PHILOSOPHICAL TRANSACTIONS A

royalsocietypublishing.org/journal/rsta

Research

Cite this article: Abdullah Hanif M, Shafique M. 2019 SalvageDNN: salvaging deep neural network accelerators with permanent faults through saliency-driven fault-aware mapping. *Phil. Trans. R. Soc. A* **378**: 20190164.
http://dx.doi.org/10.1098/rsta.2019.0164

Accepted: 22 November 2019

One contribution of 13 to a theme issue 'Harmonizing energy-autonomous computing and intelligence'.

Subject Areas:
artificial intelligence, computer-aided design, computer vision

Keywords:
neural networks, reliability, energy-efficiency, hardware, accelerators, reliable computing

Author for correspondence:
Muhammad Abdullah Hanif
e-mail: muhammad.hanif@tuwien.ac.at

SalvageDNN: salvaging deep neural network accelerators with permanent faults through saliency-driven fault-aware mapping

Muhammad Abdullah Hanif and Muhammad Shafique

Technische Universität Wien (TU Wien), Vienna, Austria

MS, 0000-0002-2607-8135

Deep neural networks (DNNs) have proliferated in most of the application domains that involve data processing, predictive analysis and knowledge inference. Alongside the need for developing highly performance-efficient DNN accelerators, there is an utmost need to improve the yield of the manufacturing process in order to reduce the per unit cost of the DNN accelerators. To this end, we present 'SalvageDNN', a methodology to enable reliable execution of DNNs on the hardware accelerators with permanent faults (typically due to imperfect manufacturing processes). It employs a fault-aware mapping of different parts of a given DNN on the hardware accelerator (subjected to faults) by leveraging the saliency of the DNN parameters and the fault map of the underlying processing hardware. We also present novel modifications in a systolic array design to further improve the yield of the accelerators while ensuring reliable DNN execution using 'SalvageDNN' and negligible overheads in terms of area, power/energy and performance.

This article is part of the theme issue 'Harmonizing energy-autonomous computing and intelligence'.

1. Introduction

Deep neural networks (DNNs) are widely used in most of the application domains including computer vision, data mining, healthcare, robotics and language processing [1]. While these networks offer state-of-the-art accuracy for many applications, they are highly compute-intensive and energy hungry. To address these

THE ROYAL SOCIETY PUBLISHING

challenges, a large body of work has been carried out in building DNN accelerators for improving the performance and energy-efficiency of the DNN-based applications [2]. A few prominent accelerator designs are the TPU [3], the MPNA [4] and the Eyeriss [5].

The hardware chips (i.e. in the case of this work, the DNN accelerators) are fabricated using nanometre CMOS technologies, which require a highly sophisticated manufacturing process. The imperfections in the process result in defects in the fabricated chips. These defects can take a wide variety of forms, from permanent faults (e.g. stuck-at faults) that affect the functionality of the chips to variations that affect just the operating characteristics of the hardware (e.g. timing errors can occur if proper variability-driven voltage and frequency guardbands are not provided) [6,7]. Moreover, technology scaling, which has played a vital role in improving the performance and efficiency of the hardware, results in increased fault rates related to both permanent and transient faults. Prior works have highlighted that permanent faults in DNN accelerators significantly affect the accuracy of the DNNs [8]. These faults can be detected using post-fabrication testing and discarding every chip with a permanent fault affects the yield and thereby increases the per-unit-cost of the chips [8].

Improving the yield is one of the foremost challenges in reducing the per-unit-cost of the DNN accelerators, which is specifically important for devices manufactured using smaller technology nodes [9]. The severity of this challenge, in general, increases with an increase in the size of the chip, aggressive technology scaling and the integration density [9]. Therefore, in the current nanometre regime, where the number of processing elements (PEs)/cores per chip has drastically increased to meet the performance requirements of the applications, many integrated circuit (IC) manufacturing companies sell their faulty chips under low performance categories after hiding the faults and without revealing this information to the end-users. Two of the most prominent techniques that are widely studied for yield enhancement are:

— *Product binning*. This approach is mainly used to categorize the manufactured products based on their characteristics observed during post-fabrication testing [10,11]. The binning process is used to reduce the large variances in the characteristics by sorting the products into categories with smaller variances. This process is commonly used by chip manufacturers like Intel and AMD to improve the yield of their manufacturing process [12,13].
— *Hardware redundancy*. In this approach, spare components are introduced in the design in order to replace the defective ones detected during post-fabrication testing [10,14,15].

To this end, Kim *et al.* in [16] and Kung *et al.* in [17] proposed techniques for enhancing the fabrication yield at the cost of performance. Recently, a Fault-Aware Pruning (FAP) method was proposed by Zhang *et al.* in [8] to cater for the impact of permanent faults in DNN accelerators while maintaining the performance. The method works on the principle of dropping the computations, which are mapped on faulty MAC units by simply *bypassing* them. Note that these faults can be detected and the fault maps can be obtained by an initial BIST (built-in-self-test) like method [18,19]. Zhang *et al.* in [8] also proposed hardware modifications in which the computations of individual MAC units in a DNN accelerator can be bypassed to support the FAP method. Further, they proposed a Fault-Aware Pruning + Training (FAP+T) method which allows the fine-tuning of a model given the fault map of the hardware. However, these methods 'tend to either be ineffective or lack efficiency' in situations where any of the following conditions hold.

(i) *The additional circuitry used for bypassing the MAC units is faulty*. In this case the faults will propagate to the output of the hardware and will result in DNN accuracy degradation.
(ii) *The number of chips manufactured is large and the number of computational units in the DNN accelerator design is also significant.* In this case, the number of faulty chips can be large with each having a distinct fault map. For example, figure 1*a* shows the possible number of distinct fault maps that can exist for different systolic array sizes and figure 1*b* shows the number of possible fault maps for different array sizes having the total number of

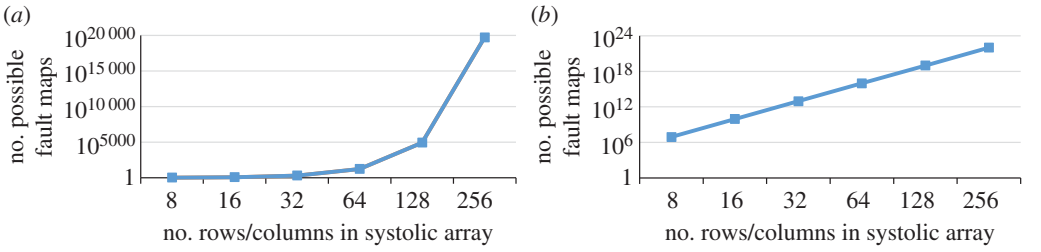

Figure 1. Trends illustrating the relation between the possible number of fault maps versus the number of rows/columns in a systolic array: (*a*) when as many as the number of PEs in the array can be faulty and (*b*) when at maximum five PEs can be faulty. (Online version in colour.)

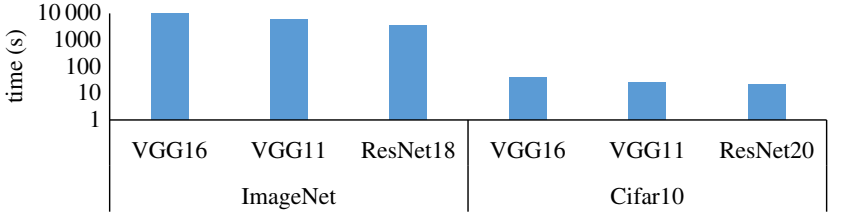

Figure 2. Time required for training different DNNs[1] (i.e. VGG16, VGG11, ResNet18 and ResNet20) for one epoch with different datasets (i.e. ImageNet and Cifar-10) on a Core i7 machine with one GTX1080Ti. (Online version in colour.)

faulty PEs less than or equal to five. As can be seen from the figure, with an increase in the size of the systolic array, the number of possible fault maps increases significantly and is greater than the number of chips that are usually manufactured of the same type. In this case, fine-tuning/training a neural network for each faulty chip seems impractical, as DNN fine-tuning/training is extremely time- and resource-consuming, specifically for deeper and complex NNs, as shown for different DNNs and datasets in figure 2.

(iii) *The training dataset is not available*, e.g. in a case where a company has released a neural network model, but they have not made the training data available.

(iv) Moreover, the above method will lack efficiency in case the number of permanent faults changes over the lifetime of the chip, e.g. due to ageing. In that case, excessive re-training cannot be done, and simply bypassing the respective computations may result in accuracy loss beyond acceptable range.

Our novel contributions: To address the aforementioned limitations, in this work, we propose:

(i) A methodology, *SalvageDNN*, for fault-aware mapping of a DNN based on the saliency (importance) characteristics of its filters/neurons. We propose and study different mapping strategies which can maintain the accuracy of the DNNs at high fault rates without the need for time- and resource-consuming fine-tuning/re-training processes. This also makes our approach training data agnostic.

(ii) Different hardware enhancements which enable us to *safely* bypass the processed computations of the faulty hardware, even if the bypass circuity of some MAC units is faulty. The designs will also allow us to trade-off between the area and the power overheads versus the accuracy loss that is experienced in case of such hardware faults.

Paper organization: In §2, we present the required preliminaries that are important for understanding the concept presented in the later sections of the paper. In §3, we present our

[1]For the analysis, we used the DNNs available in the *Neural Network distiller* [20] tool.

methodology for salvaging DNN accelerators using fault-aware mapping. Section 4 presents the results and comparison with the state of the art. Finally, §5 concludes the paper.

2. Preliminaries and background

In this section, we present a brief overview of the DNNs, the relevant methods for evaluating the saliency of filters/neurons in a NN, and the baseline DNN accelerator design used in this work.

(a) Deep neural networks

DNNs are composed of neurons that are arranged in the form of layers to progressively extract higher order features from the inputs, as shown in figure 3*a*. The basic operation performed by a neuron is a weighted sum of inputs which is then passed through an activation function to introduce non-linearity in the model. This can mathematically be written as:

$$A_i^{\langle l \rangle} = f^{\langle l \rangle}\left(\sum_j W_{(i,j)}^{\langle l \rangle} \times A_j^{\langle l-1 \rangle} + b_i^{\langle l \rangle}\right). \tag{2.1}$$

Here, $W_{(i,j)}^{\langle l \rangle}$ represents the weight of the connection between the ith neuron of the lth layer and the jth neuron of the $l-1$th layer, $b_i^{\langle l \rangle}$ represents the bias associated with the ith neuron, $A_i^{\langle l \rangle}$ represents the output (commonly known as activation) generated by the ith neuron, and $f^{\langle l \rangle}(.)$ represents the activation function used in the lth layer. The activation function can be any nonlinear function. However, the common types of activation functions used in state-of-the-art neural networks are Sigmoid, Tanh and RELU. Figure 3*a* shows an example of the fully-connected type of neural networks that can be represented using this notation. The number of neurons in the lth layer is represented by $N^{\langle l \rangle}$ and the dimension of weights of the lth fully-connected layer can be given as $N^{\langle l-1 \rangle} \times N^{\langle l \rangle}$. A neural network is considered deep (i.e. a DNN) if the number of layers is typically 3 or more [2].

A widely used type of DNN is the Convolutional Neural Network (CNN) which is based on convolutional layers. The basic operation performed in these layers is a convolution operation. The weights in these layers are arranged in the form of 3D filters which are traversed across the 2D space of the input activations to generate the outputs. Each filter is responsible for generating one feature map, and all the feature maps (generated using the filters of a layer) when combined form the input of the subsequent layer. The dimensions of the weights of the lth convolutional layer can be given as $F_{rc}^{\langle l \rangle} \times F_{rc}^{\langle l \rangle} \times C^{\langle l \rangle} \times N^{\langle l \rangle}$. Here, $N^{\langle l \rangle}$, $C^{\langle l \rangle}$ and $F_{rc}^{\langle l \rangle}$ represent the number of filters, the number of channels in each filter and the number of rows/columns of the filters of the lth layer, respectively. A specific weight in the lth convolutional layer can be represented as $W_{\{(r,c),(C,F)\}}^{\langle l \rangle}$, where r, c, C and F represent the row, the column, the channel and the filter number, respectively. Similarly, the dimensions of the output activations of the lth layer can be given as $x^{\langle l \rangle} \times y^{\langle l \rangle} \times N^{\langle l \rangle}$, where $x^{\langle l \rangle}$ and $y^{\langle l \rangle}$ represent the number of rows and columns in the output feature maps and $N^{\langle l \rangle}$ defines the number of feature maps, which is equivalent to the number of filters, in the lth layer. An example illustration of a convolutional layer is shown in figure 3*b*. A more detailed description of the DNNs can be found in [1,2].

(b) Deep neural network accelerator

Due to the compute intensive nature of DNNs, several accelerator designs have been proposed in the literature to accelerate the inference process of DNNs. These accelerators are mainly capable of performing efficient vector and matrix multiplication operations, which are the fundamental operations in the DNN inference. This is achieved by using multiple computational units which operate in parallel and by enabling local data sharing/reuse in these units.

The Tensor Processing Unit (TPU) [3] is one of the prominent DNN accelerators developed by Google for addressing the increased cloud-based DNN related workloads. At the core of the

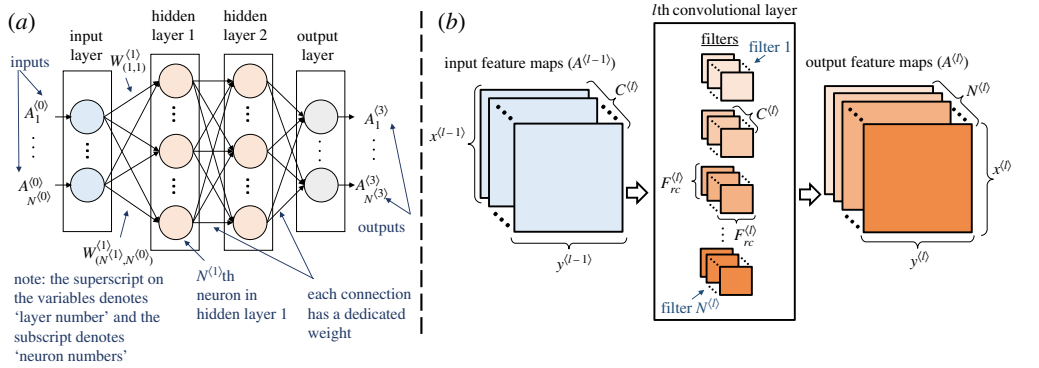

Figure 3. (*a*) An illustration of a fully-connected neural network. (*b*) An illustration of a convolutional layer used in convolutional neural networks. (Online version in colour.)

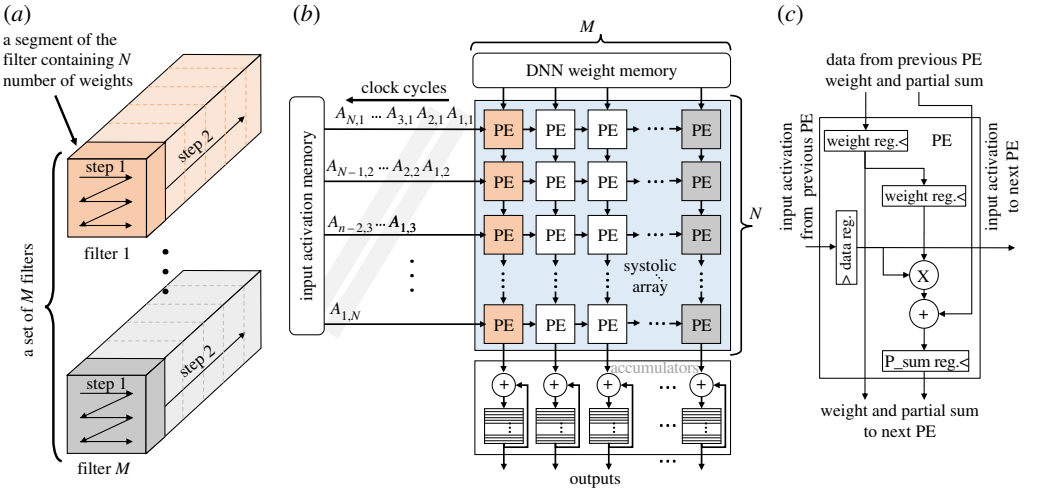

Figure 4. (*a*) Considered filter/neuron unrolling policy for mapping weights to the DNN hardware accelerator. The steps show the sequence of unrolling and mapping of the weights onto the array. (*b*) Our baseline systolic array-based DNN accelerator (similar to the design of well-know systolic arrays like Google TPU [3] and Eyeriss [5]), illustrating the mapping of a segment of filters highlighted in (*a*). (*c*) Detailed design of the processing element (PE). (Online version in colour.)

TPU is a systolic array composed of processing elements (PEs) that are connected in a 2D grid like manner. The design of our baseline systolic array is similar to that of the TPU architecture and is illustrated in figures 4*b*,*c*. Figures 4*a*,*b* show the dataflow where unrolled segments of a set of filters/neurons are mapped on columns of the systolic array, and then the vectors of input activations are fed to the array to perform the dot-product operations. The PEs work in lock steps with their neighbouring PEs to generate the output. For example, in the first clock cycle, the top left corner PE will multiply the first weight with the first input activation and then pass the partial output to its downstream neighbour and the input activation to the PE on its right side. The downstream PE (i.e. the second PE in the first column), in the next clock cycle, computes the product of the second weight with the second activation and adds the available partial product from the upstream PE to generate the partial product for its downstream PE. In the meantime, the first PE in the first column will generate the first partial product related to the second input activation vector and the first PE in the second column will generate the first partial product

related to the first input activation vector. By continuing this procedure, the result of the first dot-product from the systolic array will be available after N clock cycles and, at peak, the array can generate one result per clock cycle per column.

Note: If the number of weights in the filters/neurons is more than the number of rows in the array, the weights cannot be mapped to the array at the same time, as highlighted with the help of figures 4*a*,*b*. In such cases, the results generated by the array are not complete, and the accumulation units connected below the array are responsible for storing the partial products and accumulating them with the rest of the corresponding partial products of the filters/neurons to compute the final outputs.

(c) Saliency evaluation of neurons/filters of a DNN

Different methods have been proposed in literature to estimate the saliency of parameters (i.e. weights, neurons and filters) of a neural network. *The saliency of a network parameter defines its importance based on its contribution and/or the expected impact it can have on the output.* The widely used methods are based on L1 and L2 norms, where the norms of the parameters define their saliency [21]. Another prominent method includes neuron importance score computation based on propagation [22]. Such methods are usually considered more accurate for estimating the importance of the neurons/filters. However, they are computationally more intensive than norm-based methods, as they have to back propagate from the output to a particular neuron/filter to compute its saliency. A detailed explanation of the methods is provided in §3b.

3. SalvageDNNs: a methodology for salvaging DNN accelerators using fault-aware mapping

This section presents our methodology for salvaging DNN accelerators having permanent faults in the datapath using saliency-driven fault-aware mapping, without requiring extensive retraining as typically done in the state of the art. Figure 5 illustrates the main steps involved in the methodology. In step ❶, the DNN accelerator design is optimized such that the components having permanent faults can be disconnected from the main datapath to mitigate the effects of the faults. In step ❷, the saliency of the weights of the DNN is computed. Given a dataflow, the architectural characteristics of the modified accelerator design and the fault map of the fabricated hardware, step ❸ then proposes mapping of neurons/filters of a layer of the DNN on different segments of the hardware such that the sum of saliency of the weights that are pruned (mapped on the faulty/disconnected parts of the datapath) during inference is minimized. Step ❹ makes the required rearrangements in the DNN such that the data rearrangements, which are highly memory intensive, are not required during the DNN processing. Steps ❸ and ❹ are repeated for each layer and the resultant network is forwarded to step ❺ after setting all the weights which are to be mapped on the faulty/disconnected computational units to zero. Note that the network is traversed in a sequential order from the first layer to the last. In step ❺, required adjustments are made to the network parameters to compensate for the pruned weights/computations. The details of these steps are presented in the following subsections.

(a) Hardware design optimization/enhancements for permanent fault mitigation

Figure 6*c* shows the hardware optimization proposed in [8] for mitigating permanent faults, where an additional multiplexer is inserted in the datapath for bypassing the MAC computation in the PEs in case of faults. The baseline systolic array architecture and the design of the conventional processing element are shown in figures 6*a*,*b*, respectively. As permanent faults are randomly distributed in the hardware, they can occur in any hardware component, including the added circuitry. If a fault occurs in the multiplexer, the error cannot be mitigated and, therefore,

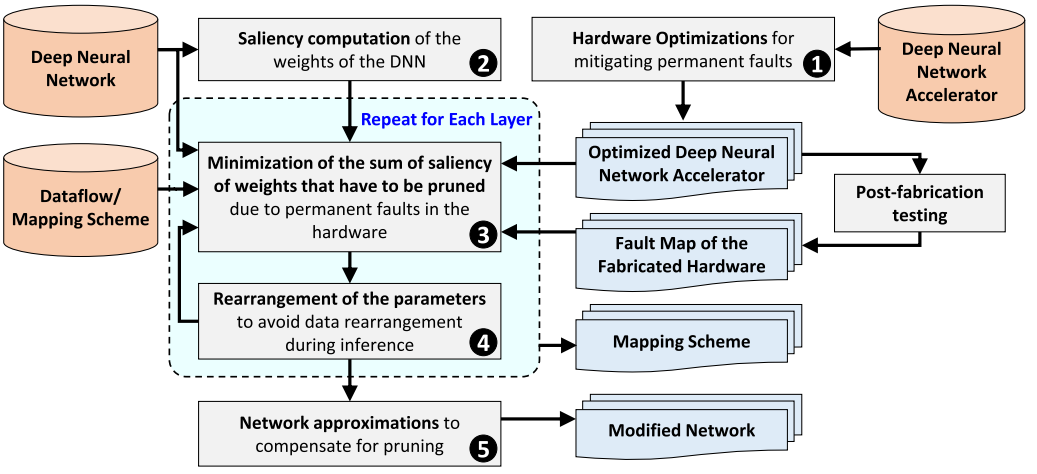

Figure 5. Overview of the proposed SalvageDNN methodology for saliency-driven fault-aware mapping of DNNs on a hardware with permanent faults. (Online version in colour.)

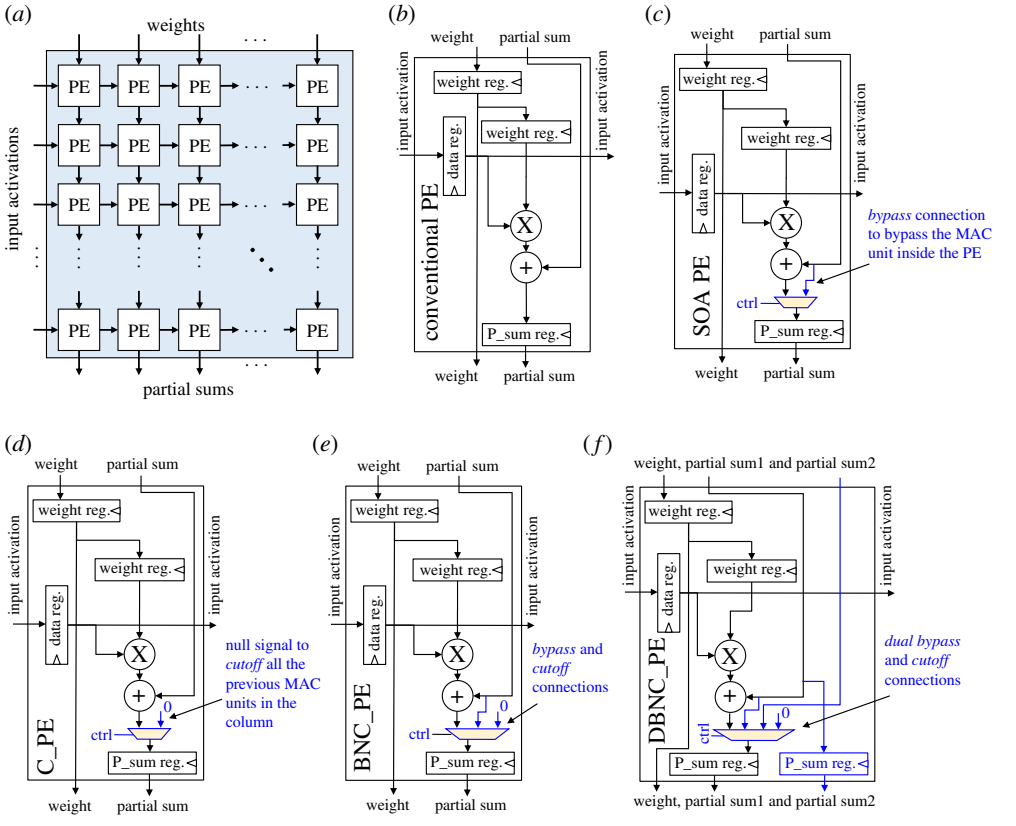

Figure 6. (*a*) The baseline systolic array design proposed in the TPU. (*b*) The conventional PE design. (*c*) The modified PE proposed in [8] for permanent fault mitigation using the FAP and the FAP+T techniques. (*d–f*) Show our novel additional PE designs for handing permanent faults. Note, the changes in the PEs with respect to the conventional PE design, i.e. (*b*), are shown in colour. (Online version in colour.)

will propagate to the output. To overcome this limitation, we propose and study different datapath enhancements which can enable us to bypass/disconnect the faulty multiplexers as well.

Figures 6*d–f* show different novel architectural modifications to mitigate the effects of permanent faults. These designs offer different trade-offs between the hardware overhead versus the average amount of circuitry disconnected in case of permanent faults, which directly impacts the accuracy of a DNN at runtime. Figure 6*d* shows the design where, in case of a fault in the MAC unit inside a PE, the complete set of MAC units above and including the MAC in the faulty PE are *cut off*. In case of a fault in the multiplexer of a PE, the MAC unit in the subsequent PE is also disconnected by using its multiplexer. This design will, henceforth, be referred to as C_PE in this paper. The advantage of this design is that the hardware overhead is significantly less. However, the accuracy of the DNNs can drop significantly depending on the locations and the number of faults, due to disconnection of a large number of PEs. Figure 6*e* illustrates a hybrid design which combines the best of both worlds, i.e. in case of a fault in the MAC unit of a PE, the MAC operation can be *bypassed* and, in case of a fault in the multiplexer, the complete set of MAC units including the MAC in the subsequent PE can be *cut off* from the computational flow. This design, from here onward, will be referred to as BNC_PE. The main advantage of this approach is that it can handle faults in the MAC units as well as the multiplexer while maintaining reasonable hardware overhead as compared to the design shown in figure 6*c*.

Figure 6*f* shows another design where an additional connection is made in the BNC_PE to *bypass two adjacent MAC units* in a column. This design also enables us to bypass the intermediate multiplexer in case it is faulty. Similarly, a design can be made where n adjacent MAC units can be bypassed. However, there are two main limitations associated with this approach: (1) the hardware overhead in this case is more as, apart from the larger multiplexer, additional registers are required for partial sums to maintain the computational flow; and (2) if n consecutive PEs have faults in their multiplexers, error can still propagate to the output. Therefore, in this work, we limit n to 2, as shown in figure 6*f*. This design, in the rest of this article, will be referred to as DBNC_PE.

(b) Saliency computation of the DNN parameters

In this article, we study the impact of two different methods that can be employed for saliency computation of the weights of a DNN on the effectiveness of the proposed methodology. Note that the proposed methodology only requires one method at a time for estimating the saliency of the DNN parameters. The methods along with their brief descriptions are as follows.

(i) *Norm-based method*. In this approach, the L1-norm or L2-norm of the weights/neurons defines their relative importance.

(ii) *Importance score propagation-based method*. In this approach the importance of a weight/neuron is computed based on its magnitude as well as the importance of the connections involved in propagating its response to the output of the NN. Let $s_i^{<l>}$ define the saliency of ith neuron/filter in the lth layer of a DNN. The saliency of a weight in the neuron/filter can be computed by multiplying the absolute value of the weight with the saliency, i.e. in the lth fully-connected layer it is computed by $|W_{(i,j)}^{<l>}| \times s_i^{<l>}$ and in the lth convolutional layer it is computed by $|W_{\{(r,c),(C,i)\}}^{<l>}| \times s_i^{<l>}$. The saliency of the neurons/filters in the lth layer of a DNN can be computed by

$$s_i^{<l>} = \sum_k |W_{(k,i)}^{<l+1>}| \times s_k^{<l+1>} \tag{3.1}$$

in case $l+1$th layer is a fully-connected layer, and is computed by

$$s_i^{<l>} = \sum_k \sum_r \sum_c |W_{\{(r,c),(i,k)\}}^{<l+1>}| \times s_k^{<l+1>} \tag{3.2}$$

in case $l+1$th layer is a convolutional layer. In this work, the saliency of all the output neurons is defined as equal, i.e. $s_i^{<n>} = 1 \forall i \in \{1, 2, \ldots N^l\}$.

Note, the aforementioned techniques are highly effective for comparing the saliencies of weights from the same layer. As in most of the DNN accelerators, the layers are processed sequentially, and we are only interested in evaluating the relative importance of weights within a layer. Hence, the above the mentioned approaches will be used for evaluating the importance of weights of a NN.

(c) Minimization of the sum of saliency

The objective of SalvageDNN is to map the least important weights on the components that will be disconnected from the datapath in case of occurrence of the permanent faults in the hardware. To achieve this, the following steps are followed for each layer of the network.

— *Generation of the Disconnection Map (DM)*. The DM is a matrix that defines the MAC units which will be disconnected from the array during processing. This is done by setting the values corresponding to the MACs which will be disconnected in DM to 1. For example, if the MAC unit in the first row and first column of the array will be disconnected during processing, the DM(1, 1) is set to 1; otherwise it will be 0. The DM is constructed using the fault map of the array. The fault map is represented using two matrices FM_{MAC} and FM_{MUX}, where FM_{MAC} keeps track of the faulty MAC units and FM_{MUX} keeps track of the faulty multiplexers in the array. Note that the size of DM, FM_{MAC} and FM_{MUX} is the same as the size of the array and there is one-to-one mapping between the elements of the matrix and respective hardware components of the systolic array.

— *Unrolling of the Saliencies of Weights of a Layer in a Matrix* $\boldsymbol{S}$. The matrix $\mathbf{S}$ represents the matrix containing the saliency of neurons/filters of a layer in flattened form, i.e. each column in the matrix contains saliency of weights from a neuron/filter that are flattened based on the given dataflow.

— *Generation of the Pruning Matrix (PM)*. The DM matrix is replicated in x and y dimensions such that the number of columns and rows in the final matrix is at least equivalent to the number of columns and rows in $\mathbf{S}$. The additional columns (from the right) and rows (from the bottom) are then removed, and the matrix is stored in the Pruning Matrix (PM). The columns containing all zeros are removed from the PM while keeping track of the original indexes of the non-zero columns in a vector $\boldsymbol{Idx}$.

— *Minimization of the Sum of Saliency of the Weights to be Pruned*. The objective of this step is to generate a mapping strategy which minimizes the sum of saliency of weights to be pruned. It can mathematically be represented as:

$$\text{argmin}_{\text{mapping}} ||S^*. * \text{PM}||. \tag{3.3}$$

Here, $\mathbf{S}^*$ represents a transformed version of $\mathbf{S}$ generated after applying rearrangement of network parameters based on the mapping strategy presented in §3d. The '.*' operation represents element-wise multiplication of the matrices. In this step, we use the proposed algorithm 1 or algorithm 2 to find the mapping sequence of neurons/filters. Algorithm 1 is based on a naive approach for fast generation of an acceptable solution. However, algorithm 2 is based on a branch and bound approach to propose an optimal solution. The detailed descriptions of algorithms 1 and 2 are as below.

Algorithm 1 presents a naive method to find a sub-optimal mapping strategy. The algorithm takes the $\boldsymbol{Costs}$ matrix, which contains the costs of mapping each neuron/filter to each faulty column of the systolic array, and $\boldsymbol{Idx}$ vector, which contains the indexes of the faulty columns of the array, as inputs. The $\boldsymbol{Costs}$ matrix is computed using matrix multiplication of *transpose* of $\boldsymbol{S}$ with $\boldsymbol{PM}$. Each row of the $\boldsymbol{Costs}$ matrix contains the cost of mapping a filter to each faulty column

Algorithm 1 A fast method to reduce the sum of saliency of the weights of a layer that have to be pruned due to permanent faults.

Inputs: A matrix ***Costs*** containing the costs of mapping neurons/filters to faulty columns of the array and a vector ***Idx*** containing the indexes of faulty columns of the array
Outputs: A matrix ***Mapping*** which defines which neuron/filter has to be mapped to which faulty column and a variable *TCost* which defines the cost of this mapping scheme
Initialize: *TCost* = 0, ***Mapping*** = Zeros(Number of Columns in ***Costs***, 2) and ***NF_Idx*** = [1, 2, 3, . . . Number of neurons/filters in the layer]

1: **for** *i* = 1 to Number of columns in ***Costs*** at the start of the loop **do**
2: [*val*, *row_id*, *col_id*] = min(***Costs***)
3: ***Mapping***(*i*,1) = ***NF_Idx***(*row_id*)
4: ***Mapping***(*i*,2) = ***Idx***(*col_id*)
5: *TCost* = *TCost* + *val*
6: ***Costs*** ⇐ ***Costs*** after removing *row_id* row and *col_id* column
7: ***Idx*** ⇐ ***Idx*** after removing *col_id* column
8: ***NF_Idx*** ⇐ ***NF_Idx*** after removing *row_id* column
9: **end for**
10: **return** *TCost* and ***Mapping***

of the array. The algorithm, at each iteration, finds the minimum value in the matrix (line 2 in algorithm 1) and then associates the corresponding neuron/filter index form ***NF_Idx*** with the index of the column of the array from ***Idx*** (lines 3 and 4). The costs associated with the selected filter and the column of the array are then removed from the ***Costs*** matrix (line 6). Similarly, the index vectors are also updated (lines 7 and 8). The resultant matrix and vectors are then used in the next iteration. The algorithm iterates for the number of filters to be mapped on the faulty columns (line 1), i.e. the number of columns in the ***Costs*** matrix, and outputs the mapping (***Mapping***) and the corresponding total cost (*TCost*).

Algorithm 2 presents a more robust approach for searching the optimal mapping which provides minimum sum of saliency of the weights to be pruned. The algorithm is based on the branch and bound method where it recursively calls (line 16) the *Branch&Bound* function (line 3) with a smaller problem, i.e. smaller ***Costs*** matrix (lines 11 till 14). The algorithm keeps track of the cost of already selected pairs in *CS_Cost* and bounds the algorithm from searching in the same branch if the overall cost is more than the cost of the reference mapping ***R_Mapping*** (lines 10 and 26). Note that at beginning of the algorithm, the reference cost is computed using algorithm 1 to speed-up the process. If, at some point, a mapping which has cost less than the cost of the reference mapping is found the reference mapping and the reference cost are updated (lines 26 till 28). As the algorithm is meant to search for all the possible branches (combinations) and can take endless amount of time for larger ***Costs*** matrices, we limit the search by applying a limit on the number of miss hits (*MH_Count*), i.e. *Termination_Limit*, (line 18) from the last reference mapping update. We also use a *per branch search limit* (line 7) to decrease the number of searches in each branch and search only on a defined number of nodes which offer minimum costs (lines 6 and 7).

(d) Rearrangement of the network parameters

In most of the DNNs, the arrangement of the neurons/filters within a layer can be changed without affecting the functionality of the network. Figure 7 illustrates this with the help of an example where two neurons in an FC layer are swapped. Figure 8 illustrates a similar example for the filters in a CONV layer. To maintain the functionality, the corresponding connections in the subsequent layer of the network are also swapped. This property helps in maintaining

Algorithm 2 A branch and bound method to optimally reduce the sum of saliency of the weights of a layer that have to be pruned due to permanent faults.

Inputs: A matrix ***Costs*** containing the costs of mapping neurons/filters to faulty columns of the array and a vector ***Idx*** containing the indexes of faulty columns of the array
Outputs: A matrix ***Mapping*** which defines which neuron/filter has to be mapped to which faulty column and a variable *TCost* which defines the cost of this mapping scheme
Initialize: *TCost* = 0, ***Mapping*** = Zeros(Number of Columns in ***Costs***, 2), ***NF_Idx*** = [1, 2, 3, ... Number of neurons/filters in the layer], and ***MH_Count*** = 0

Find a reference minimum cost *RMin_Cost* and corresponding mapping scheme ***R_Mapping*** using Algorithm 1
[*TCost*, ***Mapping***, *MH_Count*] = Branch&Bound(***Costs***, ***Idx***, ***NF_Idx***, *RMin_Cost*, ***R_Mapping***, 0, [], *MH_Count*)

Function [*RMin_Cost*, ***R_Mapping***, *MH_Count*] = Branch&Bound(***Costs***, ***Idx***, ***NF_Idx***, *RMin_Cost*, ***R_Mapping***, *CS_Cost*, ***CS_Mapping***, *MH_Count*)
if size(***Idx***)≠1 **then**
    for *i* = 1 to Number of Rows in ***Costs*** - Number of Columns in ***Costs*** + 1 **do**
        Sort i^{th} row of ***Costs*** and store indexes in ***SR_Idx***
        for *k* = 1 to minimum(Number of Columns in ***Costs***, ***Per_Branch_Search_Limit***) **do**
            MH_Count = *MH_Count* + 1
            j = ***SR_Idx***(k)
            if *CS_Cost* + ***Costs(i,j)*** < *RMin_Cost* **then**
                New_Costs ⇐ ***Costs*** after removing its i^{th} row and j^{th} columns
                New_Idx ⇐ ***Idx*** after removing its j^{th} column
                New_NF_Idx ⇐ ***NF_Idx*** after removing its j^{th} column
                New_CS_Mapping ⇐ ***CS_Mapping*** after appending the row [***NF_Idx***(*i*), ***Idx***(*j*)]
                MH_Count = *MH_Count* - 1
                [*RMin_Cost*, ***R_Mapping***, *MH_Count*] = Branch&Bound(***New_Cost***, ***New_Idx***, ***New_NF_Idx***, *RMin_Cost*, ***R_Mapping***, *CS_Cost* + ***Costs(i,j)***, ***New_CS_Mapping***, *MH_Count*)
            end if
            if *MH_Count* >= *Termination_Limit* **then**
                return *RMin_Cost*, ***R_Mapping***, *MH_Count*
            end if
        end for
    end for
else
    MH_Count = *MH_Count* + 1
    [*val*, *row_id*, *col_id*] = min(***Costs***)
    if *CS_Cost* + *val* < *RMin_Cost* **then**
        RMin_Cost = *CS_Cost* + *val*
        R_Mapping ⇐ ***CS_Mapping*** after appending the row [***NF_Idx***(*row_id*), ***Idx***(*col_id*)]
        MH_Count = 0
    end if
end if
return *RMin_Cost*, ***R_Mapping***, *MH_Count*

the original structure and the computing sequence inside the accelerator. The only exceptions are the networks which have local response normalization layers, e.g. the AlexNet [23]. In such networks, the computing sequence in the accelerator has to be slightly modified and additional

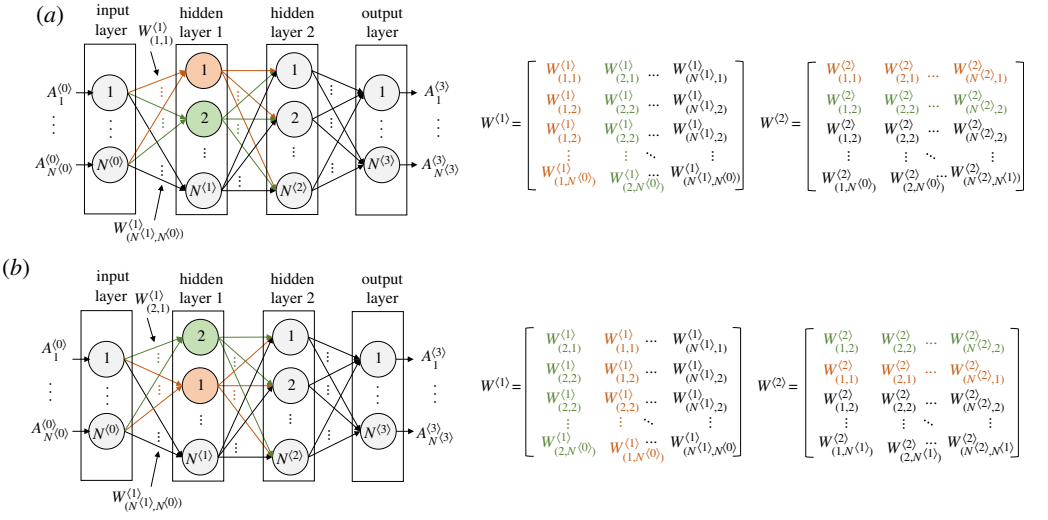

Figure 7. Impact of rearranging neurons in a layer of a fully-connected DNN on the arrangement of the weights to be mapped on the systolic array. (*a*) Shows the arrangement before swapping neurons 1 and 2 in the first hidden layer of a fully-connected DNN and (*b*) shows the arrangement after swapping the neurons. The left side of the figure illustrates the state of the neural network and the right side shows the weights of the first and second hidden layers in a manner in which they will be mapped on a systolic array. Different colours are used to show the association between the neurons and weights. (Online version in colour.)

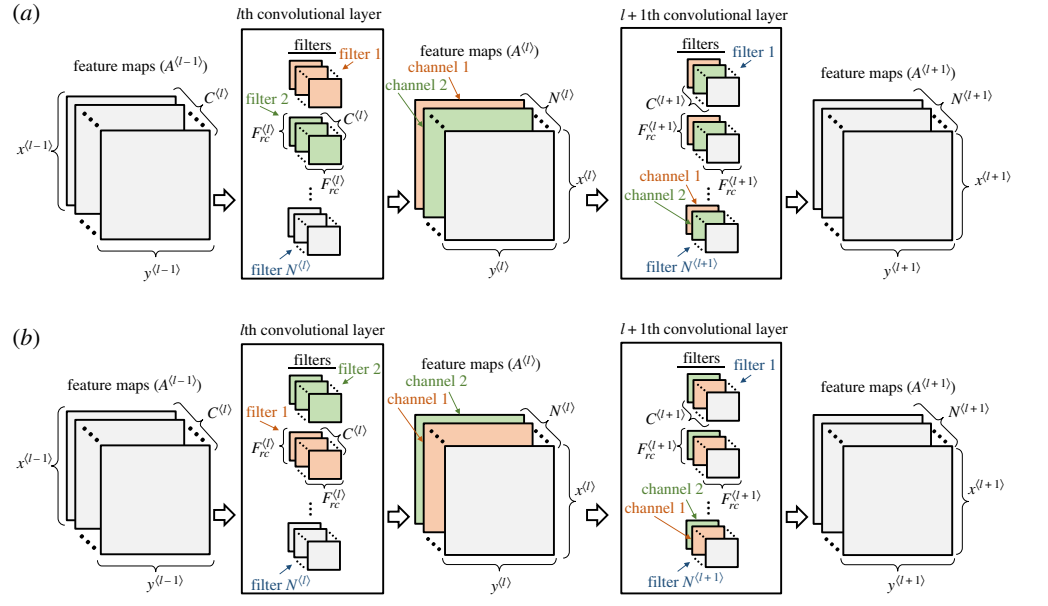

Figure 8. Impact of rearranging filters in a layer of a CNN on the arrangement of the weights in the neural network. (*a*) shows the arrangement of filters and their channels before swapping filters 1 and 2 in the *l*th convolutional layer of a CNN and (*b*) shows the arrangement after swapping the filters. Note that a swap of filters in the *l*th layer of a CNN requires a swap of the respective channels in the $l + 1$th layer of the CNN to maintain the functionality. (Online version in colour.)

data arrangement operations might be required which can affect the efficiency of the system. However, almost all the modern DNNs do not make use of this feature, and hence can directly benefit from the proposed approaches of this article.

(e) Network approximation

After setting the corresponding weights that have to be mapped on the faulty computational units to zero, we compute the mean of the output activations of the neurons and filters that are pruned using a small subset of the validation dataset. The mean values are compared with the mean values acquired using the original network, and based on their difference, the biases of the neurons/filters are adjusted to compensate for the effects of pruning. Note that, in case the training dataset is available, fine-tuning can also be exploited for regaining the lost accuracy.

(f) SalvageDNN under changing fault maps

In this section, we explain with the help of examples how the proposed methodology helps salvage a faulty DNN hardware. The practical cases can be represented using two scenarios: (1) where the fault maps are generated using post-fabrication testing and each chip has a distinct fault map; and (2) where the fault map of a chip is changing over time and can be extracted using BIST support available in the chip. Note that, for ease of understanding, in both the scenarios, we assume that only the faulty PEs are required to be bypassed to avoid error propagation during execution and that the saliency of the network parameters is computed using L1-norm, i.e. their absolute values.

Scenario 1. Different hardware chips can have different fault maps, i.e. the fault maps can vary across chips. To explain how our proposed approach handles such a scenario, we present an example where four filters (figure 9*a*) each having four values (i.e. weights) have to be mapped on a 4×4 systolic array to perform the dot-product operations. The filters are unrolled and mapped on the systolic array based on the baseline systolic array design discussed in §2b. Figure 9 highlights four different example cases each representing a systolic array belonging to a different chip. The cases are explained as follows:

- Case 1: None of the PEs in the systolic array is faulty (figure 9*b*).
- Case 2: The array has two faulty PEs (figure 9*c*).
- Case 3: The array has the same number of faulty PEs as in Case 2, but at different locations (figure 9*d*).
- Case 4: The array has different number of faulty PEs than Case 2 and Case 3, and also at different locations (figure 9*e*).

In Case 1 (figure 9*b*), the mapping does not affect the accuracy of the results, as there are no faulty PEs in the array which have to be bypassed during execution. In Case 2 (figure 9*c*), the filters are mapped onto the columns such that the sum of absolute weights mapped on the faulty PEs is minimum, which in the example is achieved by mapping filters 1, 2, 3 and 4 on columns 2, 1, 4 and 3, respectively, of the systolic array. In Case 3 (figure 9*d*), the fault map of the array is different from the ones presented in earlier two cases, therefore, the mapping can be different, depending solely on the fault map of the array and the filter values. As can be seen in figure 9*d*, to achieve the minimum sum of absolute weights, filters 1, 2, 3 and 4 are now mapped on columns 2, 3, 4 and 1, respectively. Similarly, in Case 4 (figure 9*e*) the mapping is defined based on its fault map. It is important to clarify here that, similar to the state-of-the-art FAP technique [8], our proposed technique also requires adaptation for each faulty chip based on its fault map. However, unlike the state of the art, our technique can avoid retraining, and, therefore requires much less time for the adaptation compared to the retraining-based approaches, as will be shown in §4b(v). Moreover, our technique also does not require access to the training dataset, which in most of the practical cases is not available (as stated in §1). Details regarding how our proposed approach minimizes the mapping are presented in algorithms 1 and 2.

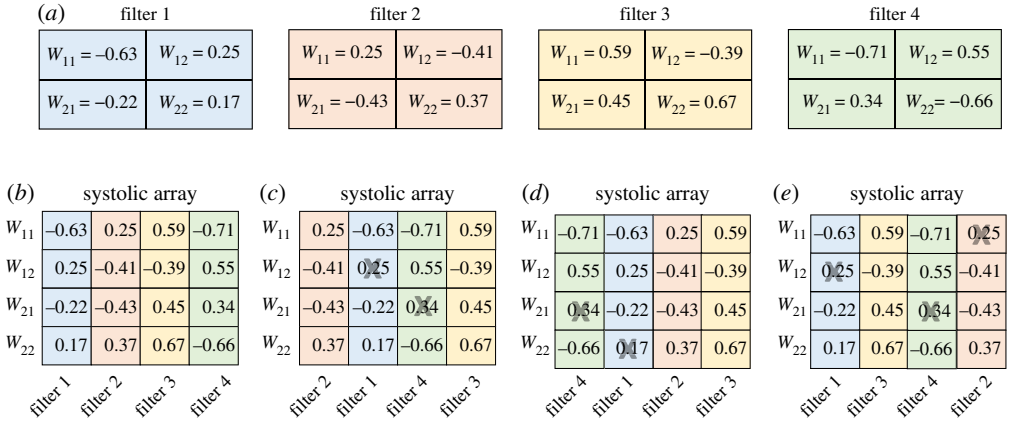

Figure 9. An example illustration of how the mapping would vary across chips having different fault maps. The grids shown in *b*, *c*, *d* and *e* corresponds to 4 × 4 systolic arrays from four different chips. Each small box inside a grid represents a single processing element (PE). The PEs with black cross over them in *c*, *d* and *e* correspond to faulty PEs. The filters considered in this example are shown in *a*. (*a*) Filters, (*b*) Case 1, (*c*) Case 2, (*d*) Case 3, and (*e*) Case 4. (Online version in colour.)

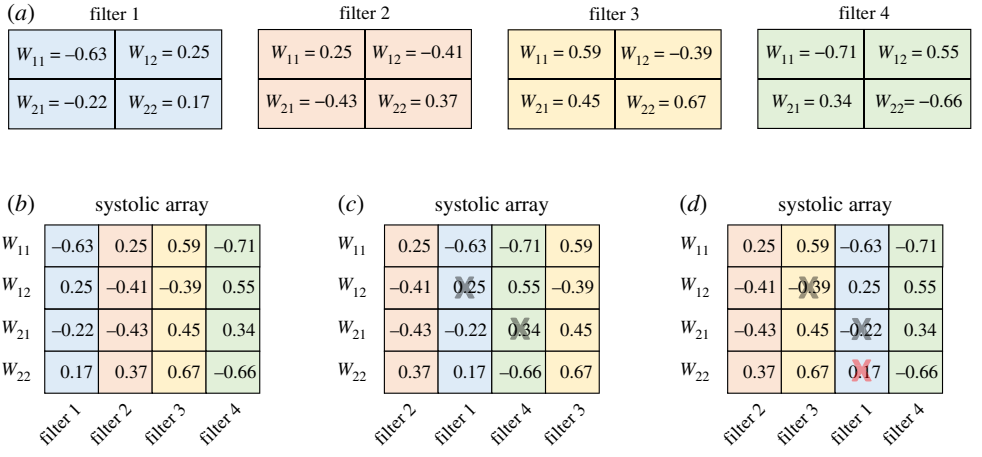

Figure 10. An example illustration of how the mapping would change if the fault map of the DNN hardware changes over time. The grid shown in *b*, *c* and *d* corresponds to a systolic array, where each small box represents a single processing element (PE). The PEs with black cross over them in (*c*) and (*d*) correspond to faulty PEs detected during post-fabrication testing and the PE with red cross over it in (*d*) corresponds to the PE which experienced fault over time due to wear-out. (*a*) Filters, (*b*) Case 1, (*c*) Case 2, and (*d*) Case 3. (Online version in colour.)

Scenario 2. The fault maps can change over time, e.g. due to wear-out. To explain how the mapping changes to take this effect of additional faults into account, we take an example where four filters (figure 10*a*), the same as shown in figure 9*a*, have to be mapped on a 4 × 4 systolic array. Figure 10 highlights three different cases:

— Case 1: None of the PEs in the systolic array is faulty (figure 10*b*).
— Case 2: The array has two faulty PEs (figure 10*c*).
— Case 3: With the passage of time, the number of faulty PEs in the array presented in Case 2 has increased to three (figure 10*d*).

Case 1 and Case 2 are the same as presented in figure 9, where, in Case 1, the mapping does not affect the accuracy of the results and, in Case 2, the filters 1, 2, 3 and 4 are mapped on columns 2, 1, 4 and 3 respectively (figure 10*c*), to minimize the sum of absolute weights that are mapped on the faulty PEs. However, in Case 3, once the fault map is updated with the help of the available BIST support, the mapping of filters should also be updated to achieve a lower sum. As can be seen in figure 10*d*, the minimum sum is achieved by mapping filters 1, 2, 3 and 4 on columns 3, 1, 2 and 4, respectively. Note that, with the mapping of Case 2 and the fault map of Case 3, the sum of absolute weights mapped on the faulty PEs would have been 1.25, while with the updated mapping, the sum is 0.78, as can be observed in figure 10*d*.

Associated testing and reconfiguring overheads: Post-fabrication testing is an essential part of the manufacturing process, which is required to verify the functional correctness of the manufactured hardware. This process is also used for diagnostics to check which module/components inside the hardware are faulty [24–26]. Therefore, no additional cost would be required to extract the fault map in Scenario 1, apart from the usual cost necessary for effective post-fabrication testing. The fault map extracted during post-fabrication testing can be used, in our case, for reconfiguring the fault tolerant systolic arrays, as well as for determining the DNN mapping using the proposed, SalvageDNN, methodology. However, storage of the fault map requires some memory, which is relatively small as it depends on the size of the systolic array, i.e. one bit per PE. On the other hand, in Scenario 2, we need boot-time/online BIST support to update the fault map after an interval of time. However, the cost of such support is high, which can be reduced by using algorithm-based fault detection and location techniques [27]. The algorithm-based techniques have proven to be very effective in the case of regular hardware composed of homogeneous components such as systolic arrays.

4. Results and discussion

(a) Hardware synthesis results

Table 1 shows the hardware characteristics of the processing elements of the baseline systolic array design (i.e. the one without fault tolerance circuitry) and the state-of-the-art fault tolerant systolic array design. The hardware results are generated following an ASIC design flow using Cadence Genus tool and a TSMC 65nm technology library. As can be seen in the table, for all the listed systolic array sizes, the MAC units consume approximately 66% of the area of a PE, and the multiplexer inside each PE (as used in the state-of-the-art fault tolerant systolic array design) consumes approximately 6% of the area of a PE. This shows that having a fault tolerant design that mitigates permanent faults in the MAC and MUX units of the array can significantly improve the yield of the manufacturing process.

Figure 11 presents a comparison of the hardware characteristics of the PEs of different sizes and types of systolic arrays as discussed in §3a. The figure shows that the fault-tolerant PEs consume slightly more area as compared to a baseline PE depending on the complexity of the bypassing/cut-off circuitry used in the design. For example, a PE with dual bypass and cut-off circuitry (i.e. DBNC_PE) consumes approximately 25% more area, and a PE with cut-off circuitry (i.e. *C*_PE) consumes approximately 1% additional area, when compared to the baseline PE used for the same size systolic array. The variation in the area with respect to the array size is due to the fact that the bit-width of the partial sums increases with an increase in the number of PEs in a single column of the array. Similar to the area characteristics, the critical path delay of a PE also depends on the design and the size of the systolic array. However, the variations in the delay are relatively small, i.e. at the maximum 6.5% for the 8x8 systolic array composed of *DBNC_PE* when compared to the baseline PE for the same array size. The trend in the power characteristics is approximately similar to the trend observed in the area characteristics of the PEs. Hence the multiple options, that are shown in figure 6, §3a, provide different design trade-offs in terms of area, latency, power and resilience.

Table 1. Hardware results of the components used in different sizes of the baseline and the state-of-the-art fault tolerant systolic arrays. The bit-widths of the partial sums in 8 × 8, 16 × 16, 32 × 32 and 256 × 256 arrays were set to 19, 20, 21 and 24, respectively, assuming the bit-width of weights and activations to be 8-bit.

		area (cell area)	delay (ns)	power (μW)
8 × 8 array	baseline PE	931	2.17	76.77
	SOA PE	972	2.21	80.10
	MAC	620	1.97	62.11
	2-to-1 MUX	56	0.24	2.69
16 × 16 array	baseline PE	950	2.23	77.09
	SOA PE	993	2.26	82.09
	MAC	632	2.01	62.01
	2-to-1 MUX	59	0.25	2.88
32 × 32 array	baseline PE	965	2.32	77.62
	SOA PE	1010	2.35	81.10
	MAC	640	2.10	61.10
	2-to-1 MUX	62	0.25	2.90
256 × 256 array	baseline PE	1006	2.65	77.40
	SOA PE	1058	2.68	80.64
	MAC	661	2.44	61.66
	2-to-1 MUX	70	0.28	3.39

(b) Impact of fault-aware mapping (SalvageDNN) on the DNN accuracy

(i) Experimental set-up

For analysing the impact of permanent faults on the accuracy of DNNs, we considered moderately deep neural networks, i.e. the VGG11 and the VGG16, trained on the Cifar-10 [28] and the ImageNet [29] datasets. The DNNs are taken from the *Neural Network distiller* [20] which is an open-source tool for neural network compression. The details of the models are provided in table 2 and a few key characteristics of the datasets are presented in table 3. We used the same tool (i.e. distiller) to quantize the neural network weights and activations to 8-bits. For fault injection, similar to the well-established works in the reliability community, we considered a random distribution of the faults across the array and performed multiple iterations of each experiment using different seeds. To achieve a fair comparison, we used the same seeds for the experiments across arrays and fault rates. Moreover, we also took into consideration the area of MAC and MUX units for distributing the faults.

(ii) Comparison with the state-of-the-art FAP approach

Figure 12 illustrates the impact of permanent faults on the accuracy of the VGG-11 network when mapped on a 256 × 256 systolic array composed of SOA_PEs using the FAP and the proposed technique. To have a fair comparison with the state-of-the-art FAP approach, we adopted a similar evaluation methodology as reported in their paper [8]. That is, in this experiment, we assumed

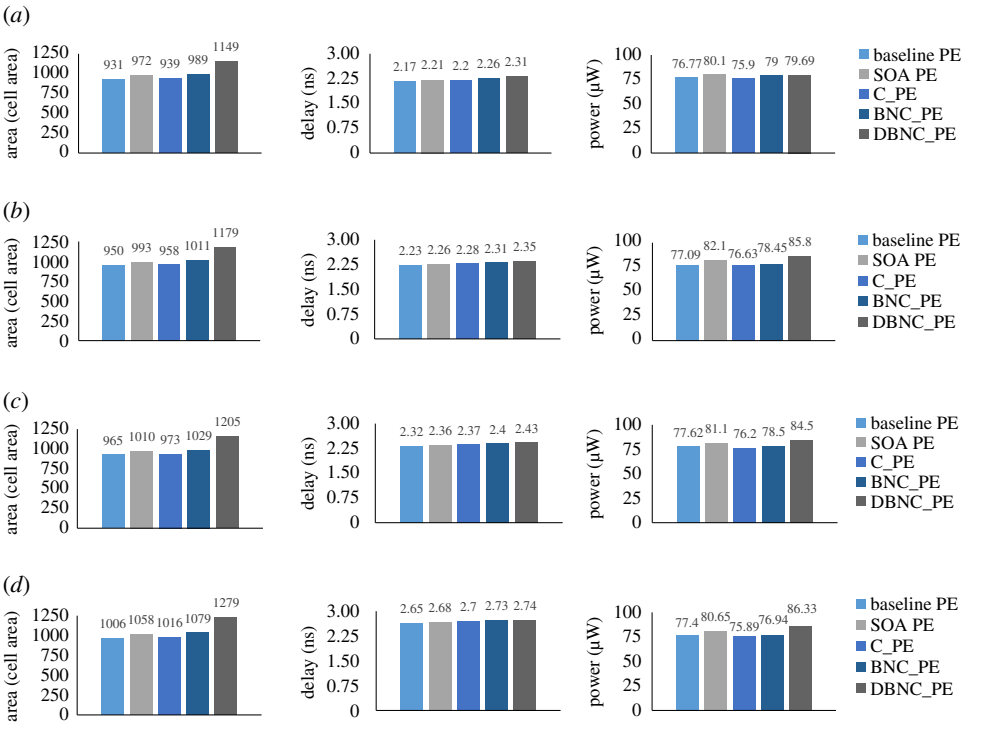

Figure 11. Hardware synthesis results of the PEs of different sizes and types of systolic array designs. (*a*) Hardware characteristics of PEs used in different types of 8 × 8 systolic arrays, (*b*) hardware characteristics of PEs used in different types of 16 × 16 systolic arrays, (*c*) hardware characteristics of PEs used in different types of 32 × 32 systolic arrays and (*d*) hardware characteristics of PEs used in different types of 256 × 256 systolic arrays. (Online version in colour.)

Table 2. DNNs and the datasets used for evaluation.

network and dataset	network architecture	baseline accuracy (%)
VGG11 for Cifar-10	layers: CONV(64, 3, 3, 3); CONV(128, 64, 3, 3); CONV(256, 128, 3, 3); CONV(256, 256, 3, 3); CONV(512, 256, 3, 3); CONV(512, 512, 3, 3); CONV(512, 512, 3, 3); CONV(512, 512, 3, 3); FC(10, 512)	85.38%
VGG11 for ImageNet	layers: CONV(64, 3, 3, 3); CONV(128, 64, 3, 3); CONV(256, 128, 3, 3); CONV(256, 256, 3, 3); CONV(512, 256, 3, 3); CONV(512, 512, 3, 3); CONV(512, 512, 3, 3); CONV(512, 512, 3, 3); FC(4096, 25088); FC(4096, 4096); FC(1000, 4096)	68.04% (Top1) 88.07% (Top5)
VGG16 for ImageNet	layers: CONV(64, 3, 3, 3); CONV(64, 64, 3, 3); CONV(128, 64, 3, 3); CONV(128, 128, 3, 3); CONV(256, 128, 3, 3); CONV(256, 256, 3, 3); CONV(256, 256, 3, 3); CONV(512, 256, 3, 3); CONV(512, 512, 3, 3); CONV(512, 512, 3, 3); CONV(512, 512, 3, 3); CONV(512, 512, 3, 3); CONV(512, 512, 3, 3); FC(4096, 25088); FC(4096, 4096); FC(1000, 4096)	70.85% (Top1) 90.0% (Top5)

that faults can only occur in the MAC units of the array and not in the MUX or other components of the PEs. From this experiment, the following key observations can be made:

— the figure illustrates that, with an increase in the fault rate, the accuracy of the network when mapped using FAP approach decreases rapidly as compared to when mapped using SalvageDNN, i.e. while considering the saliency of the weights;

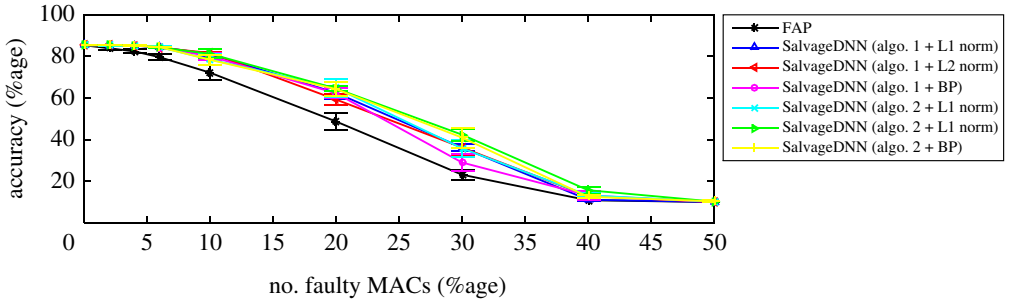

Figure 12. Impact of number of faulty MAC units in a 256 × 256 systolic array composed of SOA_PEs on the accuracy of the VGG-11 network trained on the Cifar-10 dataset. (Online version in colour.)

Table 3. A summary of the key characteristics of the datasets used in the evaluation of SalvageDNN and the comparison with the state of the art.

dataset	characteristics	
Cifar-10	number of training images	50000
	number of testing images	10000
	size of images	32 × 32 × 3
	number of classes	10
ImageNet	number of training images	∼14 million
	number of testing images	100000
	size of images	224 × 224 × 3
	number of classes	1000

— at lower fault rates (i.e. until 6% of the MAC units are faulty), which are typically common in the real-world settings if the fabrication process is mature, SalvageDNN helps in maintaining the baseline accuracy of the network while the DNN accuracy of the network when mapped using FAP starts decreasing instantly with the increase in the fault rate;
— the error bars in the figure, which shows the standard deviation of the accuracy across multiple iterations of the same experiment, highlights that the accuracy is highly dependent on the location of the faulty PEs. Therefore, FAP, which uses a fixed mapping approach, results in higher standard deviation at the same fault rate. However, the proposed SalvageDNN adapts as per the locations of the faults and offers less deviation when compared with FAP method;
— the type of saliency and the algorithm used in SalvageDNN does not significantly impact the technique. Note that algorithm 2 provides sightly better results at higher fault rates. However, based on the computational requirements algorithm 1 + L1 norm should be preferred.

(iii) Comparison with the state-of-the-art FAP+T approach

Figure 13 illustrates the impact of using SalvageDNN and then applying fault-aware retraining on the accuracy of the VGG11 network for the Cifar10 classification. The subfigures show the comparison between the 'FAP+T' [8] and the 'SalvageDNN + retraining', when the underlying hardware has faulty PEs. As can be seen from the figure, both the approaches perform equally

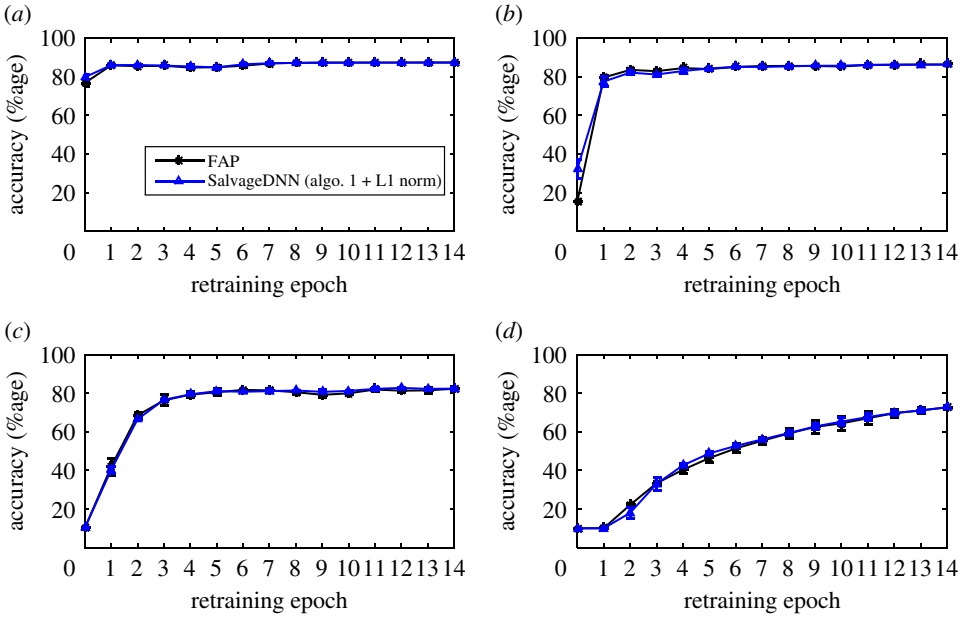

Figure 13. Impact of using SalvageDNN before applying fault-aware training on the accuracy of the VGG11 network trained for the Cifar10 classification. The subfigures show the comparison between FAP + retraining and SalvageDNN + retraining when the underlying hardware has: (*a*) 10% faulty PEs, (*b*) 30% faulty PEs, (*c*) 50% faulty PEs, and (*d*) 70% faulty PEs. (Online version in colour.)

well and if the training dataset and a significant amount of computational resources for retraining are available, a large percentage of the lost accuracy can be regained through retraining. The number of epochs required to regain the accuracy seems to be dependent on the number of faulty PEs in the array. This can be observed from figure 13, where figures 13*a*,*b*,*c* and 13*d* show the test accuracy after every epoch of retraining when 10%, 30%, 50% and 70% of the total PEs are faulty respectively. For example, in figure 13*a*, the baseline accuracy was regained after only one epoch of retraining, however, in figure 13*c*, it took five epochs to reach the saturation point.

(iv) Evaluation using DNNs trained for larger datasets

Figure 14 illustrates the comparison of the proposed approach with the state-of-the-art FAP approach when used for the VGG11 network trained on a larger dataset, i.e. the ImageNet. The systolic array considered in this scenario is a 256 × 256 sized array composed of SOA_PEs. As can be seen from the figure, for lower fault rates, i.e. around 0.06 (6%), SalvageDNN can help a network maintain accuracy close to its baseline, while the accuracy with FAP approach drops significantly even when only 2% of the total PEs are faulty. Similar results are observed in figure 15 for the VGG16 network trained on the ImageNet dataset.

(v) Comparison of the execution time of the proposed technique with the retraining-based approaches

Table 4 presents a comparison between the execution time required for the SalvageDNN and the FAP+T for different neural networks and datasets. The details of the networks and a summary of a few of the key characteristics of the datasets used in the comparison are presented in tables 2 and 3, respectively. As can be seen from the tables, the execution time depends on the complexity of the dataset (only for retraining-based approaches) as well as the size of the network. However, in all the cases SalvageDNN requires, at the very least, an order of magnitude less execution time

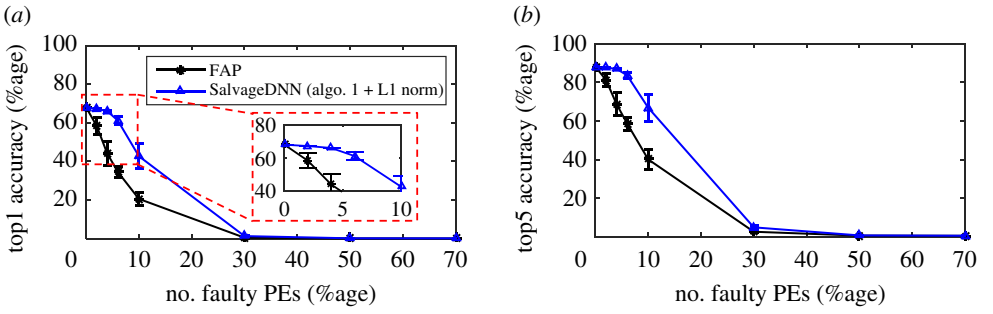

Figure 14. Comparison of SalvageDNN with the state-of-the-art FAP approach when used for the VGG11 network trained on the ImageNet dataset which has to be mapped on a 256 × 256 sized systolic array. The two commonly used accuracy metrics, i.e. the Top1 and the Top5 accuracies, are shown for cases having different number of faulty PEs, in subfigures (*a*) and (*b*) respectively. (Online version in colour.)

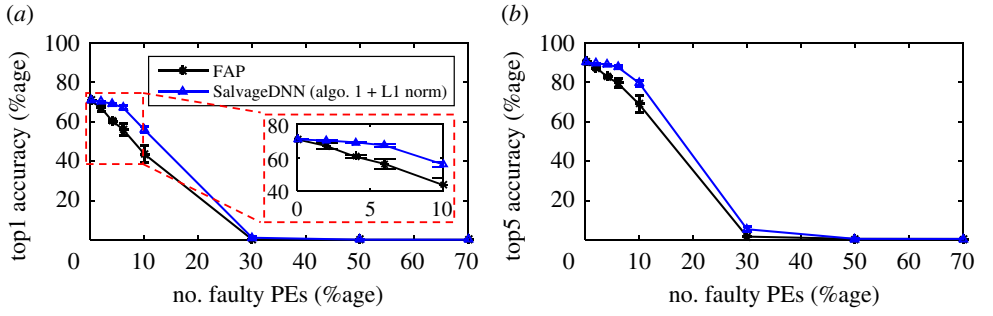

Figure 15. Comparison of SalvageDNN with the state-of-the-art FAP approach when used for the VGG16 network trained on the ImageNet dataset which has to be mapped on a 256 × 256 sized systolic array. The two commonly used accuracy metrics, i.e. the Top1 and the Top5 accuracies, are shown for cases having different number of faulty PEs, in subfigures (*a*) and (*b*) respectively. (Online version in colour.)

Table 4. Execution time comparison of SalvageDNN with retraining-based approach, i.e. FAP+T [8], for different networks trained on different datasets.

		execution time (sec)		
network	dataset	retraining (single epoch)	SalvageDNN (algo. 1 + L1-norm)	SalvageDNN speedup
VGG11	Cifar-10	24.63	1.12	21.99×
VGG11	ImageNet	6005.93	112.67	53.31×
VGG16	ImageNet	9871.38	113.26	87.16×

than the time required for a single epoch of retraining. Note that the amount of savings grow with the increase in the size and complexity of the dataset and the network. For example, the time required for SalvageDNN for the VGG16 trained on the ImageNet dataset is around 87× lesser than the time required for a single epoch of retraining.

(vi) Impact of permanent faults in the proposed systolic array designs

Figure 16 shows the impact of permanent faults on the accuracy of the VGG-11 network when executed on systolic arrays of different sizes and types (see the designs in figure 6, §3a). For these experiments, we consider that the faults can also occur in the multiplexers of a fault-tolerant PE. To have a fair comparison across arrays, we take into consideration the area of the MAC and

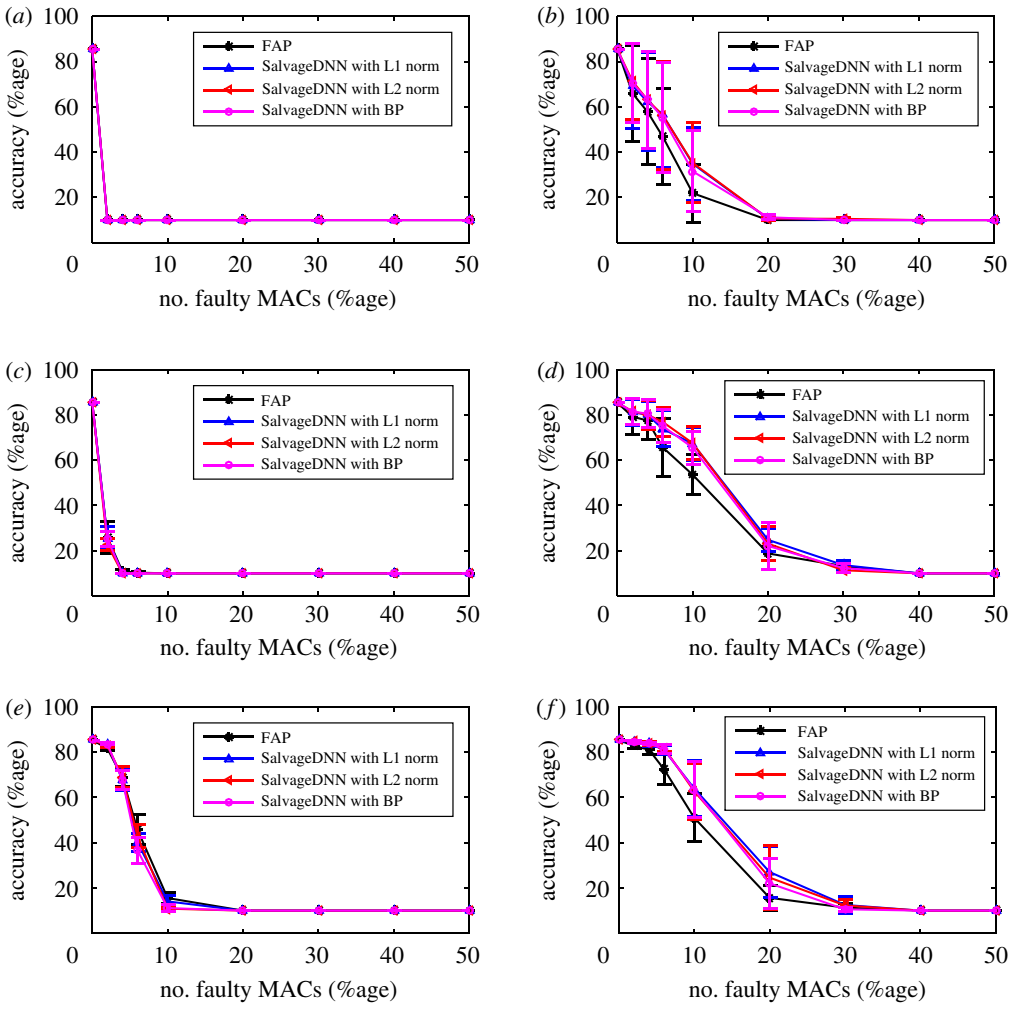

Figure 16. Impact of permanent faults in the proposed systolic array designs on the accuracy of the VGG-11 network trained on the Cifar-10 dataset. (*a*) 256 × 256 array with C_PEs, (*b*) 8 × 8 array with C_PEs, (*c*) 256 × 256 array with BNC_PEs, (*d*) 8 × 8 array with BNC_PEs, (*e*) 256 × 256 array with DBNC_PEs and (*f*) 8 × 8 array with DBNC_PEs. (Online version in colour.)

multiplexer units for injecting faults. By analysing the figure, the following observations can be made:

— SalvageDNN outperforms the state-of-the-art FAP approach in all the cases;
— the accuracy of the systolic arrays containing only C_PEs drops significantly, even at lower fault rates, as shown in figures 16*a*,*b*. This is due to the fact that a PE can get disconnected due to a fault in its downstream neighbours. Therefore, the average probability of a PE getting disconnected at a specific fault rate is significantly high in the arrays composed of *C_PEs*;
— with an increase in the number of bypass connections in a PE for fault tolerance, the average DNN accuracy at a specific fault rate increases. For example, this can be observed by analysing the DNN accuracy in figures 16*a*,*c*,*e* at where 2% of the MAC units are faulty. The *DBNC_PE* almost maintains the baseline accuracy whereas the accuracy offered by *C*_PE equals 10% (i.e. equivalent to a random selection); and
— the DNN accuracy for the same fault rate increases with the decrease in the size of the systolic array. This can be observed by comparing figures 16*a*, 16*c* and 16*e* with

figures 16*b*, 16*d* and 16*f*, respectively. This is also associated with the probability of disconnection of a MAC unit.

In summary, in order to mitigate the effects of permanent faults in the multiplexer units, designs with more bypass connections are better. However, they result in area, power/energy and delay overheads, which increase significantly with the number of bypass connections.

5. Conclusion

In this paper, we proposed a novel methodology, *SalvageDNN*, for mitigating permanent faults in DNN accelerators. The technique is based on saliency-driven fault-aware mapping of a DNN to a systolic array-based hardware accelerator without the need of re-training and without affecting the hardware level dataflow. At the core, the approach uses rearrangement of the network parameters at the software-level based on the given fault map of the hardware. This is achieved by finding the arrangement that minimizes the saliency of the parameters that will be pruned due to disconnection of the faulty MAC units / processing elements. We also proposed several hardware optimizations in the systolic array design and analysed their design trade-offs.

Data accessibility. This article does not contain any additional data.

Authors' contributions. Both authors have equal contributions to the concept development and paper writing. M.A.H. did the complete implementation of the idea.

Competing interests. We declare we have no competing interest.

Funding. This study was supported by Deutsche Forschungsgemeinschaft (GetSURE, SPP1500).

Acknowledgements. This Work is supported in parts by the German Research Foundation (DFG) as part of the GetSURE project in the scope of SPP-1500 (http://spp1500.itec.kit.edu) priority program, 'Dependable Embedded Systems'.

References

1. LeCun Y, Bengio Y, Hinton G. 2015 Deep learning. *Nature* **521**, 436. (doi:10.1038/nature14539)
2. Sze V, Chen YH, Yang TJ, Emer JS. 2017 Efficient processing of deep neural networks: a tutorial and survey. *Proc. IEEE* **105**, 2295–2329. (doi:10.1109/JPROC.2017.2761740)
3. Jouppi NP *et al.* 2017 In-datacenter performance analysis of a tensor processing unit. In *2017 ACM/IEEE 44th Annual Int. Symp. Computer Architecture (ISCA), Toronto, ON, Canada, 24–28 June 2017*, pp. 1–12. New York, NY: ACM.
4. Hanif MA, Putra RVW, Tanvir M, Hafiz R, Rehman S, Shafique M. 2018 Mpna: a massively-parallel neural array accelerator with dataflow optimization for convolutional neural networks. (http://arxiv.org/abs/1810.12910)
5. Chen YH, Yang TJ, Emer J, Sze V. 2019 Eyeriss v2: a flexible accelerator for emerging deep neural networks on mobile devices. *IEEE J. Emerg. Sel. Top. Circuits Syst. IEEE Micro* **9**, 292–308.
6. Constantinescu C. 2003 Trends and challenges in vlsi circuit reliability. *IEEE Micro* **23**, 14–19. (doi:10.1109/MM.2003.1225959)
7. Henkel J, Bauer L, Dutt N, Gupta P, Nassif S, Shafique M, Tahoori M, Wehn N. 2013 Reliable on-chip systems in the nano-era: Lessons learnt and future trends. In *Proc. 50th Annual Design Automation Conf., Austin, TX, 29 May–07 June 2013*, p. 99. New York, NY: ACM.
8. Zhang JJ, Gu T, Basu K, Garg S. 2018 Analyzing and mitigating the impact of permanent faults on a systolic array based neural network accelerator. In *2018 IEEE 36th VLSI Test Symp. (VTS), San Francisco, CA, 22–25 April 2018*, pp. 1–6. New York, NY: IEEE.
9. Koren I, Koren Z. 1998 Defect tolerance in vlsi circuits: techniques and yield analysis. *Proc. IEEE* **86**, 1819–1838. (doi:10.1109/5.705525)
10. Shivakumar P, Keckler SW, Moore CR, Burger D. 2003 Exploiting microarchitectural redundancy for defect tolerance. In *Proc. 21st Int. Conf. Computer Design, San Jose, CA, 13–15 October 2003*, pp. 481–488. New York, NY: IEEE.
11. Product binning. Online. https://en.wikipedia.org/wiki/Product_binning (accessed 20 October 2019).

12. Hruska J. Some amd rx 460s can be modded to unlock missing cores, additional performance. https://www.extremetech.com/gaming/240926-amd-rx-460s-can-modded-unlock-missing-cores-additional-performance (accessed 20 October 2019).
13. With cpu chips having billions of transistors, what happens if a few go bad? https://www.quora.com/With-CPU-chips-having-billions-of-transistors-what-happens-if-a-few-go-bad (accessed 20 October 2019).
14. Zhang L, Han Y, Xu Q, Li X. 2008 Defect tolerance in homogeneous manycore processors using core-level redundancy with unified topology. In *Proc. Conf. Design, Automation and Test in Europe, Munich, Germany, 10–14 March 2008*, pp. 891–896. New York, NY: ACM.
15. Markovsky Y, Wawrzynek J. 2007 On the opportunity to improve system yield with multi-core architectures. In *Proc. IEEE Workshop on Design Manufacturability & Yield, Santa Clara, CA, 25–26 October 2007*. New York, NY: IEEE.
16. Kim JH, Reddy SM. 1989 On the design of fault-tolerant two-dimensional systolic arrays for yield enhancement. *IEEE Trans. Comput.* **38**, 515–525. (doi:10.1109/12.21144)
17. Kung H, Lam MS. 1983 Fault-tolerance and two-level pipelining in vlsi systolic arrays. Technical report, CARNEGIE-MELLON UNIV PITTSBURGH PA DEPT OF COMPUTER SCIENCE.
18. Agrawal VD, Kime CR, Saluja KK. 1993 A tutorial on built-in self-test. 2. Applications. *IEEE Des. Test Comput.* **10**, 69–77. (doi:10.1109/54.211530)
19. Forlenza DO, Forlenza OP, Robbins BJ. 2016 *Logic-built-in-self-test diagnostic method for root cause identification*. US Patent 9,244,757.
20. Zmora N, Jacob G, Zlotnik L, Elharar B, Novik G. 2018 Neural network distiller. (doi:10.5281/zenodo.1297430)
21. Li H, Kadav A, Durdanovic I, Samet H, Graf HP. 2016 Pruning filters for efficient convNets (http://arxiv.org/abs/1608.08710).
22. Yu R, Li A, Chen CF, Lai JH, Morariu VI, Han X, Gao M, Lin CY, Davis LS. 2018 Nisp: Pruning networks using neuron importance score propagation. In *Proc. IEEE Conf. Computer Vision and Pattern Recognition, Salt Lake City, UT, 18–23 June 2018*, pp. 9194–9203. New York, NY: IEEE.
23. Krizhevsky A, Sutskever I, Hinton GE. 2012 Imagenet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems, Lake Tahoe, NV, 3–6 December 2012*, pp. 1097–1105. Red Hook, NY: Curran Associates Inc.
24. Powell MD, Biswas A, Gupta S, Mukherjee SS. 2009 Architectural core salvaging in a multi-core processor for hard-error tolerance. In *ACM SIGARCH Computer Architecture News, Austin, TX, 20–24 June 2009*, vol. 37, pp. 93–104. New York, NY: ACM.
25. Nakahara S, Higeta K, Kohno M, Kawamura T, Kakitani K. 1999 Built-in self-test for GHz embedded SRAMs using flexible pattern generator and new repair algorithm. In *Int. Test Conf. 1999. Proc. (IEEE Cat. No. 99CH37034), Atlantic City, NJ, 30–30 September 1999*, pp. 301–310. New York, NY: IEEE.
26. Schuchman E, Vijaykumar T. 2005 Rescue: A microarchitecture for testability and defect tolerance. In *32nd Int. Symp. Computer Architecture (ISCA'05), Madison, WI, 4–8 June 2005*, pp. 160–171. New York, NY: IEEE.
27. Abraham JA, Banerjee P, Fuchs W, Reddy AN *et al.* 1987 Fault tolerance techniques for systolic arrays. *Computer* **20**, 65–75. (doi:10.1109/MC.1987.1663621)
28. Krizhevsky A *et al.* 2009 Learning multiple layers of features from tiny images. Technical Report, Citeseer.
29. Deng J, Dong W, Socher R, Li LJ, Li K, Fei-Fei L. 2009 Imagenet: a large-scale hierarchical image database. In *2009 IEEE Conference on Computer Vision and Pattern Recognition, Miami, FL, 20–25 June 2009*, pp. 248–255. New York, NY: IEEE.