Hardware Accelerator for Adversarial Attacks on Deep Learning Neural Networks

Haoqiang Guo, Lu Peng, Jian Zhang, Fang Qi, Lide Duan*
Louisiana State University, *Alibaba Group
{ghaoqi1, lpeng, jz, fqi1}@lsu.edu, lide.duan@gmail.com

***Abstract*—Recent studies identify that Deep learning Neural Networks (DNNs) are vulnerable to subtle perturbations, which are not perceptible to human visual system but can fool the DNN models and lead to wrong outputs. A class of adversarial attack network algorithms has been proposed to generate robust physical perturbations under different circumstance. These algorithms are the first efforts to move forward secure deep learning by providing an avenue to train future defense networks, however, the intrinsic complexity of them prevents their broader usage.**

In this paper, we propose the first hardware accelerator for adversarial attacks based on memristor crossbar arrays. Our design significantly improves the throughput of a visual adversarial perturbation system, which can further improve the robustness and security of future deep learning systems. Based on the algorithm uniqueness, we propose four implementations for the adversarial attack accelerator (A^3) to improve the throughput, energy efficiency, and computational efficiency.

***Index Terms*—Deep Learning Visual Classification, Hardware Accelerator, Adversarial Attacks, Memristor Crossbar**

I. Introduction

Recent breakthroughs in Artificial intelligence (AI) especially Deep Learning [1] have made great advances in many application fields including Autonomous Driving, Go game, and High-frequent trading in security markets, etc. However, recent studies in the computer vision area identify that Deep-learning Neural Networks (DNNs) are vulnerable to subtle perturbations to inputs which are not perceptible to human beings but can fool the DNN models and lead to wrong outputs [2]. Many algorithms were proposed to generate robust visual adversarial perturbations under different physical circumstances [3] [4] [5] [6]. These algorithms basically include a forward-propagation process and a error-propagation process. The former is similar to an inference procedure in regular Convolution Neural Networks (CNNs), while the latter is different from conventional CNN training process in that the inputs rather than the synaptic weights are changed in the back-propagation process.

The RP2 algorithm [3] and other adversarial attack networks (AttackNet) [4] [5] [6] are the first efforts to move toward secure deep learning by providing an avenue to train future defense networks [7] [8] [9] [10]. However, the intrinsic complexity of these algorithms prevents broader usage of them. Although recent proposals for CNN training architectures may apply to AttackNet, they were not specifically designed for the unique needs of the algorithm, resulting in low efficiency. The CNN training process consists of a Forward Propagation (FP) and a Backward Propagation (BP) for both errors and weights, while AttackNet includes the FP but a different BP for errors only. For a network with depth L, the duration of the neurons of layer l in the buffer is $2(L-l)+1$ in CNN training process, which indicates that a large buffer is needed to minimize the gap between memory and the processor. Meanwhile, for the RP2 algorithm, the duration of neurons of any layer in the buffer is only 1. We quantitatively analyzed the neuron storage requirements of benchmarks used in this work in Fig.1. The experiment results have been normalized to that of RP2 training. We can see that the overall storage requirement of the RP2 training is less than 1/10 of the CNN training. Although we take RP2 as a specific example to demonstrate our designs in this work, the optimization techniques we introduced can also be applied to other adversarial attack networks.

Analog *in-situ* memristor crossbar arrays have been widely used as a potential neural network accelerator because it largely reduces the energy cost of data movement. Prior research [11] [12] [13] [14] demonstrated that memristor designs outperform GPU and ASIC designs [15] [16] in both throughput and energy saving. Although Pipelayer [14] has developed a CNN training platform based on memristor crossbar arrays, directly deploying AttackNet to Pipelayer is highly inefficient [17]. First, AttackNet does not require the same amount of on-chip buffers as what CNN training does. The back propagation process of AttackNet only involves error propagation without updating weights, which can relieve the neuron storage requirement. Unnecessary on-chip buffers consume too much energy and area. Second, the utilization of memristor crossbar arrays is low. Pipelayer utilizes dual-crossbar to store weight values, which are calculated by collecting the difference between a positive crossbar subarray and a negative crossbar subarray. In this way, the on-chip crossbar utilization has been reduced by half.

In this paper, we first identify the uniqueness of the typical visual adversarial perturbation algorithm. We analyze its dataflow and data dependence. To our knowledge, we propose the first hardware accelerator for adversarial attacks (A^3) using memristor crossbar arrays to significantly improve the throughput of the visual adversarial perturbation system. As a result, the robustness and security of future deep learning systems can also be improved. The main contributions of this paper can be summarized as follows.

- We quantitatively demonstrate that directly deploying AttackNets training on existing CNN training platforms is inefficient in both performance and energy.

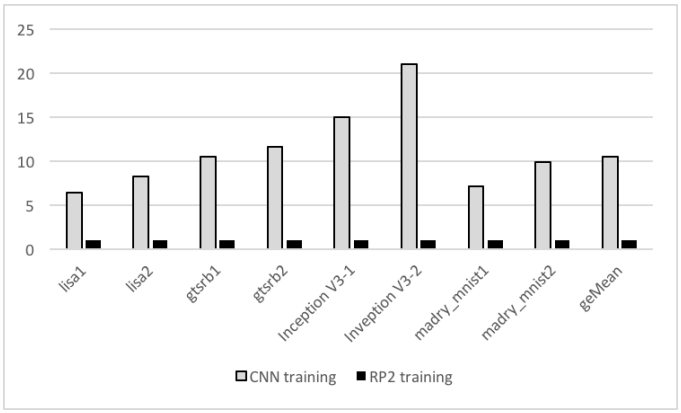


Fig. 1: Neuron storage analysis

- We explore methods of buffer reduction to increase compute engines and therefore improve the performance.
- We incorporate single crossbar storage into the training process to improve the crossbar utilization and to further boost the performance.

II. BACKGROUND

A. The training process of CNN and AttackNet Algorithms

The network architecture of both applications is cascaded, layer by layer. As shown in Fig.2, the training process of CNN can be divided into two stages, forward propagation (FP) and back propagation (BP). We denote the output of the neurons at layer $l-1$ as d^{l-1}. During FP, The output of the neurons at the next layer (layer l) can be computed by first convolving d^{l-1} with the weight matrix W^l, then going through an activation function f. There are two steps in the BP of CNN. The first step is to compute errors (also called sensitivity in some literature) and propagate them backwards layer by layer. Such process is denoted as EP in this work. In the second step, the gradients of the weights at each layer l, ΔW^l, are computed based on the error δ^l and the output d^{l-1}. Once all the gradients are available, the weight matrices for all layers can be updated in parallel using gradient descent. We refer this weight-related process as WP.

For AttackNet, its FP is roughly the same as that of the CNN. However, in contrast to CNN, whose BP includes both the EP and the WP processes, AttackNet's BP stage involves only EP, with the addition that it also computes the partial derivative of the loss function with respect to the perturbation input, in order to update the input. (The weights of the AttackNet are not updated and thus there is no need to compute ΔW^l neither.)

B. AttackNet Applications

At a high level, AttackNet takes in an adversarial image, which is a combination of the original/clean image and a perturbation image (called "mask" in some literatures), to proceed the FP process. While the loss function of a regular CNN is often based on the outputs of the FP and the target labels, the loss of an AttackNet additionally incorporates the

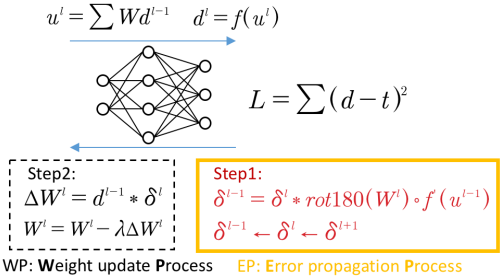


Fig. 2: CNN training algorithm ***V.S.*** AttackNet training algorithm. $*$ and $\circ$ denote the convolution operation and the element-wise production respectively.

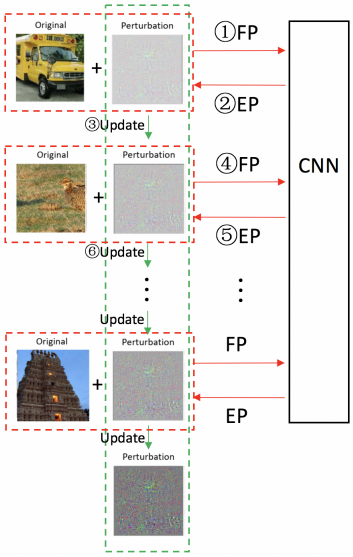


Fig. 3: AttackNet training scheme at a high level.

norm of the perturbation image. In BP, different from CNN training, AttackNet only executes the EP process. At the end of BP, it computes the error/derivative of the perturbation image and uses the derivative to update the perturbation (mask). This completes one training iteration. Multiple iterations are taken to train an optimal attack perturbation.

C. Neural Network Acceleration in Crossbar

There are a lot of work about neural network processing in memory. A neural network is composed of convolutional layers, pooling layers and full connected layers. All of above layers' computation can be transformed into a Matrix-Vector multiplication. For a convolutional layer or a full connected layer, the weights of it are programmed in crossbar in a matrix which is called weight matrix. For max pooling layer, a matrix of 4×6 (which is composed of 0,1,-1) is programmed in crossbar [12]. For average pooling layer, a matrix of 4×6 (which is composed of 1) is programmed in crossbar.

Generally, the resistance $1/g_{ij}$ (the crossing point at the *i*th row and the *j*th column) can only be positive value, while the weight value may be negative, so a 'positive' crossbar and

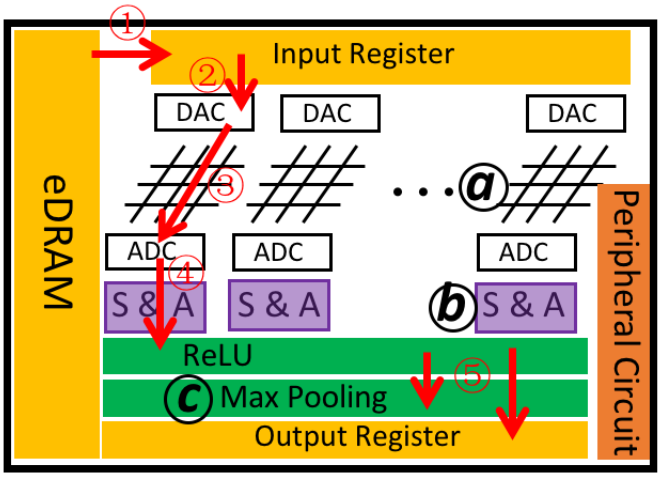


Fig. 4: A^3 architecture. Black circle ⓐ,ⓑ and ⓒ refer to crossbar, Shift&Add unit and Max pooling unit, respectively. Red number circles refer to data flow.

a 'negative' crossbar are utilized to store a matrix of rational values in dual-crossbar storage situation. If a weight value w_{ij} is negative, g_{ij-} is larger than g_{ij+} so that $g_{ij+} - g_{ij-} = w_{ij}$. If a weight value w_{ij} is positive, g_{ij+} is larger than g_{ij-} so that $g_{ij+} - g_{ij-} = w_{ij}$. Considering the limited resistance of crossbar cells, bit slicing was used to store a single weight value, which means multi-cells in the same word line are used to store a value.

The crossbar performs a matrix-vector multiplication by imposing different voltages, which equal the components of the vector, to different word lines. The current flows along the bit line can be seen as the dot product of two vectors. The above process being performed in many bit lines can represent a matrix-vector multiplication.

III. A^3 ARCHITECTURE

In this section, we first introduce the adversarial attack accelerator (A^3) architecture, followed by the first optimization with trading off buffer storage with compute engines and the second optimization about improving the crossbar utilization. Finally, we present other associated unit designs.

At a high level, A^3 is mainly composed of storage units and functional units. The functional units are composed of crossbars, DACs, ADCs, Shift&Add units, activation functional units, and max-pooling units. The storage units consist of eDRAM and input/output registers. We additionally deploy a peripheral circuit unit, in which the most important component is a finite state machine to control instruction flow. To support the training process, the peripheral circuit also covers a subtractor and a multiplier to calculate the errors at the end of FP. The overall architecture is shown in Fig.4.

The data flow of A^3 is shown in red number circles in Fig.4. During FP, inputs are loaded from eDRAM to input registers (step ①), then fed to crossbars after going through a digital-to-analog converter (DAC) (step ②). The crossbars perform

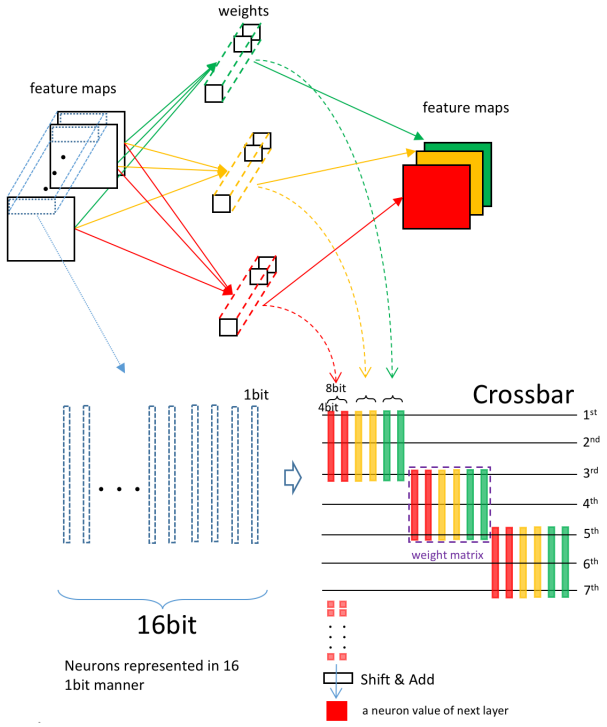


Fig. 5: Weight mapping to crossbars. Two cell in the same row to store a weight value, and each cell is 4-bit. The technique is called bit slicing.

the matrix-vector multiplication. The results of dot-product operations flow through an analog-to-digital converter (step ③), shift and accumulate the adjacent bit line in Shift&Add units (step ④). Next, the dot-product results are forwarded to activation units. All the above operations are performed in a logical cycle. After the activation, there are two paths. If the layer is followed by a pooling layer, the data flows into another cycle to perform the pooling operation. At the end of this cycle, the results are stored to output registers. Otherwise, the results bypass the pooling unit and are stored to output register directly (step ⑤).

The weights of a network layer are programmed into crossbars in the form of a matrix. The method to transform the computations of a convolutional layer can be found in cuDNN documents [18]. The strategy of mapping weights to crossbars is similar to [11], as shown in Fig.5. The weight matrices corresponding to the connections of the same channel (red) of the next layer are mapped to a logical bit line, followed by matrices of other channels (yellow, green) of the same layer. All the logical bit lines of the same layer make up of a logical mapping matrix. To maximize the usage of overlapping neurons of adjacent convolution operations, multiple copies of the logical mapping matrix are put after each other when programming the weight matrix to the crossbar. In A^3, we use two physical bit lines to represent a logical line. Different from traditional PIM platforms such as Pipelayer, A^3 exploits memory (eDRAM, input/output registers) optimization with

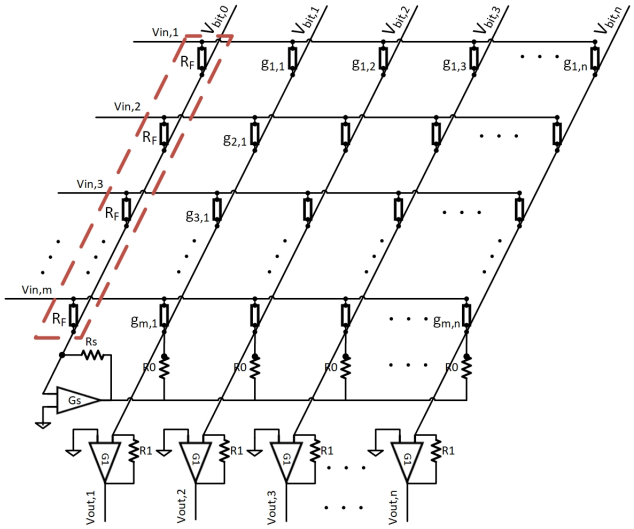


Fig. 6: Weight storage using single crossbar

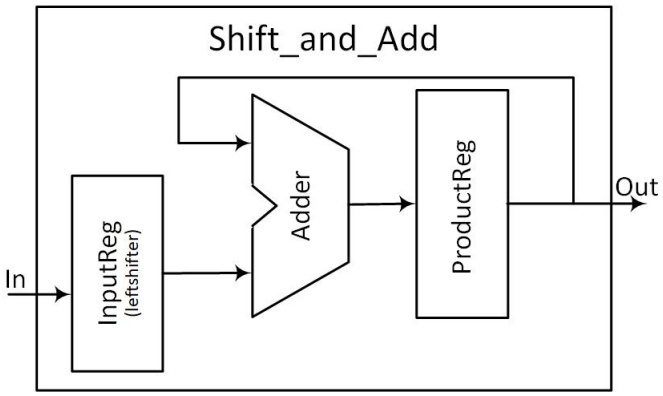


Fig. 7: Shift&Add unit design

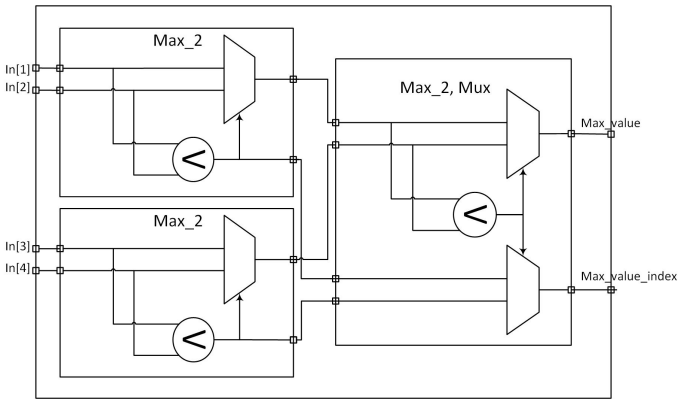


Fig. 8: Max pooling unit design

the same power budget and the same area. It is unnecessary to allocate as large a buffer space as Pipelayer because A^3 only performs the error propagation (EP) process. Reducing memory brings power and area reduction at the same time. The power reduction or area reduction makes room for crossbars. To further increase the throughput, A^3 uses a single crossbar to store weight values, nearly doubling the utilization of crossbars. Furthermore, we redesign the Shift&Add units and Max pooling units to support the modification. More details will be shown in the following sections.

A. *Trading off storage with compute engines:* $\boldsymbol{A^3p}$ *and* $\boldsymbol{A^3r}$

As illustrated in Fig.2, CNN needs to compute the gradient of weights before updating the corresponding weight matrix. The neurons of feature maps on layer $l-1$ are involved when the system computes the gradient of weights of layer l, so the feature maps of each layer should be stored in the buffer for gradient computing. In pipeline cases, buffer requirement increases with the network depth. In the training process of AttackNet, weights no longer need to be updated, so neurons of feature maps of a layer can be dropped once the next layer gets its neurons. The reason lies in that the system does not compute the gradient of weights anymore. The buffer requirement of AttackNet relies on two types of data, derivatives of the ReLU function with respect to feature maps and errors of each layer. ReLU is $max(x, 0)$ whose derivative is a 0-1 matrix, which means the system is capable of storing elements of a matrix using a bit map rather than using a floating matrix. In this way, the required on-chip buffer can be significantly reduced.

In this paper, we will use Pipelayer as the baseline because it is the only processing-in-memory (PIM) platform supporting CNN training. The power and area of the baseline are summarized in Table I. According to the observation shown in Fig.1, it is unnecessary to equip A^3 with large buffers. Therefore, we can trade off the buffer space with more crossbars for faster computations. Assuming that the buffer being taken away accounts for $\boldsymbol{P}$ and $\boldsymbol{A}$ for power and area respectively, we can make use of the same power or area to add more crossbars separately. We denote the power of a crossbar and associated units as $\boldsymbol{p}$, and the area of a crossbar and associated units as $\boldsymbol{a}$.

Under different constraints, we can have two design options. The first design $\boldsymbol{A^3p}$ is to increase the number of crossbars and associated units by $\boldsymbol{P/p}$, leading to the same power budget with the baseline. The second design $\boldsymbol{A^3r}$ increases the number of crossbars and associated units by $\boldsymbol{A/a}$, thus resulting in the same area space to the baseline. As we will see in the methodology section, $\boldsymbol{A^3r}$ will incur power overhead, but the performance efficiency of it is slightly higher than that of baseline for some benchmarks.

B. *Improving crossbar utilization:* $\boldsymbol{A^3px}$ *and* $\boldsymbol{A^3rx}$

Traditionally, dual-crossbar is used to store weight matrices [11] [12] [14]. In this manner, the difference between the two crossbars equals the actual weight values. The drawback of this method is that only half of crossbars on-chip are used to perform dot-production.

As illustrated in Fig.6, the crossbar of A^3 adds a column of constant-term circuits $1/R_F$ on the left of the custom crossbar. The memristor's conductance at the crossing point of the *i*th

row and *j*th column is $g_{i,j}$. $V_{in,m}$ is the voltage applied to the *m*th row. According to Kirchhoff's current law, the current along the *k*th bit line is

$$V_{out,k} = -\left[\sum_{s=1}^{m}(R_1 \cdot g_{s,k} - \frac{R_1}{R_F})V_{in,s}\right]$$

This value can be directly fed to the analog-to-digital unit instead of going through a subtractor. The overhead incurred by the constant-term $1/R_F$ and the OP amp is negligible [19]. With this minor modification, we can store weights using a single crossbar instead of two crossbars and improve resource utilization significantly.

The single crossbar optimization can be incorporated with the proposed $\boldsymbol{A^3p}$ and $\boldsymbol{A^3r}$ designs to further improve the crossbar utilization. We refer the combined techniques as $\boldsymbol{A^3px}$ and $\boldsymbol{A^3rx}$ respectively.

C. Shift&Add Unit and Max pooling Unit Design

The multiplication of two binary numbers comes down to calculating partial products. The Shift&Add unit is shown in Fig.7. The most significant 4 bits enter the unit, are shifted left, then being accumulated with the least significant 4 bits. The output of the adder is the product of the original two binary numbers.

The max pooling unit design is presented in Fig.8. Overall, the unit is implemented in a tree-like manner with depth equaling two. In the first stage, it takes in four inputs and compares the two pair of numbers in parallel. The outputs of the first stage not only include the comparison result of each comparator but also cover the input indices of the comparator results. Both the indices are fed to a multiplexer whose output is determined by another comparator in the second stage. Finally, the max pooling unit outputs the maximum value of the four inputs, and the index corresponds to that maximum value.

Differences with PipeLayer: $\boldsymbol{A^3}$ explores buffer optimization due to the uniqueness of AttackNet algorithms, and redesigns functional units to accommodate weight storage in single crossbars. All optimization techniques enable $\boldsymbol{A^3}$ to outperform Pipelayer in throughput and energy efficiency compared to deploying AttackNets on PipeLayer.

IV. Pipeline Analysis

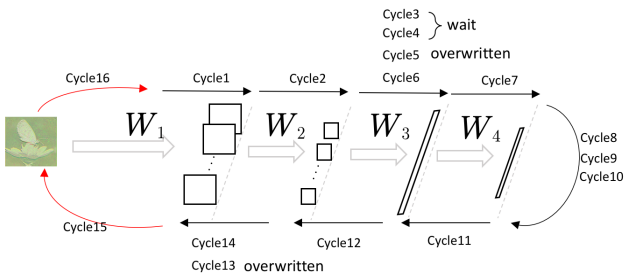


Fig. 9: An AttackNet training example: two convolutional layers and two fully connected layers. The batch size is 3.

In this section, we take a network of four layers as an example to illustrate how crossbar counts affect performance. The AttackNet example in Fig.9 is composed of two convolutional layers (L_1,L_2) and two fully connected layers (L_3,L_4).The weight matrix of each layer is W_1, W_2, W_4 and W_4. Each layer accounts one cycle when its weight matrix is available. Otherwise, it should wait for its weight matrix, which can overwrite other weight matrices if and only if they have been used by the last image in the same batch.

Assuming a memristor crossbar based CNN training platform is equipped with C crossbars, we can only pre-program W_1 and W_2 into these C crossbars. In this case, Fig.10 shows the pipeline details when batch size equals three. Fig.10(b) focuses on image1 of the first batch. The system processes L_1 (corresponding to stage1) in the 1st cycle, followed by stage2 that proceeds the computation of L_2 in the 2nd cycle. The system should wait for W_1 and W_2 being used by image3, meanwhile stall image1 in the following two cycles and image2 in the 4th cycle, then is capable of programming W_3 and W_4 into crossbars in the 5th cycle. After that, the system performs the computation of the two fully connected layers, L_3 and L_4, in the 6th cycle and the 7th cycle respectively. In the 8th cycle and the 9th cycle, the system computes the error of L_4 for image1 and image2 respectively. During the 10th cycle, the system computes the error of L_4 for image3 and accumulate errors for all images in this batch. Note that the error computation of L_4 needs target labels instead of a weight matrix. The system can process L_3 and L_2 in the following two cycles (the 11th and 12th cycle) because W_4 and W_3 are already in crossbars. After the 12th cycle, the system needs to wait for an extra cycle to program W_2 and W_1 into crossbars. Finally, the system computes the error of $input$ in the 15th cycle, then updates the mask of inputs in the next cycle. All the above processes are illustrated in Fig.10(b) and summarized in Fig.11.

As crossbar counts increase, $\boldsymbol{A^3}$ significantly decreases the overwritten times, thus reducing the processing time of a batch. The Fig.10(c) shows the pipeline when $\boldsymbol{A^3}$ can store the whole network parameters (the weight matrix of all layers). More crossbars delete stall cycles, thus reducing the processing time of one batch of inputs. Further increasing crossbars enables more than one set of the network parameters to be programmed into crossbars. Fig.10(d) shows the pipeline when $\boldsymbol{A^3}$ can store three copies of the whole network parameters. As we can see, the pipeline of multiple input sets is similar to that of the super-scalar processor.

V. Methodology

Power and Area Model: In this work, we use CACTI 7.0 [20] at 32 nm to model the power and area of the SRAM buffer (input/output registers). The consumption of power and area of eDRAM are measured using Destiny [21] at 32 nm. The area and energy for memristor-based crossbar are adapted from [22] [23]. The area and power of DAC and ADC are modeled from the analysis [24] and ISAAC [13]. For simplicity, we use a 1-bit DAC. The power and area of the Shift&Add unit, the Max

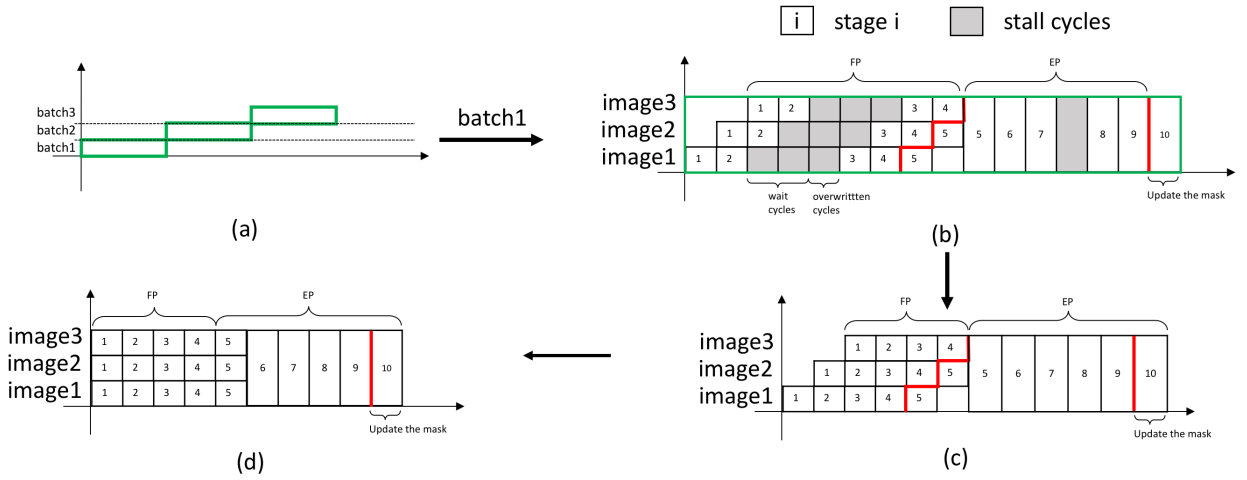


Fig. 10: Pipeline analysis for the network example.

Cycle1	Stage1	Cycle9	stage5 (error for image2)
Cycle2	Stage2	Cycle10	stage5 (error for image3), Calculating average errors
Cycle3,4	Waiting for image3 to finish stage 2	Cycle11	Stage6
Cycle5	Programing W_3,W_4 into crossbar	Cycle12	stage7
Cycle6	stage3	Cycle13	Programing W_2,W_1 into crossbar
Cycle7	Stage4	Cycle14	Stage8
cycle8	stage5 (error for image1)	Cycle15	Stage9

Fig. 11: Pipeline stages for the AttackNet training example.

Benchmarks	Input Size	#ConvLayers	#Pooling	#FC	Batch size
lisa1	1*32*32	4	0	1	3
lisa2	1*32*32	6	0	2	3
gtsrb1	1*32*32	7	3	4	5
gtsrb2	1*32*32	12	6	2	5
Inception V3-1	3*299*299	95	13	1	30
Inception V3-2	3*299*299	43	7	1	30
madry_mnist1	3*28*28	2	2	2	4
madry_mnist2	3*256*256	5	5	2	40

Fig. 12: Benchmarks and the configurations.

Baseline (Pipelayer)						
	Size	Power(w)	Area(mm^2)	#count	Total Power	Total Area
eDRAM buffer	32MB	4.49	16.364	1	4.49	16.364
Output Register	128KB	0.04	0.175	1	0.037	0.175
Input Register	128KB	0.037	0.1752	1	0.037	0.175
crossbar	128*128	0.0003	0.000025	16128	4.84	0.4
DAC	1*128	0.0005	0.00002125	16128	8.064	0.34272
ADC	8bits	0.002	0.0012	16128	32.256	19.3536
SUM					49.72	36.814

TABLE I: Baseline (Pipelayer) Configuration

A^3p						
	Size	Power(w)	Area(mm^2)	#count	Total Power	Total Area
eDRAM buffer	2MB	1.36	2.45	1	1.36	2.45
Output Register	16KB	0.01	0.015	1	0.01	0.01
Input Register	16KB	0.01	0.015	1	0.01	0.01
crossbar	128*128	0.0003	0.000025	17265	5.1795	0.43
DAC	1*128	0.0005	0.00002125	17265	8.63	0.3669
ADC	8bits	0.002	0.0012	17265	34.53	20.718
SUM					49.719	23.992

TABLE II: A^3p Configuration

A^3r						
	Size	Power(w)	Area(mm^2)	#count	Total Power	Total Area
eDRAM buffer	2MB	1.36	2.45	1	1.36	2.45
Output Register	16KB	0.01	0.015	1	0.01	0.01
Input Register	16KB	0.01	0.015	1	0.01	0.01
crossbar	128*128	0.0003	0.000025	27553	8.27	0.689
DAC	1*128	0.0005	0.00002125	27553	13.78	0.586
ADC	8bits	0.002	0.0012	27553	55.11	33.064
SUM					78.53	36.813

TABLE III: A^3r Configuration

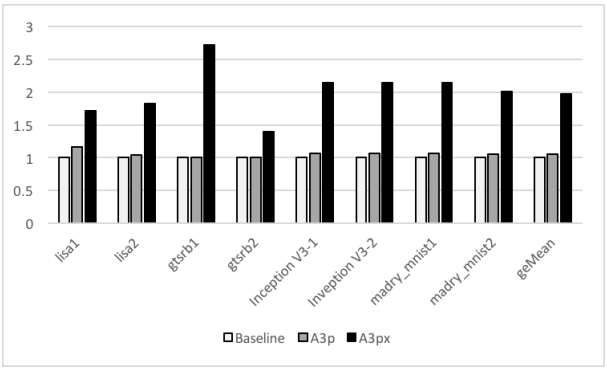
pooling unit are negligible when compared with other units. We developed the baseline accelerator following Pipelayer as described in [14] with configurations listed in table I. A^3p and A^3r are shown in table II and table III respectively.

Performance Model: To simulate the process, we developed an in-house simulator to model the training process of adversarial attack algorithms, and to estimate the throughput performance. The cycle time in A^3 is 50.88ns, which is consistent with Pipelayer [14]. The metrics we use are power efficiency (PE, the number of 16-bit operations performed per watt, $GOPs/W$) and computational efficiency (CE, the number of 16-bit operations performed per second per mm^2, $GOPs/T \times mm^2$), both of which are defined in ISAAC [13].

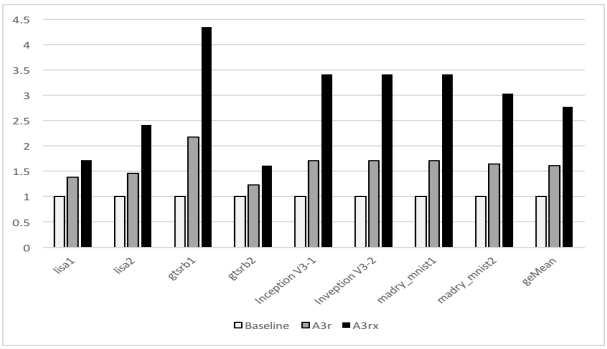
Benchmarks: The benchmarks we used in this paper are selected from from [3] [25]. The benchmark configurations are listed in Fig.12. lisa1, gtsrb1, Inception v3-1 and madry_mnist1 are the same as that in the original literature. lisa2, gtsrb2, Inception v3-2 and madry_mnist2 are modified from lisa1, gtsrb1, Inception v3-1 and madry_mnist1 to make the test networks' architecture more diverse.

VI. Experiment Results

According to the previous discussion, $\boldsymbol{A^3p}$ and $\boldsymbol{A^3r}$ have the same power and the same area to the baseline respectively. To further increase the throughput, we apply single crossbar storage to both designs. All the performance results are shown in Fig.13. From Fig.13(a), reducing the number of buffers



(a) Performance improvement on the same power budget



(b) Performance improvement on the same area budget

Fig. 13: Speedup results over the baseline. All results are normalized to the baseline.

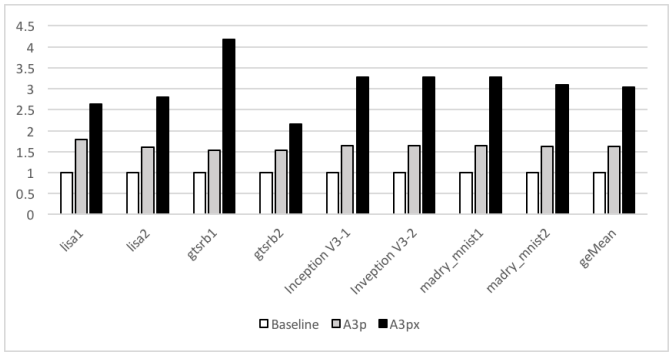


Fig. 14: Computational efficiency improvement on A^3p

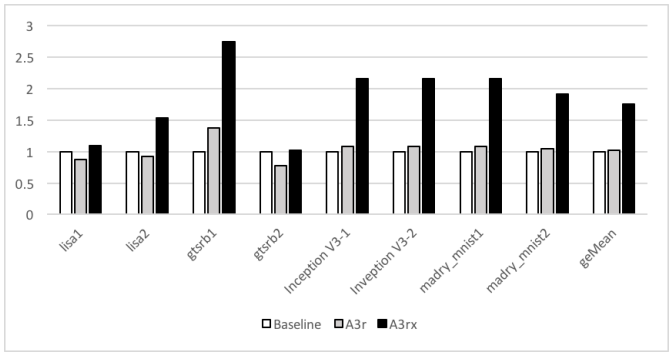


Fig. 15: Power efficiency improvement on A^3r

to make room for crossbars can bring a little performance improvement under the same power budget due to the limited power saving brought by buffer reduction. In contrast, single crossbar storage significantly increases performance. The results of keeping the same area budget are presented in Fig.13(b). The speedup of all benchmarks on $\boldsymbol{A^3r}$ is larger than that of $\boldsymbol{A^3p}$. The area reduction brought by buffer reduction enables more crossbars to be added to $\boldsymbol{A^3}$. As can be seen in table II and table III, $\boldsymbol{A^3r}$ owns more crossbar than $\boldsymbol{A^3p}$.

Since the power of $\boldsymbol{A^3p}$ is the same as that of the baseline, we only compare the computational efficiency results in Fig.14. Compared with 13(a), CE speedups of all benchmarks on $\boldsymbol{A^3p}$ are more obvious. The reason can be found in table I and II. Keeping the same power budget, the area of $\boldsymbol{A^3p}$ is smaller than that of the baseline.

We only compare the power efficiency of all benchmarks on $\boldsymbol{A^3r}$ in Fig.15 because the area of $\boldsymbol{A^3r}$ is the same as that of the baseline. Though increasing crossbars in $\boldsymbol{A^3r}$ incurred power overheads, the geometric mean of the PE is better than that of the baseline. However, some benchmarks' PE on $\boldsymbol{A^r}$ are slightly worse than that on the baseline.

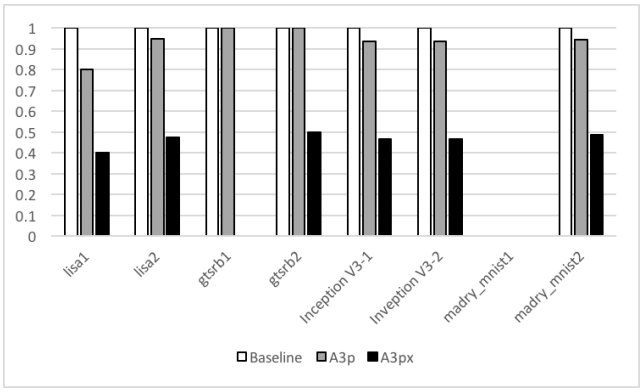
Fig.16 shows the results of weight overwritten times of all benchmarks. The overwritten times of $\boldsymbol{A^3p}$, $\boldsymbol{A^3px}$, $\boldsymbol{A^3r}$ and $\boldsymbol{A^3rx}$ are normalized to that of the baseline. Relatively, $\boldsymbol{A^3px}$ and $\boldsymbol{A^3r}$ reduce the numbers of weight overwritten much more than $\boldsymbol{A^3p}$ and $\boldsymbol{A^3r}$ do. Madry_minst1 does not have any weight overwritten in all cases. Gtsrb1 does not need overwrite weights on $\boldsymbol{A^3r}$, $\boldsymbol{A^3px}$, and $\boldsymbol{A^3rx}$. Overall, we can observe that reducing overwritten times can bring performance benefits for all benchmarks. This figure is largely inverse to the speedups shown in Fig. 13.

Fig.17 provides the breakdown of the energy consumption by different components. The total energy consumption is divided into five parts, ADC energy, DAC energy, Buffer energy, crossbar energy and others. From Fig.17, we can see clearly the proportion of ADC consumption and DAC consumption in $\boldsymbol{A^3px}$ and $\boldsymbol{A^3rx}$ are larger than that in $\boldsymbol{A^3}$. The reason is that both $\boldsymbol{A^3px}$ and $\boldsymbol{A^3rx}$ own more crossbar than $\boldsymbol{A^3}$. The number of ADC and DAC are in proportional to crossbar numbers.

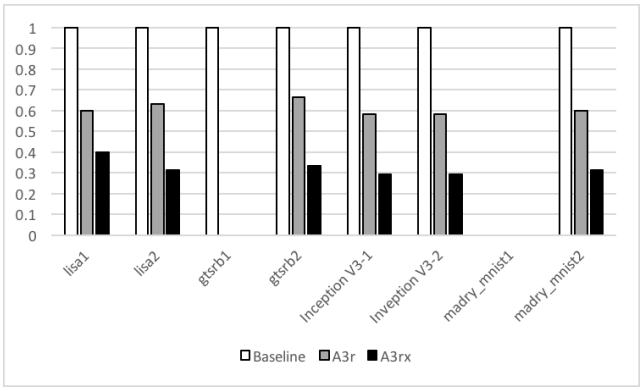
Accuracy and convergence discussion: AttackNet aim to misclassifying inputs, thus generating a mask for training a robust deep learning network, so they are less sensitive to accuracy than CNN training. The accuracy of AttackNet should take convergence into account, which means results should be evaluated by pairing the iteration with the corresponding misclassifying rate. The bit-widths of neurons and weights in this work are constant with ISAAC, and have no effects on AttackNet according to our observations.

VII. Conclusion

Adversarial Attacks to CNNs are an emerging topic in the deep neural networks area. In this paper, we propose the



(a) The percentage of overwritten on the same power budget



(b) The percentage of overwritten on the same area budget

Fig. 16: Weight overwritten proportions vary with optimization. All the results are normalized to the baseline.

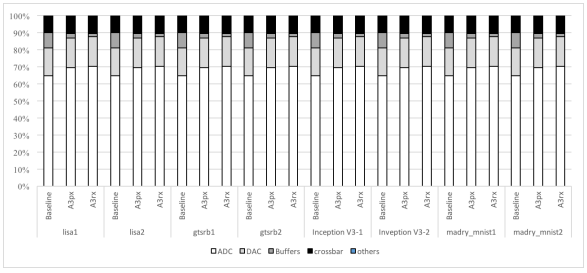


Fig. 17: Energy Breakdown

first hardware accelerator for adversarial attacks based on memristor crossbar arrays. Compared to conventional CNN training architectures, the proposed design significantly improves performance and energy efficiency.

REFERENCES

[1] LeCun, Yann, Yoshua Bengio, and Geoffrey Hinton. "Deep learning." nature 521.7553 (2015): 436.

[2] Akhtar, Naveed, and Ajmal Mian. "Threat of adversarial attacks on deep learning in computer vision: A survey." IEEE Access 6 (2018): 14410-14430.

[3] Eykholt, Kevin, et al. "Robust physical-world attacks on deep learning visual classification." Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition. 2018.

[4] Xie, Cihang, et al. "Adversarial examples for semantic segmentation and object detection." Proceedings of the IEEE International Conference on Computer Vision. 2017.

[5] Norton, Andrew P., and Yanjun Qi. "Adversarial-Playground: A visualization suite showing how adversarial examples fool deep learning." 2017 IEEE Symposium on Visualization for Cyber Security (VizSec). IEEE, 2017.

[6] Shen, Sijie, et al. "Fooling neural networks in face attractiveness evaluation: Adversarial examples with high attractiveness score but low subjective score." 2017 IEEE Third International Conference on Multimedia Big Data (BigMM). IEEE, 2017.

[7] Krotov, Dmitry, and John J. Hopfield. "Dense associative memory for pattern recognition." Advances in neural information processing systems. 2016.

[8] Abbasi, Mahdieh, and Christian Gagn. "Robustness to adversarial examples through an ensemble of specialists." arXiv preprint arXiv:1702.06856 (2017).

[9] Cisse, Moustapha, et al. "Parseval networks: Improving robustness to adversarial examples." Proceedings of the 34th International Conference on Machine Learning-Volume 70. JMLR. org, 2017.

[10] Akhtar, Naveed, Jian Liu, and Ajmal Mian. "Defense against universal adversarial perturbations." Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition. 2018.

[11] Hao Yan, Hebin R. Cherian, Ethan C. Ahn, and Lide Duan. 2018. CELIA: A Device and Architecture Co-Design Framework for STT-MRAM-Based Deep Learning Acceleration. In Proceedings of the 2018 International Conference on Supercomputing (ICS '18). ACM, New York, NY, USA, 149-159.

[12] Ping Chi, et al. 2016. PRIME: a novel processing-in-memory architecture for neural network computation in ReRAM-based main memory. In Proceedings of the 43rd International Symposium on Computer Architecture (ISCA '16). IEEE Press, Piscataway, NJ, USA, 27-39.

[13] Ali Shafiee, et al. 2016. ISAAC: a convolutional neural network accelerator with in-situ analog arithmetic in crossbars. In Proceedings of the 43rd International Symposium on Computer Architecture (ISCA '16). IEEE Press, Piscataway, NJ, USA, 14-26.

[14] Song, Linghao, et al. "Pipelayer: A pipelined reram-based accelerator for deep learning." 2017 IEEE International Symposium on High Performance Computer Architecture (HPCA). IEEE, 2017.

[15] Tianshi Chen, et al. 2014. DianNao: a small-footprint high-throughput accelerator for ubiquitous machine-learning. SIGPLAN Not. 49, 4 (February 2014), 269-284. DOI: https://doi.org/10.1145/2644865.2541967

[16] Chen, Yunji, et al. "Dadiannao: A machine-learning supercomputer." Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture. IEEE Computer Society, 2014.

[17] Guo, Haoqiang et al. (2019). Fooling AI with AI: An Accelerator for Adversarial Attacks on Deep Learning Visual Classification. 136-136. ASAP 2019.

[18] Chetlur, Sharan, et al. "cudnn: Efficient primitives for deep learning." arXiv preprint arXiv:1410.0759 (2014).

[19] Truong, Son Ngoc, and Kyeong-Sik Min. "New memristor-based crossbar array architecture with 50-% area reduction and 48-% power saving for matrix-vector multiplication of analog neuromorphic computing." Journal of semiconductor technology and science 14.3 (2014): 356-363.

[20] Rajeev Balasubramonian, et al. 2017. CACTI 7: New Tools for Interconnect Exploration in Innovative Off-Chip Memories. ACM Trans. Archit. Code Optim. 14, 2, Article 14 (June 2017).

[21] Poremba, Matt, et al. "Destiny: A tool for modeling emerging 3d nvm and edram caches." Proceedings of the 2015 Design, Automation&Test in Europe Conference&Exhibition. EDA Consortium, 2015.

[22] Hu, Miao, et al. "Dot-product engine for neuromorphic computing: Programming 1T1M crossbar to accelerate matrix-vector multiplication." Proceedings of the 53rd annual design automation conference. ACM, 2016.

[23] Zangeneh, Mahmoud, and Ajay Joshi. "Design and optimization of nonvolatile multibit 1T1R resistive RAM." IEEE Transactions on Very Large Scale Integration (VLSI) Systems 22.8 (2014): 1815-1828.

[24] B. Murmann, "ADC Performance Survey 1997-2018," [Online]. Available: http://web.stanford.edu/ murmann/adcsurvey.html.

[25] https://www.tensorflow.org/guide/performance/benchmarks