AEP: An Error-bearing Neural Network Accelerator for Energy Efficiency and Model Protection

Lei Zhao*, Youtao Zhang†, Jun Yang†
*†Department of Computer Science, University of Pittsburgh
‡Department of Electrical and Computer Engineering, University of Pittsburgh
Email: *lez21@pitt.edu, †zhangyt@cs.pitt.edu, ‡juy9@pitt.edu

***Abstract*—Neural Networks (NNs) have recently gained popularity in a wide range of modern application domains due to its superior inference accuracy. With growing problem size and complexity, modern NNs, e.g., CNNs (Convolutional NNs) and DNNs (Deep NNs), contain a large number of weights, which require tremendous efforts not only to prepare representative training datasets but also to train the network. There is an increasing demand to protect the NN weight matrices, an emerging Intellectual Property (IP) in NN field. Unfortunately, adopting conventional encryption method faces significant performance and energy consumption overheads.**

In this paper, we propose AEP, a DianNao based NN accelerator design for IP protection. AEP aggressively reduces DRAM timing to generate a device dependent error mask, i.e., a set of erroneous cells while the distribution of these cells are device dependent due to process variations. AEP incorporates the error mask in the NN training process so that the trained weights are device dependent, which effectively defects IP piracy as exporting the weights to other devices cannot produce satisfactory inference accuracy. In addition, AEP speeds up NN inference and achieves significant energy reduction due to the fact that main memory dominates the energy consumption in DianNao accelerator. Our evaluation results show that by injecting 0.1% to 5% memory errors, AEP has negligible inference accuracy loss on the target device while exhibiting unacceptable accuracy degradation on other devices. In addition, AEP achieves an average of 72% performance improvement and 44% energy reduction over the DianNao baseline.

I. Introduction

Neural Networks (NNs) have recently gained popularity in a wide range of modern application domains, e.g., computer vision [12] and speech recognition [11]. NNs are machine learning (ML) approaches that, by exploiting the significant increase of computing power of modern computers, achieve superior inference accuracy improvements over traditional ML approaches. A typical NN contains a large number of layers and their weight matrices, making it difficult to train and infer twenty years ago. For example, the VGG-16 DNN is a 16-layer image classification NN that has 138M parameters. Even using four Nvidia Titan black GPUs, it still takes around two to three weeks to train [27].

The intrinsic computation and memory intensity of NNs has driven the development of hardware accelerators for high performance and energy efficiency. For example, the DianNao accelerator [3] achieves 118× performance speedup and 21× energy reduction over an SIMD accelerator. Our paper is built on top of the DianNao accelerator. Memory accesses remain a major bottleneck for hardware NN accelerators, as shown in [3].

With ML problems growing in both complexity and size, it becomes increasingly challenging to design effective NNs that achieve high inference accuracy, e.g., ImageNet has yearly competition for large scale image recognition [12]. An NN is composed of its layer structure as well as the corresponding weight matrices in each layer. While the former is either well known or hinted by the weight matrices, the latter is more problem dependent — it requires not only the efforts to prepare effective training datasets [9], sometimes including sensitive and/or proprietary data, but also the efforts to train the weights. Leaking the model may result in huge losses, e.g., the object tracking and recognition NN of an autonomous driving car may be pirated, which speeds up the development of new systems from the competitors. Therefore, there is an increasing demand to protect NN models, in particular, their weight matrices.

For the NNs that are deployed at the client side, it is challenging to design effective protection solutions. A simple solution is to adopt XOM approach [20] to store encrypted weights and decrypt them before use. Unfortunately, due to extremely high memory bandwidth demand in NN accelerators, e.g., 250GB/s in DianNao accelerator, an encryption based approach not only requires the integration of expensive encryption engines but also introduces significant encryption and decryption latency and energy consumption overheads.

In this paper, we develop AEP, a novel error-bearing NN training and inference approach that effectively protects the NN model with low overheads. The following summarizes our contributions.

- We propose to enable effective IP protection through device dependent weight matrices. Adopting such weights achieves high inference accuracy only on the target device. For this purpose, we integrate a device dependent error mask in the NN training phase and utilize batch normalization to effectively mitigate the impact of errors during training. To the best of our knowledge, we are the first to integrate IP protection with weight training process.
- We propose to aggressively reduce DRAM timing to generate a device dependent error mask, i.e., we get a set of erroneous cells while the distribution of these cells are device dependent due to process variations. We propose a table driven tRAS adjustment mechanism to defeat probing attacks.
- We extensively evaluate the effectiveness of the AEP design. Our results showed that, by injecting 0.1% to 5% memory errors, AEP has negligible inference accuracy loss on the target devices but exhibiting unacceptable degradation on other devices. In addition, by aggressively reducing tRAS timing, AEP achieves, on average, 72% performance improvement and 44% energy reduction over the DianNao baseline.

In the rest of the paper, we briefly discuss the background in Section II. We then motivate the AEP design in Section III and elaborate the details in Section IV. We present the experiment methodology in Section V and discuss the results in Sec-

tion VI. Additional related work is discussed in Section VII. We conclude the paper in Section VIII.

II. BACKGROUND

In this section, we discuss the background to facilitate our design. We first discuss neural networks (NNs) and the approximation in NN training. We then briefly introduce the DianNao NN accelerator that our design is based on.

A. Neural Networks

A Neural Network (NN) is a layered structure containing multiple layers. Each layer consists of multiple neurons while each neuron computes

$$y = WX + b$$

where X is an input vector to the neuron; y is the a scalar output value from the neuron. W is the weight matrix, and b is a scalar value called bias. The outputs of one layer's neurons are the inputs of the following layer (except the last layer whose output is the NN output). The outputs of one layer are also referred to as its feature map. A feature map is a three dimensional structure: $W \times H \times D$ where W and H are its width and height, and D is the number of the feature maps (depth).

Convolutional layers. A convolutional layer is composed of one or several three dimensional filters. The width and height dimensions of the filter (K_x, K_y) are usually the same and are on the order of 10s. The filter's depth dimension (K_z) equals to that of the input feature maps (D_{in}). Each filter slides over the width and height dimensions (W_{in}, H_{in}) of the input feature maps with stride S. In each step, a filter produces a single output neuron by computing the dot product with the overlapped input neurons. The produced output feature maps are of dimension $W_{out} \times H_{out} \times D_{out}$:

$$\begin{aligned} W_{out} &= (W_{in} - K_x)/S + 1, \\ H_{out} &= (H_{in} - K_y)/S + 1, \\ D_{out} &= N_k, \end{aligned}$$

where N_k is the number of filters in this layer.

In DNN, the filter in each step is private, i.e. the filter uses a different set of values for each step when it slides along the input feature maps.

Pooling layers. Pooling layers are usually used in an interleaved fashion with convolutional layers to reduce the feature map size. Similar to convolutional layers, it slides a filter along the width and height dimensions of the input with stride S. However, the filter is two dimensional ($K_x \times K_y \times 1$) so that it applies the filter on each depth of the input separately. Another difference is that there are no weights in pooling layer's filters, it only computes the max or average over the covered input neurons. The dimension of output feature maps are $W_{out} \times H_{out} \times D_{out}$:

$$\begin{aligned} W_{out} &= (W_{in} - K_x)/S + 1, \\ H_{out} &= (H_{in} - K_y)/S + 1, \\ D_{out} &= D_{in}. \end{aligned}$$

Classifier layers. Classifier layers are usually located at the end of NNs. The inputs and outputs in classifier layers are regarded as one dimensional vectors. Each output neuron is fully connected to all neurons in the input.

B. Training vs. Inference

An NN has two different computing phases: *training* and *inference*. The training phase is to adjust the weights of each layer to make the NN fit a specific function. Then the trained weights are used in the inference phase to perform tasks, such as image classification, speech recognition, etc. Figure 1a shows the training process in an example NN that classifies images. The black arrows show the forward propagation pass. The first layer takes the image as input I together with this layer's weights W_1 to calculate the output neurons N_1, which are then fed into the next layer. After the last layer computes its neurons N_4, the $Loss$ function calculates the difference between the network's output and the ideal accelerator output T. Then the loss goes back through the network layer by layer in the backward propagation pass, denoted by the blue arrows. Each layer takes the gradients of its output neurons as input and produces the neuron's gradients of previous layer. At each step, the gradients of the current layer's weights are also produced. Finally, each layer's weights are updated by applying the equation $W_i = W_i - \eta \frac{\partial Loss}{\partial W_i}$ $(1 \le i \le 4)$, where η is the learning rate.

The inference phase uses the weights trained by the training phase, and only executes the forward propagation pass to classify an input image. The output of the last layer (N_4) is the predicted label of the input image.

C. Training with Approximation

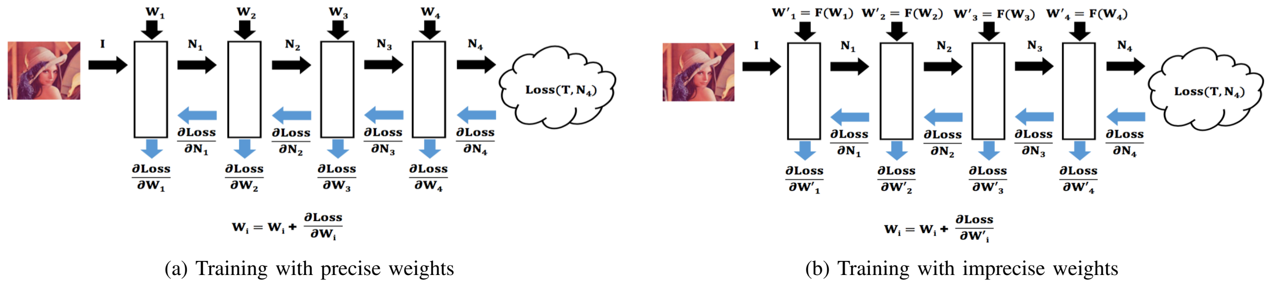
It has been proven that full precision computation (32/64-bit floating point) is not necessary in NNs [10], [14]. Most hardware accelerators use fixed point representation for weights and/or neurons [3], [4]. Chen *et al.* [4] used different fixed point widths to train and inference CNNs. They found 16-bit fixed point is sufficient in inference phase, however, in order to make the training phase converge, at least 32-bit fixed point should be used in training. Courbariaux *et al.* [7] also investigated different low precision operations in training deep NNs. They can achieve negligible accuracy loss with lower precision fixed point computations. Courbariaux *et al.* [8] then proposed to use binary weights (1 bit) in inference. The proposed training process in their work is shown in Figure 1b. They keep two sets of weights, one is the full precision floating point version W_i, the other one is the low precision fixed point weights W_i'. They first binarize the weights through function F:

$$W_i' = F(W_i) = sign(W_i), (1 \le i \le 4)$$

and use W_i' in the forward propagation pass. During the backward propagation pass, they calculate the gradients w.r.t. W_i', but only update the full precision weights W_i. In inference phase, only W_i' are used, so the floating point weights W_i can be discarded.

D. DianNao

DianNao [3] is an ASIC accelerator for speeding up NN computing. Figure 2 shows the structure of the accelerator. It is composed of one Neural Functional Unit (NFU) and three SRAM buffers, i.e. input buffer (NBin), output buffer (NBout) and synaptic buffer (SB). The SB is divided into 16 lanes. There are 64 entries in NBin, NBout and each SB lane. One entry can store 16 16-bit fixed point data. First the inputs of a layer are loaded into NBin, and the corresponding weights are loaded into SB. In each cycle, the 16 SB lanes provide 256 weights which are grouped into 16 16-weight groups to NFU. At the same time, the 16 input neurons from one NBin entry are broadcasted to each group. Every group of 16 neurons and 16 weights first multiply with each other in NFU-1 stage, then the 16 products are summed up in NFU-2 stage into one partial result and stored into NBout. If there is a partial sum of the same output neuron calculated in the previous cycle (buffered in NBout), it is also summed up in this stage. Once all the partial sums of an output neuron have been added together, it goes through the NFU-3 stage to do a nonlinearity function, e.g. sigmoid. Thus, 16 output neuron's



(a) Training with precise weights

(b) Training with imprecise weights

Fig. 1: The training process of an NN.

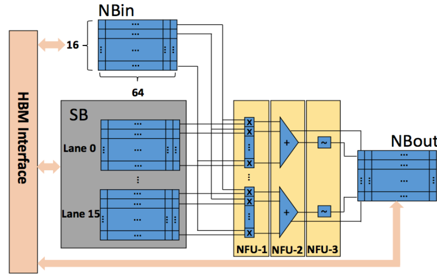


Fig. 2: The DianNao architecture.

partial sum are computed in parallel. In order to support this high parallelism, Chen *et al.* [3] uses a memory system that can provide up to 250GB/s, which is significantly higher than the capability of traditional DDR memories. In our experiments, we employ HMB [13] as the baseline memory system that provides 256GB/s bandwidth.

III. Motivation

Security model. In this paper, we focus on the IP (intellectual property) protection. The GPU-pool based server needs to train an NN model (including its layer structure and the trained weight matrices) and then deploy it to the DianNao-based accelerator at the client side. After deployment, the client has full control of device and may pirate and/or tamper with the saved weights. A simple piracy would be directly copying the weights to another device, treating the model as a black box.

The design goal of the paper is to protect the NN model, in particular, its weight matrices, at the client side.

Encryption-based protection is too expensive. A naive solution to protect NN models is to use encryption. To convoy the weights to the accelerator without being exposed to the third party during the process, we may adopt the XOM design [20] and the privacy enhancement [33]. It works briefly as follows. We first upgrade the accelerator with a small TCB (trusted computing base) that contains a crypto-engine and PKI (public key infrastructure) support, i.e., the private key is kept secret while the public key is released to the end users and servers. The server encrypts the weights using a session key and then encrypts the session key with the public key of the accelerator. Thus, only the target accelerator can decrypt the session key and then decrypt the encrypted weights before computation.

Unfortunately, this simple solution faces severe scalability issues. Given CNNs and DNNs are increasingly adopted to accomplish challenging tasks in various application domains, their sizes, i.e., the number of weights in the network, grow rapidly. For example, Le *et al.* [17] created an NN with 1 billion weight parameters, Catanzaro *et al.* [6] used an NN that has 11 billion parameters. More recently, Ni *et al.* [23] built an NN that contains over 15 billion parameters, which requires more than 30GB if using 16-bit fixed point. It consists of two convolutional layers and one classifier layer, which have 3GB, 18GB and 12GB weights, respectively.

An NN accelerator, due to its limited on-chip memory, needs to load and decrypt the large amount of weights when starting a new layer. Adopting HBM improves the maximal bandwidth but not the decryption latency. For example, without encrypting and decrypting the weights, the second layer of the above NN finishes in 504μs using DianNao. However, even we optimistically assume that the weights of the third layer (i.e., 12GB) can be loaded in parallel with the execution of the second layer, decrypting the weights with a fully-unrolled, pipelined AES implementation needs 349ms [1]. Here we optimistically assume the AES decryption engine consists of multiple copies such that it can match the peak HBM bandwidth, i.e., 256GB/s. The weight decryption can easily become the latency bottleneck of the whole system. Therefore, the encrypted based scheme is less preferred for NN model protection.

IV. Design Details

In this section, we first present an overview of the design and then elaborate the details of each component in AEP.

A. Overview

Figure 3 illustrates the overview of our AEP design. To deploy an NN to a device, e.g., a car manufacturer may need to deploy an object tracking and recognition NN to its autonomous driving car, the server, i.e., the car manufacturer in the example, extracts a device dependent memory error mask from each car that it manufactures, as ① in the figure. The server then integrates the error mask of the target device in the NN training process (using GPUs), which generates a set of device dependent weights as part of the NN model, as ② in the figure. Next, the NN model and its weights are sent to the (DianNao-based) target device. The target device can employ the NN model to conduct inference tasks and achieve desired accuracy. However, while an attacker may port the weights to another device, as ③ in the figure, no matter if the pirate device has memory errors or not, the inference accuracy is below acceptable threshold.

With the focus on IP protection in this paper, we assume the NN is deployed before the target devices are released to clients. If there is a need to upgrade the NN after release, the manufacturer needs to identify the corresponding error mask from its error mask database, and then generate new device dependent weights. There might be user privacy concerns in

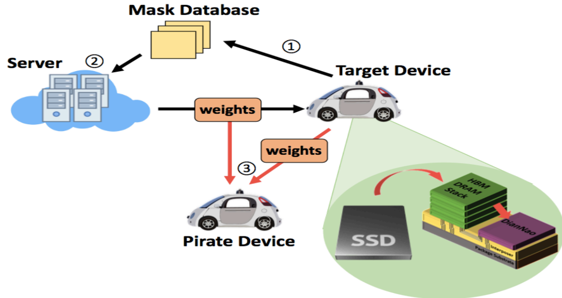


Fig. 3: An overview of AEP design.

this process, which demands PKI-based piracy enhancement [33]. We leave it to the future work.

We next elaborate the design details.

B. Device Dependent Weight Matrices

Modern NNs exhibit significant error resilience. As we discussed in Section 2.3, either replacing full precision weights with 16-bit fixed point values, or reducing refresh frequency to having a number of refresh errors [21], the NN inference accuracy is often little affected.

However, to make the AEP design possible, we need to solve a new problem that has not been studied before, i.e, ***is it possible to train device dependent NN models that only produce satisfactory inference results on target devices but not on other devices?*** To the best of our knowledge, we are the first to study this problem and our positive findings are highly valuable for real deployment.

Our baseline training process uses two sets of weights — the imprecise weights W_e are used for forward computation, and the precise weights W_f are used for weight update, which are similar to W_i' and W_i in Figure 1(b) respectively. Assuming we have a memory error mask M, a naive approach to train device dependent weights is to apply the mask when we generate the imprecise weights. That is,

$$\begin{aligned} W_e &= f(W_f) = W_f \& M \\ W_f &= W_f + \frac{\partial Loss}{\partial W_f}. \end{aligned} \quad (1)$$

W_e is also the weights to be used in inference at the client side.

We tested the effectiveness of this strategy on benchmark `CNN1` and summarized the inference accuracy during training in Figure 4. The setting details can be found in Section 5. The `Normal` line denotes the training in the baseline. After five rounds, the training inference accuracy is close to 100%. The `withoutBN` line denotes the strategy that simply apply Equation (1) with a 5% error rate mask M to W_f during training. From the figure, we observe that this simple strategy does not work — `withoutBN` cannot improve the inference accuracy above 20%.

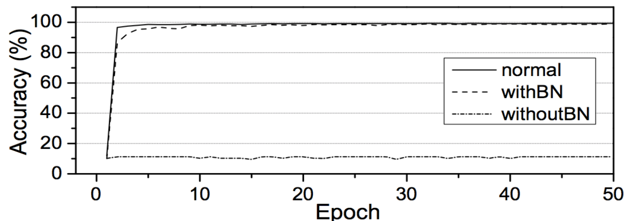


Fig. 4: Training with and without batch normalization.

We then studied the training process and found that errors bits, when appearing in the integer bits of the weights, introduce large errors that cannot be corrected. For this reason, we adopt batch normalization [8], a layer to mitigate the impact of bit errors in weights. That is

$$y = \frac{x - \mu}{\sqrt{\sigma^2 + \epsilon}} \gamma + \beta \quad (2)$$

where x is the output of previous convolutional or classifier layer and input into the batchnorm layer. By substracting the mean (μ) and dividing standard deviation (σ) from x, the input is standard normalized (with zero mean and unit deviation). γ and β are two learnable parameters like the weights in convolutional and classifier layers. ϵ is a small constant defined prior the training.

We tested this new training process which adds batch normalization to the NN. The `withBN` line in Figure 4 denotes this strategy. As we can see from the figure, the loss of inference accuracy is negligible (0.22%).

Device dependent weight matrices. Given that the error mask is deeply integrated in each round of the training process, we next check its dependency on devices.

After the training process, the server sends the weights W_f, instead of the weights W_e used in training, to the target device. This is because the target device has the error mask M, it can get W_e when loading the weights (W_f) to DRAM.

For IP protection, the weights W_f are visible to attackers. Without knowing the secrect mask M used in generating the weights, the attackers have two options to abuse the weights: one is to use W_f directly for inference on other devices; the other is to apply a different error mask M_X (the error mask of the pirate device), i.e.,

$$W_x = W_f \ \& \ M_X. \quad (3)$$

TABLE I: Inference Accuracy on Different Devices

	Target device	Pirate device	Error-free device
`CNN1`	99.16%	10.15%	21.66%

For the same benchmark `CNN1`, Table I summarizes the inference accuracy when adopting W_f on different devices. From the table, the trained weights W_f are applicable only to the target device, and produces unacceptable inference accuracies on either pirate or error-free devices.

C. Device Dependent Error Masks

The preceding section addresses the challenge of producing device dependent weights based on device dependent error masks. Next, we generate one such error mask as a proof of concept. In practice, there are many different approaches to generate device dependent error masks and signatures, e.g., those exploited in physical unclonable functions [29].

In this paper, our device dependent error masks are generated from DRAM restore errors. Given DRAM reads are destructive operations that destroy the stored values, they need to restore the values back to the cells after read. JEDEC defines the timing that is required to reliably store the value in DRAM cells — two timing parameters tRAS and tWR are directly related for restore after read and for write operations, respectively. We only adjust tRAS for proof of concept in this paper.

The tRAS timing defined in JEDEC is often very conservative. Lee *et al.* [19] and Zhang *et al.* [32] proposed to safely reduce tRAS (and tWR) to improve DRAM performance, i.e., without introducing DRAM errors. In this paper, we propose to further reduce tRAS timing to introduce DRAM errors. Due to process variations, the set of the DRAM errors and their distribution are device dependent. Figure 5 presents the

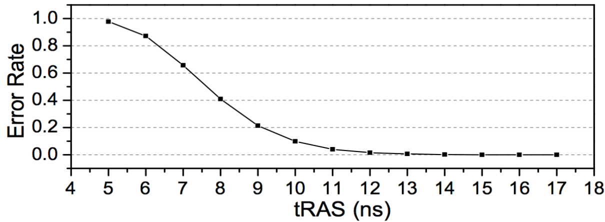


Fig. 5: Reducing tRAS aggressively introduces memory errors.

percentage of error bits for different tRAS values. we adopted the same models as reported in [31], [32].

Choosing tRAS timing is closely coupled with the error tolerance of the NN, i.e., the intrinsic layer structure and problem difficulty. As shown in Figure 6, different NNs often exhibit different error tolerance. For high performance NN applications, we often have a small bound on how much loss on inference accuracy can be tolerated. If we set the threshold to be 1%, `CNN2` and `CNN3` can tolerate 0.1% bit errors, `MLPS`, `MLPM`, and `MLPL` can tolerate up to 1% bit errors, while `CNN1` can tolerate as high as 5% bit errors. For `CNN1`, we may reduce tRAS from 35ns in the baseline to 11ns, as shown in Figure 5.

Recently, Song *et al.* [28] observed that different NNs has different accuracy requirement, and for some applications higher accuracy is not always preferable. Their observation is orthogonal to our design — while we set a fixed threshold (1%) in this paper, we will study NNs with different thresholds in our future work.

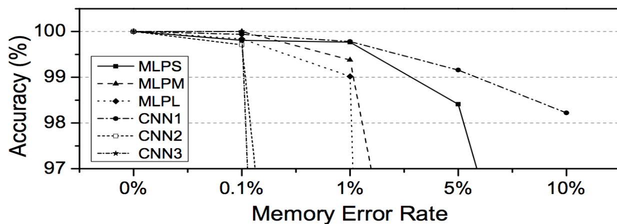


Fig. 6: Different NNs have different error tolerance.

While DRAM restore errors manifest as empty cells with no charge, a DRAM module may map cell state to 0 and 1 differently — some rows map no-charge cells as 1s while other map them as 0s. For this reason, the error mask consists of two submasks $M1$ and $M0$. They are used to identify the cells that are stuck at 1s and 0s, respectively.

The server took the post-fabrication test as shown in [19], [32] to extract the masks. This is done before deployment. Recent studies found a small number of cells show variable retention time (VRT) [24], which may manifest as errors at different times. In our study, AEP proactively introduces 0.1% to 5% errors while VRT has a very low error rate, e.g., below 10^{-6} [24]. The impact from VRT is negligible.

D. Securing The Error Mask At The Client Side

In AEP, the NN weights in AEP depend on the memory error mask and thus it becomes important to prevent attackers from extracting the mask at the client side. AEP defends the memory error mask using two approaches.

The first approach is that we disable the read path before release. The DianNao based accelerator and its DRAM module are packaged together such that only the accelerator can fetch data from DRAM to compute. A persistent attacker may probe the error mask by loading his own NNs and correlating the output to the inputs to detect possible error bits. It is much more difficult and takes longer than memory march test [19].

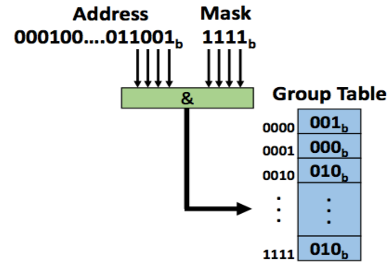


Fig. 7: Dynamic tRAS adjustment.

The second approach is to adopt a light weight error mask selection mechanism as follows. We integrate a small table in the memory controller, e.g., the table in Figure 7 has 16-entries while each entry contains 3 bits. When accessing a DRAM row, we map its row address to an entry in the table, using the XOR of the address' last 4 bits and a 4-bit mask. The 3-bit entry content, e.g., '010_b'=2 indicates how to adjust the tRAS value at runtime. For example, if tRAS is set to 13ns globally, the memory controller effectively reduces the tRAS timing for this row to 11ns if the mapped table entry contains 2; and there is no adjustment if the entry contains 0. Note that because NN has a regular memory access pattern which is predetermined, the table look up can be done before the access, thus it is not on the critical path.

The mask and the table requires 7 bytes. It is sensitive information that is encrypted by the public key of the accelerator. The accelerator decrypts the mask/table before execution. In this way, even the attacker can faithfully probe the error mask of the system, (S)He cannot determine the actual error mask that is used in training.

The mask/table mechanism not only secures the error mask but also provide a better control of the error rate in training. There are about 0.02% and 0.1% memory errors when we set tRAS to 16ns and 15ns, respectively. If an NN has error tolerance at about 0.05%, choosing either tRAS=16 or tRAS=15 may not give us the optimal configuration. Instead, the server can set the tRAS to be 16ns globally and then set the values of some entries in the above table to 0 while others to 1, which effectively set the tRAS for these rows to 16ns while others being 15ns. The mix of these lines gives an average error rate that is 0.05%.

V. Methodology

To evaluate the effectiveness of our proposed scheme, we used the Theano [30] framework to explore the error tolerance of different NNs. We tested three datasets with six NN structures. MNIST [18] is a widely used gray scale image dataset for handwritten digit recognition. SVHN [22] is a real world house number recognition dataset obtained from Google Street View images. Cifar10 [16] is a color image classification dataset containing 10 different object classes. We built 3 different CNNs on MNIST, SVHN, and Cifar10. In addition, we also built three different size multilayer perceptrons (MLP-S/M/L). Table II lists the detailed network structures.

For performance evaluation, we developed a cycle accurate simulator for the baseline DianNao and our proposed AEP design. We modified DRAMsim2 [25] to simulate HBM module according to [13]. The parameters of our simulator are listed in Table III.

In our evaluation, we compared three schemes.

- `DianNao` indicates the baseline accelerator proposed in [3]. It has no IP protection.

TABLE II: Datasets and Networks

Benchmark	Dataset	Neural Network
MLPS	MNIST	240-240-10
MLPM	MNIST	784-500-250-10
MLPL	MNIST	784-1500-1000-500-10
CNN1	MNIST	con5x20-pool2-conv5x50-pool2-500-10
CNN2	SVHN	conv5x32-pool3-conv4x64-pool3-1000-400-10
CNN3	Cifar10	conv4x32-pool3-conv4x32-pool3-conv3x64-conv3x64-conv3x64-pool3-500-250-10

TABLE III: Simulator Setup

DianNao	
NFU core	32nm 606MHz
NBin & NBout	2KB SRAM
SB	32KB SRAM
Data Format	16 bit fixed point (1 interger bit)
HBM	
Size	32GB 256GB/s
Timing	tRCD=12ns tWR=12ns tRP=12ns tRAS = { 35ns Baseline; 24ns AL; 11ns/13ns/15ns AEP

- AL indicates the scheme that reduces tRAS for performance improvement [19]. AL is built on top of DianNao.
- AEP indicates the design proposed in this paper. While both AEP and AL reduce tRAS, we compare them to study, in addition to model protection, what additional performance and energy benefits we may gain from aggressive reduction of tRAS.

VI. Results

A. IP Protection Effectiveness

We first evaluated the effectiveness of AEP in IP protection. We trained the NNs using the device dependent error mask from one device and tested the trained weight matrices on error-free device as well as a device with different error mask. Table IV summarizes our results from different benchmarks. The results are normalized to the case that we train the weights for error-free devices and perform the inference on error-free devices.

TABLE IV: Inference Accuracy with Different Error Masks

	Target Device	Pirate Device	Error-Free Device
MLPS	99.77%	39.35%	81.5%
MLPM	99.38%	28.31%	71.43%
MLPL	99.02%	10.41%	24.53%
CNN1	99.16%	10.15%	21.66%
CNN2	99.71%	89.94%	97.6%
CNN3	100%	60.95%	97.77%

From the table, we observed that the trained weights exhibit strong device dependency — while having negligible accuracy loss on the target device, they show unacceptable accuracy degradation on other devices.

B. Performance

In addition to IP protection, AEP improves inference performance due to shortened memory access latency. Figure 8 compares the execution time of AL and AEP with the results normalized to DianNao. From the figure, AEP achieves, on average, 72% and 32% performance improvements over DianNao and AL, respectively. The large improvement comes mainly from tRAS reduction. Given all NN benchmark programs exhibit extreme memory access intensity, reducing tRAS timing is very effective in improve inference performance.

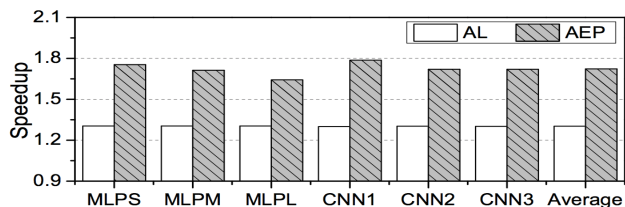


Fig. 8: The performance comparison.

C. Energy Consumption

DRAM based memory is notorious for its refresh operations, which affect both performance and energy consumption. It is reported that refresh takes 20% energy of the entire memory system[2]. However, energy distribution is different in neural network applications. Neural networks are highly parallelizable, since the majority operations in neural networks are matrix multiplications. So the memory is more frequently read and written at all times. By reducing tRAS time in read operations, both AL and AEP can improve energy consumption significantly. However, because AEP uitlizes the intrinsic high error tolerance in NNs, tRAS can be further reduced to 11ns/13ns/15ns for different NN structures (compared to 24ns in AL). As a result, AEP's energy consumption is only 56% of the baseline, while AL's energy consumption is 76% of the baseline.

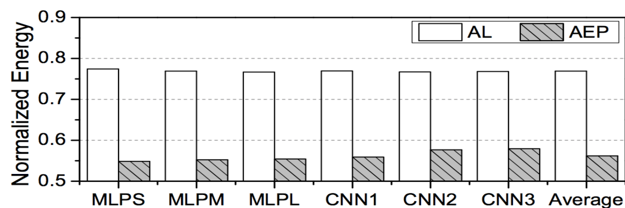


Fig. 9: The energy consumption comparison.

D. Training Overhead

A big concern regarding training device dependent weight matrices is its training overhead. Figure 10 compares different training approaches. Normal indicates the traditional error-free training process. Scratch indicates the error-bearing training process, i.e., we integrate a device dependent mask from the beginning. Continue indicates the optimized training process that we integrate device dependent mask after Normal.

From the figure, Scratch and Normal have comparable training speed — Scratch is slightly worse. However, Continue has much faster training speed than the other two. It converges in 24 epochs while the other two converge in 57 and 82 epochs, respectively.

The reason that we developed Continue is that, when deploying an NN to multiple devices, we do not have to train the NN from scratch for each error mask. Adopting Continue can significantly reduce the training overhead at the server side.

E. AEP Robustness

The last experiment that we performed is to study the robustness of the design. In this paper, the error mask depends on the memory errors with different tRAS timing parameters. Due to process variations, the number of erroneous cells and their distribution are device dependent. However, due to VRT and chip voltage fluctuation at runtime, the real mask might show a small deviation from the profiled mask.

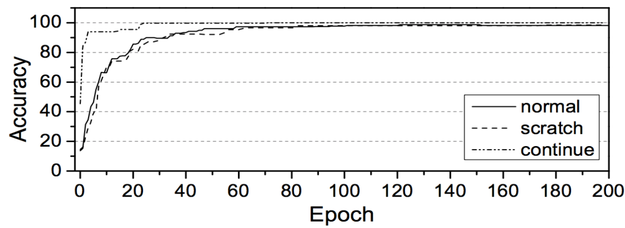


Fig. 10: The server side training overhead.

Figure 11 studies the inference accuracy degradation due to the difference between real mask and profiled mask on the target device. X-axis indicates the percentage of difference; y-axis shows the accuracy normalized to the NNs that are trained on error-free weights.

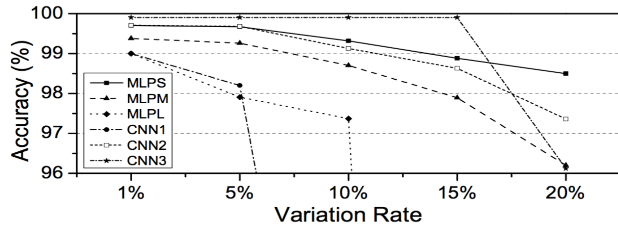


Fig. 11: The AEP robustness.

From the figure, different NNs have different tolerance levels on error mask deviation. For example, CNN3 can tolerate up to 15% error mask deviation while MLPL shows significant degradation with 5% difference. Recent study shows that the runtime deviation is often small, comparing to the number of proactively introduced errors by tRAS reduction. For example, VRT error rates are in the order of 10^{-6} [24], which have negligible impact on inference accuracy.

VII. Related Work

Recently many NN accelerators were developed to speed up NN inference and training phases. The Eyeriss [4] achitecture focuses on minimizing data movement in CNNs to improve energy efficiency. Neurocube [15] builds an NN accelerator in HMC to achieve in-memory neuromorphic computing. By exploiting ReRAM's intrinsic ability to do MAC operations, PRIME [5] and ISAAC [26] achieves significant throughput improvements. At the client side, NN accelerators are deployed mainly for achieving high inference performance. However, none of the existing accelerators take the IP protection of weights into account in their design.

Recent studies in ML reserach investigate the designs to protect dataset during training phase. For example, Dowlin *et al.* [9] proposed CryptoNets to use encrypted dataset during training. By feeding into the NN with cypher text directly, CryptoNets can improve throughout largely and still achieve a high accuracy. AEP different from CryptoNets in that we focus on protecting the trained weights deployed at the client side.

VIII. Conclusions

In this paper we proposed AEP, a DianNao based NN accelerator design for IP protection. By extracting device dependent error masks, AEP trains device dependent weight matrices such that adopting the weights can produce high inference accuracy on target devices. This effectively defeats NN weight piracy. In addition, by aggressively reducing tRAS timing, AEP achieves, on average, 72% performance improvement and 44% energy reduction over the DianNao baseline.

IX. Acknowledgments

This work was funded by NSF grant 1535755 and 1617071 to University of Pittsburgh.

References

[1] N. Ahmad, *et al.*, "Low-power Compact Composite Field AES S-Box/Inv S-Box Design in 65nm CMOS using Novel XOR Gate," *Integration, the VLSI Journal*, 2013.
[2] I. Bhati, *et al.*, "DRAM Refresh Mechanisms, Penalties, And Trade-Offs," *IEEE Transactions on Computers (TC)*, 2016.
[3] T. Chen, *et al.*, "DianNao: A Small-footprint High-throughput Accelerator For Ubiquitous Machine-learning," in *ASPLOS*, 2014.
[4] Y.-H. Chen, *et al.*, "Eyeriss: A Spatial Architecture For Energy-Efficient Dataflow For Convolutional Neural Networks," in *ISCA*, 2016.
[5] P. Chi, *et al.*, "PRIME: A Novel Processing-in-Memory Architecture For Neural Network Computation In ReRAM-Based Main Memory," in *ISCA*, 2016.
[6] A. Coates, *et al.*, "Deep learning with COTS HPC systems," in *ICML*, 2013.
[7] M. Courbariaux, *et al.*, "Training Deep Neural Networks With Low Precision Multiplications," arXiv:1409.1556, 2014.
[8] M. Courbariaux, *et al.*, "Binaryconnect: Training Deep Neural Networks with Binary Weights During Propagations," in *NIPS*, 2015.
[9] N. Dowlin, *et al.*, "Cryptonets: Applying Neural Networks to Encrypted Data with High Throughput and Accuracy," in *ICML*, 2016.
[10] S. Gupta, *et al.*, "Deep Learning with Limited Numerical Precision," in *ICML*, 2015.
[11] G. Hinton, *et al.*, "Deep Neural Networks for Acoustic Modeling in Speech Recognition: The Shared Views of Four Research Groups," in *IEEE Signal Processing Magazine*, 2012.
[12] ImageNet, "http://image-net.org/challenges/LSVRC/," .
[13] JEDEC, "High Bandwidth Memory (HBM) DRAM," *Technical report*, JESD235A JC-42.3, 2015.
[14] P. Judd, *et al.*, "Reduced-precision Strategies for Bounded Memory In Deep Neural Nets," arXiv:1511.05236, 2015.
[15] D. Kim, *et al.*, "Neurocube: A Programmable Digital Neuromorphic Architecture With High-Density 3D Memory," in *ISCA*, 2016.
[16] A. Krizhevsky, *Learning Multiple Layers of Features from Tiny Images*, Master's thesis, Univ. of Toronto, 2009.
[17] Q. V. Le, "Building High-level Features Using Large Scale Unsupervised Learning," in *ICASSP*, 2013.
[18] Y. LeCun, *et al.*, "Gradient-Based Learning Applied to Document Recognition," *Proc. of The IEEE*, 1998.
[19] D. Lee, *et al.*, "Adaptive-latency DRAM: Optimizing DRAM Timing For The Common-case," in *HPCA*, 2015.
[20] D. Lie, *et al.*, "Architectural Support for Copy and Tamper Resistant Software," in *ASPLOS*, 2000.
[21] S. Liu, *et al.*, "Flikker: Saving DRAM Refresh-power through Critical Data Partitioning," in *ASPLOS*, 2012.
[22] Y. Netzer, *et al.*, "Reading Digits in Natural Images with Unsupervised Feature Learning," in *NIPS*, 2011.
[23] K. Ni, *et al.*, "Large-scale deep learning on the YFCC100M dataset," arXiv:1502.03409, 2015.
[24] M. K. Qureshi, *et al.*, "AVATAR: A Variable-Retention-Time (VRT) Aware Refresh For DRAM Systems," in *DSN*, 2015.
[25] P. Rosenfeld, *et al.*, "DRAMSim2: A Cycle Accurate Memory System Simulator," in *IEEE Computer Architecture Letters*, 2011.
[26] A. Shafiee, *et al.*, "ISAAC: A Convolutional Neural Network Accelerator With In-Situ Analog Arithmetic In Crossbars," in *ISCA*, 2016.
[27] K. Simonyan, *et al.*, "Very Deep Convolutional Networks for Large-scale Image Recognition," arXiv:1409.1556, 2014.
[28] M. Song, *et al.*, "Towards Pervasive And User Satisfactory CNN Across GPU Microarchitectures," in *HPCA*, 2017.
[29] G. E. Suh, *et al.*, "Physical Unclonable Functions for Device Authentication and Secret Key Generation," in *DAC*, 2007.
[30] TheanoTeam, "Theano: A Python Framework for Fast Computation of Mathematical Expressions," arXiv:1605.02688, 2016.
[31] X. Zhang, *et al.*, "Exploiting DRAM Restore Time Variations In Deep Sub-micron Scaling," in *DATE*, 2015.
[32] X. Zhang, *et al.*, "Restore Truncation For Performance Improvement In Future DRAM Systems," in *HPCA*, 2016.
[33] Y. Zhang, *et al.*, "Architectural Support for Protecting User Privacy on Trusted Processors," in *ASPLOS*, 2004.