Kufa Journal of Engineering
Vol. 17, No. 3, July 2026, P.P. 350-364
Article history: Received 20 June 2025, last revised 19 September 2025,
accepted 19 September 2025

TERNARY NEURAL NETWORK WITH SPARSE WEIGHT OPTIMIZATION FOR ENERGY-EFFICIENT MEMRISTOR CROSSBAR DEPLOYMENT WITH DEFECT TOLERANCE

Tien Van Nguyen

Institute of Engineering - Technology, Thu Dau Mot University, Ho Chi Minh City, Viet Nam, Email: tiennv01@tdmu.edu.vn.

https://doi.org/10.30572/2018/KJE/170320

ABSTRACT

Ternary neural network with three values of the weight (-1, 0, +1) is an effective solution for artificial intelligence applications in edge computing due to their energy efficiency, which is achieved through their deployment on memristor crossbars. However, the implementation on a memristor crossbar often has problems related to programming energy and defects. Therefore, this paper proposes a training method that uses both weight pruning and defect-aware training to optimize the number of programming pulses while maintaining a high recognition rate for the network. By using this method, at a 20% defect rate and 80% sparsity, the network achieves a 95% recognition rate on MNIST and 90% on CIFAR-10, compared to only 73.7% and 60.4% without using defect-aware training. At the same sparsity, the number of programming pulses, which is indicative of programming energy, is reduced by 49.1% compared to 20% sparsity.

KEYWORDS

Defects, Edge Devices, Memristor Crossbar, Neural Network, Sparsity.

1. INTRODUCTION

Nowadays, neural networks are used in many fields, particularly excelling in natural language processing, image recognition, and object classification (Aidi et al., 2025), (Alaa et al., 2024), (Mohammed et al., 2022), (Pouyanfar et al., 2018), (Shrestha and Mahmood, 2019). Their widespread use in various applications demonstrates the need for more powerful and efficient computing systems. Standard computing systems based on CMOS technology used to be able to meet these needs (Bohr and Young, 2017). However, CMOS scaling, the process of shrinking the dimensions of transistors to increase performance and reduce costs, is becoming increasingly difficult due to several physical and engineering limitations. Moreover, the drawbacks in the Von Neumann architecture, especially the memory bottleneck that happens when data is constantly moving between memory and processing units (Linn et al., 2012), (Wright et al., 2013), (Indiveri and Liu, 2015), (Sebastian et al., 2019). Due to the two blocks being separated from each other, the frequent and mass memory access can increase power consumption dramatically (Hu et al., 2018). To solve these problems, memristors become a promising solution for realizing the neural networks on hardware (Jo et al., 2010). Memristors were first introduced in 1971 by Leon Chua and demonstrated how they work in 2008 (Strukov et al., 2008). Memristors are non-volatile devices and can change their resistance levels. Therefore, memristors can store the differential states (Truong et al., 2016), (Song et al., 2017), (Pham et al., 2019). Memristor crossbars also can be made in a 3D layer that looks like the way neurons are arranged in the brain (Adam et al., 2017), (Chakrabarti et al., 2017), (Wang et al., 2020), (Lin et al., 2020). Besides, the integration of CMOS devices on the same wafer with back-end-of-line (BEOL) processes enables the development of compact and expandable designs (Sheng et al., 2019), (Graves et al., 2020). Because of these characteristics, memristor crossbars are highly effective at speeding up neural network calculations. Specifically, Ternary neural network (TNN), with their simplified weight structure of -1, 0, and +1. Compared to Binary neural network (BNN), TNN adds a zero-weight state, allowing the network to effectively pruning unimportant weights, a key factor in reducing programming energy and improving fault tolerance in memristor crossbars (Pham et al. 2018). Meanwhile, quantized neural network (QNN) requires high-precision programming, which is difficult under hardware fault conditions (Song et al. 2017). In contrast, TNN only require two resistance levels high resistance state (HRS) or low resistance state (LRS), thereby saving significant energy. With their balanced combination of high accuracy, hardware simplicity, and resilience to defects, TNNs is an efficient solution for low-power computing systems (Alemdar et al., 2017).

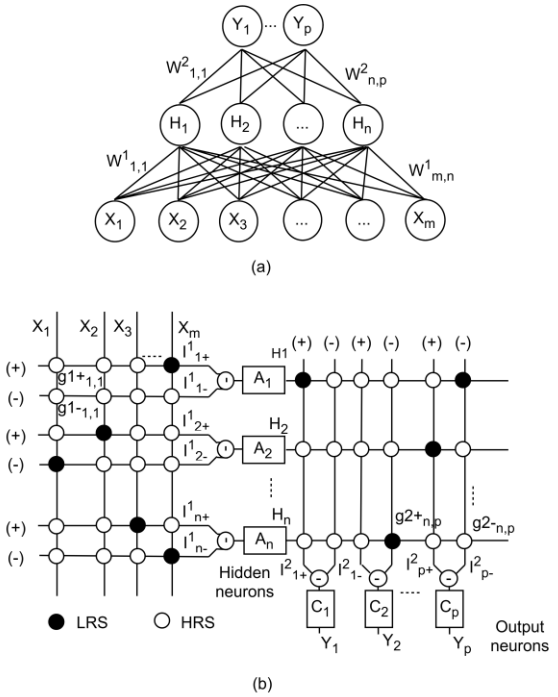

Fig. 1. (a) The ternary neural network architecture, (b) the realization of the neural network using the memristor crossbar.

As shown in Fig. 1(a), the TNN structure includes input neurons, hidden neurons, and output neurons, all of which are illustrated as circles. Here, the input neurons are denoted as X_1, X_2,..., X_m, the hidden neurons (H_1, H_2, ..., H_n), and the output neurons (Y_1, Y_2, ..., Y_p). Where m, n and p are the number of input neurons, hidden neurons, and output neurons, respectively. $W^1_{1,1}$ is the weight that connects input neuron X_1 to hidden neuron H_1 in the first layer. $W^1_{m,n}$ connects X_m to H_n. Similarly, $W^2_{1,1}$ connects hidden neuron H_1 to output neuron Y_1 in the second layer, and $W^2_{n,p}$ connects H_n to Y_p. Here, the input neurons are multiplied with the weights in the first layer and then use an activation function to calculate the values of hidden neurons. Next, the values of the output neurons are calculated by multiplying the hidden neurons and the weights in the second layer. Finally, compare the values of all the neurons and select the one with the highest value. Fig. 1(b) shows the implementation of TNN using the memristor crossbar. In the figure, the input neurons X_1, X_2,..., X_m are voltage inputs that range from 0 to 1V. The weights of the TNN are shown by the memristor cells in the crossbar. Each cell can be in a high resistance state (HRS) or a low resistance state (LRS), which are shown by open and solid circles, respectively. The memristor crossbar is divided into positive and negative crossbars to show the three weight values: -1, 0, and +1. A weight of -1 is represented by HRS-LRS pair, 0 means HRS-HRS pair, and +1 means LRS-HRS pair. The currents of the first column of the positive and negative memristor crossbar are denoted as $I^1_{1+}$and I^1_{1-}, respectively. Then the current of $I^1_{1+} - I^1_{1-}$ is transferred to the activation circuits A_1 to obtain the output of the hidden

neuron H_1. The outputs of H_2 to H_n are obtained from activation circuits A_2 to A_n. After that, these outputs are fed into the second memristor crossbar, which shows the weights of the second layer. A current-to-voltage circuit C_1 is responsible for converting the currents $I_{1+}^2 - I_{1-}^2$ into voltage signals, which provide the value of output neuron Y_1. This process is repeated for other output neurons.

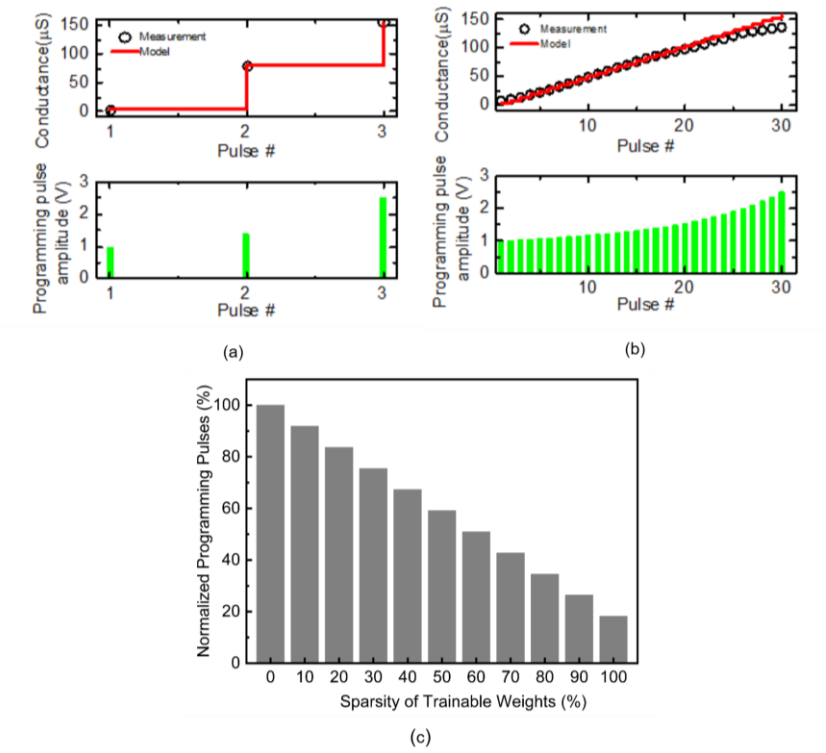

Fig. 2. (a) Coarse programming scheme, (b) fine programming scheme, (c) the simulation of normalized programming pulses for varying the sparsity of trainable weights.

In memristor crossbars, programming pulses are the electrical signals applied to modify the resistance state of memristor cells, such as switching from HRS to LRS or vice versa. The total number of these pulses directly influences the energy required to program the weights. As previous work, two programming schemes are proposed to program the value of the memristor (HRS or LRS) (Pham et al., 2019). In which, Fig. 2(a) shows a coarse programming scheme, which is just needs 3 pulses. This scheme is less accurate but uses less energy. While Fig. 2(b) shows a fine programming scheme, which uses 30 pulses to get better accuracy but uses more energy. For balancing energy efficiency and accuracy of the memristor value, coarse programming is applied for HRS state and fine programming is used for LRS state. This is because LRS state has a bigger effect on current flow in the crossbar. So, if the number of HRS cell in the crossbar is increased, the number of programming pulses for the crossbar is decreased. As shown in Fig. 2(c), the relationship between the sparsity of trainable weights and normalized programming pulses is evaluated. At 0% sparsity, it means that all weights are non-

zero (-1 or +1), yielding an HRS:LRS ratio of 50:50 due to the representation of -1 (HRS-LRS) and +1 (LRS-HRS). At this point, the highest normalized programming pulse is 100%. When the sparsity is 20%, it means that 20% of the weights are 0 (HRS-HRS), which gives an HRS:LRS ratio of 60:40, the normalized programming pulses reduce to 83.6%. Similarly, at 80% sparsity, HRS:LRS ratio is 90:10, the normalized programming pulses is 34.5%. Compared to the 20% sparsity, normalized programming pulses of 80% sparsity is lower than 49.1%. When the sparsity is 100%, it means that all weights are 0, the normalized programming pulses is 18.18%. However, in this case, the network is unable to train. From this, increasing sparsity of trainable weights in the network, the number of programming pulses is decreased, but need to consider the performance of the network.

One more problem in memristor crossbars is the defects which are caused by fabrication processes. These defects, commonly known as stuck-at-faults, occur when cells become permanently stuck in either the High Resistance State (HRS) or Low Resistance State (LRS), even when programming pulses are applied. For example, the percentages of defective cells are 10% and 4% for LRS and HRS, respectively. Despite their energy and performance advantages, memristor crossbars remain vulnerable to these fabrication-induced reliability issues (Yeo et al., 2019). Due to defects, the performance of the network can degrade significantly. To address the impact of defective cells, a defect-aware training approach is employed. Unlike redundancy or error correction code (ECC) techniques that rely on additional hardware or coding overhead, defect-aware training uses a defect map to retrain the network without modifying the hardware. In addition, the proposed method combines the defect-aware training with pruning technique to remove unimportant weights by setting them to zero. This increases the number of HRS cells, which reduces the number of required programming pulses and thus lowers the programming energy. Together, these techniques provide a scalable and energy-efficient solution for deploying networks on memristor crossbars.

The remainder of the paper is structured as follows: in section 2, the materials and methods are explained in more detail. The result is tested and discussed for MNIST dataset (Modified National Institute of Standards and Technology database) and CIFAR-10 dataset (Canadian Institute For Advanced Research), shown in section 3 (Deng et al., 2012), (Krizhevsky et al., 2025). Finally, section 4 is the conclusions.

2. METHODOLOGY

2.1. Materials

Fig. 3 shows the current–voltage behavior of the fabricated memristor and its Verilog-A model, measured using a Keithley-4200 system (Truong et al., 2016). In Fig. 3, the HRS/LRS ratio is

100, with the black line representing the Verilog-A behavioral model and the red line showing the measured data. This model was used for circuit simulations of the memristor-CMOS hybrid circuit in this paper. The subfigure displays a cross-sectional view of the memristor, which has a film structure consisting of a Pt/laalo3/Nb-doped srtio3 stacked layer and a microscope image of the memristor crossbar size 10x10 cells measurement (Jang et al., 2018).

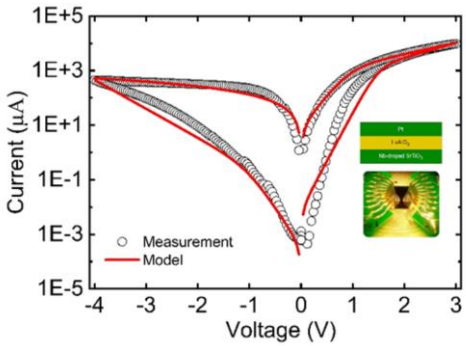

Fig. 3. The current-voltage behavior of the fabricated memristor and its Verilog-A model.

2.2. Pruning Weight and Defect aware training

Fig. 4(a) shows a 5×5 crossbar array, which is a part of the Ternary neural network. The weights (W) can be 0, -1, or +1. Here, W = 0 is denoted as an open circle, W = +1 as a solid square, and W = -1 as a solid triangle. Some of the weights in the TNN have defects, which means they are stuck at some fixed value (0, -1, or +1) and cannot be changed during programming. Defects are shown with red stars on top of the corresponding shapes.

Fig. 4(b) shows a defect map based on Fig. 4(a), indicating the location and value of defective weights. Squares with red borders show where the weights are defective, which shows what kind of defect (stuck at 0, -1, or +1). Open squares without values indicate non-defective or trainable weights, which can be changed during programming.

Fig. 4(c) shows a 5×5 crossbar array that was mapped from Fig. 4(b). The values of the defective weights are kept, and they are shown by squares with red borders. In Fig. 4(c), the trainable weights have the least impact on error of the network are chosen for pruning and assigned a value of 0, as depicted by the open squares with the value 0 inside. The remaining weights are trainable weights that are employed to recover the performance of network by alleviating the effects of pruned and defective weights.

A flowchart describes how to train a TNN using the pruning and defect-aware training method as shown in Fig. 5. The first step is to train the TNN without the information of the defective weights. The defect map is then determined by direct measurements on the memristor crossbars. In the defect map, defect positions and defect types, which can be either 0, -1, or +1. The values of defective weights are fixed. Therefore, they cannot be changed during training.

Next, for the non-defective weights, they are ranked based on their importance, which is evaluated by using the gradient squared sum. This metric shows how much each weight contributes to the loss function over time by accumulating the square of gradients across training steps. Compared to magnitude-based pruning, which removes weights with small absolute values, the gradient-based method provides a more informed decision. Based on this, the weights with the lowest importance scores are gradually set to zero until the total sparsity reaches the desired target. Then, the accuracy and sparsity are checked. If they meet the specific target, finish the process. If they are unsatisfied, the training continues by using the defect map to change the weights that are non-defective.

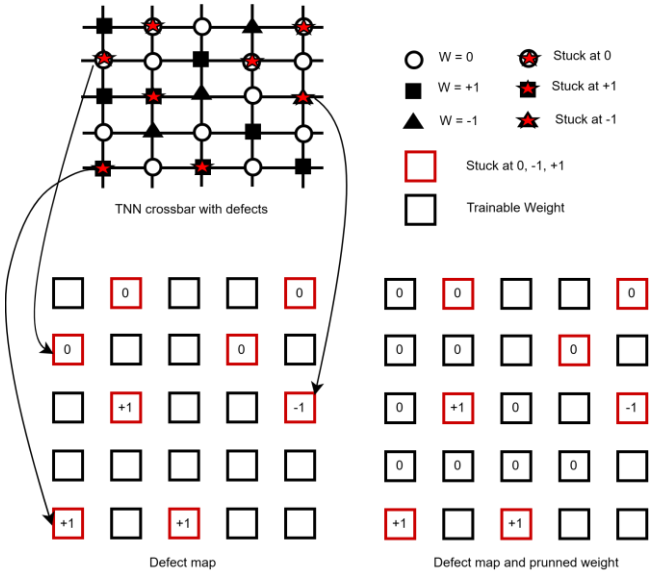

Fig. 4. (a) A TNN crossbar with defects, (b) defect map includes locations and type of defects information, (c) pruned weights and defect map method.

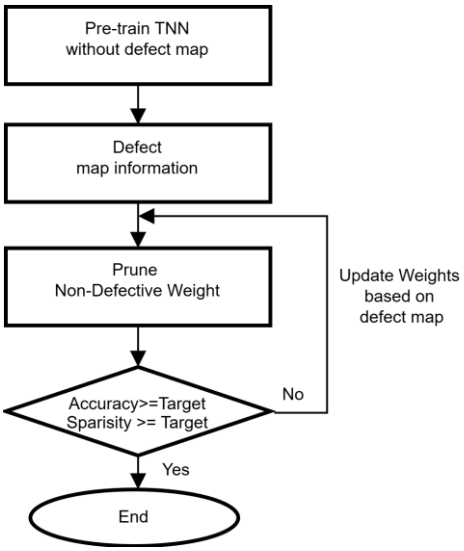

Fig. 5. A flowchart of the training process using the pruning and defect-aware training method.

2.3. Memristor crossbar circuit for implementing the ternary neural network

Fig. 6(a) shows a memristor crossbar composed of CMOS and memristors. Here, X_1, X_2, etc., represent input voltages. Two columns are used in the crossbar to calculate both positive and negative weights, which are denoted as (+) and (−) symbol. M^1_{1+} And M^1_{1-}Are memristors for positive and negative columns, respectively. Here, ternary weights are used in Fig. 6(a). For the weight = 0, both memristors on positive and negative columns are HRS. For the weight = 1, the positive and negative cells are LRS and HRS, respectively. If the positive and negative columns have HRS and LRS, the weight can be −1, respectively. Here, the ratio of HRS/LRS = 100, with HRS = 100 kΩ and LRS = 1 kΩ. Here, the low ratio such as HRS/LRS = 10 reduces to the performance of the network (Pham et al., 2019). I^1_{1+} And I^1_{1-} Represent positive and negative column currents. The current $I^1_{1+} - I^1_{1-}$ Is then transferred to the activation function circuit A_1, as shown in Fig. 6(b). The outputs of the activation function circuits are H_1, H_2 Etc., which represent output voltages of hidden neurons. The column control block selects the column containing the memristor to be programmed. In the Fig. 6(a), parasitic resistances such as RS, RN, and RW are considered. Here, RS, RN, and RW represent source resistance, neuron resistance, and line resistance, respectively. The parasitic resistances can affect to the performance of the network. To overcome it, the correction circuit can be included in the memristor crossbar for compensating the nonideal effects (Nguyen et al., 2021). Fig. 6(b) shows the relu activation function circuit. In the activation circuit, OP_1 Acts as a current-to-voltage converter, OP_2 Is an inverting amplifier. Here, V_{Cc} = 1V, R_1 = 1kΩ and R_2 = R_3 = 100 kΩ. Fig. 6(c) shows the current-voltage characteristic of the relu activation function circuit. The minimum voltage is 0V, and the maximum voltage is 1V.

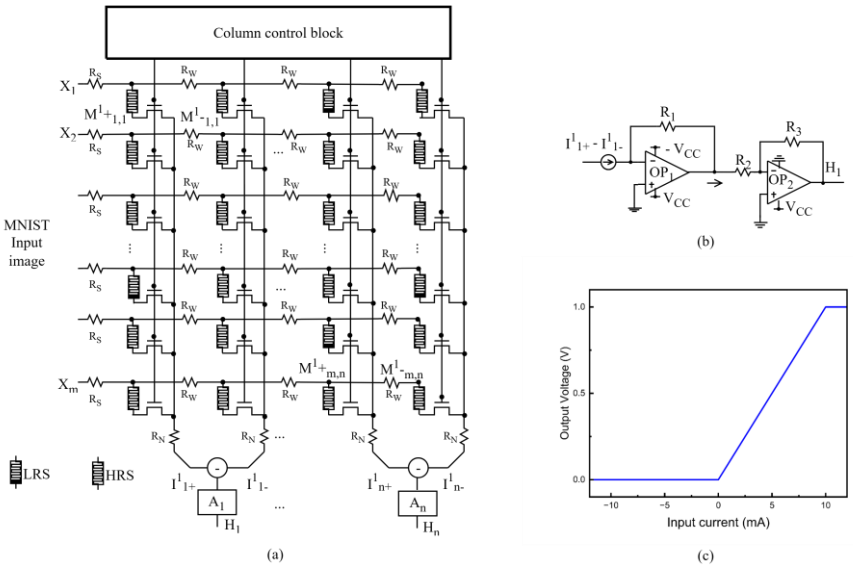

Fig. 6. (a) The schematic of memristor crossbar, (b) the ReLU activation function circuit, (c) the input current – output voltage curve of the ReLU activation function circuit.

3. RESULTS AND DISCUSSION

For recognizing the MNIST dataset, this assumes that the neural network has 784 input neurons, 100 hidden neurons, and 10 output neurons. The 784 × 100 and 100 × 10 crossbars are used to set up the network. Fig. 7(a) shows the relationship between the recognition rate and the sparsity of the trainable weights. Here, assume that there are no defective weights in the network, so 100% of the weights is the trainable weights. In addition, the parasitic resistances R_S = 1KΩ, R_N = 2KΩ and R_W = 1Ω are added to verify the network. As shown in Fig. 7(a), the recognition rate is high and stable at approximately 95% ± 0.5 as sparsity increases from 0% to 80%. After this point, especially at 90% and 100% sparsity, the recognition rate drops significantly. This shows the trade-off between the recognition rate and the sparsity, where too much pruning makes the performance become worse. Fig. 7(b) shows the ratio of recognition rate to normalized programming pulses with varying the trainable weight sparsity. The ratio shows the hardware efficiency. The efficiency of this ratio steadily improves as the sparsity increases from 0% to approximately 80%. This shows that the model can stay very accurate while the number of programming pulses is reduced. At 90% sparsity, the ratio keeps high. However, the recognition rate at this sparsity is very low. When sparsity reaches 100%, the ratio drops sharply because 100% of weights are set to 0. The network is no longer useful, even though the number of pulses is low at this point. The peak of the ratio is around 80% sparsity, suggests that this is the best sparsity range for balancing recognition rate and hardware savings.

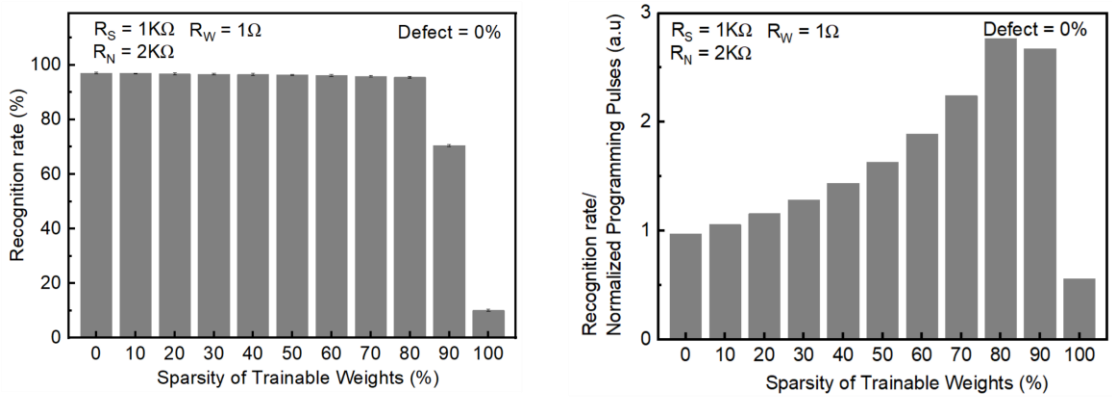

Fig. 7. (a) The simulation of recognition rate for varying the sparsity of trainable weights. Here, 100% weights in the network can be trained, (b) the metric of recognition rate and normalized programming pulse with varying of the sparsity of trainable weights.

To show the effect of defect on the performance of the network, Fig. 8 is tested with 10% and 20% defect. Here, without using the defect-aware-training, the performance of the network drops significantly due to the information of defect map is not used in the training. So, the recognition rates of 10% and 20% defects are as low as 85.3% and 73.7%, respectively. As shown in Fig. 8(a), with the defect-aware-training, the recognition rate is verified with varying

the percentage of the trainable weights when defect rate is 10%, which means that 90% of the weights of the network can still be programmed. The results indicate that the recognition rate is as high as 95% ± 0.5, even when sparsity increases from 0% to about 80% and defect rate is 10%. The result shows that the defect-aware-training method can recover well the recognition rate at defect rate = 10% when comparing the result in Fig. 7(a) at defect rate = 0%. When the sparsity is 90%, it drops sharply to around 65%. Finally, when the sparsity is 100%, it almost completely decreases to 10%, which means that there are no trainable weights available for training. The above results are obtained because gradient-based pruning helps keep important weights. These weights can recover the performance of the network when the sparsity is from 0 to 80%. Meanwhile, with higher sparsity, important weights are also removed, reducing the recovery ability of the network, leading to a sharp decrease in recognition rate. Fig. 8(b) shows the relationship between the recognition rate and the normalized programming pulses. The ratio is highest at about 80% sparsity. This suggests that this level of sparsity is the best for balancing energy efficiency and recognition rate. Similarly, Figs. 8(c) and 8(d) show the same results when the defect is 20% which means that only 80% of weights can still be trained. Fig. 8(c) shows that the recognition rate is still high even when the sparsity is 80%. The efficiency metric is still highest at about 80% sparsity, but it is lower than the metric at 10% defect as shown in Fig. 8(d).

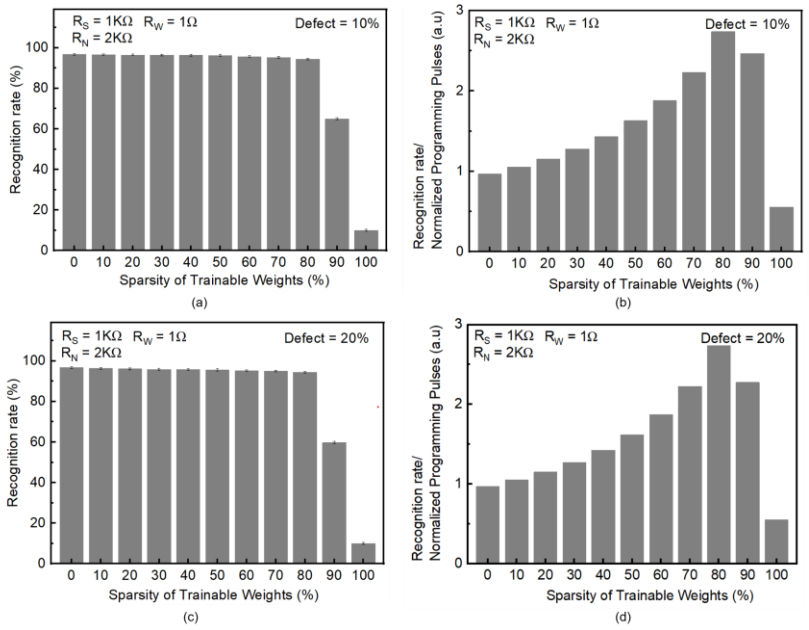

Fig. 8. (a) The simulation of recognition rate of MNIST dataset for varying sparsity of trainable weights with percentage of defect is 10%, (b) the metric of recognition rate and normalized programming pulse with varying of sparsity of trainable weights with percentage of defect is 10%, (c) the simulation of recognition rate for varying sparsity of trainable weights with percentage of defect is 20%, (d) the metric of recognition rate and normalized programming pulse with varying of sparsity of trainable weights with percentage of defect is 20%.

Next, the CIFAR-10 dataset is used to test the performance of the network. Here, CIFAR-10 dataset has 50,000 training images and 10,000 testing images. The size of the image is 32x32 RGB pixels. The ResNet CNN architecture is used for testing the CIFAR-10 dataset. Here, only the fully connected layers in the ResNet are implemented with the memristor crossbars. In the fully connected layer in ResNet, the number of hidden neurons is 100. The neural network has 10 output neurons (He et al., 2016). Firstly, without using the defect-aware-training, the recognition rates of 10% and 20% defects are as low as 79.8% and 60.4%, respectively. Next, with using defect-aware-training, Fig. 9(a) and 9(c) show the recognition rate with varying percentages of the trainable weights when defect rates are 10% and 20%, respectively. The recognition rate can be recovered to 90% ± 0.6 when the sparsity increases from 0 to 80%. When the sparsity is 90% and 100%, it drops significantly. Fig. 9(b) and 9(d) show the relationship between the recognition rate and the normalized programming pulses with defect rates of 10% and 20%, respectively. The ratios remain highest at approximately 80% sparsity. It means that even if the defect rate is up to 20%, the network can obtain the high recognition rate and reduce the number of programming pulses or programming energy by using defect-aware training and the pruning method.

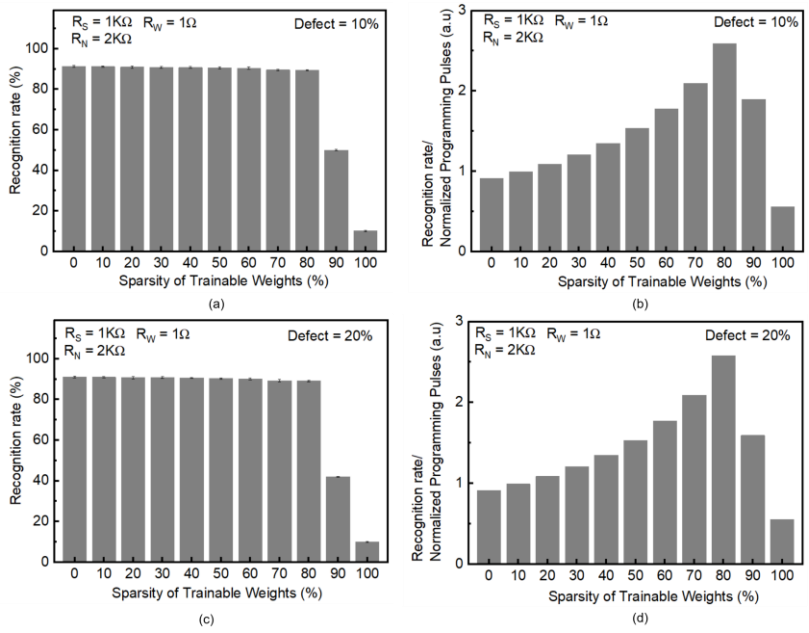

Fig. 9. (a) The simulation of recognition rate of CIFAR-10 dataset for varying sparsity of trainable weights with percentage of defect is 10%, (b) the metric of recognition rate and normalized programming pulse with varying of sparsity of trainable weights with percentage of defect is 10%, (c) the simulation of recognition rate for varying sparsity of trainable weights with percentage of defect is 20%, (d) the metric of recognition rate and normalized programming pulse with varying of sparsity of trainable weights with percentage of defect is 20%.

The results from Fig. 8 and 9 show that an 80% sparsity is optimal for the ternary neural network deployed on the memristor crossbar when recognizing the MNIST and CIFAR-10 datasets. At

this sparsity level, the network maintains high recognition rate even in the percentage of defects is 10% or 20%. Besides, the normalized programming pulses decrease significantly 49.1% at 80% sparsity compared to 20% sparsity. However, when the sparsity is larger than 80%, such as at 90% and 100%, the recognition rate drops significantly. This indicates that too much pruning weights, the network has no more weights to train. Therefore, an 80% sparsity level is optimal, suitable for energy-efficient edge computing applications, ensuring both high accuracy and optimal hardware efficiency even under defect conditions.

4. CONCLUSIONS

The demand for high processing performance and low energy consumption has been a significant challenge for traditional hardware architectures in the context of the increasing deployment of artificial intelligence systems on edge devices. Ternary neural network with three values of weight (-1, 0, and +1) is an effective solution for simplifying computations. However, when implemented on a crossbar memristor network - a potential hardware architecture for in-memory computing - new challenges arise, such as energy consumption during weight programming and the presence of fixed defects during manufacturing.

This paper has proposed a training method that combines two techniques: (1) weight pruning to increase the number of zero-weights, thereby reducing the number of programming pulses, and (2) defect-aware training to maintain network performance. By using defect maps, the network can identify and ignore defect positions while focusing on retraining important weights. The results on the MNIST and CIFAR-10 datasets show that with the sparsity of around 80% and the defect rate of 20%, the network still achieves a high recognition rate while the number of programmed pulses, which is indicative of programming energy, is reduced by more than 49.1% compared to the 20% sparsity. In future work, the proposed method can be further evaluated on large-scale memristor crossbars and higher defect rates to assess its robustness. Scaling up the network size is expected to improve recognition rates due to increased learning capacity, but excessive size in physical crossbars may introduce significant parasitic resistances, leading to voltage drops and reduced energy efficiency. Additionally, co-designing pruning with fault-tolerant architectures, such as reconfigurable or redundant crossbars, could enhance both accuracy and hardware reliability.

5. REFERENCES

Adam, G.C. et al. (2017) '3-D Memristor Crossbars for Analog and Neuromorphic Computing Applications', IEEE Transactions on Electron Devices, 64(1), pp. 312–318. DOI: https://doi.org/10.1109/TED.2016.2630925

Aidi Sharif, M. et al. (2025) 'Artificial Intelligence in Robotic Manipulators: Exploring Object Detection and Grasping Innovations', Kufa Journal of Engineering, 16(1), pp. 136–159. DOI: https://doi.org/10.30572/2018/KJE/160109

Alaa, R. et al. (2024) 'Object detection algorithms implementation on embedded devices: challenges and suggested solutions', Kufa Journal of Engineering, 15(3), pp. 148–169. DOI: https://doi.org/10.30572/2018/KJE/150309

Alemdar, H. et al. (2017) 'Ternary neural networks for resource-efficient AI applications', in International Joint Conference on Neural Networks (IJCNN), USA, pp. 2547–2554. DOI: https://doi.org/10.1109/IJCNN.2017.7966166

Bohr, M.T. and Young, I. (2017) 'CMOS Scaling Trends and Beyond', IEEE Micro, 37(6), pp. 20–29. DOI: https://doi.org/10.1109/MM.2017.4241347

Chakrabarti, B. et al. (2017) 'A multiply-add engine with monolithically integrated 3D memristor crossbar/CMOS hybrid circuit', Scientific Reports, 7, pp. 1–10. DOI: https://doi.org/10.1038/srep42429

Deng, L. (2012) 'The MNIST Database of Handwritten Digit Images for Machine Learning Research', IEEE Signal Processing Magazine, 29(6), pp. 141–142. DOI: https://doi.org/10.1109/MSP.2012.2211477

Graves, C.E. et al. (2020) 'In-Memory Computing with Memristor Content Addressable Memories for Pattern Matching', Advanced Materials, 32(38), 2003437. DOI: https://doi.org/10.1002/adma.202003437

He, K. et al. (2016) 'Deep residual learning for image recognition', in Proceedings of the 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), USA, pp. 770–778. DOI: https://doi.org/10.1109/CVPR.2016.90

Hu, M. et al. (2018) 'Memristor-based analog computation and neural network classification with a dot product engine', Advanced Materials, 30(9), 1705914. DOI: https://doi.org/10.1002/adma.201705914

Indiveri, G. and Liu, S.-C. (2015) 'Memory and Information Processing in Neuromorphic Systems', Proceedings of the IEEE, 103(8), pp. 1379–1397. DOI: https://doi.org/10.1109/JPROC.2015.2444094

Jang, J.T. et al. (2018) 'Effect of oxygen content of the LaAlO3 layer on the synaptic behavior of Pt/LaAlO3/Nb-doped SrTiO3 memristors for neuromorphic applications', Solid-State Electronics, 140, pp. 139–143. DOI: https://doi.org/10.1016/j.sse.2017.10.032

Jo, S.H. et al. (2010) 'Nanoscale memristor device as synapse in neuromorphic systems', Nano Letters, 10(4), pp. 1297–1301. DOI: https://doi.org/10.1021/nl904092h

Krizhevsky, A., Nair, V. and Hinton, G. (2018) CIFAR-10 and CIFAR-100 Datasets. https://www.cs.toronto.edu/~kriz/cifar.html [accessed on 8 August 2025]

Linn, E. et al. (2012) 'Beyond von Neumann - Logic operations in passive crossbar arrays alongside memory operations', Nanotechnology, 23(30), 305205. DOI: https://doi.org/10.1088/0957-4484/23/30/305205

Lin, P. et al. (2020) 'Three-dimensional memristor circuits as complex neural networks', Nature Electronics, 3(5), pp. 225–232. DOI: https://doi.org/10.1038/s41928-020-0397-9

Mohammed, H. et al. (2022) 'A comparative evaluation of deep learning methods in digital image classification', Kufa Journal of Engineering, 13(4), pp. 53–69. DOI: https://doi.org/10.30572/2018/KJE/130405

Nguyen, T.V., An, J. and Min, K.-S. (2021) 'Memristor-CMOS hybrid neuron circuit with nonideal-effect correction related to parasitic resistance for binary-memristor-crossbar neural networks', Micromachines, 12(7), 791. DOI: https://doi.org/10.3390/mi12070791

Nguyen, T.V., Pham, K.V. and Min, K.-S. (2019) 'Hybrid circuit of memristor and complementary metal–oxide–semiconductor for defect-tolerant spatial pooling with Boost-factor Adjustment', Materials, 12(13), 2122. DOI: https://doi.org/10.3390/ma12132122

Pham, K.V. et al. (2018) 'Memristor binarized neural networks', Journal of Semiconductor Technology and Science, 18(5), pp. 568–577. DOI: https://doi.org/10.5573/JSTS.2018.18.5.568

Pham, K.V., Tran, S.B., Nguyen, T.V. and Min, K.-S. (2019) 'Asymmetrical training scheme of binary-memristor-crossbar-based neural networks for energy-efficient edge-computing nanoscale systems', Micromachines, 10(3), 141. DOI: https://doi.org/10.3390/mi10020141

Pouyanfar, S. et al. (2018) 'A survey on deep learning: Algorithms, techniques, and applications', ACM Computing Surveys, 51(1), pp. 1–36. DOI: https://doi.org/10.1145/3234150

Sebastian, A., Le Gallo, M. and Eleftheriou, E. (2019) 'Computational phase-change memory: Beyond von Neumann computing', Journal of Physics D: Applied Physics, 52(44), 443002. DOI: https://doi.org/10.1088/1361-6463/ab37b6

Sheng, X. et al. (2019) 'Low-Conductance and Multilevel CMOS-Integrated Nanoscale Oxide Memristors', Advanced Electronic Materials, 5(3), 1800876. DOI: https://doi.org/10.1002/aelm.201800876

Shrestha, A. and Mahmood, A. (2019) 'Review of Deep Learning Algorithms and Architectures', IEEE Access, 7, pp. 53040–53065. DOI: https://doi.org/10.1109/ACCESS.2019.2912200

Song, C. et al. (2017) 'A quantization-aware regularized learning method in multilevel memristor-based neuromorphic computing system', in IEEE 6th Non-Volatile Memory Systems and Applications Symposium (NVMSA), Hsinchu, Taiwan, 16–18 August, pp. 1–6. DOI: https://doi.org/10.1109/NVMSA.2017.8064465

Strukov, D.B. et al. (2008) 'The missing memristor found', Nature, 453(7191), pp. 80–83. DOI: https://doi.org/10.1038/nature06932

Truong, S.N. et al. (2016) 'New pulse amplitude modulation for fine tuning of memristor synapses', Microelectronics Journal, 55, pp. 162–168. DOI: https://doi.org/10.1016/j.mejo.2016.07.010

Wang, T. et al. (2020) 'Three-Dimensional Nanoscale Flexible Memristor Networks with Ultralow Power for Information Transmission and Processing Application', Nano Letters, 20(6), pp. 4111–4120. DOI: https://doi.org/10.1021/acs.nanolett.9b05271

Wright, C.D. et al. (2013) 'Beyond von-Neumann computing with nanoscale phase-change memory devices', Advanced Functional Materials, 23(18), pp. 2248–2254. DOI: https://doi.org/10.1002/adfm.201202383

Yeo, I. et al. (2019) 'Stuck-at-fault tolerant schemes for memristor crossbar array-based neural networks', IEEE Transactions on Electron Devices, 66(7), pp. 2937–2945. DOI: https://doi.org/10.1109/TED.2019.2914460