Article

ACIMS: Analog CIM Simulator for DNN Resilience

Dong Ding *, Lei Wang *, Zhijie Yang, Kai Hu and Hongjun He

College of Computer Science and Technology, National University of Defense Technology, Changsha 410000, China; yangzhijie@nudt.edu.cn (Z.Y.); hukai18@nudt.edu.cn (K.H.); hhj_hi@sina.com (H.H.)
* Correspondence: dingdong@nudt.edu.cn (D.D.); leiwang@nudt.edu.cn (L.W.); Tel.: +86-138-7312-2396 (D.D.);
+86-138-7499-7430 (L.W.)

Abstract: Analog Computing In Memory (ACIM) combines the advantages of both Compute In Memory (CIM) and analog computing, making it suitable for the design of energy-efficient hardware accelerators for computationally intensive DNN applications. However, their use will introduce hardware faults that decrease the accuracy of DNN. In this work, we take Sandwich-Ram as the real hardware example of ACIM and are the first to propose a fault injection and fault-aware training framework for it, named Analog Computing In Memory Simulator (ACIMS). Using this framework, we can simulate and repair the hardware faults of ACIM. The experimental results show that ACIMS can recover 91.0%, 93.7% and 89.8% of the DNN’s accuracy drop through retraining on the MNIST, SVHN and Cifar-10 datasets, respectively; moreover, their adjusted accuracy can reach 97.0%, 95.3% and 92.4%.

Keywords: DNN accelerator; compute in memory; analog computing; fault resilience

1. Introduction

Recent hardware development in the field of deep learning has exhibited a migration from general-purpose designs to more specialized hardware in order to improve the computational efficiency. Analog Computing In Memory (ACIM) is a good solution to this problem that combines Computing In Memory (CIM) and analog computing, both of which are frontiers of the DNN hardware accelerator. CIM reduces the movement of data between memory and the arithmetic unit of Von Neumann Architecture, which can remove the impact of the memory wall to some extent [1]. Analog computing can execute matrix operations in constant time by exploiting Kirchoff’s law, rather than a sequence of bit operations depending on clock frequency [2].

The ACIM architecture includes an array made up of isomorphic computing elements. Each computing element contains a local memory unit and a calculating unit, adopts an analog computing method. Analog computing utilizes continuous physical quantities such as voltage, current, frequency and pulse-width. Both CIM and analog computing speed up the DNN algorithm; however, they also reduce the DNN’s reliability. CIM introduces additional logic and control circuits to the traditional memory unit; however, this is a new structural and physical design that is neither mature nor stable. Furthermore, analog computing has an inherent random error and a low noise margin, both factors that significantly reduce the resilience of DNN.

Several alternative techniques have been suggested to improve DNN’s resilience when deployed on ACIM devices, including Triple Modular Redundancy (TMR), Error Correction Code (ECC) and fault-aware training [3]. However, TMR introduces large overhead in hardware design, especially for massively parallel processing array. Moreover, ECC can only be used in a memory unit and not in a logic unit owing to its inability to maintain the error correction message transfer between the layers [4]. For its part, fault-aware training has been tested and found to be effective for many DNN inference accelerator fault-patterns, while also having little impact on hardware design.
Our main purpose is to improve the resilience of DNN when deployed on ACIM devices. Thus, we adopt fault-aware training to improve DNN’s resilience. In this paper, we take Sandwich-Ram as an example ACIM device. Sandwich-Ram is a novel ACIM architecture proposed by Jun Yang et al. [5]. It is an energy-efficient in-memory binary weight network architecture with pulse-width modulation. First, we analyze the cause of the accuracy drop when a well-trained DNN is deployed on Sandwich-Ram. Second, we establish a mathematical model of Sandwich-Ram’s fault pattern. Moreover, to reproduce the fault pattern and perform further studies in a more convenient software environment, we propose the Analog CIM Simulator (ACIMS). ACIMS is an injection and training framework that supports ACIM fault, based on a previous work (Ares) proposed by B. Reagen et al. [6]. In the software ACIM fault environment, ACIMS can perform fault-aware training more flexibly than hardware debugging. With the aid of ACIMS, we recover 90% of DNNs’ accuracy drop on ACIM devices, making them reliable and practical. For most datasets, the accuracy can be adjusted to 92%. During this process, we also identify some interesting characteristics of DNN resilience.

The key contributions of this paper can be summarized as follows:

- We take Sandwich-Ram as an example in order to study the fault-pattern of ACIM architecture. We are the first to analyze the cause of ACIM fault. Through verification, our fault-pattern can fit Sandwich-Ram’s fault-pattern.
- We design ACIMS, a configurable fault injection framework that provides three injection methods: specially weight injection, module modification and activation injection. It incorporates a special mechanism, named Hook, to simulate ACIM behavior. ACIMS is capable of supporting multiple fault patterns, including single-bit error, stuck at error and ACIM error.
- With the help of ACIMS, we develop a training-adjusted procedure that can repair the dropped accuracy into the tolerance range. Through this procedure, the adjusted DNN can recover 90% of the accuracy drop of MNIST, SHVN and CIFAR-10 when deployed on ACIM devices. Moreover, the adjusted accuracy of these three datasets can reach 92%.

2. Background

2.1. DNN

DNN is a complex fitting function with a large number of parameters. Its basic expression is a function between input $X$ and label $Y$, as expressed in Equation (1):

$$Y = F(X)$$

$$a^l = \sigma(z^l) = \sigma(a^{l-1}a^{-1} + b^{-1}) \text{ for } l \in [1, N], a^1 = X, a^N = Y$$

Equation (2) is the per-layer formula. Here, $\sigma$ represents the nonlinear components, while $z$ represents the linear component. Given sufficient scale, DNN can fit any low-dimensional function or classification problem provided that this problem is inherently regular [6]. For example, a single-layer neural network can fit a linear function, and a multilayer neural network can fit curves and surfaces function. We suppose that ACIM error has a certain mathematical expression so that DNN can adapt the error as well as to accomplish classification. This ability is the DNN’s inherent resilience, as discussed in previous works (e.g., [7]).

2.2. Sandwich-Ram

Jun Yang et al. proposed a novel architecture named Sandwich-Ram [5], which is an energy-efficient in-memory binary weight network architecture with pulse-width modulation. We use this architecture as an example of CIM to conduct our research. The main mechanism of Sandwich-Ram is Pulse-Width Modulation calculation. This is an Analog Computing mechanism. A schematic diagram is presented in Figure 1 below.
Sandwich-Ram uses pulse-width to represent an analog value. Its feature is an 8-bit unsigned feature and its weight is 1-bit signed. In the computing process, the feature will be transferred to the width of the pulse, while the weight’s sign will be transferred into either high or low voltage. Figure 1a illustrates the first stage of one PWM core. It can perform multiplication between the last two digits of the feature and the weight. Feature [0, 1] encodes four switches, V1–V4, which are linked to the grid voltage of the NMOS transistors. These can control the strength grade of the velocity at which current leaks to the ground and further generate four levels of rising edge delay from $T$ to $4T$ at $P_{Out}$. The remaining three stages of one PWM core cascade behind Figure 1a using inverter $I$ with successive $\times 4$ current velocity discharge. Weight ‘s sign control the voltage’s high or low as shown in Figure 1b,c. When subtracting, the effective voltage is high; the rising edge delay causes the pulse-width to shorten. When add, the effective voltage is low; the rising edge delay causes the pulse-width to elongate.

Sandwich-Ram accomplishes matrix multiplication through the use of three components, Pulse-Width Generator (PWG), Pulse-Width Modulator (PWM) and Pulse-Width Quantizer (PWQ), as shown in Figure 2. One PWM core has local memory space for the feature and weight. The Sandwich-Ram array calculates matrix multiplication from left to right once per row. PWG, on the left-hand side, provides an initial pulse width to prevent negative pulse. PWM, in the central array, modifies the pulse width based on the product of its local feature and weight. PWMs modify the pulse widths one at a time. After one row of PWM is finished, the pulse-width value is converted to a digital value by the PWQ on the right. At the end, accumulators are used to add up each row’s result. The local weights can also be passed between the adjacent units in the data preparation stage so that Sandwich-Ram can accomplish the convolution operation on the basis of matrix multiplication.

![Figure 1. Pulse-Width Modulator (PWM) and pulse-width operation.](image1)

(a) The first stage one PWM core, it can control the pulse-width from $P_{In}$ to $P_{Out}$ according to feature [0, 1]; (b) is an example of subtraction of two units of pulse-width; (c) is an example of add of one units of pulse-width.

![Figure 2. Sandwich-Ram array for matrix multiplication.](image2)

The array combines Pulse-width Generator (PWG), Pulse-Width Modulator (PWM) and Pulse-Width Quantizer (PWQ). Each PWM has a local feature and weight.
3. ACIM Fault

In this section, we analyze the cause of ACIM fault and present a mathematical Fault Pattern enabling Sandwich-Ram to fit the actual hardware.

3.1. Hardware Error

All hardware has a process deviation. As represented by the grayscale part of Figure 2, Sandwich-Ram’s calculation also involves hardware error; this is equivalent to measuring an object with an incorrectly calibrated ruler. Process deviation is the most significant source of hardware error among all error sources. Although Sandwich-Ram attempts to adopt a consistent design on each pulse-width unit, differences in each component still arise during the chip production process. This deviation may lead to inconsistencies in leakage current velocity, as well as to differences in the intrinsic pulse-width of each unit. Process deviation is randomly distributed among a batch of chips; when considering one particular taped-out chip, however, its intrinsic pulse-width is stable and regular. We can thus obtain the first stage of Sandwich-Ram’s fault pattern:

\[ Y_i = \left( \text{Pre} \times G_i + \sum_{i=1}^{M} \sum_{j=1}^{N} M_{ij} \times W_{ij} \times X_{ij} \right) / Q_i - \text{Pre} \] (3)

Equation (3) is a normal matrix multiplication, while Equation (4) is the result of Sandwich-Ram. \( Y_i \) is the accumulated sum of row \( i \). As shown in Figure 2, \( G_i, M_{ij} \) and \( Q_i \) represent the intrinsic pulse-width of PWG, PWM and PWQ. Pre represents the reserved amount, subtracted after quantizing, which prevents the pulse-width value from being less than zero.

3.2. Environmental Error

When a particular sandwich chip is in operation, there are also other disturbances in the environment, such as random errors caused by temperature, supply voltage, cosmic rays, etc. These forms of environmental interference are largely normally distributed and manifest in the pulse-width. We use the Gaussian distribution \( \mathcal{N}(\mu, \sigma^2) \) in Equation (5) to represent the influence of environmental errors:

\[ Y_i = \left( \text{Pre} \times G_i + \sum_{i=1}^{M} \sum_{j=1}^{N} M_{ij} \times W_{ij} \times X_{ij} \times \mathcal{N}(\mu, \sigma^2) \right) / Q_i - \text{Pre} \] (5)

3.3. Rounding Error

Sandwich-Ram adopts 8-bit unsigned feature and 1-bit weight, as well as analog computing. Before calculation, the well-trained DNN parameters are quantized into either 8-bit or 1-bit; in our model, we use the Uniform Affine Quantizer [8] to quantize the parameters. After calculation, the output analog value is quantized into a digital value. The third stage fault pattern takes \( W, X \) and quantization into consideration, such that we obtain Equation (6):

\[ Y_i = \left\lfloor \left( \text{Pre} \times G_i + \sum_{i=1}^{M} \sum_{j=1}^{N} M_{ij} \times \text{Sign}(W_{ij}) \times \left\lfloor \frac{X_{ij}}{8} \right\rfloor \times 8 \times \mathcal{N}(\mu, \sigma^2) \right) / Q_i \right\rfloor - \text{Pre} \] (6)

It is important to note that environmental error introduces the noise, while low data precision also increases the noise margin. There is a constraint relationship between the environmental error and the rounding error. Thus, they are secondary errors in contrast to hardware error.
3.4. Error and Verification

Generation of the ACIM error in the simulator takes place over two phases. In the first phase, we generate the hardware fault. We use the Gaussian distribution \( \mathcal{N}(1, \frac{1}{6}) \) to create the intrinsic pulse-width parameters \( G_i, Q_i \) and \( M_{ij} \). Most of the parameter data are distributed from \(-50\%\) to \(50\%\). We use these data in line with the research of Yang et al. [5], who measured the pulse-width variation to be \(-35.1\%\) at ff70 and \(63.0\%\) at ss0 compared to tt25. Here, ff70, tt25 and ss0 are typical stages in chip manufacturing process. Moreover, ff70 and ss0 are the limiting cases of process deviations, while tt25 is the statistical center. Our randomly distributed selection can cover most error margins in a real situation. Furthermore, \( Pre = 7140 = 255 \times 28 \) is set to prevent overflow from occurring case of a whole line of \( 0xff \times -1 \). In the second phase, we generate the environment fault with \( \mathcal{N}(1, \frac{1}{12}) \), which is a relatively weak error. The rounding fault is generated naturally by means of the simulator’s data format.

The two phases are carried out at different times. The first phase is executed only once before the DNN operation. Although there are random deviations during chip production, for a particular ACIM chip, its intrinsic pulse-width is fixed. The intrinsic matrix \( M_{ij} \) will remain constant during the entire DNN operation. The second phase is carried out in every single layer operation.

We further design a verification process focusing on the Hardware fault. This experiment can verify the correctness of the intrinsic pulse-width matrix in our fault model. We compare the one-row vector product between the RTL design developed by Yang’s team and our simulator’s hardware fault in Python. We set the operation with a 28 unsigned 8-bit 0xff feature vector and a 28 1-bit weight vector of +1 or \(-1\). We further test these operations in three corners (ff70, tt25 and ss0) given by Sandwich-Ram’s RTL design. The results are shown in Figure 3.

In the verification, we choose tt25 as the standard condition and modify the delay parameters in Sandwich-Ram’s RTL design, along with the deviation matrix in the Python simulator. The deviation range is set from \(-35.1\%\) to \(63.0\%\). This experiment indicates that the pulse-width deviation impacts the calculation results and that our fault model can detect the deviation.

![The Vector Product](image)

**Figure 3.** Verification of pulse-width deviation model.
4. ACIMS

The previous fault injection framework Ares [6] only modified the stored weights and changed the results of layers. It is not therefore suitable for ACIM fault simulation today, as new types of ACIM fault vary from design to design. To meet the requirement of new device simulation, we propose the Analog CIM Simulator (ACIMS) based on PyTorch.

4.1. ACIMS Framework

Figure 4 presents the Flow Diagram of ACIMS. Given a well-trained DNN model, it will obtain a GoldenAcc on a CPU or GPU platform. We name the classification accuracy on CPU the GoldenAcc because the CPU has been verified to be sufficiently stable. ACIM devices such as Sandwich-Ram damage DNN’s classification accuracy. Moreover, the accuracy obtained from the physical ACIM device is referred to as the PhysicalAcc. Acc obtained from Sandwich-Ram is notably lower than the GoldenAcc obtained from CPU due to Sandwich-Ram’s fault pattern. However, it is somewhat inconvenient for researchers to study the fault pattern on hardware directly.

Accordingly, we design ACIMS to simulate physical devices. The fault pattern extracted in Section 3 is injected into ACIMS. This enable us to obtain a StrickenAcc from this simulator. The StrickenAcc is a recurrence of the physical accuracy; it can demonstrate the accuracy drop of a well-trained DNN model first affected by ACIM fault.

ACIMS then begins to implement the retraining environment. Fault awareness training is a good choice for adjusting the stricken DNN model. Through retraining, we can obtain an adjusted fault resilience DNN model, the classification accuracy of which has been adjusted, named AdjustedAcc.

![Image of Flow Diagram of ACIMS](image)

**Figure 4.** Flow Diagram of ACIMS. ACIMS provides four kinds of accuracy: GoldenAcc (from traditional processors), PhysicalAcc (from ACIM devices), StrickenAcc (from fault injected model) and AdjustedAcc (from the adjusted fault model).

4.2. Fault Injection Method

Most DNN faults occur in the areas of weights, neuron modules and activations [9]. ACIMS provides three fault injection methods: weight injection, module modification and activation injection. As Figure 5 shows, we use a mask to inject a proper fault pattern into the appointed layers.

Weight Injection is a static injection method. It can inject faults into the stored parameters of DNN, which can simulate memory faults such as single-bit error. Module modification is a dynamic injection that simulates faults occurring in layers with parameters in every forward and backward propagation. Module modification replaces the standard DNN module with a customized module. For example, ACIMS implements a customized ConvQ module with multiplication between the 8-bit unsigned feature and 1-bit weight rather than using the standard fp32 model to simulate Sandwich-Ram’s data format. Activation injection is another dynamic injection method that simulates activation layers without parameters.
To enhance the universality of the simulator, the ACIMS framework provides basic configurable variables, including network models, datasets, optimization methods and other DNN basic topologies. The three injection methods can support a range of faults including single-bit error, symbol reversal, Gaussian noise, etc.

4.3. Hardware Simulator

Hook is an original method created by ACIMS from PyTorch’s forward_hook mechanism, as shown in the bottom right of Figure 5. A standard module defines the forward and backward calculation procedures. When rewriting the forward computing procedure, its backward procedure is automatically changed by the AutoGrad mechanism of PyTorch. While this mechanism is useful, it is also not appropriate for hardware fault simulation. Many DNN accelerators implement only a forward circuit. When a fault occurs in the forward circuit, the DNN algorithm is unable to detect it. It is in this case that Hook comes into effect. Using the forward_hook mechanism, Hook acquires input data through the bypass. It then calculates the results according to the Sandwich-Ram fault pattern and returns the results to the next layer. The procedure of Hook is invisible to AutoGrad.

5. Evaluation

5.1. Experimental Setup

Given a dataset and DNN model, ACIMS will go through two stages of training and obtain three test accuracies. In the first training stage, ACIMS trains the original model and obtains the first test accuracy, GoldenAcc, which is taken as a baseline. ACIMS then injects Sandwich-Ram fault into the model and obtains the second test accuracy, StrickenAcc. In the second training stage, ACIMS trains the fault model using the fault-aware training method and obtains the third test accuracy, AdjustAcc.
To evaluate the effect of fault-aware training, two indexes, \( \text{ResRate} \) and \( \text{AdjRate} \) are derived in Formulas (7) and (8):

\[
\text{ResRate} = \frac{\text{GoldenAcc} - \text{StrickenAcc}}{\text{GoldenAcc}} \tag{7}
\]

\[
\text{AdjRate} = \frac{\text{AdjustedAcc} - \text{StrickenAcc}}{\text{GoldenAcc} - \text{StrickenAcc}} \tag{8}
\]

\( \text{ResRate} \) is used to measure the resistance ability of the original DNN model. \( \text{ResRate} \) represents the accuracy drop ratio of the fault strike: the smaller, the better. Moreover, the \( \text{AdjRate} \) is used to represent the adjustment efficiency. \( \text{AdjRate} \) represents the accuracy recovery ratio of fault-aware training: the higher, the better. \( \text{AdjRate} \) is correlated with DNN topology, quantization precision, fault injection strength and retraining parameters, among other factors.

5.2. Models Configuration and Software Platform

We assembled a collection of three datasets and four kinds of DNN models, as summarized in Table 1. To train our networks, we selected three datasets that are commonly used in image classification: MNIST, SVHN and CIFAR-10. In the first group, we deploy MNIST to three different scales of Multilayer Perceptron (MLP) networks: small, middle and large. The small one, MLP-S, has two hidden layers of 10–15 nodes. The middle one, MLP-M, has 10–30 nodes. Finally, the large one MLP-L has 100–300 nodes. The second group of experiments combined two datasets and three kinds of models. The goal here is to compare the difference between the performance of LeNet, Vgg and ResNet on SVHN and CIFAR10.

| Dataset  | Model       | MLP-S | MLP-M  | MLP-L  |
|---------|-------------|-------|--------|--------|
| MNIST   | MLP-S       | 10–15 | MLP-M  | 10–30  |
|         | MLP-M       |       |        | 100–300|
| SVHN    | LeNet       | VGG13/16 | ResNet18/34|
| CIFAR10 | LeNet       | VGG13/16 | ResNet18/34|

The training parameters remain the same for all experiments. The loss function is cross-entropy; the optimizer is SGD; the initial learning rate is 0.1; and the LR-scheduler is CosineAnnealingWarmRestarts. Our operational environment is Ubuntu 18.4, Cuda 11.0, PyTorch 1.7 with two NVIDIA Tesla V100.

5.3. Experimental Result

In this section, we use ACIMS to quantify \( \text{ResRate} \) and \( \text{AdjRate} \) with different models and datasets. In Section 5.3.1, ACIMS faults are injected into different scales of MLP Net, and the dataset used is MNIST. Section 5.3.2 compares the effect of multiply models on the datasets SVHN and CIFAR-10.

5.3.1. Scaled MLP Model

Figure 6 presents our ACIMS experiment of MNIST on the scaled MLP network. In the first 100 epochs, ACIMS trained the original MLP network. The dot represents the best test accuracy of the original model, which we refers to as GoldenAcc. At the 100th epoch, the AIMC fault is injected into MLP networks. The dot at the 100th epoch represents the results of MLP with faults, which is StrickenAcc. In the last 50 epochs, ACMIS begins the retraining stage. After the parameters are adjusted by means of fault perception training, ACIMS obtains AdjustedAcc (at the third dot).
From the line chart, we can see that, when the original network is stricken by fault injection, its testing accuracy sharply declines. Subsequently, though retraining, the accuracy is gradually increased. We can determine that a larger MLP generally performs better than a smaller one in GoldenAcc, StrickenAcc and AdjustedAcc. Moreover, it is a coincidence that MLP-S and MLP-L obtain the best testing accuracy in the same epoch; this is due to the consistent update cycle of the learning rate.

The ResistRate and AdjustRate results are presented in Table 2. The number in brackets indicates the number of training epochs at which ACMIS reaches the best test accuracy in that stage. Several conclusions can be drawn from these experiments. First, the information contained in a particular dataset is limited, and the larger network has redundancy in processing this dataset. From MLP-S to MLP-M, the GoldenAcc improves by about 6% while the model size doubles. From middle MLP to large MLP, the accuracy improves by only 3% with a tenfold increase in size. The theory that DNN robustness varies according to the number of neurons was proposed by Clay [10]. Second, ACIM fault does have some regularity, which can be learned by DNN’s redundancy. All three scaled models repair ACIM faults. In short, the MLP network can learn the mathematical model of ACIM fault, and a larger MLP has more redundancy and recovers more quickly.

Figure 6. The ACIMS experiment on MNIST with different scaled models. In the first 100 epochs, ACIMS trains the original MLP model; the best accuracy is dotted and labeled GoldenAcc. The vertical dashed line indicates that the fault pattern is injected at the 100th epochs and the point on the line marks the StrickenAcc. In the last 50 epochs, ACIMS retrains the fault injected model; the best accuracy is dotted and labeled AdjustedAcc.

Table 2. The ResRate and AdjRate of MNIST in different scales MLP.

|                  | MLP-S     | MLP-M     | MLP-L     |
|------------------|-----------|-----------|-----------|
| GoldenAcc        | 88.12%    | 94.32%    | 97.78%    |
|                  | (89) *    | (99)      | (89)      |
| StrickenAcc      | 53.09%    | 61.73%    | 89.04%    |
| AdjustAcc        | 79.62%    | 88.42%    | 97.00%    |
|                  | (144)     | (137)     | (144)     |
| ResRate          | 0.397     | 0.345     | 0.089     |
| AdjRate          | 0.757     | 0.818     | 0.910     |

* The number in the brackets is the epoch at which the model reaches the best accuracy in the corresponding training stage.
5.3.2. Various Models

In this section, we test SVHN and CIFAR-10 on various models using ACIMS. In the first 300 epochs, ACIMS trains the original model. Error is injected in the 300th epoch. ACIMS will spend 100 epochs conducting fault-aware training. The experimental results are represented in Tables 3 and 4.

As can be seen from the statistical data, LeNet’s accuracy cannot reach 90% on SVHN and CIFAR-10; thus, it is not sufficient for these two datasets, as they contain more information than LeNet can learn. Moreover, ResNet performs better than VGG in terms of GoldenAcc and AdjustAcc; this reinforces the theory that redundancy is caused by network complexity.

Table 3. SVHN with various models.

| SVHN | GoldenAcc | StrickenAcc | AdjustAcc | ResRate | AdjRate |
|------|------------|-------------|-----------|---------|---------|
| Lenet | 89.37% (24) * | 73.66% | 88.62% (386) | 0.175 | 0.952 |
| VGG13 | 95.55% (239) | 85.39% | 94.82% (381) | 0.106 | 0.927 |
| VGG16 | 95.75% (290) | 86.99% | 95.70% (353) | 0.091 | 0.994 |
| ResNet18 | 95.73% (270) | 63.58% | 94.99% (331) | 0.335 | 0.976 |
| ResNet34 | 95.75% (245) | 89.15% | 95.34% (376) | 0.068 | 0.937 |

* The number in the brackets is the epoch at which the model reaches the best accuracy in the corresponding training stage.

We further note that a deeper VGG and ResNet cannot always perform better than a shallower one. ResNet34’s ResRate is worse than ResNet18’s on CIFAR-10, while the GoldenAcc is also lower. We think there are several reasons for this: specifically, the model’s depth, width and training duration. In this experiment, we assume that the whole model is deployed on an ACIM device; thus, ACIMS injects errors in all layers of the model. Furthermore, ResNet34 has the same width as ResNet18 but a greater depth. In other words, ResNet34 suffers more breakdowns with the same redundancy. We should however also note that ResNet34 achieves the best accuracy in the training and retraining stages at the ending epochs, which implies that the deep network model has not been fully trained. VGG13 and VGG 16 exhibit a similar phenomenon on the SVHN dataset.

Table 4. CIFAR-10 with various models.

| CIFAR-10 | GoldenAcc | StrickenAcc | AdjustAcc | ResRate | AdjRate |
|----------|------------|-------------|-----------|---------|---------|
| LeNet | 75.50% (224) * | 60.22% | 70.56% (397) | 0.202 | 0.676 |
| VGG13 | 92.55% (298) | 77.85% | 91.22% (355) | 0.158 | 0.909 |
| VGG16 | 92.32% (265) | 80.51% | 91.04% (388) | 0.127 | 0.891 |
| ResNet18 | 94.09% (257) | 85.51% | 92.98% (364) | 0.091 | 0.870 |
| ResNet34 | 93.47% (297) | 83.66% | 92.47% (397) | 0.104 | 0.898 |

* The number in the brackets is the epoch at which the model reaches the best accuracy in the corresponding training stage.

Despite the presence of some problems that should be addressed in future work, the goal of this paper is achieved. For common datasets, the fault they suffered from (hardware) can be repaired in ACIMS. The experimental results show that ACIMS can recover 91.0%, 93.7% and 89.8% of the DNN accuracy drop through retraining on the MNIST, SVHN and Cifar-10 datasets. The adjusted accuracy can reach 97.0%, 95.3% and 92.4%. Through ACIMS with fault-aware training, we thus improve the resilience of DNN in ACIM devices.
6. Conclusions

In this paper, we propose ACIMS, which can improve the resilience of DNN deployed on analog CIM devices. We randomly instantiate a set of parameters according to the ACIM fault model, which can be thought of as encountering a particular ACIM chip that contains errors. After retraining, ACIMS can learn the fault’s regulation contained in that error chip. This information is incorporated into the DNN model’s parameters such as VGG16 and ResNet18. Finally, the adjusted DNN model can reach high accuracy in the particular error chip.

Moreover, the ACIMS framework is a DNN training procedure implemented in PyTorch. Its function and operation are consistent with the hardware accelerator. This procedure can be easily carried out on real hardware and achieve a similar effect. In future work, we aim to explore more efficient adjusting strategies. The redundancy related to model topologies also merits further investigation.

Author Contributions: Conceptualization, D.D. and L.W.; methodology, D.D. and L.W.; software, D.D.; validation, D.D., K.H. and Z.Y.; formal analysis, D.D. and L.W.; investigation, D.D.; resources, D.D.; data curation, D.D.; writing—original draft preparation, D.D.; writing—review and editing, D.D., Z.Y. and L.W.; visualization, D.D.; supervision, H.H. and L.W.; project administration, L.W.; and funding acquisition, L.W. and H.H. All authors have read and agreed to the published version of the manuscript.

Funding: This research was funded by the National Key R&D Program of China (grant number 2018YFB2202600).

Data Availability Statement: Publicly available datasets were analyzed in this study. This data can be found here: Mnist from http://yann.lecun.com/exdb/mnist/ (accessed on 14 March 2021). SVHN from http://ufldl.stanford.edu/housenumbers (accessed on 14 March 2021). CIFAR10 from http://www.cs.toronto.edu/kriz/cifar.html (accessed on 14 March 2021).

Conflicts of Interest: The authors declare no conflict of interest.

Abbreviations

The following abbreviations are used in this manuscript:

ACIM Analog Computing In Memory
ACIMS Analog Computing In Memory Simulator
CIM Computing In Memory
DNN Deep Neural Network
ECC Error Correction Code
LR Learning Rate
MLP Multilayer Perceptron
TMR Triple Modular Redundancy
PWG Pulse-Width Generator
PWM Pulse-Width Modulator
PWQ Pulse-Width Quantizer

References

1. Xue, C.; Chang, M. Challenges in Circuit Designs of Nonvolatile-memory based Computing-In-Memory for AI Edge Devices. In Proceedings of the 2019 International SoC Design Conference (ISOCC), Jeju, Korea, 6–9 October 2019; pp. 164–165.
2. Haensch, W.; Gokmen, T. The Next Generation of Deep Learning Hardware: Analog Computing. Proc. IEEE 2019, 107, 108–122. [CrossRef]
3. Mittal, S. A survey on Modeling and Improving Reliability of DNN Algorithms and Accelerators. J. Syst. Archit. 2020, 104, 101689. [CrossRef]
4. Fernandes dos Santos, F.; Draghetti, L. Evaluation and Mitigation of Soft-Errors in Neural Network-Based Object Detection in Three GPU Architectures. In Proceedings of the 2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W), Denver, CO, USA, 26–29 June 2017; pp. 169–176.
5. Yang, J.; Kong, Y. 24.4 Sandwich-RAM: An Energy-Efficient In-Memory BWN Architecture with Pulse-Width Modulation. In Proceedings of the 2019 IEEE International Solid-State Circuits Conference-(ISSCC), San Francisco, CA, USA, 17–21 February 2019; pp. 394–396.

6. Reagen, B.; Gupta, U. Ares: A Framework for Quantifying the Resilience of Deep Neural Networks. In Proceedings of the 2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC), San Francisco, CA, USA, 24–28 June 2018; pp. 1–6.

7. Yarotsky, D. Error Bounds for Approximations with Deep ReLU Networks. *Neural Netw.* 2017, 94, 103–114. [CrossRef] [PubMed]

8. Krishnamoorthi, R. Quantizing Deep Convolutional Networks for Efficient Inference: A Whitepaper. *arXiv* 2018, arXiv:abs/1806.08342. [CrossRef]

9. Torres-Huitzil, C.; Girau, B. Fault and Error Tolerance in Neural Networks: A Review. *IEEE Access* 2017, 5, 17322–17341. [CrossRef]

10. Clay, R.D.; Sequin, C.H. Fault Tolerance Training Improves Generalization and Robustness. In Proceedings of the 1992 IJCNN International Joint Conference on Neural Networks, Beijing, China, 3–6 November 1992; pp. 769–774.