NASCaps: A Framework for Neural Architecture Search to Optimize the Accuracy and Hardware Efficiency of Convolutional Capsule Networks

ABSTRACT

Deep Neural Networks (DNNs) have made significant improvements to reach the desired accuracy to be employed in a wide variety of Machine Learning (ML) applications. Recently the Google Brain's team demonstrated the ability of *Capsule Networks (CapsNets)* to encode and learn spatial correlations between different input features, thereby obtaining superior learning capabilities compared to traditional (i.e., non-capsule based) DNNs. However, designing CapsNets using conventional methods is a tedious job and incurs significant training effort. Recent studies have shown that powerful methods to automatically select the best/optimal DNN model configuration for a given set of applications and a training dataset are based on the *Neural Architecture Search (NAS)* algorithms. Moreover, due to their extreme compute and memory requirements, DNNs are employed using the specialized hardware accelerators in IoT-Edge/CPS devices.

In this paper, we propose **NASCaps**, an automated framework for the hardware-aware NAS of different types of DNNs, covering both traditional convolutional DNNs and CapsNets. We study the efficacy of deploying a multi-objective Genetic Algorithm (e.g., based on the NSGAA-II algorithm). The proposed framework can jointly optimize the network accuracy and the corresponding hardware efficiency, expressed in terms of energy, memory, and latency of a given hardware accelerator executing the DNN inference. Besides supporting the traditional DNN layers (such as, convolutional and fully-connected), our framework is the first to model and supports the specialized capsule layers and dynamic routing in the NAS-flow. We evaluate our framework on different datasets, generating different network configurations, and demonstrate the tradeoffs between the different output metrics. We will open-source the complete framework and configurations of the Pareto-optimal architectures at https://BlindedLink.

KEYWORDS

Deep Neural Networks, DNNs, Capsule Networks, Evolutionary Algorithms, Genetic Algorithms, Neural Architecture Search, Hardware Accelerators, Accuracy, Energy Efficiency, Memory, Latency, Design Space, Multi-Objective, Optimization.

1 INTRODUCTION

Deep Neural Networks (DNNs) are advanced machine learning-based algorithms that claim to surpass the human-level accuracy in a certain set of tasks, such as image classification, object recognition, detection, and tracking, when extensively trained over large datasets [3][6][22]. Designing the best DNN for a given set of applications and a given dataset is an extremely challenging effort. Highly-accurate DNNs require a significant amount of hardware resources, which may not be feasible on resource-constrained IoT-Edge devices [15][27]. The advanced DNN models, called Capsule Networks (CapsNets) [23], are able to learn the hierarchical and spatial information of the input features in a closer manner to our current understanding of the human brain's functionality. However, the layers made of capsules introduce an additional dimension w.r.t. the matrices of the convolutional and fully-connected layers of traditional Convolutional Neural Networks (CNNs), which, besides the dynamic routing computations, put an extremely heavy computation and communication workload on the underlying hardware [14]. In Fig. 1 we compare the CapsNet [23] with the LeNet [12] and the AlexNet [11], and analyze their memory footprints and total number of multiply-and-accumulate (MAC) operations required to perform an inference pass. Note, the MACs/memory ratio is a good indicator to show the computational complexity of the network, thus demonstrating the higher compute-intensive nature of CapsNets, compared to traditional CNNs.

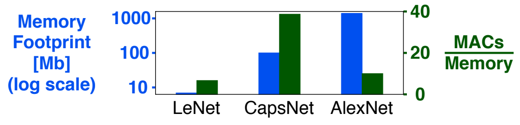


Figure 1: Memory footprint and (Multiply-and-Accumulate operations vs. memory) ratio (MACs/Memory) for the LeNet [12], CapsNet [23] and AlexNet [11].

1.1 Research Problem & Associated Challenges

In the literature, many DNN models/architectures have been proposed [5][11][24][28][29]. In the pioneering age of deep learning, the DNN architectures were designed manually. However, their structures became very complex. Therefore, Neural Architecture Search (NAS) methodologies emerged as an attractive procedure to select the optimal DNN model for a given set of applications and a training dataset [32]. Evolutionary Algorithm (EA) based methodologies [26] were proposed for learning small DNNs. However, for more complex DNNs, EAs have been used for improving certain parameters of single/individual DNN layers [18], as well as global DNN hyperparameters [17], or the connection between submodules [31].

Most of the automatic tools based on a NAS algorithm that have been proposed in the literature only focus on optimizing the DNN accuracy [26][33]. Only a few of them have recently introduced the hardware constraints in the optimization problem, for instance, considering the hardware resources (e.g., #FLOPs, memory requirements, etc.) available for performing the DNN inference [8][9][13][25]. To the best of our knowledge, none of them include in the design space the possibility of employing capsule layers and dynamic routing, which are inevitable for automatically designing the CapsNets.

Toward this, we propose **NASCaps**, a framework for the NAS of DNNs, that not only incorporates the most common types of DNN layers (such as convolutional, fully-connected) but also, *for the first time, the different types of capsule layers*. Our framework supports multi-objective hardware-aware optimizations because it investigates the network accuracy, and it accounts for different hardware efficiency parameters (such as memory usage, energy consumption, and latency) that are crucial for embedded DNN inference accelerators.

However, the huge variety of possible configurations that should be explored to obtain an exhaustive set of Pareto-optimal solutions might dramatically explode. In addition to this, despite adopting the most advanced learning policies and employing high-end GPU clusters, complex CapsNets and CNNs typically require long training time [4][16]. Complete detailed post-synthesis hardware measurements are not feasible for this search due to their long simulation times. The above-discussed limitations challenge the applicability of such an exploration in real-case HW/SW co-design searches, with stringent time-to-market constraints.

1.2 Our Key Research Contributions

To address the above challenges, we devise different optimizations and integrate them into our *NASCaps* framework (Fig. 2). The steps are summarized in the following **novel contributions**:

- We present a framework, called *NASCaps*, to automatically search the DNN model architecture configurations, based on convolutional layers and capsule layers. (**Section 3**)
- We model the operations involved in the CapsNet architectures in a parametric way, including the different types of capsule layers and the dynamic routing. (**Section 3.1**)
- We model the functional behavior of a given specialized CNN and CapsNet hardware accelerator at a high level, to quickly estimate the memory usage, energy consumption, and latency, when different DNN architectural models are executed. (**Section 3.2**)
- Based on the NSGA-II method [2], we developed a specialized multi-objective genetic algorithm for solving the optimization problem targeted in this paper, i.e., a multi-objective Pareto-frontier selection of DNN architectures while optimizing the neural network's accuracy, energy consumption, memory usage, and latency. (**Section 3.3**)
- To reduce the training time for the exploration of different solutions, we propose a methodology to evaluate the accuracy of partially-trained DNNs. The number of training epochs is chosen based on the tradeoff between training time and Pearson correlation coefficient w.r.t. fully-trained DNNs. (**Section 4.2**)
- During the exploration phase, we trained and evaluated more than 600 candidate DNN solutions running on the GPU-HPC computing nodes equipped with four high-end Nvidia V100-SMX2 GPUs. The Pareto-optimal solutions generated by our *NASCaps* framework are competitive w.r.t. the previous SoA accuracy values for CapsNets, i.e., the DeepCaps [21], while improving the corresponding hardware efficiency, thereby opening new avenues towards the deployment of high-accurate DNNs at the edge. (**Section 4.4**)
- Towards encouraging fast advancements in the DNN research community, we will open-source the complete *NASCaps* framework and configurations of the Pareto-optimal architectures at https://BlindedLink.

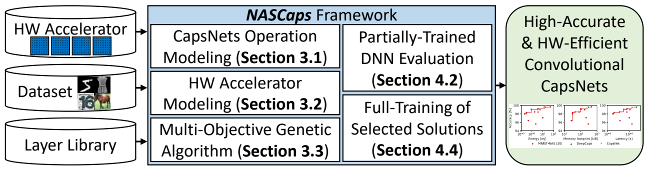

HW Accelerator
Dataset
Layer Library
NASCaps Framework
CapsNets Operation Modeling (Section 3.1)
HW Accelerator Modeling (Section 3.2)
Multi-Objective Genetic Algorithm (Section 3.3)
Partially-Trained DNN Evaluation (Section 4.2)
Full-Training of Selected Solutions (Section 4.4)
High-Accurate & HW-Efficient Convolutional CapsNets

Figure 2: Overview of our *NASCaps* framework.

Before proceeding to the technical sections, we provide a brief overview of the CapsNets and the hardware accelerators executing CapsNet inference. (**Section 2**)

2 BACKGROUND: CAPSULE NETWORKS

The idea of grouping the neurons to form a *capsule* was first proposed by G. Hinton et al. in [7]. The purpose of the capsules is to retain the instantiation probability of an entity or a feature, together with information on its instantiation parameters, such as the position, rotation, or width. On the contrary, traditional neurons can only detect features without any knowledge of their spatial characteristics and inter-correlation.

The first DNN with capsules, i.e., a *Capsule Network* (CapsNet), was proposed in [23] by the Google Brain's team. In this model, the neurons inside a capsule are arranged into the shape of a vector. Each neuron encodes a spatial parameter of the entity associated with the capsule, and the length of the vector represents the probability that the object is present. The CapsNet in [23] consists of three layers, as shown in Fig. 3.a:

(1) **Conv Layer**: a convolutional (Conv) layer that applies a 9x9 kernel with stride 1 to the input image and produces 256 output channels.
(2) **PrimaryCaps layer**: a Conv layer that applies a 9x9 kernel with stride 2 and produces 256 output channels. The output channels are divided into 32 channels of 8-D capsules. Since the length of the capsules encodes a probability, the *squash function* (Eq. 1) is applied to force the length in the range [0,1].

$$y = \frac{|\mathbf{x}|^2}{(1+|\mathbf{x}|)^2}\frac{\mathbf{x}}{|\mathbf{x}|} \tag{1}$$

(3) **DigitCaps layer**: a fully-connected (FC) layer of capsules with 10 output 16-D capsules (for a 10-classes dataset). The DigitCaps layer performs the *dynamic routing*, an iterative algorithm that associates coupling coefficients to the predictions obtained from the PrimaryCaps layer. The iterations of the algorithm maximize the coupling coefficients of the capsules predicting the same result (similar spatial parameters) with greater confidence (higher probability of instantiation).

CapsNets belong to a particular category of DNNs, i.e., *autoencoders*, that can reconstruct the image they receive as input. Therefore, the three layers listed above are followed by a decoder, consisting of three FC layers that reconstruct the input image from the output of the DigitCaps layer. During training, two losses are computed, the loss of the classification network and

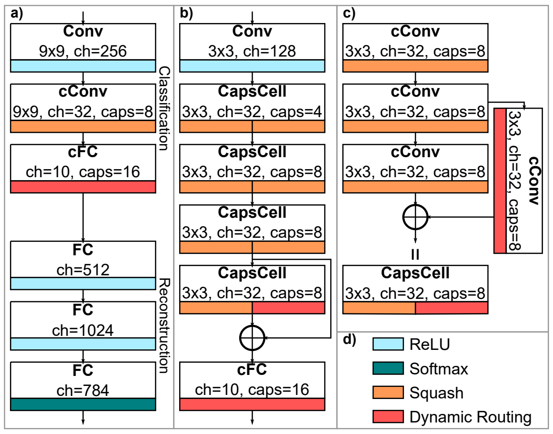


Figure 3: (a) CapsNet model [23] with the decoder for image reconstruction; (b) DeepCaps model [21] (the decoder is omitted); (c) CapsCell used in the DeepCaps model; (d) legend of the activation functions used in each layer.

the reconstruction network. When the inference is performed for classification purposes, the reconstruction network can be removed.

The DeepCaps, a novel deeper CapsNet architecture, has been recently proposed in [21]. The architecture, shown in Fig. 3.b, is formed by a traditional Conv layer, FC capsule layers (cFC), and Conv capsule layers (cConv). The latter are arranged in blocks, here referred to as *CapsCells* (Fig. 3.c), where a cConv layer runs in parallel to two cConvs layers. The decoder used as a reconstruction network consists of a series of de-convolutional layers. The DeepCaps also introduce a skip connection between the last CapsCells to mitigate the vanishing gradient effects that affect deeper networks.

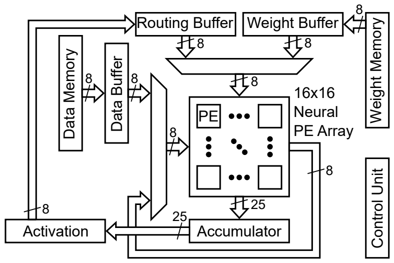


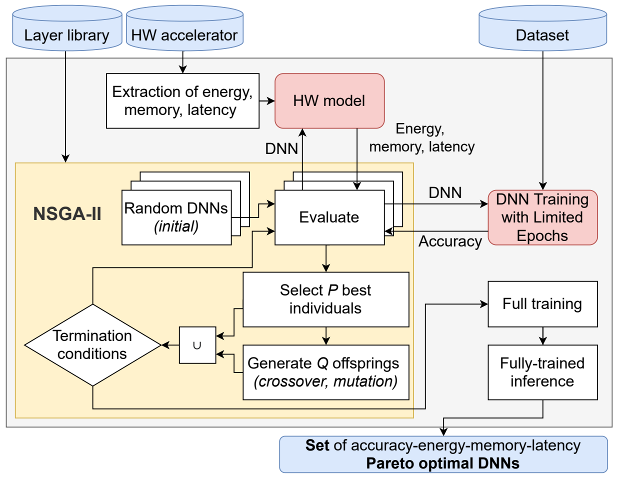
Figure 4: Architectural view of the CapsAcc [14] accelerator.

The capsule layers involve operations that are not performed by the traditional DNNs, and consequently, they are not supported by already existing DNNs hardware accelerators. Moreover, CapsNets require modified data mapping. CapsAcc, a hardware platform targeting CapsNets acceleration, is proposed in [14] and shown in Fig. 4. The computational core of CapsAcc consists of an array of processing elements (PEs), followed by an accumulator that properly adds the partial sums. There is then an activation unit that can apply ReLU, softmax, or squash functions to the output of the accumulator. The activations and weights are stored in data and weight memories, respectively, and there are three buffers used during the computation to exploit data reuse and minimize the accesses to the larger memories. In particular, a data and a weight buffer store the activations and the weights, and a routing buffer is used to store partial results of the dynamic routing iterations. A control unit selects the paths for the mapping of different layers onto the PE array.

3 NASCAPS: NEURAL ARCHITECTURE SEARCH OF CONVOLUTIONAL CAPSNETS

Our multi-objective *NASCaps* framework generates and evaluates convolutional- and capsule-based DNNs, by performing a multi-objective NAS, to find a set of accurate yet resource-efficient DNN models, i.e., jointly considering the DNN validation accuracy, energy consumption, latency, and memory footprint. The search is based on our specialization of the genetic NSGA-II algorithm [2], to enable a search with multi-objective comparison and selection among the generated candidate DNNs.

The overall structure and workflow of the *NASCaps* framework is shown in Fig. 5. As input, it receives the configuration of the underlying hardware accelerator (that would execute the generated DNN in the real-world scenario) and a given dataset used for DNN training, as well as a collection of the possible types of layers that can be used to form different candidate DNNs. First, we create a layer library that includes convolutional layers, capsule layers (as defined in [23]), and the CapsCell and FlatCaps layers defined in [21]. We envision that, due to the modular structure of our framework, other types of layers can easily be integrated into its future versions to further extend the search-space, also thanks to the use of a simple modular representation of the candidate networks relying on the combination of single-layer descriptors, as discussed in Section 3.1.



Layer library
HW accelerator
Dataset
Extraction of energy, memory, latency
HW model
DNN
Energy, memory, latency
NSGA-II
Random DNNs (initial)
Evaluate
DNN
DNN Training with Limited Epochs
Accuracy
Select P best individuals
Termination conditions
∪
Generate Q offsprings (crossover, mutation)
Full training
Fully-trained inference
Set of accuracy-energy-memory-latency Pareto optimal DNNs

Figure 5: Overview of our *NASCaps* framework, showing different components and their interconnections defining the workflow.

The automated search is initialized with N randomly-generated DNNs used as input to start the evolutionary search process. Each candidate DNN is evaluated in terms of its validation accuracy after being trained for a limited number of epochs. As we will discuss in Section 4.2, this optimization is designed to curtail the computational cost and to reduce the required computational time for the search, while keeping a good level of correlation w.r.t. the full-training accuracy, measured with the Pearson correlation coefficient. Moreover, each DNN under test is also characterized for its energy consumption, latency, and memory footprint, by modeling its inference processing considering the final real-world use case of executing the generated DNN on a specialized DNN hardware accelerator. At this evaluation point, the genetic algorithm proceeds to the next step, finding at each iteration a new Pareto-frontier that contains the best candidate DNN solutions. At the end of this selection process, the Pareto-optimal DNN solutions are fully-trained[1], to make an exact accuracy evaluation. In the following sub-sections, we discuss the key components of our framework in detail.

3.1 Parametric Modeling of Capsule Network Layers and Architectures

The proposed genetic-based *NASCaps* framework is relying on an explicit position-based representation for each layer of the candidate DNNs. This representation allows to define the key parameters of each layer uniquely.

The DNN layers have been constructed using a *layer descriptor*, which encodes the information needed to build and model a given candidate network, in a very compact form. Each layer descriptor is a 9-element position-based structure, thus guaranteeing the modularity for constructing any given candidate DNN architecture. **The elements of the layer descriptor are listed as follows:**

(1) type of layer,

[1]A complete training up to the 100th epoch for the MNIST, Fashion-MNIST, and SVHN datasets, and up to the 300th epoch for the CIFAR-10 dataset is conducted.

(2) size of the input feature maps n_{in},
(3) number of input channels ch_{in},
(4) number of input capsules $caps_{in}$,
(5) kernel size $kernel_{size}$,
(6) stride size $stride_{size}$,
(7) size of the output feature maps n_{out},
(8) number of output channels ch_{out},
(9) number of output capsules $caps_{out}$.

Such a representation allows to describe even more complex structures by simply defining a new layer *type*. For instance, a layer descriptor can define a more complex repeating structure, composed of multiple elements, like a CapsCell in the DeepCaps architecture. In this way, the DeepCaps architecture has been described with six layer descriptors. The first one for the single convolutional layer, four CapsCell blocks, and a final Class Capsule layer.

The complete DNN architecture description is then completed by two non-layer terms, that allow to encode the position of a skip connection, and an indicator, called *resize flag*, to explicitly indicate if the resizing of the inputs is required. Fig. 6 shows the format proposed to describe a candidate DNN architecture, which is, from now on, referred to as the *genotype*.

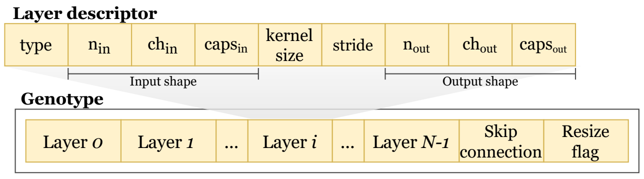


Figure 6: Proposed structure of the genotype.

3.2 Modeling the Execution of CapsNets in Hardware Accelerators

The *NASCaps* framework can receive any given hardware accelerator executing DNN inference as an input. For illustration, we showcase the modeling of the CapsAcc accelerator; this choice is related to the fact that it supports the execution of all the capsule layers. Starting from the RTL-level description of the CapsAcc architecture, we extract and model the different micro-architectural configurations at a higher abstraction level, which constitutes the inputs for our model. First, the description of the operation-specific parameters of the layers is presented. Afterward, the global parameters, that are strictly related to the CapsAcc accelerator, are discussed.

3.2.1 Operation-Specific Modeling for different Layers.

The operation-specific parameters that can be extracted from the execution of different operations in the hardware are the following:

- *weights*: number of weights in the layer,
- *sums_per_out*: number of terms to be added for an output value,
- *data_per_weight*: number of feature maps that are multiplied by the same weight.

For each operation, these parameters are computed by different equations, due to the different nature of the respective types of computations, see Table 1. Note that, by setting $caps_{in}$ and $caps_{out}$ to 1, the *ConvCaps* and *ClassCaps* layers become a traditional convolutional layer and fully-connected layer, respectively.

3.2.2 Global Parameter Modeling.

Our models estimate the latency and the energy consumption of the inference of one input, for a given *CapsNet*, while the memory footprint is computed as the sum of the number of weights for each layer. They are modeled for each operation in a modular way (i.e., bottom-up). First, the weights must be loaded onto the PE array, then reused as long as they need to be multiplied by other inputs. Afterward, the next group of weights is loaded until all the computations of the layers are done (see Eqs. 2-4). The adopted model parameters are the following:

- *w_load_cycles*: number of clock cycles required to load the weight onto the PE array,
- *w_loads*: number of groups of weights loaded onto the PE array,
- $cycles(l)$: number of cycles required to execute the layer l,
- *ma*: number of memory accesses,
- en_{mem}: energy consumption of a single memory accesses,
- pwr_{PEA}: power consumption of the PE array.

$$w\_load\_cycles = 16 \tag{2}$$

$$w\_loads = \left\lceil \frac{weights}{16 \cdot \min(16, sums\_per\_out)} \right\rceil \tag{3}$$

$$cycles(l) = w\_load\_cycles \cdot w\_loads + data\_per\_weight \tag{4}$$

The overall latency is then computed as the sum of the contributions of the layers (Eq. 5).

$$latency = \sum_{l \in L} cycles(l) \cdot T \tag{5}$$

In the Eq. 6, the number of memory accesses is computed by distinguishing whether the operation is a convolutional layer or not. Such a distinction has been implemented by analyzing the value of *data_per_weight*, which is greater than 1 for convolutional layers and 1 otherwise.

$$ma = \begin{cases} 256, & \text{if } data\_per\_weight = 1 \\ 16 \cdot \max(sums\_per\_out - 15, 1), & \text{otherwise} \end{cases} \tag{6}$$

The energy of the accelerator (Eq. 7) is estimated as the sum of the energy of memory accesses and the sum of the power consumption of each layer processed in the PE array, multiplied by its latency (period T and the number of cycles). Note that the average power consumption of the PE array is used in our model.

$$energy = \left\lceil \frac{ma \cdot 8}{128} \right\rceil \cdot en_{mem} + \sum_{l \in L} cycles(l) \cdot T \cdot pwr_{PEA} \tag{7}$$

3.3 The Multi-Objective NSGA-II Algorithm

The selection of the Pareto-optimal solutions for the *NASCaps* framework is based on the evolutionary algorithm NSGA-II [2]. It has a main loop (lines 2-14 of Algorithm 1) whose iterations represent a single generation of the overall evolution process of an initial population. The initial population (sized n) is randomly generated and can be referred to as P_1 (line 1 of Algorithm 1). This set of solutions represents the initial parent generation of the algorithm. The crossover among the solutions belonging to P_t (line 3) allows the generation of a new set of offspring individuals Q_t. At this point, the population $P_t \cup Q_t$ is sorted according to a non-domination criterion. For each iteration of the inner loop (lines 6-13), the candidate solutions are grouped into different fronts F_i. The ones included in the first front F_1 represent the best-found solutions of the overall population. Each subsequent front $(F_2, F_3, \dots)$ is instead constructed by removing all the preceding fronts from the population and then finding a new Pareto-front.

Table 1: Equations for the operation-specific modeling of CapsNets.

Operation	*weights*	*sums_per_out*	*data_per_weight*
ConvCaps layer	$(ch_{in} \cdot kernel_{size}^2 + 1) \cdot ch_{out} \cdot caps_{out} \cdot caps_{in}$	$(kernel_{size}^2 + 1) \cdot ch_{in} \cdot caps_{in}$	$(n_{out})^2 \cdot ch_{in} \cdot caps_{in}$
ConvCaps3D layer	$(ch_{in} \cdot kernel_{size}^3 + 1) \cdot ch_{out} \cdot caps_{out} \cdot caps_{in}$	$(kernel_{size}^3 + 1) \cdot ch_{in} \cdot caps_{in}$	$(n_{out})^2 \cdot ch_{in} \cdot caps_{in}$
ClassCaps layer	$(ch_{in} \cdot n_{in}^2 + 1) \cdot ch_{out} \cdot caps_{out} \cdot caps_{in}$	$(n_{in}^2 + 1) \cdot ch_{in} \cdot caps_{in}$	1
Dynamic Routing	$ch_{in} \cdot kernel_{size}^2 \cdot ch_{out}$	$caps_{in}$	1

Since the first front may be composed of less than n individuals, also the solutions from subsequent fronts will be selected to be part of the next parent generation.

To have exactly n parents in the output set, the solutions that are part of the last front are ranked using the crowded distance comparison approach (line 11), which consists of sorting the population of that front according to each objective function value in ascending order. These steps are shown in Fig. 7. Only half of the population becomes part of the next parent generation, while the other half is discarded.

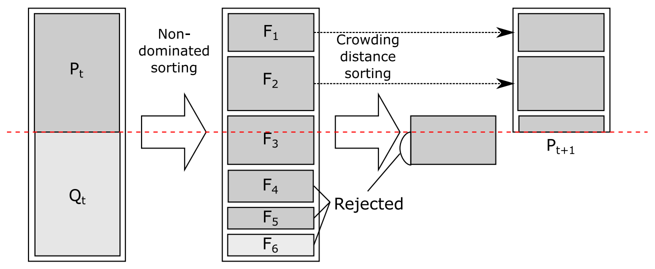


Figure 7: Sorting of the population.

These steps repeat for a certain number g of generations. The complete pseudocode is reported in Algorithm 1, where the following procedures are used:

- $RandomConfigurations(n)$ randomly generates n configurations belonging to the search space.
- $CrossoverAndMutate(X, n)$ generates n new offsprings from parents P by crossover and mutation (described in Section 3.3.1).
- $EstimateParameters(X)$ evaluates the new candidate solutions from a set X.
- $PickPareto(X)$ selects the Pareto-optimal solutions from a set X, and these solutions are removed from the set.
- $DistanceCrowding(X, n)$ returns n solutions from a set X (as described in [2]).

The advantage of a multi-objective algorithm lies in the fact that it re-constructs the Pareto-front at each generation, aiming to cover all the possible solutions. The algorithm's output is a set of non-dominated solutions.

3.3.1 *Crossover and mutation operations.*

The two key operators in the progression of a genetic algorithm are crossover and mutation. The standard single-point **crossover** operation allows to generate the offspring solutions, given two parent solutions P_a and P_b that have been previously randomly picked among the current population candidates. The genotypes of the two parent individuals are split into two parts each. The splitting point is pseudo-randomly selected. Initially, a cut point is randomly chosen. Then, a series of checks are performed to verify the validity of the output genotypes. The following criteria have been applied to choose the splitting point correctly:

Algorithm 1 : The genetic NSGA-II algorithm used in our *NASCaps* framework.

```
Require: search space $S$, sizes of population $|P|, |Q|$, number of generations $g$
Ensure: Pareto set $F \subseteq P_1 \times P_2 \times \cdots \times P_k$
 1: $P_1 \leftarrow RandomConfigurations(|P|)$
 2: for $g = 1 \ldots G$ do
 3:     $Q_i \leftarrow CrossoverAndMutate(P_i, |Q|)$
 4:     $T \leftarrow EstimateParameters(P_i \cup Q_i)$
 5:     $P_{i+1} \leftarrow \emptyset$
 6:     while $|P_{i+1}| < |P|$ do
 7:         $F = PickPareto(T)$
 8:         if $|P_{i+1}| + |F| \leq |P|$ then
 9:             $P_{i+1} \leftarrow P_{i+1} \cup F$
10:         else
11:             $P_{i+1} \leftarrow P_{i+1} \cup DistanceCrowding(F, |P| - |P_{i+1}|)$
12:         end if
13:     end while
14: end for
15: return $PickPareto(P_g)$
```

- the cut-points are chosen to ensure that the generated DNN is made up of at least one initial convolutional layer and a minimum of 2 capsule layers,
- no convolutional layer is placed between two capsule layers.

Note, the reason behind the second constraint lies in the fact that capsules aim to derive higher-level information w.r.t. convolutional layers. At this point, the actual crossover operation is performed. As shown in Fig. 8, the last parts of the parent genes P_a and P_b are switched.

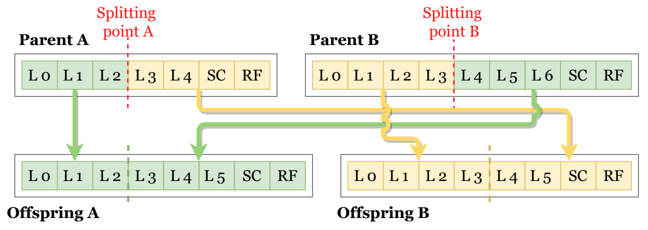


Figure 8: Example of crossover between two genotypes.

The second key operation performed by the algorithm is **mutation**. As it has been implemented for our *NASCaps* framework, the operator performs a mutation by randomly choosing one of the layer descriptors from the genotype of the input candidate network, and by randomly modifying one of the main parameters of the selected layer with a probability p_m. In particular, the parameters that can be affected by a mutation are the kernel size, the strides, the number of output capsules, and the position of the skip connection.

After these two operations, a further step is performed to ensure the validity of the output genotypes that, in a large number of cases, will represent an invalid DNN. This correction step allows to properly adjust the input and output tensor dimensions for every layer for genotypes derived from a mutation or a crossover operation, which can randomly modify or join different parent genotypes together.

4 EVALUATING OUR NASCAPS FRAMEWORK

4.1 Experimental Setup

The overview of our experimental setup and tool-flow is shown in Fig. 9. The training and testing for accuracy of the candidate DNNs have been conducted with the TensorFlow library [1], while extensive experiments are performed using the GPU-HPC computing nodes equipped with four NVIDIA Tesla V100-SXM2 GPUs. Our proposed *NASCaps* framework has been evaluated for the MNIST [12], Fashion MNIST (FMNIST) [30], SVHN [19] and CIFAR-10 [10] datasets. The implementation of the HW model is based on an open-source SoA Capsule Network accelerator (CapsAcc) [14], as described in Section 3.2. The core processing elements were synthesized using the Synopsys Design Compiler with a 45nm technology node and a clock period T of 3ns.

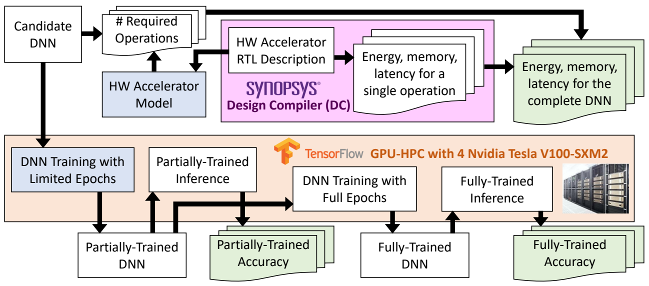

Candidate DNN
# Required Operations
HW Accelerator Model
HW Accelerator RTL Description
Energy, memory, latency for a single operation
SYNOPSYS
Design Compiler (DC)
Energy, memory, latency for the complete DNN
TensorFlow
GPU-HPC with 4 Nvidia Tesla V100-SXM2
DNN Training with Limited Epochs
Partially-Trained Inference
DNN Training with Full Epochs
Fully-Trained Inference
Partially-Trained DNN
Partially-Trained Accuracy
Fully-Trained DNN
Fully-Trained Accuracy

Figure 9: Setup and tool-flow for conducting our experiments.

The experiments were divided into three steps. (1) In the first step, a basic random search has been performed, to investigate how many training epochs are necessary to train the candidate DNNs and evaluate their accuracy in the loop of the genetic NSGA-II algorithm. (2) During the second step, the search algorithm for finding Pareto-optimal DNN architectures for the energy, memory, latency, and accuracy objectives is executed. (3) Finally, the selected Pareto-optimal DNNs have been fully-trained. To evaluate the transferability of the selected DNNs w.r.t. different datasets, the selected DNNs have been fully-trained also for the other datasets. Moreover, the following settings have been used to conduct the experiments:

- Initial parent population size $|P| = 10$
- Offspring population size: $|Q| = 10$
- Maximum number of generations for the genetic loop $g = 20$
- Mutation probability: $p_m = 10\%$
- $kernel_{size} \in \{3 \times 3, 5 \times 5, 9 \times 9\}$
- $stride_{size} \in \{1, 2\}$
- $ch_{out} = \{1, 2, \ldots, 64\}$
- $caps_{out} = \{1, 2, \ldots, 64\}$

4.2 Results for Reduced Training Epochs for Full-Training Accuracy Estimation

One of the most crucial aspects linked to the NAS problem lies in its high computational exploration cost. This is due to the *large number of candidate networks* that constitute the population and the *time-consuming training steps needed to evaluate the accuracy.* To limit the time needed to perform the complete search and consequently, its computational cost, we propose a two-stage evaluation approach. (1) The first step consists of training the population of candidate networks with a limited number of epochs, producing a set of partially-trained DNNs. The validation accuracy obtained by the partially-trained DNNs has been used for the evaluation of the Pareto-fronts in the NSGA-II algorithm, as discussed in Section 3.3. The choice of the number of epochs has been determined carefully by analyzing the impact of different epoch sizes over the achieved accuracy for different datasets. (2) Afterward, the candidate networks that show their accuracy and hardware efficiency in a Pareto-front are fully-trained to evaluate their actual validation accuracy.

Hence, this approach allowed to use only a reduced number of training epochs to predict the full-training accuracy of the DNNs. *This approach has been tested using 66 randomly generated DNNs (in addition to CapsNet and DeepCaps architectures) and performing a full-training on them*, while recording the obtained validation accuracy at each training epoch. The Pearson correlation coefficient (*PCC*) [20] has been computed to analyze the correlation between the accuracy of the fully-trained DNNs and the accuracy of the same DNNs at the intermediate steps.

Table 2 shows the values of the *PCC*, computed between the accuracy of the DNNs after n training epochs and their accuracy after a full training. The median cumulative training time needed to perform an n epoch training is also reported. This study allowed to determine, as expected, that more complex datasets require a larger number of training epochs to distinguish the most promising networks from the rest correctly. For the case of the MNIST dataset, 5 epochs are sufficient to reach a *PPC* equal to 0.9999. Instead, for the CIFAR-10 dataset, such a high value of confidence is never reached within the first few epochs. In this case, 10 training epochs are selected, which ensure a *PCC* equal to 0.9334. This choice leveraged the tradeoff between the correlation coefficient and the required training time. Of course, a larger number of training epochs can also be selected, but it would drastically increase the exploration time due to the DNN training, which is a crucial parameter to consider when large populations and/or number of generations are explored by the *NASCaps* framework. On the other hand, the selection performed after 10 epochs of training allowed to discard more Pareto-dominated candidate networks than what would have been discarded after 5 epochs. For the Fashion-MNIST and SVHN datasets, the selection stage has also been performed after 5 epochs of training.

Table 2: Pearson correlation coefficient (PCC) and median cumulative training time expressed in seconds (MCTT) for the MNIST, Fashion-MNIST (FMNIST), SVHN and CIFAR-10 datasets.

Epoch n.		**1**	**3**	**5**	**10**	**15**	**20**
MNIST	PCC	0.8407	0.9998	0.9999	1.0000	1.0000	1.0000
	MCTT	55.4	166.2	277.0	554.0	831.0	1108.0
FMNIST	PCC	0.8306	0.8963	0.9013	0.9935	0.9989	0.9998
	MCTT	86.2	258.7	431.1	862.3	1293.4	1724.6
SVHN	PCC	0.6812	0.8733	0.9518	0.9531	0.9667	0.9876
	MCTT	128.3	385.0	641.6	1283.3	1924.9	2666.6
CIFAR-10	PCC	0.2969	0.4259	0.7279	0.9334	0.9518	0.9879
	MCTT	61.6	184.7	307.9	615.8	923.6	1231.5

Note, a certain set of networks can be discarded relatively early, i.e., after a few training epochs, since they do not improve their accuracy much. The candidate networks that pass the selection stage can then complete their training. A second selection stage is beneficial for performing a more fine-grained selection of the candidate networks, and avoiding the tedious and computational-hungry full-training of Pareto-dominated DNNs.

4.3 NASCaps Results for the Partially-Trained DNNs

Our *NASCaps* framework is first applied to the MNIST dataset to evaluate its efficiency and correct behavior. The number of generations is set at 20, but a maximum time-out of 12 hours has been imposed in the cases of the MNIST and Fashion-MNIST datasets, while a 24-hour maximum search time has been used for the CIFAR-10 and SVHN datasets.

The search for the *MNIST-NAS* lasted for 20 complete generations, and the single candidate networks were trained for 5 epochs. This setup led to train and evaluate a total of 210 DNNs. The resulting individual solutions are compared to the two reference SoA solutions, that are the CapsNet and DeepCaps architectures. In Fig. 10a, each individual DNN architecture is represented w.r.t. the four objectives of the search.

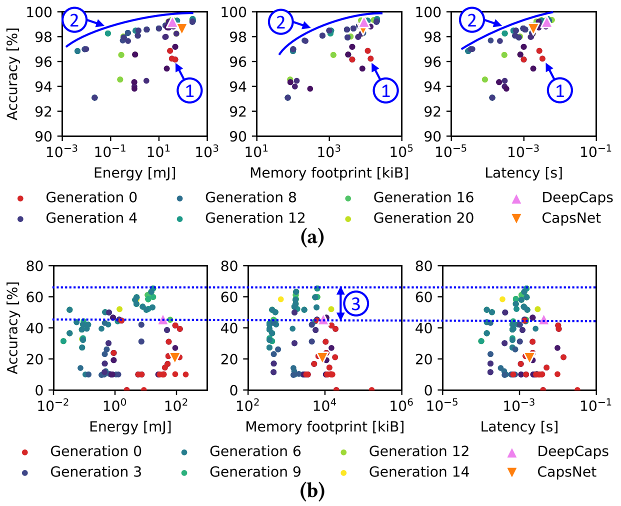


Figure 10: Partially-Trained DNN NAS for (a) the MNIST dataset, and (b) the CIFAR-10 dataset. The color shows in which generation the solution occurs first.

The Fashion-MNIST search ended at its 19th generation (in 12 hours) and evaluated a total of 200 candidate architectures. The search for the SVHN dataset lasted for 12 generations, and it allowed to evaluate 130 architectures. For the CIFAR-10 dataset, the search reached its 14th generation, with a total of 150 tested architectures.

Fig. 10 shows how the evolutionary search algorithm progressed for the MNIST and CIFAR-10 datasets. Note that the red dots, i.e., the initial population at the generation 0 (see pointer ① in Fig. 10a), represent *randomly generated* DNNs. The objectives significantly improve during the following iterations (see pointer ②), when our evolutionary algorithm finds better candidate DNN architectures using crossover and mutation operations iteratively. The reduced epoch training allowed to evaluate a large number of candidate networks (a total of nearly 700 architectures) based on convolutional and capsule layers. This method led to finding multiple candidate architectures that have been able to reach an accuracy up to 30.86% higher than the best among the partially-trained SoA solutions, i.e., within the limits of a strongly reduced training time. For instance, the NAS for the CIFAR-10 dataset produced a network with an accuracy of 76.46% after 10 epochs, while the DeepCaps architecture reached only 45.60% accuracy within the same training interval (see pointer ③ in Fig. 10b). This corroborates the fact that our *NASCaps* can generate networks with higher accuracy compared to DeepCaps-like structures, when both are subjected to short training time constraints.

4.4 NASCaps Results for the Selected Fully-Trained DNNs

After the first selection stage, the candidate DNNs belonging to the Pareto-optimal subsets have been fully-trained to evaluate their final accuracy. Fig. 11 shows the Pareto-optimal solutions at the end of the full-training process.

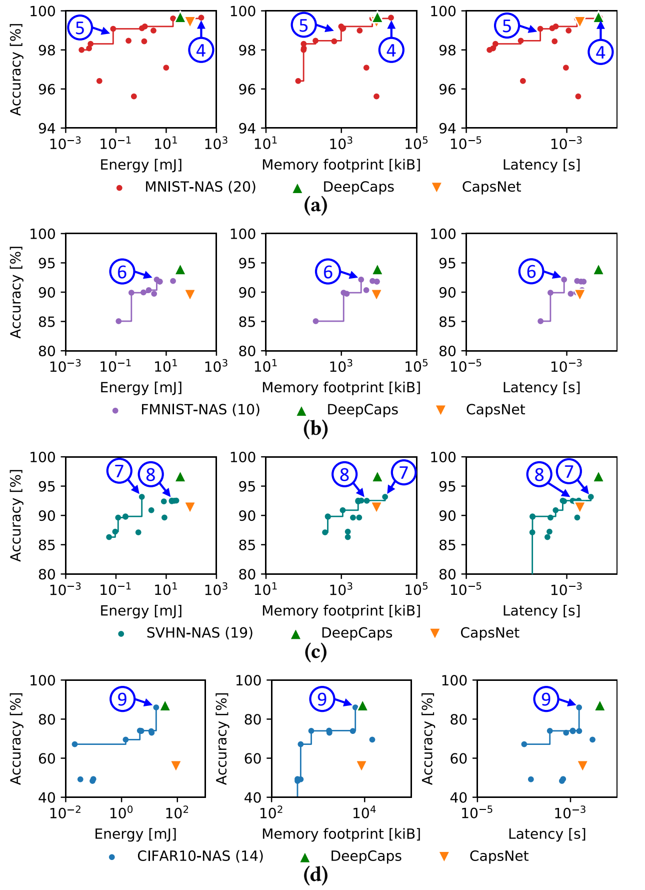


Figure 11: Fully-trained DNN results for (a) the MNIST, (b) the Fashion-MNIST, (c) the SVHN, and (d) CIFAR-10 datasets.

4.4.1 *NASCaps for the MNIST Dataset.*

The highest-accuracy architecture (see pointer ④ in Fig. 11a) found during the MNIST search allowed to reach an accuracy of 99.65% in 93 epochs of training. However, that particular solution requires 2.8× more energy, 2.5× more time, and 2.4× more memory w.r.t. the CapsNet architecture. The red front (see pointer ⑤) also highlights other interesting solutions belonging to the derived Pareto-optimal front, with a slightly lower accuracy, but *up to a couple of orders of magnitude lower energy, memory and latency* achieved by our identified solutions.

4.4.2 *NASCaps for the Fashion-MNIST Dataset.*

One of the best solutions (see pointer ⑥ in Fig. 11b) achieves an accuracy of 92.15% in 51 epochs. This solution improved the latency (-79.38%), energy (-88.43%), and memory footprint (-63.05%) compared to both the CapsNet and DeepCaps architectures, with almost the same accuracy as the last one, which is 93.94%.

4.4.3 *NASCaps for the SVHN Dataset.*

The set of experiments for the SVHN dataset produced a solution (see pointer ⑦ in Fig. 11c) that reached an accuracy of 93.17% in 56 epochs. This solution also significantly reduced the energy by 97.05% and latency by 29.56%, compared to the DeepCaps, but it requires 1.6x more memory. On the other hand, another interesting solution (see pointer ⑧) reached an accuracy of 92.53%, while requiring 30.59% lower energy, 59.63% lower latency and 62.70% lower memory, compared to the DeepCaps.

4.4.4 *NASCaps for the CIFAR-10 Dataset.*

A solution found by the CIFAR10-NAS (see pointer ⑨ in Fig. 11d) achieved an accuracy of 85.99% after 300 epochs of training, while significantly improving all the other objectives, compared to the DeepCaps architecture. This particular solution (*NASCaps-C10-best* in Table 3) reduced the energy consumption by 52.12%, the latency by 64.34% and the memory footprint by 30.19%, compared to the DeepCaps executed on the CapsAcc accelerator, while encountering a slight accuracy drop of about 1%, while using the same training settings. Table 3 reports also other Pareto-optimal DNN architectures found by our *NASCaps* framework for the CIFAR-10 dataset.

Table 3: Selected CIFAR-10 architectures after 300-epoch training.

Architecture	Accuracy	Energy	Latency	Memory
DeepCaps [21]	87.10%[2]	36.30 mJ	4.29 ms	9,052 kiB
NASCaps-C10-best ⑨	85.99%	17.38 mJ	1.53 ms	6,319 kiB
NASCaps-C10-a0d	74.11%	4.53 mJ	1.12 ms	1,718 kiB
NASCaps-C10-9fd	74.00%	5.11 mJ	0.36 ms	713 kiB
NASCaps-C10-658	73.91%	5.06 mJ	1.54 ms	5,573 kiB
CapsNet [23]	55.85%[2]	88.80 mJ	1.82 ms	8,573 kiB

4.4.5 *Transferability of the Selected DNNs Across Different Datasets.*

To test the transferability of the DNN solutions found by our *NASCaps* framework, the dataset-specific found DNNs have been also trained and tested on the rest of the considered datasets. Table 4 reports the matrix of highest-accuracy solutions, obtained by this transferability analysis.

The *NASCaps-C10-best* architecture of Table 3 resulted also particularly accurate for the other datasets. For the MNIST dataset, it achieved an accuracy of 99.72% in 37 epochs of training, which is also higher than the solutions found by the MNIST-NAS. For the Fashion-MNIST dataset, it reached an accuracy of 93.87% in 32 epochs of training, which is even higher than the DeepCaps after 100 epochs of training. When tested on the SVHN dataset, it reached an accuracy of 96.59%, thus outperforming the highest-accuracy DNN found during the SVNH-NAS. The *NASCaps-C10-best* architecture is similar to the DeepCaps, but it has two initial convolutional layers and three CapsCell blocks, without skip connection. The highest-accuracy architecture found by the MNIST-NAS also proved to work well with the Fashion-NMIST dataset, reaching an accuracy of 93.34% after 91 epochs of training.

[2]Note: The accuracy reported for the DeepCaps and CapsNet do not 100% match with the ones reported in [21]. This can be attributed to the differences in the training hyper-parameter setup, as their papers do not disclose the complete in-depth information about the training that can ensure reproducibility of their results.

Table 4: Highest-Accuracy DNNs found by the dataset-specific NAS, which are then trained for the other datasets for 100 epochs.

Architecture	MNIST	FMNIST	SVHN	CIFAR-10
NASCaps-MNIST-best ④	99.65%	93.34%	96.36%	71.44%
NASCaps-FMNIST-best ⑥	99.49%	92.15%	93.12%	68.34%
NASCaps-SVHN-best ⑦	99.51%	91.43%	93.17%	63.72 %
NASCaps-C10-best ⑨	**99.72%**	**93.87%**	**96.59%**	**76.46%**

The results reported in Table 4 show how the solution *NASCaps-C10* is the best overall architecture found during the four searches performed. This is due to multiple reasons: the evolutionary process was based on a random initial parent population that has been newly generated at each search. Moreover, the small size of the initial parent population may have contributed to a non-convergence of the four dataset-specific searches that have been performed. Also, not each one of the four searches reached the same generation at the end of the experiments.

4.5 Summary of Key Results

The above results show how our *NASCaps* framework has been able to explore multiple solutions with diverse tradeoffs, thanks to the usage of an evolutionary algorithm for a multi-objective search. It has been possible to generate and test 690 candidate networks for the four dataset-specific searches. Our approach allowed to outperform many objectives of the SoA solutions when performing the full-training, despite the strict time constraints applied to the single searches. In summary, our framework allowed to:

- Derive some interesting architectures, such as the above-discussed *NASCaps-C10-best* that reached an almost similar accuracy as of the SoA, while significantly improving all other objectives of the search, i.e., energy, memory and latency.
- Perform early candidate selection while still achieving high accuracy after performing the full training.
- Achieve good transferability between different datasets, as demonstrated by the fact that the *NASCaps-C10-best* DNN, which is found for the CIFAR10-specific search, outperforms other dataset-specific searches also on other datasets.

5 CONCLUSION

In this paper, we presented *NASCaps*, a framework for the Neural Architecture Search (NAS) of Convolutional Capsule Networks (CapsNets). The set of optimization goals for our framework are the network accuracy and the hardware efficiency, expressed in terms of energy consumption, memory footprint, and latency, when executed on the specialized hardware accelerators. We performed a large-scale NAS using GPU-HPC nodes with multiple Tesla V100 GPUs, and found interesting DNN solutions that are hardware-efficient yet highly-accurate, when compared to SoA solutions. Our framework is even more beneficial when the design times are short, training resources at the design center are limited, and the DNN design is subjected to short training durations. Our *NASCaps* framework can ease the deployment of DNNs based on capsule layers in resource-constrained IoT/Edge devices. We will open-source our framework at https://BlindedLink.

REFERENCES

[1] M. Abadi et al. Tensorflow: A system for large-scale machine learning. In *OSDI*, 2016.

[2] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan. A fast and elitist multiobjective genetic algorithm: Nsga-ii. *IEEE transactions on evolutionary computation*, 2002.

[3] S. Grigorescu, B. Trasnea, T. Cocias, and G. Macesanu. A survey of deep learning techniques for autonomous driving. *Journal of Field Robotics*, 2019.

[4] A. Harlap et al. Pipedream: Fast and efficient pipeline parallel dnn training. *ArXiv*, 2018.

[5] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. *CVPR*, 2015.

[6] K. He, X. Zhang, S. Ren, and J. Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *ICCV*, 2015.

[7] G. E. Hinton, A. Krizhevsky, and S. D. Wang. Transforming auto-encoders. In *ICANN*, 2011.

[8] W. Jiang et al. Accuracy vs. efficiency: Achieving both through fpga-implementation aware neural architecture search. In *DAC*, 2019.

[9] W. Jiang et al. Device-circuit-architecture co-exploration for computing-in-memory neural accelerators. *IEEE Transactions on Computers*, 2020.

[10] A. Krizhevsky. Learning multiple layers of features from tiny images. Technical report, Department of Computer Science, University of Toronto, 2009.

[11] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. *Commun. ACM*, 2017.

[12] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 1998.

[13] Z. Lu, K. Deb, and V. N. Boddeti. Muxconv: Information multiplexing in convolutional neural networks. *ArXiv*, 2020.

[14] A. Marchisio, M. A. Hanif, and M. Shafique. Capsacc: An efficient hardware accelerator for capsulenets with data reuse. In *DATE*, 2019.

[15] A. Marchisio et al. Deep learning for edge computing: Current trends, cross-layer optimizations, and open research challenges. In *ISVLSI*, 2019.

[16] A. Marchisio et al. X-traincaps: Accelerated training of capsule nets through lightweight software optimizations. *ArXiv*, 2019.

[17] R. Miikkulainen et al. Evolving deep neural networks. *arXiv*, 2017.

[18] V. Mrazek et al. Alwann: Automatic layer-wise approximation of deep neural network accelerators without retraining. In *ICCAD*, 2019.

[19] Y. Netzer et al. Reading digits in natural images with unsupervised feature learning. *NIPS*, 2011.

[20] K. Pearson. Note on regression and inheritance in the case of two parents. *Proceedings of the Royal Society of London*, 1895.

[21] J. Rajasegaran et al. Deepcaps: Going deeper with capsule networks. In *CVPR*, 2019.

[22] J. Redmon, S. K. Divvala, R. B. Girshick, and A. Farhadi. You only look once: Unified, real-time object detection. *CVPR*, 2016.

[23] S. Sabour, N. Frosst, and G. E. Hinton. Dynamic routing between capsules. In *NIPS*, 2017.

[24] K. Simonyan and A. Zisserman. Very deep convolutional networks for large-scale image recognition. In *ICLR*, 2015.

[25] D. Stamoulis et al. Single-path nas: Designing hardware-efficient convnets in less than 4 hours. In *ECML/PKDD*, 2019.

[26] Y. Sun, B. Xue, M. Zhang, and G. G. Yen. Automatically designing cnn architectures using genetic algorithm for image classification. *IEEE transactions on cybernetics*, 2020.

[27] V. Sze, Y.-H. Chen, T.-J. Yang, and J. S. Emer. Efficient processing of deep neural networks: A tutorial and survey. *Proceedings of the IEEE*, 2017.

[28] C. Szegedy, S. Ioffe, V. Vanhoucke, and A. A. Alemi. Inception-v4, inception-resnet and the impact of residual connections on learning. In *AAAI*, 2017.

[29] C. Szegedy et al. Going deeper with convolutions. In *CVPR*, 2015.

[30] H. Xiao, K. Rasul, and R. Vollgraf. Fashion-mnist: a novel image dataset for benchmarking machine learning algorithms. *ArXiv*, 2017.

[31] L. Xie and A. L. Yuille. Genetic cnn. *ICCV*, 2017.

[32] B. Zoph and Q. V. Le. Neural architecture search with reinforcement learning. In *ICLR*, 2017.

[33] B. Zoph, V. Vasudevan, J. Shlens, and Q. V. Le. Learning transferable architectures for scalable image recognition. In *CVPR*, 2018.