Microprocessors and Microsystems 73 (2020) 102989

Contents lists available at ScienceDirect

Microprocessors and Microsystems

journal homepage: www.elsevier.com/locate/micpro

DeepMaker: A multi-objective optimization framework for deep neural networks in embedded systems

Mohammad Loni[a], Sima Sinaei[a], Ali Zoljodi[b], Masoud Daneshtalab[a,*], Mikael Sjödin[a]

[a] School of Innovation, Design and Engineering, Mälardalen University, Västerås, Sweden
[b] Shiraz University of Technology, Shiraz, Iran

ARTICLE INFO

Article history:
Received 18 February 2019
Revised 15 November 2019
Accepted 2 January 2020
Available online 3 January 2020

Keywords:
Convolutional Neural Networks (CNNs)
Design Space Exploration (DSE)
Embedded systems
Multi-Objective Optimization (MOO)

ABSTRACT

Deep Neural Networks (DNNs) are compute-intensive learning models with growing applicability in a wide range of domains. Due to their computational complexity, DNNs benefit from implementations that utilize custom hardware accelerators to meet performance and response time as well as classification accuracy constraints. In this paper, we propose DeepMaker framework that aims to automatically design a set of highly robust DNN architectures for embedded devices as the closest processing unit to the sensors. DeepMaker explores and prunes the design space to find improved neural architectures. Our proposed framework takes advantage of a multi-objective evolutionary approach that exploits a pruned design space inspired by a dense architecture. DeepMaker considers the accuracy along with the network size factor as two objectives to build a highly optimized network fitting with limited computational resource budgets while delivers an acceptable accuracy level. In comparison with the best result on the CIFAR-10 dataset, a generated network by DeepMaker presents up to a 26.4x compression rate while loses only 4% accuracy. Besides, DeepMaker maps the generated CNN on the programmable commodity devices, including ARM Processor, High-Performance CPU, GPU, and FPGA.

1. Introduction

In recent years, deep learning, which uses deep neural networks as the learning model, has shown excellent performance on many challenging artificial intelligence and machine learning tasks, such as image classification [1], speech recognition [2], and unsupervised learning tasks [3]. In particular, Convolutional Neural Networks (CNNs) propose massive success in visual recognition tasks in the past few years and are applied to various computer vision applications [4]. CNNs have penetrated in a broad spectrum of platforms from workstations to embedded devices due to influential learning capabilities.

However, as CNN architectures become increasingly complex in order to improve accuracy, also the energy consumption for inference is becoming a bottleneck. Dealing with enormous computing throughput demand of up-coming complex learning models in the context of big data will be more acute where the failure of traditional energy and performance scaling paradigm in affording of modern applications requirements leads computing landscape towards inefficiency [42]. On the other hand, leveraging high-performance cloud infrastructures for providing required computational capacity is not always feasible, especially for mission-critical applications due to limited network bandwidth, privacy constraints, low-power efficiency, and not guaranteeing worst-case response-time.

Generally, two approaches are presented to tackle these challenges: 1) diminishing the network size by leveraging network pruning techniques during the training phase [1] and 2) employing customized hardware accelerators [13,9,35]. However, optimizing the network architecture at design time should be taken into account as the third approach since the choice of the architecture strongly impacts on both the performance and the output quality of DNNs. To benefit from this opportunity, we propose a neural acceleration framework, named DeepMaker, which automatically generates a robust DNN in terms of network accuracy and network size, then maps the generated network to an embedded device. Unlike previous neural architectural solutions that their focus is only on improving the accuracy level, DeepMaker also considers network size as the second objective of the search space in order to adaptively find a fit DNN for limited resource embedded devices. For this, DeepMaker is equipped with a Multi-Objective Optimization (MOO) method to solve the neural architectural search problem by finding a set of Pareto-optimal surfaces. The design space

* Corresponding author.
E-mail address: masdan@kth.se (M. Daneshtalab).

https://doi.org/10.1016/j.micpro.2020.102989

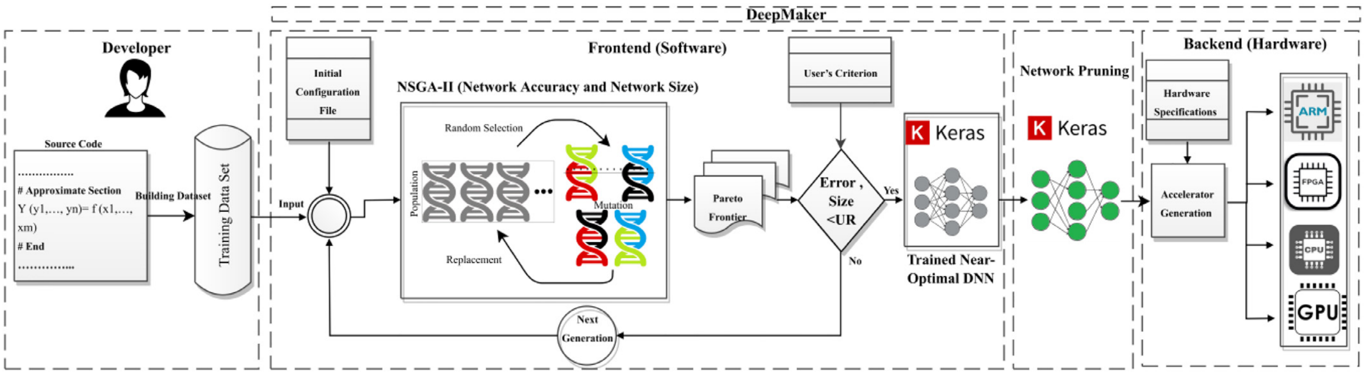

Fig. 1. An overview of the DeepMaker framework.

has been pruned by taking inspirations from a cutting-edge architecture, DenseNet [6], to boost the convergence speed to an optimal result.

The proposed DeepMaker framework uses a multi-objective neuro-evolutionary approach for the space exploration of finding optimal deep neural architectures while mapping the generated network to the given hardware. An overview of the proposed framework is illustrated in Fig. 1. The configuration file of DeepMaker comprises predefined parameters for the MOO algorithm and network training parameters. As shown in Fig. 1, the input of the framework is a dataset for generating a neural network.

To approximate an application, developers first need to identify the approximation region of the code, then provide a training dataset for the specified code block in order to be mimicked by a DNN generated by DeepMaker. The approximation region of the code should be both hotspot and less sensitive to a quality loss in both data and operations. We can define a hotspot as a code region that consumes considerable energy or occupies a significant part of execution time [7].

The output of the DeepMaker framework is a set of optimized architectures. Network pruning is a popular solution for diminishing the amount of network computation. In addition to the design space exploration, DeepMaker can apply a network pruning method on a dense architecture to accelerate finding the optimal neural networks. In a nutshell, our main contributions in DeepMaker are as follows:

- Developed a multi-objective neuro-evolutionary method to discover near-optimal DNN architectures in terms of the accuracy and the network size.
- We applied a network pruning method [39] on the optimized neural network architectures designed by DeepMaker to obtain a higher level of network compression rate.
- Supporting both Multi-Layer Perceptron (MLP) and Convolutional Neural Network (CNN) models fitting with the required accuracy of diverse applications from mathematical function to image classification.
- Adaptive finding the best architecture regarding resource budget and execution time constraints. Then, mapping the generated network on different platforms to evaluate the applicability of DeepMaker is our last contribution.

The remainder of this paper is organized as follows: Section 2 gives preliminaries on CNN and the MOO algorithm. Details of the proposed framework are presented in Section 3, which consists of two solutions for network optimization: Design Space Exploration and Neural Network Pruning. The experimental results are presented in Section 4. Section 5 reviews related work in this scope, after which Section 6 concludes the paper.

2. Related work

2.1. Automatic design of deep neural network architecture

State-of-the-art approaches pointing to design the architecture of DNNs automatically that could be categorized into the hyperparameter optimization, reinforcement learning, and evolutionary approaches.

a) **Hyperparameter Optimization**: From the machine learning point of view, we can model the DNN architecture designing problem as a hyperparameter optimization problem. There have been proposed many hyperparameter optimization methods, such as Grid Search (GS) [16], gradient search [17], Random Search (RS) [18], and Bayesian optimization-based method [19]. However, GS is relatively slow. Using RS is challenging due to extremely random sampling in the search space, and Bayesian-based methods suffer from immense computational cost. Besides, these methods are suitable only for search models with a fixed length space and hard to design more flexible architectures from scratch [20].

b) **Reinforcement Learning:** Recently, there has been much work at the intersection of reinforcement learning and deep learning, which show better results for image classification applications compared to best hand-craft DNN accuracy results. Baker et al. [21] have proposed a meta-modeling approach based on reinforcement learning to produce CNN architectures. In this paper, A Q-learning agent explores and exploits a space of model architectures with greedy strategy and experience replay. In [22], a recurrent neural network (RNN) was used to generate neural network architectures, and the RNN was trained with reinforcement learning to maximize the expected accuracy on a learning task. This method uses asynchronous parameter updates and distributed training and with 800 graphic processing units (GPUs) to accelerate the reinforcement learning process. In [23], a block-wise network generation pipeline called BlockQNN has been provided to automatically build high-performance networks using the Q-Learning paradigm with the epsilon-greedy exploration strategy. Despite their success, these models are too slow and require huge computational resources in both training and prediction steps, e.g., MetaQNN [21] contains 11.18M trainable parameters and used 10 GPUs for 8–10 days to train a CIFAR-10 classifier.

c) **Evolutionary-based approaches**: Suganuma et al. [24] tried to automatically construct CNN architectures for an image classification task based on Cartesian genetic programming (CGP). The CNN structure and connectivity represented by the CGP encoding method are optimized to maximize the validation accuracy. Sun et al. [20] proposed a new method using genetic algorithms for evolving the architectures and connection weight initialization values of a deep CNN. In their proposed algorithm, an efficient variable-length gene encoding strategy is designed to represent the different building blocks and the unpredictable optimal depth in convolutional neural networks. In addition, a new representation scheme is developed for effectively initializing connection weights expected to avoid networks getting stuck into local minima. Real et al. [25] Proposed simple evolutionary techniques at unprecedented scales to discover models for the CIFAR-10 and CIFAR-100 datasets. They used novel and intuitive mutation operators that navigate large search spaces. [45] introduces NSGA-Net, an evolutionary-based method for designing neural architecture search with the goal of optimizing multiple conflicting objectives. Compared to the NSGA-Net, DeepMaker proposes more dense architecture on both Cifar-10 and Cifar-100 datasets. Moreover, they did not guarantee the convergency of their proposed optimization. Finally, DeepMaker provides a fast optimization method compared to NSGA-Net due to leveraging a template-based exploration method.

2.2. Neural network pruning

In [36], a method is proposed that prune CNN filters that identified to have a small effect on output accuracy. By removing all inessential filters and their connecting feature maps in the network, the computation costs are reduced resulting in non-sparse connectivity patterns. In [40], a data-free approach is proposed to carry out CNN model compression. They managed to avoid employing any training data by minimizing the expected squared difference of logits. Compared to the previous works, they removed a neuron at a time instead of removing weights. Plus, all the reduction is implemented on fully connected layers. In [37], a pruning approach by applying L1/L2-norm regularizations is introduced to remove the small weights. The basic idea of their work is that weight connectivity should be pruned if it is less than a predefined threshold. Both convolutional layers and fully connected layers could be pruned by using this strategy. Their method achieved the compression ratio on Lenet-5 by a factor of 12×. Although the performance is inspiring, the pruning would result in unstructured patterns in weights connectivity. This shortcoming requires long fine-tuning time which may exceed the original network training by a factor of 3 ×. The paper presented in [38] introduces the *Average Percentage of Zeros* to assess the importance of each filter since small differences in weights are affected too much. A filter may be selected as unimportant one if most outputs of the filter are zero. Although their solution seems more reasonable, this work needs lots of extra calculations, and the compression ratio is not satisfying.

2.3. Automatic code approximation frameworks

After reviewing literature, various code approximation frameworks have been found [32–35,46]. The main idea of these frameworks is to generate a DNN for a subsection of the code, then accelerate the generated DNN on a custom hardware platform. However, the main weakness of them is the NN architecture selection procedure. Prior work mainly used a simple search methodology to explore a small design space that is not applicable to real-world applications. Moreover, they generate a deep multi-layer network that is obsolete and does not produce e accurate competitive results for modern applications such as object recognition.

3. Preliminaries

3.1. Convolutional Neural Networks (CNNs)

A Convolutional Neural Network (CNN) is a multi-layer neural network that is composed of neurons ordered in a layered structure. The neurons in different layers perform different kinds of computations and have different connection structures. The four essential layers of CNNs are convolutional layers (*Conv*), activation layers (*Act*), pooling layers (*Pool*), and fully-connected layers (*FC*). A typical NN structure is composed of several stacks of {*Conv-Act-Pool*} at the beginning, and a few stacks of {*FC-Act*} at the end. Each layer gets feature maps information from previous layers and generates new output feature maps by using a filter kernel. The convolution, pooling, normalization, and activation layers are used for feature extraction, and fully-connected layers are responsible for classification. The performance criteria of a DNN include the ability to classify data that has never seen before, inference time, and learning rate, which all depend on the multiple hyper-parameters of network architecture.

The *Conv* and *FC* layers are the most computation-intensive layers in CNNs. They have the same basic operations: $b_j = \sum_i a_i.w_{i,j}$, i.e., the weighted sum of the inputs. The weights $(w_{i,j})$ are learned from the training phase, and the inputs (a_i) are from the previous layer. While the *Conv* layers use small groups of weights (called kernels) to slide over the inputs, the *FC* layers use a full connection between input and output neurons. The *Act* layers apply a non-linear function, e.g., ReLU (max *(0, x)),* Sigmoid ($^1/_{1+e^{-x}}$) on each neuron. The *Pool* layers are used to decrease the feature dimension size by either selecting the largest neuron (i.e., max-pooling) or computing the mean value (i.e., mean pooling) from a subset of neurons in a local region.

3.2. Multi-Objective Optimization (MOO)

The problem of finding the best configuration (s) of a parameterized system *S* with *n* different parameters with respect to *m* different objectives is called a MOO Problem [41]. The set of all possible configurations is called the Design Space, whereas each point *C* in this space (each configuration C) is called a solution to the MOO problem.

Each point *C* in the design space can be shown by an n-tuple $<v_1, v_2, \ldots, v_n>$ in which v_i is the value of *i-th* parameter for that solution. The values of all design objectives for a specific solution can be shown by an m-tuple $<a_1, a_2, \ldots, a_m>$ in which a_i is the value of *i-th* objective function corresponding to that solution.

Definition 1. Let $<a_1, a_2, \ldots, a_m>$ and $<b_1, b_2, \ldots, b_m>$ be the objective values corresponding to solutions *A* and *B*, respectively. The solution *A* is said to dominate the solution *B*, if and only if (Eqn 1):

$$\forall i: \quad a_i \le b_i \qquad and \qquad \exists j: \quad a_j < b_j \tag{1}$$

Definition 2. A solution is said to be *Pareto Optimal* if and only if any other solution to the problem does not dominate it. For example, assuming that the design space of Fig. 2. Fig. 2. contains only the seven points mentioned above, solutions A, C, D, and F are Pareto optimal since any other solution dominates none of them.

Definition 3. The set of all Pareto optimal solutions in a MOO is called *Pareto Front*. Nevertheless, in almost all practical design space exploration situations, the size of the design space is exponential, and finding the exact *Pareto Front* is not feasible [41] in a reasonable time. Thus, the goal of design space exploration is modified as: To the *Pareto Front* or a good approximation of it.

In this paper, MOO is used to solve the neural architectural search problem by finding a set of Pareto-optimal sets of network hyperparameters. The key design objectives which are considered in this paper for the network optimization are classification accuracy and network size. In this work, the Non-Dominated Sorting Genetic Algorithm (NSGA-II) [8] has been used to solve the exploration problems. NSGA-II is a robust meta-heuristic

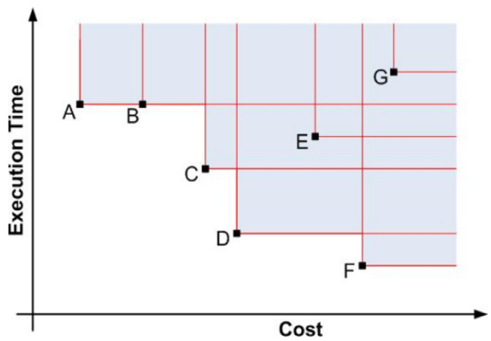

Fig. 2. The notion of domination and Pareto optimality in objective space. All other points dominate solution G, A dominates B, and G, while any other solution does not dominate the solutions A, C, D, and F.

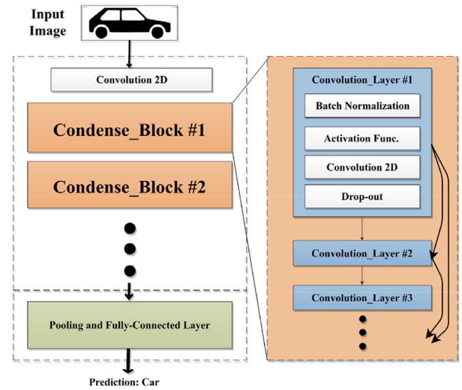

Fig. 3. The generic structure of a neural architecture.

population-based evolutionary algorithm solving MOO problems that aim to adaptively fit a set of candidates to Pareto frontier.

NSGA-II works as follows: In the first step, an offspring population U_t is formed from a parent population P_t by using Genetic Programming, both with size *N*. Then we combine U_t and P_t to devise a third population R_t of size $2*N$. Next, NSGA-II extracts a population (with size *N*) from R_t by employing multiple objectives, non-dominated sorting, and crowding distance comparison. The main aim of non-dominated sorting is to find a set of solution which cannot dominate each other. Moreover, by doing crowding distance sorting, we can orchestrate the density of solution for each Pareto front. NSGA-II selects the best *N* candidates for generating the next population called P_t+1. This procedure is repeated for the next generations until it exceeds a predefined maximum number of generations or satisfies the developer's criterion including the desired level of accuracy/network size. Although DeepMaker walks toward an optimal solution, it does not always guarantee to reach the developer's criterion.

4. The proposed framework

This section explains the DeepMaker framework. DeepMaker uses two solutions for network optimization: 1) *Design Space Exploration* for designing network architecture and 2) *Neural Network Pruning* for compressing the model size, which will be presented in Sections 4.1 and 4.2, respectively.

The DeepMaker framework is composed of frontend and backend layers. The frontend is responsible for generating the optimized DNN while the backend layer deals with hardware configuration and mapping. The hand-craft designing of DNN architectures needs deep expertise and a large number of trial and error imposing a considerable design cost and efficiency risk. Thereby, tailoring the DNN architecture has emerged as an efficient alternative solution in the machine learning community. This approach is considered for the frontend layer of our framework in which we propose an evolutionary-based approach to search the design space inspired by DenseNet to vanish the probability of generating huge design space. This decision leads DeepMaker to generate compact-inclined networks in a reasonable time by gaining from human experience in designing efficient DNNs.

The underlying template architecture of the network is shown in Fig. 3. The generated network consists of back-to-back Condense Blocks for feature extraction while each layer consists of multiple Convolution Layers. Each Convolution Block includes Batch Normalization, Activation Function, 2D Convolution, and Dropout layers, respectively. The final classification is integrated by the max-pooling and the fully connected layers as the output layer with the softmax activation function. For sharing maximum information between layers in the network, all the layers are connected in a feed-forward manner such that each layer receives the additional feature map information from the whole former layer and combining them by using a concatenation layer. This structure leads us to enlarge sharing information and shorten the path from the first layer to the last layer.

4.1. Design Space Exploration

Design Space Exploration (DSE) is the process of finding a set of optimal or near-optimal design configurations for a system subject to one or more design criteria. As discussed in the introduction, the design objectives are considered as accuracy and network size. After computational analysis of a popular CNN, VGG16 [31,44], we have concluded that convolutional layers (Conv.) are extremely computationally intensive. Thus, for optimizing a CNN architecture, convolutional parameters including the number of convolutional layers, the sizes of each layer, and the filter size should be considered as the network optimization hyperparameters. Moreover, the choice of activation functions in DNNs outstandingly influence on the training performance since the heart of neural networks is an activation function applied to a linear transformation. So, the activation function is also considered as a pivotal metric in designing the DNN architecture.

Table 1 lists the main architectural hyperparameters of DNNs Table 1. For cutting back the search space, the range of each hyperparameter is limited. Different combinations of these parameters form several architectures with various performances. Finding

Table 1
The cnn hyperparameters used as searching neural design space parameters.

Parameters	Value range
Activation Function	Hard-sigmoid, relu, elu, tanh, sigmoid, softplus, linear, selu
# Condense_Block	1, 2, 3, 4
# Convolution_Layer	16, 28, 40, 52
Learning Rate	0.001, 0.0001, 0.00001
Kernel Size	3 × 3, 5 × 5
Optimizar	Rmsprop, adam, sgd, adagrad, adadelta, adamax, nadam

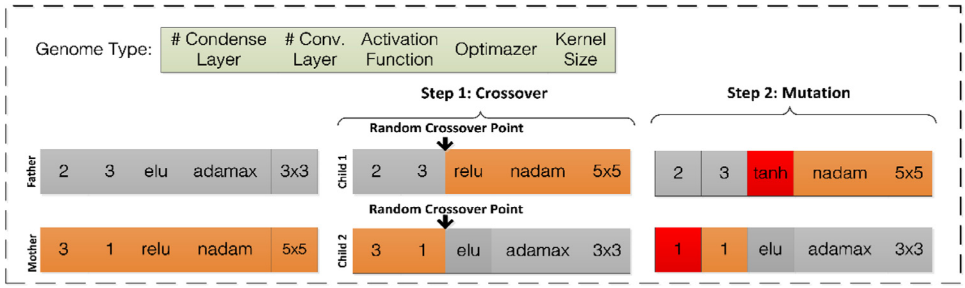

Fig. 4. Genome type.

a near-optimal network architecture of the combination of these hyperparameters is the main goal of the search algorithm. In another word, we can model the DNN architecture selection problem as the hyperparameter optimization problem. DeepMaker is equipped with the fast and multi-objective GP, NSGA-II, to discover a near-optimal set of hyperparameters considering both the accuracy and the network size as the objectives. Total trainable network weights are defined as the network size objective since the performance and energy efficiency of the backend accelerator highly rely on inner product operations, which are execution bottleneck of DNNs [9].

Network hyperparameters are represented as a string of genomes using direct encoding, and the recombination of these genes occurs with a one-point crossover operation shown in Fig. 4. The neural architectural exploration algorithm is explained in four steps as follows:

1. After generating a random initial parent population P_t with size N, DeepMaker generates a network model based on the hyperparameters of each genome in the parent population. Then DeepMaker trains each model to calculate the network accuracy and network size for all the models. In this paper, the roulette wheel selection method is leveraged used due to giving selection chance to all the available individuals.
2. The offspring populating U_t will be created by using GP, including crossover and mutation steps.
3. The NSGA-II sorts the combination of U_t and P_t to find the next generation parent population of N acceptable individuals that cannot dominate each other in terms of accuracy and network size.
4. This process will continue until attaining the predefined maximum number of generations.

Algorithm 1 summarizes the entire exploration procedure. Compare to DenseNet, DeepMaker generates more accurate networks with superior flexibility regarding resource limitation of the backend platform. To increase the rate of optimal discovering, we monitor all genomes in all previous generations. The output of the frontend layer is an asset of improved network architectures on the Pareto curve with different network accuracies and sizes. Efficient mapping of the generated network on hardware is the next step. We only evaluated the impact of proposed design space

Algorithm 1 Pseudo Code of DeepMaker's Design Space Exploration Procedure.

Input: **N**: Population Size, **G**: Max. Number of Generations, **H**: Input Set of Hyperparameters
Output: A Set of Optimal Architectures on Pareto Frontier

Function DeepMaker(N, G, H):
P_0= **Random Population** (N, H);
//Creating initial random solutions with size N
Objectives Function (P_0, Size (P_0));
//Evaluating the objectives of each solution in the population
U_0= **Selection Crossover Mutation** (P_0);
//Generating the offspring population by doing random crossover and mutation
$t = 1$;
while ($t < G$) or (Criterion Not satisfied) **do**
R_t = **Combine** (P_t, U_t);
//Merging Parent and Offspring population, the size of P_{t+1} is $2*N$
Objectives Function (P_{t+1}, Size (P_{t+1}));
Sort_t = **NonDominant Sort** (P_{t+1});
//Sorting the first population in fronts
$pfs[t]$ = **Crowding Distance Sorting** (Sort_t);
//Symmetric disturbing offspring population by crowding distance sort to build Pareto frontier and save it in pfs
$P_{t+1} = pfs[t]$;
// Creating Next Population
U_{t+1}=**Selection Crossover Mutation** (P_{t+1});
Objectives Function (U_{t+1}, Size (U_{t+1}));
return $pfs[G]$;
Function Objectives_Function(*Population P, Size N*):
$i = 1$;
while ($i<N$) **do**
List [i]= **Extract Network Parameters** (P_i);
model[i]= Create Model (List [i]);
//Generating a DNN model using network hyperparameters
Acc.[i], #Params[i]=**Train_Evaluate**(*model[i]*);
// Train the ntwork to get validation accuracy and num. network parameters
return *Accuracy, #Parameters;*

Algorithm 2 Pseudo-Code of the Network Pruning.

Input: A trained CNN model, i is layer number, and α is a threshold for pruning rate
Output: Pruned CNN model

1. Initialize k=N_i -1
2. While (Pruning rate $< \alpha$)
a. Use k-means++ to force the W_n^i $(1 \le n \le N_i)$ into k clusters
b. Keep the filter that is the most closet to the centroid for each cluster, then prune the other filters and feature maps
c. Fine-tuning the pruned model as a training process
d. k = =k-1
3. Save the weights of pruned CNN model

exploration on the hardware platforms since designing network architecture is the main contribution of the paper. Thus, we only provide the hardware results of the generated network after the optimization.

Using Application-Specific Integrated Circuit (ASIC) as a customized DeepMaker's backend accelerator can gain considerable power and performance efficiency. Nonetheless, ASIC cannot be reconfigured and reprogrammed. Graphic Processing Units (GPUs) are famous performance centric accelerators considered as another possibility to cope with diminishing the efficiency trend in the multi-core era [11]. Although GPUs offer a higher level of programmability and memory bandwidth, they suffer from huge power consumption and are efficient only for data parallel kernels and dense data structures [12]. On the other hand, the combination of supporting arbitrary forms of parallelism, flexibility, and power efficiency of off-the-shelf Field-Programmable Gate Arrays (FPGAs) provide a promising opportunity for efficient neural network implementation. Unfortunately, on-chip memory limitation, relatively primitive memory abstraction model, and the lack of efficient high-level APIs are the major bottlenecks of FPGA as a neural-based accelerator [9]. Each of these hardware devices offers various capabilities for real work problems. Section 5.4 presents implementation results on different processing platforms.

4.2. Neural network pruning

In general, neural network pruning techniques try to reduce the storage and computation required by neural networks without considerable affecting the network accuracy by learning the important weights. The pruning method used in this paper is based on the work presented in [39]. This technique tries to select and remove redundant filters, which affected zero or very low in the network results. To select appropriate filters for pruning, many strategies are possible candidates, such as random selection. In this work, the K-means++ algorithm has been utilized. First, the pruning algorithm employs the k-means++ algorithm to enforce the filters to enter specific clusters. Second, it will retain the filter which is the closest to the cluster center, and prune the others in every cluster. Then the pruned model will be fine-tuned to recover accuracy. A filter may be selected as a negligible filter if most outputs of the filter are zero. In Algorithm 2 presents the pseudo-code of the pruning procedure.

5. Experimental results

In this section, first, the used datasets for the experiments will be introduced. After that, the experimental results of design space exploration and network pruning will be presented, respectively. In the end, the hardware implementation of the proposed framework on four prevalent hardware platforms, Xilinx UltraScale plus FPGA, NVIDIA Tesla M60 GPU, Intel Core i7-7820, and ARM Cortex-A15 is discussed.

5.1. Training datasets

The DeepMaker framework has been evaluated by using well-known datasets and cutting-edge architectures. The experiments have been performed on the following data sets: a) **MNIST** [14]: This is a dataset of black and white images for handwritten digit recognition containing 60,000 training and 10,000 testing images, respectively. Each image in the MNIST dataset is a 28 × 28 pixel with ten labeled output as 0–9 numbers. b) **CIFAR-10** [15]: This is a complex colorful benchmark dataset of natural images, each with 32 × 32 pixels, which are mainly used for object recognition. This benchmark contains ten labeled output classes. CIFAR-10 training and testing datasets contain 50,000 and 1000 images, respectively. c) **CIFAR-100** [15]: CIFAR-100 is similar to CIFAR-10, but with 100 classes, while each class has 500 instead of 5000 as in CIFAR10 making the classification more challenging.

5.2. Design Space Exploration

DeepMaker searches the optimal network architecture using partial training by using just 16 epochs since this epoch number is enough for deciding since we got roughly 90% of the maximum achievable accuracy after 16 epochs (Fig. 5). Fig. 5. plots the validation loss and validation accuracy progression by increasing the number of epochs for Net-CNN-Arch.3 with 0.14 million parameters. DeepMaker utilizes the Keras Library [10] with TensorFlow backend [43] for training the network and implementation on the hardware platforms.

The design space contains 5300 different design points. By employing NVIDIA GTX1080ti, the training time is half an hour on average for each design point. Therefore, exhaustive search is not a feasible solution. Since the design exploration takes 2700 GPU-hours, while DeepMaker only needs 75 GPU-hours for generating

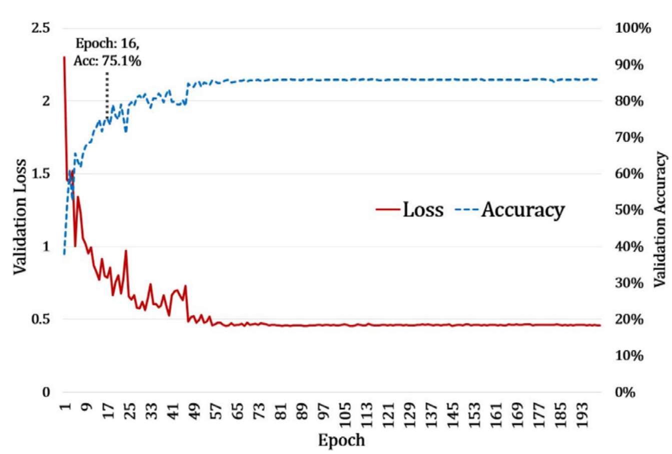

Fig. 5. Validation accuracy and validation loss for Net-CNN-3 with 0.14M parameters.

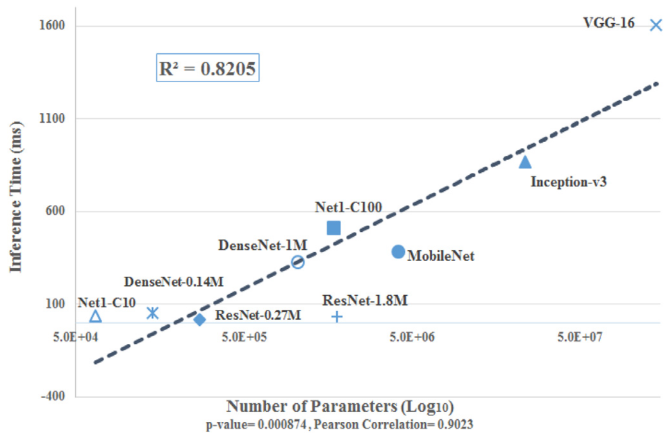

Fig. 6. Inference time vs. network size.

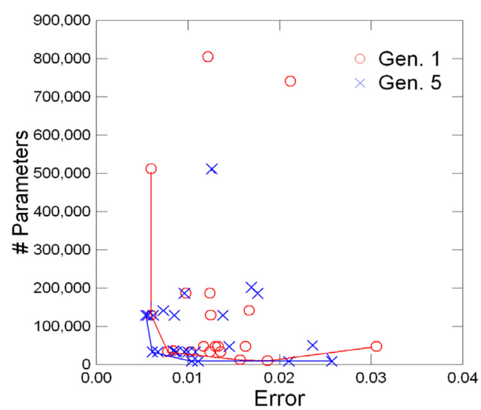

Fig. 7. Pareto frontier plots for CNN architecture generated for the MNIST dataset.

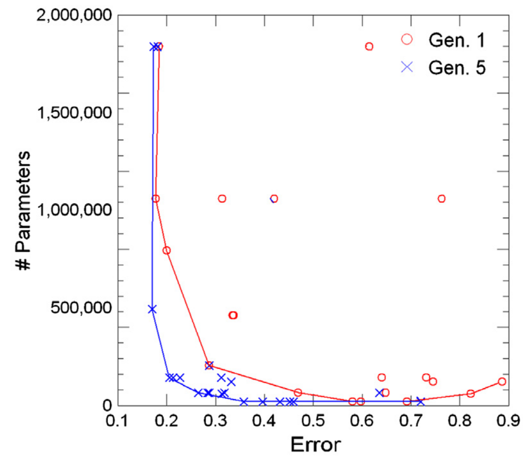

Fig. 8. Pareto frontier plots for CNN architecture generated for the CIFAR-10 dataset.

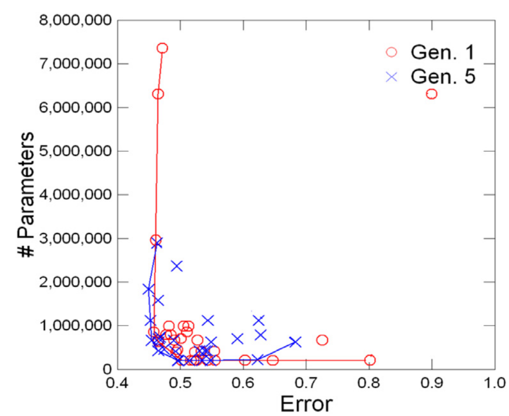

Fig. 9. Pareto frontier plots for CNN architecture generated for the CIFAR-100 dataset.

the experimental results, leading to reduce exploration time by the factor of 35x.

As mentioned before, the objectives of the network optimization and design space exploration are Accuracy and Network size. We realized a strong relationship between inference time and the network size of a CNN. Fig. 6. illustrates the relationship between inference time per each forward query and the number of parameters executed on an NVIDIA Quadro K5100M GPU (R2=0.8205, p-value=0.000874, Pearson correlation=0.9023). The results are plotted in the logarithmic scale to improve visual comprehension. These results imply that the network size is a reliable proxy for network architectural complexity [5,6,18]. These experimental results indicate that DeepMaker efficiently decreases inference time by considering network size as its objective.

The near-optimal Pareto frontier results are illustrated in Figs. 7–9. on MNIST, CIFAR-10, and CIFAR-100 datasets after just five generations. These results are obtained with the following setting in DeepMaker's configuration: dropout=0.2, epoch=16, batch size=128, number of generations=5, and random initial population with the size equal to 30. As can be seen, Pareto-optimal curves shifted toward left, implying that we achieved an improved set of network architectures.

The effectiveness of DeepMaker has been evaluated compared to the error rate and the total number of trainable parameters of the other cutting-edge approaches shown in Table 2. The results of this table are obtained by a full train of the networks with 300 epochs. Network architectures with the highest accuracy are employed as the baseline of the comparisons. Compare to a reinforcement learning solution such as MetaQNN, we lost 0.06% accuracy for the MNIST dataset while we have a 43x compression rate. Not only compare to MetaQNN but also the superiority of DeepMaker's optimization rate is clear for the MNIST dataset.

Net-CNN-Arch.1, Net-CNN-Arch.2, and Net-CNN-Arch.3 are three different nodes of Pareto frontier selected from fifth generation. These three nodes have different network objectives, which give broad authority to DeepMaker to select the most appropriate architecture based on the execution time constraints or resource limitation of the target hardware platform. Net-CNN-Arch.1 loses 4% accuracy compared to the most accurate networks [26], while has 26.4x fewer parameters. Moreover, the MLP model presents comparable accuracy for MNIST, but it cannot provide acceptable accuracy for CIFAR-10 and CIFAR-100, revealing we need more complex architectures for the modern datasets. In a nutshell, DeepMaker strikes better the balance between network accuracy and network size compare to reinforcement learning-based solutions, evolutionary-based approaches, and hand-craft designs.

5.3. Neural network pruning

The pruning method has been evaluated on our Net-CNN-Arch.3 for processing the Cifar10 dataset. First, the network is trained with 85.9% accuracy. Then, we used the filter pruning technique [39] by assuming the '*number of fine-tune epochs* = 5' and α as the maximum pruning percentage is equal to {10, 20, 30, 40, 50, 60, 70}. α is the threshold on weight pruning. Fig. 10. illustrates the impact of the network pruning on the accuracy level of the densest architecture. Obviously, the number of network parameters will be decreased by increasing the pruning rate. On the other hand, there is not a linear correlation between pruning rate and accu-

Table 2
Comparison results of error rate on mnist and cifar-10 datasets.

Dataset	Approach	Network Architecture	#Params (x10^6)	Error (%)
MNIST	RL	MetaQNN [21]	5.59	0.35
	EC	EDEN [28]	1.8	1.6
	Hand-Crafted	SimpleNet [29]	0.3	0.25
	Hand-Crafted	Wan et al. [30]	–	0.21
	MO^2-EC	**Our MNIST-MLP**	0.19	1.2
	MO^2-EC	**Our MNIST-CNN**	**0.13**	**0.41**
Cifar-10	RL	NAS-v1/v3 [22]	4.2/37.4	5.5/3.65
	Hand-Crafted	SimpleNet [29]	5.48	4.68
	Hand-Crafted	VGG-16 [31]	138.0	7.55
	Hand-Crafted	DenseNet ($K = 12$)−40 [6]	1.0	7.0
	Hand-Crafted	DenseNet ($K = 12$)−100 [6]	7.0	5.77
	Hand-Crafted	DenseNet ($K = 24$)−100 [6]	27.2	5.83
	EC	EDEN [28]	0.17	25.6
	Hand-Crafted	ResNet-20 [27]	0.27	8.75
	Hand-Crafted	ResNet-110 [27]	1.7	6.43
	EC	Masanori et al. [24]	1.68	5.98
	RL	Block-QNN-22 L [23]	39.8	3.54
	RL	MetaQNN [21]	6.92	11.18
	EC	Real et al. [25]	5.4	5.4
	Hand-Crafted	Gastaldi et al. [26]	26.4	2.86
	MO^2-EC	**Our Net-MLP**	0.66	37.0
	MO^2-EC	**Our Net-CNN-Arch.1**	**1.0**	**6.9**
	MO^2-EC	**Our Net-CNN-Arch.2**	**0.49**	**8.7**
	MO^2-EC	**Our Net-CNN-Arch.3**	**0.14**	**14.1**
Cifar-100	RL	MetaQNN [21]	11.18	27.14
	RL	Block-QNN-22 L [23]	6.1	20.65
	Hand-Crafted	DenseNet ($K = 12$)−40 [6]	1.0	27.55
	Hand-Crafted	DenseNet ($K = 12$)−100 [6]	7.0	23.79
	Hand-Crafted	SimpleNet [29]	5.48	26.58
	MOO-EC	**Our C100-Net.1**	**1.1**	**26.63**
	MOO-EC	**Our C100-Net.2**	**1.89**	**24.87**

The considerable results achieved by DeepMaker are in **Bold**.

Table 3
The specification of hardware platforms.

Platform	CPU	GPU	ARM	FPGA
Frequency (GHz)	2.9	1.178	1.9	0.8
Technology (nm)	14	28	28	16 (FinFER+)
TDP (W)	45	300	5	–
Cores/Total Thread	4/8	4096 CUDA Cores	8/8	FF = 2.5(x10^6) LUT = 1.18(x10^6) DSP = 6800
Memory	8 MB Cache	16GB GDDR5	2.5 MB Cache	BRAM = 75.5 Mb
Approx. Price (USD)	378$	7532$	60$/board	–

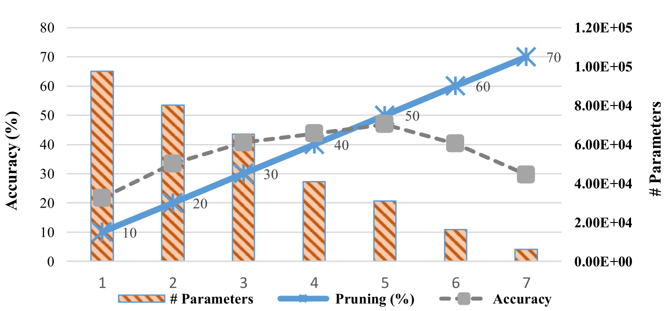

Fig. 10. The impact of network pruning on Accuracy of dense architecture.

racy level since accuracy is also influenced by other factors such as over-fitting, weight initialization and etc.

5.4. Hardware implementation

To verify the practical impact of DeepMaker, we used four prevalent hardware platforms, Xilinx UltraScale plus FPGA, NVIDIA Tesla M60 GPU, Intel Core i7-7820, and ARM Cortex-A15. Table 3.

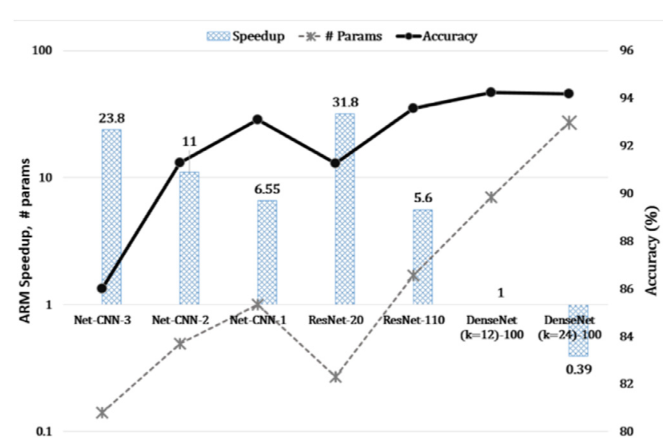

Fig. 11. Speedup of DeepMaker generated networks in comparison to network size and accuracy on ARM platforms.

summarizes the specification of test platforms. We picked out four congruent networks offering better accuracy per parameters, including ResNet-20, ResNet110, DenseNet ($k = 12$)-100, and

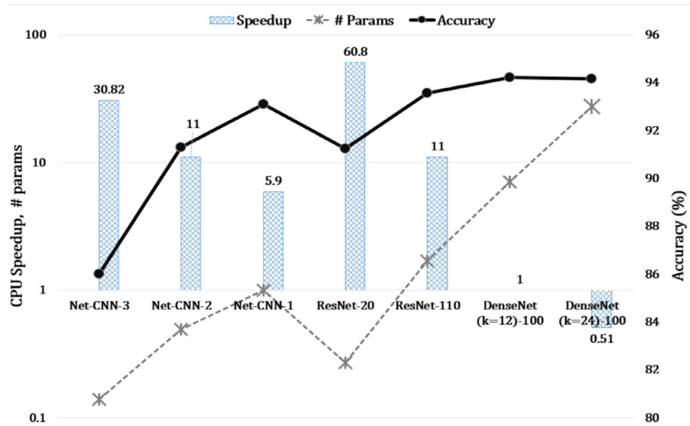

Fig. 12. Speedup of DeepMaker generated networks in comparison to network size and accuracy on CPU platforms.

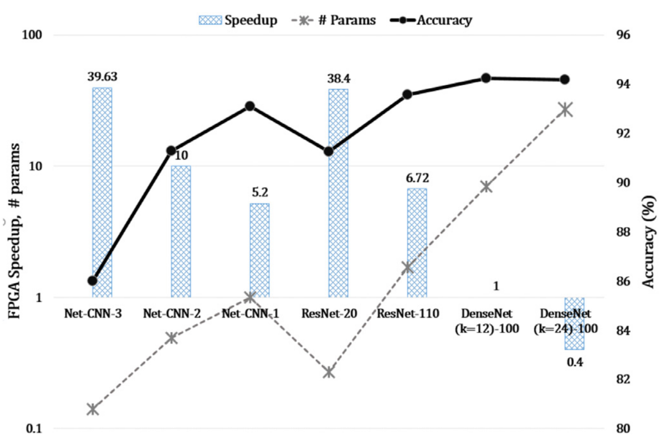

Fig. 13. Speedup of DeepMaker generated networks in comparison to network size and accuracy on FPGA platforms.

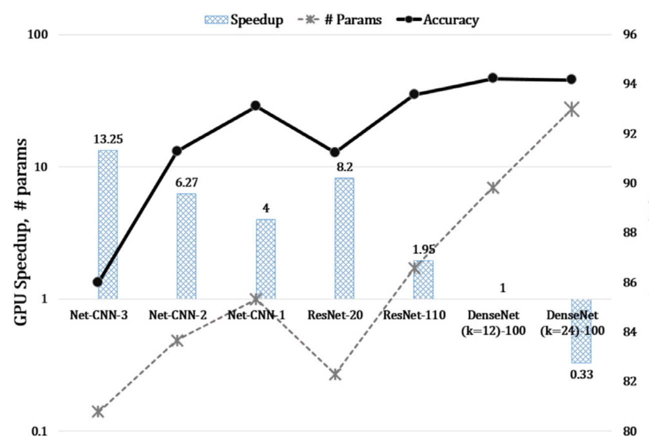

Fig. 14. Speedup of DeepMaker generated networks in comparison to network size and accuracy on GPU platforms.

DenseNet ($k = 24$)-100 to compare with the generated networks by DeepMaker. We did not leverage any network compression technique to assess the influence of network architecture on inference time. Due to the sake of brevity, we present the implementation results of the more complex dataset, CIFAR-100.

In general, Keras uses the TensorFlow backend for training models and mapping on the hardware platform. TensorFlow supports a wide range of hardware platforms from x86 CPU processors to ARM-based platforms. Tensorflow automatically uses cuDNN to compile a neural network for GPU. For obtaining FPGA results, the Amazon EC2 deep learning F1.2xlarge instance has been used. Besides, TensorFlow-Lite is installed on the ARM processor. Thus, Keras can easily map generated models on the ARM processors with the TensorFlow-Lite as the backend library.

Unlike CPUs, we do need an initialization phase to copy data to GPU/FPGA's internal memory, before lunching processing kernel. Usually, kernel time is used for reporting runtime results; however, considering the communication time is vital for embedded implementations, especially for mission-critical applications since these applications are mainly latency oriented. Due to this reason, the total execution time must be taken into account as the evaluation metric. In addition, we believe compacting a network potentially could diminish the overhead of communication time since a smaller number of data packets need to be copied via PCI-Express bus. To increase the precision of results, we got them for 10,000 times, and the average time is leveraged for presenting the results. Figs. 11–14. plot accuracy, the logarithmic scale (to improve visual comprehension) of the number of parameters and the speedup compared to the baseline, DenseNet ($k = 12$)-100. The main reason for selecting DenseNet ($k = 12$)-100 as the ideal baseline is that it delivers better accuracy-parameters tradeoff in comparison with the other networks. Unlike accuracy and the number of parameters, execution time is a platform aware metric and highly depends on hardware implementation, compiler, and the software stack. Therefore, there is no exact speedup similarity among different hardware platforms. The results show that for each hardware platform, there is a firm relation among inference time, network accuracy and network parameters. In a nutshell, we can conclude: 1) the networks with more parameters have higher accuracy, 2) after getting a network more complex, the speedup rate will be decrease, e.g. we got maximum speedup up to 39% on FPGA platform with minimum number of parameters for Net-CNN-3, while DenseNet ($k = 24$)-100 with the best accuracy result always has shown at least 0.33 speed-down. 3) The execution time is scaled by changing the number of parameters demonstrating the considering network size as a design objective decreases both the communication and kernel execution times.

6. Conclusions

The technology of CNNs are ever-evolving and more complex processing models are developed, which becomes an obstacle for embedded systems where memory and energy are often constraining resources. To handle this problem, we proposed DeepMaker, a framework that automatically generates a highly optimized CNN for commercial embedded devices. DeepMaker alleviates the huge computational cost of CNNs by benefiting from minimizing the network architecture at design time. To reach this goal, DeepMaker integrates a multi-objective optimization strategy to explore the design space of CNN architectures. Moreover, the proposed framework can prune the model for obtaining more compressing rate with acceptable accuracy level. Experimental results show that, in comparison with the best results on CIFAR-10/CIFAR100 datasets, DeepMaker presents up to 1.59x/3.46x speedup while loses 2.4%/−0.6% accuracy.

Declaration of Competing Interest

None.

Acknowledgment

KKS has supported this work within the projects DeepMaker and DPAC.

Supplementary materials

Supplementary material associated with this article can be found, in the online version, at doi:10.1016/j.micpro.2020.102989.

References

[1] A. Krizhevsky, I. Sutskever, G.E. Hinton, Imagenet classification with deep convolutional neural networks, Adv. Neural Inf. Process. Syst. (2012) 1097–1105.
[2] G. Hinton, L. Deng, D. Yu, G.E. Dahl, A.R. Mohamed, N. Jaitly, B. Kingsbury, Deep neural networks for acoustic modeling in speech recognition: the shared views of four research groups, IEEE Signal Process. Mag. 29 (6) (2012) 82–97.
[3] V. Mnih, K. Kavukcuoglu, D. Silver, A.A. Rusu, J. Veness, M.G. Bellemare, S. Petersen, Human-level control through deep reinforcement learning, Nature 518 (7540) (2015) 529.
[4] R. Zhang, P. Isola, A.A. Efros, Colorful image colorization, in: European Conference on Computer Vision, 2016, pp. 649–666.
[5] J. T., W.D. Song Han, J. Pool, Learning both weights and connections for efficient neural networks, Adv. Neural Inf. Process. Syst. 50 (2) (2015) 1135–1143.
[6] G. Huang, Z. Liu, K.Q. Weinberger, L. van der Maaten, Densely connected convolutional networks, in: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, 1, 2017, p. 3.
[7] A. Yazdanbakhsh, D. Mahajan, H. Esmaeilzadeh, P. Lotfi-Kamran, AxBench: a multiplatform benchmark suite for approximate computing, IEEE Des. Test. 34 (2) (2017) 60–68.
[8] K. Deb, A. Pratap, S. Agarwal, T. Meyarivan, A fast and elitist multiobjective genetic algorithm: NSGA-II, IEEE Trans. Evol. Comput. 6 (2) (2002) 182–197.
[9] C. Zhang, Z. Fang, P. Zhou, P. Pan, J. Cong, Caffeine: towards uniformed representation and acceleration for deep convolutional neural networks, in: Proc. 35th Int. Conf. Comput. Des., 2016, p. 18.
[10] F. Chollet, Keras, github, 2015. [Online]. Available: https://github.com/fchollet/keras.
[11] H. Esmaeilzadeh, E. Blem, R. St. Amant, K. Sankaralingam, D. Burger, Power challenges may end the multicore era, Commun. ACM 56 (2) (2013) 93.
[12] B. Falsafi, B. Dally, D. Singh, D. Chiou, J.J. Yi, R. Sendag, FPGAs versus GPUs in data centers, IEEE Micro 37 (1) (2017) 60–72.
[13] H. Sharma, J. Park, E. Amaro, B. Thwaites, P. Kotha, A. Gupta, J. Kyung, K. Mishra, H. Esmaeilzadeh, DNNWEAVER: from high-level deep network models to FPGA acceleration, IEEE Int. Conf. Mechatron. Electron. Autom. Eng. (2) (2015) 76–80.
[14] Y. LeCun, L. Bottou, Y. Bengio, P. Haffner, Gradient based learning applied to document recognition, Proc. IEEE 86 (11) (1998) 2278–2324.
[15] A. Krizhevsky and G. Hinton. Cifar-10 dataset. https://www.cs.toronto.edu/kriz/cifar.html.
[16] J. Bergstra, R. Bardenet, Y. Bengio, B. Kgl, Algorithms for hyperparameter optimization, in: Advances in Neural Information Processing Systems, 2011, pp. 2546–2554.
[17] Y. Bengio, Gradient-based optimization of hyperparameters, Neural Comput. 8 (2000) 1889–1900.
[18] J. Bergstra, U. Yoshua Bengio, Random search for hyper-parameter optimization, J. Mach. Learn. Res. (13) (2012) 281–305.
[19] J. Snoek, H. Larochelle, R.P. Adams, Practical bayesian optimization of machine learning algorithms, Adv. Neural Inf. Process. Syst. (25) (2012) 2960–2968.
[20] Y. Sun, B. Xue, and M. Zhang, Evolving deep convolutional neural networks for image classification (2017). arXiv:1710.10741.
[21] B. Baker, O. Gupta, N. Naik, and R. Raskar, Designing neural network architectures using reinforcement learning (2016) arXiv Prepr 116.
[22] B. Zoph, and Q.V. Le, Neural architecture search with reinforcement learning (2016) arXiv prepr. arXiv:1611.01578.
[23] Z. Zhong, J. Yan, and C.L. Liu, Practical network blocks design with Q-Learning (2017) arXiv prepr. arXiv:1708.05552.
[24] M. Suganuma, S. Shirakawa, T. Nagao, A genetic programming approach to designing convolutional neural network architectures, Genet. Evol. Comput. Conf. (2017) 497–504.
[25] E. Real, S. Moore, A. Selle, S. Saxena, Y.L. Suematsu, Q. Le, and A. Kurakin, Large-scale evolution of image classifiers (2017). arXiv:1703.01041.
[26] X. Gastaldi, Shake-shake regularization (2017). arXiv:1705.07485.
[27] K. He, X. Zhang, S. Ren, J. Sun, Deep residual learning for image recognition, in: 2016 IEEE Conference on Computer Vision and Pattern Recognition, 2016, pp. 770–778.
[28] E. Dufourq, and B.A. Bassett, EDEN: evolutionary deep networks for efficient machine learning (2017). arXiv:1709.09161.
[29] S.H. Hasanpour, M. Rouhani, M. Fayyaz, and M. Sabokrou, Lets keep it simple, using simple architectures to outperform deeper and more complex architectures (2016). arXiv:1608.06037.
[30] L. Wan, M. Zeiler, S. Zhang, Y. LeCun, and R. Fergus, Regularization of neural networks using dropconnect, 1 (2013) 109–111.
[31] K. Simonyan, A. Zisserman, Very deep convolutional networks for large-scale image recognition, Int. Conf. Learn. Represent. (2015) 114.
[32] B. Grigorian, G. Reinman, Accelerating divergent applications on simd architectures using neural networks, in: 32nd IEEE International Conference on Computer Design, 2014, pp. 317–323.
[33] Z. Du, A. Lingamneni, Y. Chen, K.V. Palem, O. Temam, C. Wu, Leveraging the error resilience of neural networks for designing highly energy efficient accelerators, IEEE Trans. Comput. Des. Integr. Circt. Syst. 34 (8) (2015) 1223–1235.
[34] T. Moreau, M. Wyse, J. Nelson, A. Sampson, H. Esmaeilzadeh, L. Ceze, M. Oskin, SNNAP: approximate computing on programmable SOCS via neural acceleration, in: 2015 IEEE 21st International Symposium on High-Performance Computer Architecture, 2015, pp. 603–614.
[35] A. Yazdanbakhsh, J. Park, H. Sharma, P. Lotfi-Kamran, H. Esmaeilzadeh, Neural acceleration for GPU throughput processors, in: Proceedings of the 48th International Symposium on Microarchitecture – MICRO48, 2015, pp. 482–493.
[36] H. Li, A. K., I. Durdanovic, H. Samet, and H.P. Graf, Pruning filters for efficient convnets (2016). arXiv:1608.08710.
[37] S. Han, J. Pool, J. Tran, W. Dally, Learning both weights and connections for efficient neural network, in: Advances in neural information processing systems, 2015, pp. 1135–1143.
[38] H. Hu, R. Peng, Y.W. Tai and C.K. Tang, Network trimming: a data-driven neuron pruning approach towards efficient deep architectures (2016) arXiv preprint arXiv:1607.03250.
[39] L. Li, Y. Xu, J. Zhu, Filter level pruning based on similar feature extraction for convolutional neural networks, IEICE Trans. Inf. Syst. 101 (4) (2018) 203–1206.
[40] S. Srinivas and R.V. Babu, Data-free parameter pruning for deep neural networks (2015). arXiv:1507.06149.
[41] E. Zitzler, L. Thiele, Multiobjective optimization using evolutionary algorithms—a comparative case study, in: International Conference on Parallel Problem Solving From Nature, 1998, pp. 292–301.
[42] H. Esmaeilzadeh, E. Blem, R. St. Amant, K. Sankaralingam, D. Burger, Dark silicon and the end of multicore scaling, IEEE Micro 32 (3) (2012) 122134.
[43] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, Tensorflow: a system for large-scale machine learning, in: 12th USENIX Symposium on Operating Systems Design and Implementation, 2016, pp. 265–283.
[44] M. Loni, M. Daneshtalab, M. Sjodin, ADONN: adaptive design of optimized deep neural networks for embedded systems, in: Euromicro Conference on Digital System Design, 2018, pp. 397–404.
[45] Z. Lu, I. Whalen, V. Boddeti, Y. Dhebar, K. Deb, E. Goodman, W. Banzhaf, NSGA-Net: neural architecture search using multi-objective genetic algorithm, in: Proceedings of the Genetic and Evolutionary Computation Conference, 2019, pp. 419–427.
[46] H. Mahdiani, A. Ghanbari, A. Khadem, M. Modarressi, M. Daneshtalab, ΔNN: power-efficient neural network acceleration using differential weights, IEEE Micro (2020).

Mohammad Loni is a Ph.D. student at the School of Innovation, Design, and Engineering at Mälardalen University since October 2017. He received his B.Sc. degree in computer hardware engineering and the M.Sc. degree in computer science from Shiraz University in 2017. He is a member of the Dependable Platforms for Autonomous Systems and Control (DPAC), HERO, DeepMaker and FAST-ARTS projects at Mälardalen University. He is working on efficient implementation of neural networks on FPGA.

Sima Sinaei received the B.S. and M.S. and P.h.D degrees in (Hardware and computer Architecture) computer engineering from Shahid-bahonar University of Kerman, University of Tehran, University of Tehran, Iran, in 2008 and 2011, 2018 respectively. She is currently working as a Postdoc researcher at IDT, in Malardalen University, Sweden. Her current research interests include Machine Learning, Deep Learning, Neural Network Architecture Optimization, Design Methodology for heterogeneous embedded systems, Design Space Exploration and Mapping Algorithms for multiprocessor systems.

Ali Zoljodi is a Master student at Shiraz technical University. He is working on efficient implementation of stereo-vision algorithms on multi-core platforms.

Masoud Daneshtalab: (http://www.idt.mdh.se/~md/) is currently a tenured associate professor at Mälardalen University (MDH) in Sweden while co-leading the Heterogeneous System research group (www.es.mdh.se/hero/). He has represented Sweden in the management committee of the EU COST Actions IC1202: Timing Analysis on Code-Level (TACLe). Since 2016 he is in Eurimicro board of Director and a member of the HiPEAC network. His research interests include interconnection networks, hardware/software co-design, deep learning architectures, and multi-objective optimization. He has published 2 book, 7 book chapters, and over 200 refereed international journals and conference papers.

Mikael Sjödin: The current research goal is to find methods that will make software development cheaper, faster and yield software with higher quality. Concurrently, Mikael is also been pursuing research in analysis of real-time systems, where the goal is to find theoretical models for real-time systems that will allow their **timing behavior** and **memory consumption** to be calculated. Mikael received his PhD in computer systems 2000 from Uppsala University (Sweden). Since then he has been working in both academia and in industry with embedded systems, real-time systems, and embedded communications. Previous affiliations include Newline Information, Melody Interactive Solutions and CC Systems. In 2006 he joined the MRTC faculty as a full professor with specialty in real-time systems and vehicular software-systems.