Network via backpropagation (with the weights of the Critic Network frozen).
- Step 3: *train the Target Actor Network*. Let δ be a small number, such as $\delta = 0.01$. Let w_{actor}^{target} be a weight of the current Target Actor Network, and let w_{actor} be the corresponding weight of the updated Actor Network. We update w_{actor}^{target} as: $w_{actor}^{target} \leftarrow w_{actor}^{target} + \delta(w_{actor} - w_{actor}^{target})$. We update all weights of the Target Actor Network in the same way.
- Step 4: *train the Target Critic Network*. We update its weights in the same way as we did with the Target Actor Network, except that here we consider the Target Critic Network and the Critic Network.

In summary, the Critic Network learns to predict the future rewards given the current state and the action to be taken. The Actor Network learns to take the best action based on the future rewards predicted by the Critic Network. The Target Critic Network (respectively, the Target Actor Network) follows the learning of the Critic Network (respectively, the Actor Network), except that it updates its weights at a slower pace, which is a conservative method that helps the DRL algorithm converge. The DRL algorithm ends when the four networks' performance converges or when a preset number of training steps is reached.

III. Experimental Evaluation and Analysis

In this section, we present experimental evaluation of the Selected Protection scheme. We focus on two important deep neural networks in computer vision: ResNet-18 [13] and VGG16 [27]. (Both are widely used CNNs. ResNet-18 has 26 edge layers and 11.69 million weights. VGG16 has 16 edge layers and 138 million weights.) We consider two well-known datasets for image classification: the CIFAR-10 dataset [19] and the MNIST dataset [20]. We use two important data representation schemes for the weights: the IEEE-754 floating-point representation, and the fixed-point representation. We explore two types of error correcting codes: an ideal ECC that reaches the Shannon capacity, and a practical finite-length BCH code. (When the BSC has BER p, the ideal ECC have a code rate of $1 - H(p)$, matching the channel's capacity. We use the code to protect all the selected important bits, and assume that decoding always succeeds. The practical finite-length BCH code is chosen as a (8191, 6722) BCH code when the IEEE-754 floating-point representation is used, which can correct 115 errors, and as a (8191, 6787) BCH code when the fixed-point representation is used, which can correct 110 errors. When $p = 0.01$, a practical BER for storage systems, both codes can decode with sufficiently small failure probabilities, thus causing minimal degradation for the neural network's performance.) We study the performance of two methods for the SP scheme – the *BitMask* method and the *TopBits* method – and compare them to a *baseline* method, which simplifies *TopBits* by protecting the same number of bits for all layers. Given a solution of the SP scheme, we generate random errors 100 times for all the weights, and evaluate

the neural network's average performance (i.e. classification accuracy). The performance was found to be stable over different experiments.

The experimental results for the *redundancy-performance tradeoff* are shown in Figure 3 and Figure 4. Both are for BER $p = 0.01$. The figures show that once the redundancy drops below a certain threshold, the performance drops sharply. It can be seen clearly that, overall, both the *BitMask* method and the *TopBits* method *significantly* outperform the *baseline* method. It can also be seen that when the IEEE-754 representation is used, the *BitMask* method outperforms the *TopBits* method substantially overall. (When the fixed-point representation is used, the two become more comparable, with *TopBits* sometimes outperforming.) It is a very interesting observation because the *TopBits* method always chooses the first few bits of each weight, which are usually considered more significant than the remaining bits. It implies that the *BitMask* method can find less significant bits (LSBs) that are more important than MSBs for a DNN's overall performance. (Some typical examples of the bits selected by *BitMask* are illustrated in Fig. 5.) In the following, we analyze this surprising result.

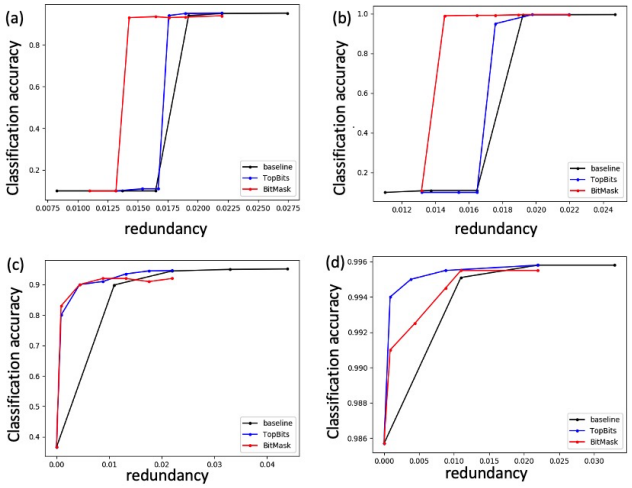

Fig. 3. The redundancy-performance tradeoff for the SP scheme when ideal ECC is used. Here "baseline", "*TopBits*" and "*BitMask*" denote the baseline algorithm (where all layers protect the same set of bits), the *TopBits* method and the *BitMask* method, respectively. (a) The neural network is ResNet-18, the dataset is CIFAR-10, and the data representation scheme is IEEE-754. (b) The neural network is VGG16, the dataset is MNIST, and the data representation scheme is IEEE-754. (c) The neural network is ResNet-18, the dataset is CIFAR-10, and the data representation scheme is fixed-point. (d) The neural network is VGG16, the dataset is MNIST, and the data representation scheme is fixed-point.

We discover that two major factors determine the importance of bits. First, 0-to-1 errors and 1-to-0 errors have an *asymmetric impact* on the DNN's performance. For example, consider a 0-to-1 error that changes an exponent bit b_i from 0 to 1 in the IEEE-754 representation. It will change the weight w to $w_{0-to-1} = 2^{2^{8-i}} \times w$. Instead, with a 1-to-0 error that changes b_i from 1 to 0, the weight w will change to $w_{1-to-0} = 2^{-2^{8-i}} \times w$. Since each neuron takes a linear combination of its incoming values before passing it to an activation function, the absolute value of the weight plays an important role in the function of the neuron. It is easy to see that 0-to-1 errors can change weights much more significantly, leading to a higher impact on DNN's performance.

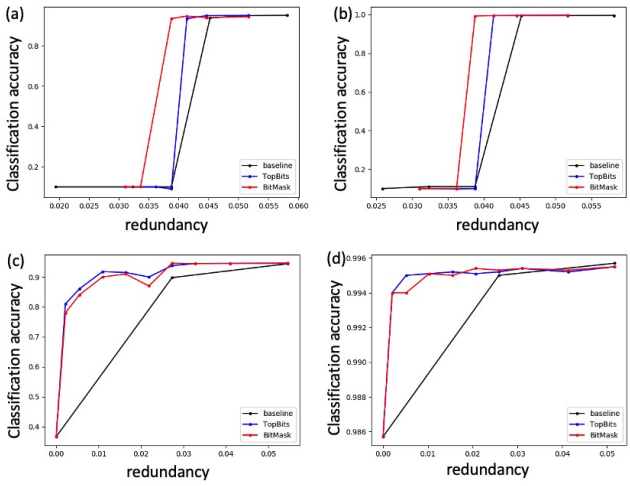

Fig. 4. The redundancy-performance tradeoff for the SP scheme when BCH codes are used. (a) The neural network is ResNet-18, the dataset is CIFAR-10, and the data representation scheme is IEEE-754. (b) The neural network is VGG16, the dataset is MNIST, and the data representation scheme is IEEE-754. (c) The neural network is ResNet-18, the dataset is CIFAR-10, and the data representation scheme is fixed-point. (d) The neural network is VGG16, the dataset is MNIST, and the data representation scheme is fixed-point.

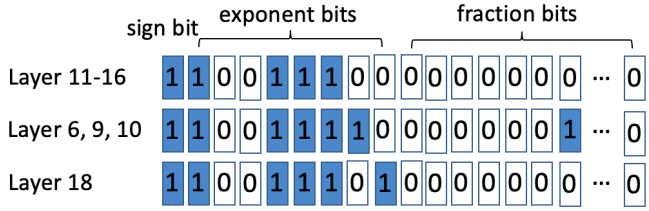

Fig. 5. Typical examples of the bit-mask vector in some edge layers, with the IEEE-754 floating-point representation and the *BitMask* method. Here the neural network is ResNet-18, the dataset is CIFAR-10 and the ECC is the ideal ECC. The positions of the selected bits for ECC protection correspond to the 1's in the bit-mask vector (of the blue color). Notice that among the exponent bits, some less significant bits are selected instead of more significant bits.

The second factor is that for large subsets of the weights, their bits in the same bit position of the IEEE-754 representation can have a highly imbalanced probability distribution. The bits in some positions are much more likely to be 1s, while in some other positions, the bits are more likely 0s. The overall importance of bits is affected by both factors substantially. For a more detailed analysis, please refer to our full paper [14].

IV. Conclusions

The redundancy-performance tradeoff for noisy neural networks can be improved significantly by protecting bits non-uniformly with ECCs. Two methods based on DRL, *BitMask* and *TopBits*, are presented to optimize such a tradeoff.

ACKNOWLEDGMENT: This work was supported in part by NSF Grant CCF-1718886.

REFERENCES

[1] Y. Cassuto, S. Kvatinsky and E. Yaakobi, "Write Sneak-Path Constraints Avoiding Disturbs in Memristor Crossbar Arrays," in *Proc. IEEE International Symposium on Information Theory (ISIT)*, 2016.

[2] S. Cavalieri and O. Mirabella, "A Novel Learning Algorithm which Improves the Partial Fault Tolerance of Multilayer Neural Networks," in *IEEE Transactions on Neural Networks*, vol. 12, no. 1, pp. 91-106, 1999.

[3] P. Chandra and Y. Singh, "Fault Tolerance of Feedforward Artificial Neural Networks – A Framework of Study," in *Proceedings of the IEEE International Joint Conference on Neural Networks*, vol. 1, pp. 489-494, 2003.

[4] Y. M. Chee, H. M. Kiah, A. Vardy, V. K. Vu and E. Yaakobi, "Coding for Racetrack Memories," in *IEEE Transactions on Information Theory*, vol. 64, no. 11, pp. 7094-7112, 2018.

[5] C. T. Chiu, K. Mehrotra, K. M.Chilukuri and S. Rankat, "Training Techniques to Obtain Fault-tolerant Neural Networks," in *IEEE International Symposium on Fault-Tolerant Computing*, pp. 360-369, 1994.

[6] V. Choudhary, E. Ledezma, R. Ayyanar and R. M. Button, "Fault Tolerant Circuit Topology and Control Method for Input-series and Output-parallel Modular DC-DC Converters," in *IEEE Transactions on Power Electronics*, vol. 23, no. 1, pp. 402-411, 2008.

[7] F. M. Dias and A. Antunes, "Fault Tolerance Improvement through Architecture Change in Artificial Neural Networks," in *Proc. International Symposium on Intelligence Computation and Applications*, pp. 248-257, 2008.

[8] M. El-Mhamdi, R. Guerraoui and S. Rouault, "On the Robustness of a Neural Network," in *Proc. IEEE 36th Symposium on Reliable Distributed Systems(SRDS)*, pp. 84-93, 2017.

[9] A. Gal and M. Szegedy, "Fault Tolerant Circuits and Probabilistically Checkable Proofs," in *Proc. 10th Annual IEEE Conference on Structure in Complexity Theory*, pp. 65-73, 1995.

[10] S. Han, H. Mao and W. Dally, "Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding," arXiv preprint arXiv:1510.00149, 2015.

[11] A. Hareedy and R. Calderbank, "Asymmetric LOCO Codes: Constrained Codes for Flash Memories," in *Proc. Annual Allerton Conference on Communication, Control and Computing (Allerton)*, pp. 124-131, Monticello, IL, 2019.

[12] Y. He and S. Han, "ADC: Automated Deep Compression and Acceleration with Reinforcement Learning," CoRR, abs/1802.03494, 2018.

[13] K. He, X. Zhang, S. Ren and J. Sun, "Deep Residual Learning for Image Recognition," CoRR, abs/1512.03385, 2015.

[14] K. Huang, P. H. Siegel and A. Jiang, "Functional Error Correction for Reliable Neural Networks," available online at $http : //faculty.cse.tamu.edu/ajiang/Publications/FunctionEC.pdf$.

[15] K. A. S. Immink, K. Cai and Jos H. Weber, "Dynamic Threshold Detection Based on Pearson Distance Detection," in *IEEE Transactions on Communications*, vol. 66, no. 7, pp. 2958-2965, 2018.

[16] H. Ito and T. Yagi, "Fault Tolerant Design using Error Correcting Code for Multilayer Neural Networks," in *Proc. IEEE International Workshop on Defect and Fault Tolerance in VLSI Systems*, pp. 177-184, 1994.

[17] Y. Kim, Y. Cassuto and L. R. Varshney, "Boosting Classifiers with Noisy Inference," available at $https : //arxiv.org/pdf/1909.04766.pdf$, 2019.

[18] J. Kirkpatrick, R. Pascanu, N. C. Rabinowitz, J. Veness, G. Desjardins, A. A. Rusu, K. Milan, J. Quan, T. Ramalho, A. Grabska-Barwinska, D. Hassabis, C. Clopath, D. Kumaran and R. Hadsell, "Overcoming Catastrophic Forgetting in Neural Networks," CoRR, abs/1612.00796, 2016.

[19] A. Krizhevsky, G. Hinton, *et al.*, "Learning Multiple Layers of Features from Tiny Images," Technical report, available online at https://www.cs.toronto.edu/kriz/learning-features-2009-TR.pdf, 2009.

[20] Y. Lecun, L. Bottou, Y. Bengio and P. Haffner, "Gradient-based Learning Applied to Document Recognition," in *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278-2324, Nov. 1998.

[21] T. Lillicrap, J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver and D. Wierstra, "Continuous Control with Deep Reinforcement Learning," arXiv preprint at arXiv:1509.02971, 2015.

[22] Y. Liu, L. Wei, B. Luo and Q. Xu, "Fault Injection Attack on Deep Neural Network," in *Proc. IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pp. 131-138, Nov. 2017.

[23] J. Luo, J. Wu, and W. Lin, "Thinet: A Filter Level Pruning Method for Deep Neural Network Compression," in *Proc. IEEE International Conference on Computer Vision*, pp. 5058-5066, 2017.

[24] V. Piuri, "Analysis of Fault Tolerance in Artificial Neural Networks," in *Journal of Parallel and Distributed Computing*, vol. 61, no. 1, pp. 18-48, 2001.

[25] M. Qin, C. Sun and D. Vucinic, "Robustness of Neural Networks Against Storage Media Errors," CoRR, abs/1709.06173, 2017.

[26] A. S. Rakin, Z. He and D. Fan, "Bit-Flip Attack: Crushing Neural Network with Progressive Bit Search," arXiv e-prints at arXiv:1903.12269, March 2019.

[27] K. Simonyan and A. Zisserman, "Very Deep Convolutional Networks for Large-scale Image Recognition," arXiv preprint at arXiv:1409.1556, 2014.

[28] C. E. Stroud, "Reliability of Majority Voting Based VLSI Fault-tolerant Circuits," in *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 2, no. 4, pp. 516-521, Dec. 1994.

[29] C. Torres-Huitzil and B. Girau, "Fault and Error Tolerance in Neural Networks: A Review," in *IEEE Access*, vol. 5, pp. 17322-17341, 2017.

[30] P. Upadhyaya, X. Yu, J. Mink, J. Cordero, P. Parmar and A. Jiang, "Error Correction for Noisy Neural Networks," Information Theory and Its Applications (ITA) Workshop, Feb. 2019.

[31] K. Wang, Z. Liu, Y. Lin, J. Lin and S. Han, "HAQ: Hardware-aware Automated Quantization," CoRR, abs/1811.08886, 2018.

[32] N. Wei, S. Yang and S. Tong, "A Modified Learning Algorithm for Improving the Fault Tolerance of BP Networks," in *Proc. International Conference on Neural Networks (ICNN)*, vol. 1, pp. 247-252, 1996.

[33] Y. Yehezkeally and M. Schwartz, "Limited-magnitude Error-correcting Gray Codes for Rank Modulation," in *IEEE Transactions on Information Theory*, vol. 63, no. 9, pp. 5774-5792, 2017.