Genetic Programming and Evolvable Machines (2022) 23:351–374
https://doi.org/10.1007/s10710-022-09441-z

Evolutionary approximation and neural architecture search

Michal Pinos[1] · **Vojtech Mrazek**[1] · **Lukas Sekanina**[1]

Published online: 11 June 2022

Abstract

Automated neural architecture search (NAS) methods are now employed to routinely deliver high-quality neural network architectures for various challenging data sets and reduce the designer's effort. The NAS methods utilizing multi-objective evolutionary algorithms are especially useful when the objective is not only to minimize the network error but also to reduce the number of parameters (weights) or power consumption of the inference phase. We propose a multi-objective NAS method based on Cartesian genetic programming for evolving convolutional neural networks (CNN). The method allows approximate operations to be used in CNNs to reduce the power consumption of a target hardware implementation. During the NAS process, a suitable CNN architecture is evolved together with selecting approximate multipliers to deliver the best trade-offs between accuracy, network size, and power consumption. The most suitable $8 \times N$-bit approximate multipliers are automatically selected from a library of approximate multipliers. Evolved CNNs are compared with CNNs developed by other NAS methods on the CIFAR-10 and SVHN benchmark problems.

Keywords Approximate computing · Convolutional neural network · Cartesian genetic programming · Neuroevolution · Energy efficiency

✉ Lukas Sekanina
sekanina@fit.vutbr.cz

Michal Pinos
ipinos@fit.vutbr.cz

Vojtech Mrazek
mrazek@fit.vutbr.cz

[1] Faculty of Information Technology, Brno University of Technology, Brno, Czech Republic

1 Introduction

Deep neural networks (DNNs) are complex computational models that must be trained using application-specific data to provide requested performance [1]. The DNN design consists of creating DNN's architecture (including its sizing and selection of appropriate layers) and training, i.e., optimizing trainable parameters of the architecture (the so-called weights) to minimize an application-specific loss function. As current DNNs can have thousands of layers and millions of weights [2], their design is difficult and time-consuming even for experts.

DNNs are typically trained on graphics processing units (GPUs) and operated in data centers. A new trend is to adapt them for resource-constrained devices such as mobile phones, consumer electronics, and Internet of Things (IoT) nodes to enable intelligent processing and analysis of images, speech, text, and other types of data directly in the end devices [2]. Particularly, the inference process of a fully trained DNN is usually accelerated in hardware to meet real-time and other challenging constraints required by modern applications. The current approach to hardware implementations of DNNs is typically based on semi-automated simplification of a network model, which was initially developed for a GPU and trained on GPU without considering any hardware implementation aspects. One of the most successful techniques adopted for hardware-aware DNN design is approximate computing [3]. By introducing suitable approximations at the level of DNN architecture, data representation, arithmetic operations, and memory subsystem, DNN implementations showing excellent trade-offs between the error and resources have been obtained for common benchmark problems [4, 5]. To reduce the human effort, automated DNN design methods known as the *neural architecture search* (NAS) have been developed in recent years [6–8]. The evolutionary design of neural networks, introduced for smaller networks over three decades ago [9], is now extended for DNNs, mainly because it can easily be implemented as a multi-objective design method [10, 11]. In addition to the quality (the error) of resulting DNNs, recently proposed hardware-aware NAS methods [12–14] try to produce DNNs optimized in terms of latency and resources (such as the area on a chip, memory usage, and power consumption).

This paper is focused on the hardware-aware NAS applied to the automated design of *convolutional neural networks* (CNNs) for image classification. Because multiplication is performed quite often in CNNs, approximate multipliers have been introduced to CNNs [15, 4, 16] to reduce their power consumption. However, except our initial study [17], there is not any research dealing with the automated design of CNN architectures in which standard multipliers can be replaced by approximate multipliers. The proposed method, EvoApproxNAS (Evolutionary Approximation and Neural Architecture Search), is based on multi-objective Cartesian genetic programming (CGP), optimizing the accuracy and network size. Moreover, it can automatically select the most suitable approximate multiplier for each convolutional layer of the CNN to minimize the energy needed by multiplications. Candidate approximate multipliers are chosen from a library of approximate circuits called EvoApproxLib [18]. While CGP delivers the network topology and selects suitable multipliers, the weights are obtained by a standard gradient method implemented in

TensorFlow [19]. The NAS supporting approximate multipliers in CNNs is obviously more computationally expensive than the NAS for common CNNs. The reason is that TensorFlow does not support the fast execution of CNNs that contain non-standard operations such as approximate multipliers. We propose eliminating this issue by employing TFApprox [20], which extends TensorFlow to support approximate multipliers in CNN training and inference. EvoApproxNAS is devoted to smaller CNNs (with around 1 million trainable parameters) that can be adopted in resource-constrained embedded devices. EvoApproxNAS is evaluated on CIFAR-10 and SVHN data sets and compared with other relevant methods.

The key contributions and the extensions concerning our initial study [17] are as follows:

- We present the EvoApproxNAS method capable of an automated design of CNN topology with an automated selection of suitable approximate multipliers for each convolutional layer. The methodology uniquely integrates a multi-objective CGP and TFApprox-based training and evaluation of CNNs containing approximate circuits.
- We implemented a common version of CGP while the previous work was just inspired by CGP; for example, it did not allow mutation of layer types. We also used data augmentation techniques to reduce the resulting error and improve the learning progress.
- For each convolutional layer, EvoApproxNAS determines the most suitable approximate multiplier; 35 versions of approximate and accurate 8×4, 8×5, 8×6, 8×7, and 8×8-bit multipliers are supported. Only 8×8-bit approximate multipliers were used in [17].
- Compared to [17], the top-1 accuracy on CIFAR-10 was improved from 83.98 to 93.2% for the same number of evaluations per CGP run. Evolved CNNs were compared with CNNs developed by other NAS methods and semi-automated ALWANN method [21] targeting the same design specification. The experiments are also reported for the SVHN data set.

The rest of the paper is organized as follows. Section 2 surveys related work in the area of CNN design, CNN optimization and neural architecture search. The proposed method, EvoApproxNAS , is presented in Sect. 3. We conducted a number of experiments to evaluate EvoApproxNAS ; their results are reported in Sect. 4. Conclusions are given in Sect. 5.

2 Related work

We briefly introduce CNNs, implementation options for CNNs, and methods developed to optimize these implementations. Then, the basic principles of the NAS methods and hardware-aware NAS methods are presented in the context of the proposed method.

2.1 Convolutional neural networks and their implementation

Convolutional neural networks are deep neural networks employing, in addition to other layer types, the so-called *convolutional layers* [1]. Each convolutional layer generates, by applying one or several convolutional *kernels* (filters), a successively higher level of abstraction of the input data, called a *feature map*. The core computational procedure of a convolutional layer is a high-dimensional convolution. The number of trainable parameters in the convolutional kernels is small compared to the common fully connected layers processing feature maps of the same size. A nonlinear activation function typically follows each convolutional layer. *Pooling layers* combine, by applying the averaging or maximum operators, a set of input values into a small number of output values to reduce the dimension of feature maps. Layers can be connected to form specialized structures (e.g., residual modules, see Sect. 3.1) used as building blocks of complex CNNs. The designer's task is to compose a suitable set of layers and their sizes to establish a CNN for a given application. During the training, which is usually based on a gradient method, the weights are optimized to minimize an application-specific loss function.

Because CNNs can contain hundreds of layers and millions of network elements, training is very time-consuming and can take days on a cluster of GPUs. However, even the inference phase of trained CNN is expensive. For example, the ResNet-50 network requires performing $3.9 \cdot 10^9$ multiply-and-accumulate (MAC) operations to classify one single input image when the challenging ImageNet benchmark is considered [22].

In addition to general-purpose multicore CPUs and GPUs, various specialized accelerators, ranging from high-performance devices in data centers to low-power chips in IoT nodes, have been developed for CNNs. These accelerators are implemented either as application-specific integrated circuits [23–25] (including Tensor Processing Units—TPUs [26]), in field-programmable gate arrays (FPGAs) [27], or by extending common processors [28].

CNNs are typically applied in error-resilient applications in which a minor error introduced by inexact computing is often invisible to the end-user. Hence, CNNs can be simplified to reduce hardware resources, power consumption, or latency [29]. Many of these techniques can be seen as approximation techniques. They focus on optimizing the data representation, pruning less important connections and neurons, approximating arithmetic operations, compression of weights, and employing various smart data transfer and memory storage strategies [30, 4, 5]. For example, the Ristretto tool is specialized in determining the number of bits needed for arithmetic operations [31] because the standard 32-bit floating-point arithmetic is too expensive and unnecessarily accurate for CNNs. Further savings in energy are obtained not only by the bit width reduction of arithmetic operations but also by introducing approximate operations, particularly to the multiplication circuits [15, 21, 16]. This approach is motivated by the fact that arithmetic operations conducted in the inference are responsible for 10% to 40% of total energy (depending on a particular CNN and a hardware platform used to implement it) [2]. A strategy enabling the cross-layer approximation has been proposed for the RAPID-AI accelerator in which software, architecture, and hardware are approximated and optimized together [5].

2.2 Neural architecture search

The NAS has been introduced to automate the neural network design process [6–8]. NAS methods can be classified according to the *search mechanism* which is usually based on reinforcement learning [8] or evolutionary algorithms (EA) [32]; however, many other search methods have been utilized [7]. During the search process, a common approach is to train and validate each candidate CNN to determine its accuracy. However, this is computationally expensive, even for data sets of moderate size. Hence, various alternative approaches such as surrogate models, one-shot search, and differentiable NAS have been proposed to accelerate the NAS method [6, 7].

NAS methods were initially constructed as single-objective methods to minimize the classification error [33, 34]. By introducing the hardware-aware NAS, the search process becomes multi-objective, and other objectives such as latency for a mobile phone or power consumption for an IoT node can be optimized. The best-performing CNNs obtained by NAS currently show superior quality concerning human-designed CNNs if the multi-objective scenario is considered [35].

As our NAS method employs genetic programming, we briefly discuss the main components of the EA-based approaches. Regarding the *problem representation*, direct [35, 10, 34] and indirect (generative) [11] encoding schemes have been investigated. The selection of *genetic operators* is tightly coupled with the chosen problem representation. While mutation is the key operator for CGP [34], the crossover is crucial for binary encoding of CNNs as it allows population members to share common building-blocks [35, 10]. The multi-objective search often employs the *non-dominated sorting*, known from, e.g., the NSGA-II algorithm [36], which enables to maintain diverse trade-offs between conflicting design objectives. The evolutionary search is usually combined with *learning* because it is very inefficient to let the evolution find the weights. A candidate CNN, constructed using the information available in its genotype, is trained using common learning algorithms available in popular DNN frameworks such as TensorFlow [19]. The number of epochs and the training data size have to be carefully chosen to reduce the training time, although by doing so, the fitness score can wrongly be estimated. The CNN accuracy, which is obtained using test data, is interpreted as the fitness score.

The entire *neuro-evolution* is very time and resources demanding and, hence, only several hundreds of candidate CNNs can be generated and evaluated in one EA run. All mathematical operations carried out by common software tools for DNNs (such as TensorFlow) are highly optimized and work with standard floating-point numbers on GPUs. If one needs to replace these operations with approximate operations, these non-standard operations have to be expensively emulated. The CNN execution is then significantly slower than with the floating-point operations. This problem can partly be eliminated by using TFApprox in which all approximate operations are implemented as look-up tables and accessed through a texture memory mechanism of CUDA capable GPUs [20].

The most recent hardware-aware NAS methods optimize network architecture together with the accelerator configuration [14]. The search space is thus extended by a set of parameters that define, for example, the quantization setup, the number of

processing elements of the accelerator, the size of buffers, and the dataflow organization. In addition to determining the accuracy of a candidate CNN, its hardware implementation has to be modeled or simulated to obtain latency, power consumption, and other hardware-related metrics. Lu et al. [37] introduces the hardware search space and a joint exploration of the space of neural architectures, hardware implementations, and layer-wise quantization. CNNs are evaluated in terms of accuracy, the area used in the FPGA, throughput, and memory usage. Another method, NAAS, shows how the accuracy and EDP (Energy Delay Product) trade-offs can be improved if the search algorithm considers only the network architecture, then the network architecture and accelerator sizing, and finally the network architecture, accelerator sizing and compiler mapping optimization [38].

3 Evolutionary NAS with approximate multipliers

Our method is inspired by paper [34] whose authors used CGP to evolve CNNs. In our initial study [17], we introduced a multi-objective search to the CGP-based NAS and optimized the selection of an 8-bit approximate multiplier for convolutional layers. The proposed EvoApproxNAS method fully implements CGP, employs data augmentation techniques to improve the learning progress, and selects the most suitable $8 \times$ N-bit approximate multiplier for each convolutional layer.

The role of CGP is to provide a suitable CNN architecture (utilizing approximate multiplication in convolutional layers) for a hardware CNN accelerator. The weights are obtained using a common gradient method implemented in TensorFlow. We assume that the accelerator employs a two-dimensional array of processing elements (PE). A typical PE multiplies the input with its weight and updates the sum maintained in each layer. The PE array can be operated in several ways, see [2]; we suppose that the CNN accelerator is organized as a generic PE array, and each CNN layer can be configured to use a different approximate multiplier.

In this section, we will first describe the proposed CGP-based NAS, which is developed for CNNs with floating-point arithmetic operations executed on GPU. Section 3.5 deals with the integration of approximate multipliers into the design process.

3.1 CNN representation

CGP was developed to automatically design programs and circuits that are modeled using directed acyclic graphs [39]. A candidate solution is represented using a two-dimensional array of $n_c \times n_r$ nodes, consuming n_i inputs and producing n_o outputs. In the case of the evolutionary design of CNNs, each node represents either one layer (e.g., convolutional, batch normalization, activation, pooling) or a module (e.g., residual, bottleneck residual, or inverted bottleneck residual) of a CNN. Each node of j-th column reads a tensor coming from column $1, 2, \ldots, j-1$ and produces another tensor. In our case study, CNNs accept one 4D input tensor (of shape [batch_size, height, width, depth]) holding one batch of input images and produce

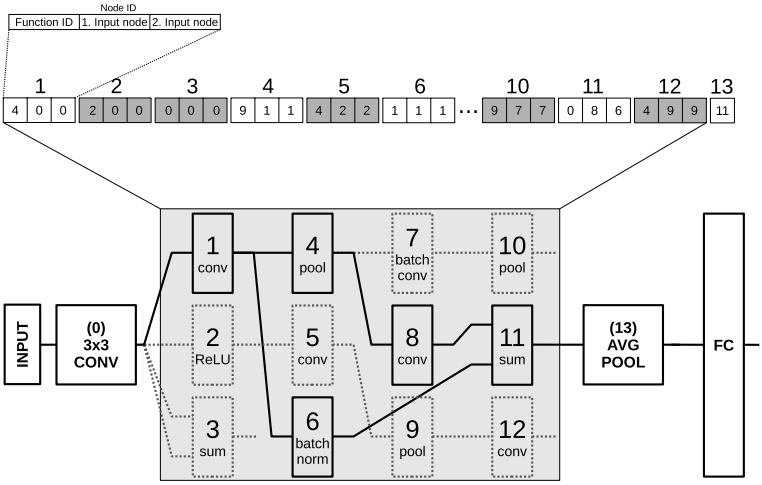


Fig. 1 Example of a candidate CNN whose front and rear parts are fixed, and the middle layers are represented as nodes of the CGP array (Bottom). Active nodes are denoted by the white color (in the chromosome) and solid lines (in the CNN), whereas inactive nodes are distinguished by grey color (in the chromosome) and dotted lines (in the CNN). Every node is encoded using three integers (ID, input 1, input 2) in the chromosome (Top). For clarity of the drawing, the ID represents the entire node record, i.e., all the parameters defining one layer of a candidate CNN

one 2D output tensor (of shape [batch_size, class_probs]), which is treated as a matrix, in which each row corresponds to a vector of class probabilities.

Figure 1 shows how a candidate CNN is encoded in the chromosome. The input of the CNN model (i.e., an image) undergoes a pre-processing phase implemented with the 3x3 convolution. This part of CNN is fixed. The output of this pre-processing phase serves as the primary input (node 0) to the CGP array of nodes. A unique identifier (an address) is assigned to each node; see 1 to 12 in our example shown in Fig. 1. The rear part of CNN consisting of node 13 (an average polling layer) and a fully connected layer is not evolved because almost all CNN-based image classifiers employ such a structure. Every evolvable node is encoded using three integers. Two of them define the nodes to which node's inputs are connected. The third integer is a pointer to a record that holds all the node parameters, including its function code. Table 1 contains all functions (i.e., layer or module types) that a node can implement. Depending on the function type, various parameters can be set up, for example, activation function type, kernel size, stride size, and used approximate multiplier. In Sect. 4.1, approximate multipliers that are supported by the nodes will be introduced. The last integer of the chromosome specifies the output node of the CGP array.

Figure 1 also shows that there exist two types of nodes—*active* and *inactive*. Active nodes, denoted by the white color in the chromosome and solid line in the phenotype, contribute to the output of the CNN. Inactive nodes are denoted by the grey color in the chromosome and dotted line in the phenotype.

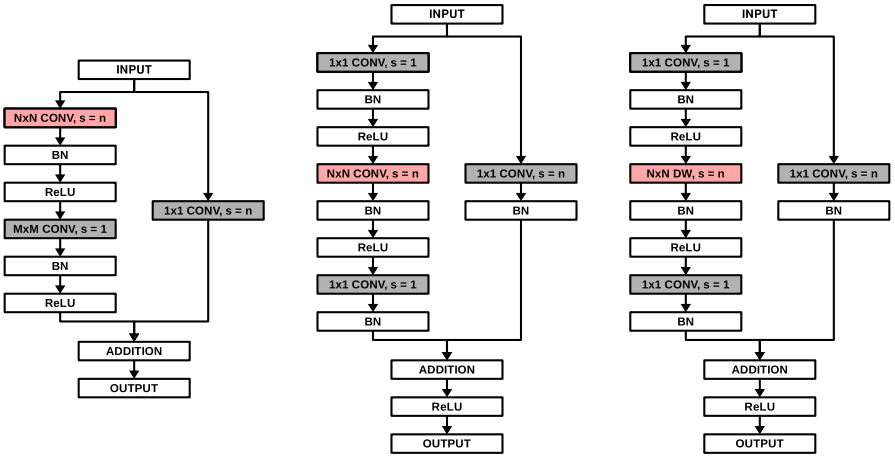
Table 1 CGP function table for evolutionary design of CNNs

ID	Name	Arity	Parameters	Values
0	summation	2	–	–
1	batch_norm	1	–	–
2	activation	1	activation	{*relu*, *elu*, *leakyrelu*, *prelu*}
3	depthwise_conv2d	1	kernel_size	$\{2\times2, 3\times3\}$
			strides	$\{1\times1, 2\times2\}$
			activation	{*relu*, *elu*}
			use_bias	{*True*, *False*}
			approx_mul_file	{*list_of_ax_mults*}
4	conv2d	1	kernel_size	$\{2\times2, 3\times3\}$
			channels	{32, 40, 48, 64, 80, 96, 112, 128}
			strides	$\{1\times1, 2\times2\}$
			activation	{*relu*, *elu*}
			use_bias	{*True*, *False*}
			approx_mul_file	{*list_of_ax_mults*}
5	basic_residual	1	kernel_size	$\{3\times3, 4\times4\}$
			channels	{32, 40, 48, 64, 80, 96, 112, 128}
			strides	$\{1\times1, 2\times2\}$
			preact	{*True*, *False*}
			approx_mul_file	{*list_of_ax_mults*}
6	residual_bottleneck	1	kernel_size	$\{3\times3, 4\times4\}$
			factor	{1.0, 1.5, 2.0, 2.5, 3.0, 3.5, 4.0}
			strides	$\{1\times1, 2\times2\}$
			approx_mul_file	{*list_of_ax_mults*}
7	residual_inverted	1	kernel_size	$\{3\times3, 4\times4\}$
			factor	{1.0, 1.5, 2.0, 2.5, 3.0, 3.5, 4.0}
			strides	$\{1\times1, 2\times2\}$
			approx_mul_file	{*list_of_ax_mults*}
8	dropout	1	min	{0.05, 0.10, 0.15}
			max	{0.20, 0.25, 0.30}
9	max_pooling	1	–	–

Approximate multipliers in the list *list_of_ax_mults* are indexed using integers

The functions listed in Table 1 are either standard layers (such as convolutional layer, depthwise convolutional layer, activation layer etc.) or various versions of residual module [40] (shown in Fig. 2). Selected layers and modules are introduced in the following paragraphs; the remaining ones are standard.

The *summation layer* accepts tensors t_1 and t_2 with $shape(t_1) = (h_1, w_1, c_1)$ and $shape(t_2) = (h_2, w_2, c_2)$, where h_x, w_x and c_x are height, width and the number of channels respectively ($x \in \{1, 2\}$). It outputs t_o, i.e. the sum of t_1 and t_2, defined as



(a) Basic residual. **(b)** Bottleneck residual. **(c)** Inverted residual.

Fig. 2 Diagrams of **a** basic residual module, **b** bottleneck residual module, and **c** inverted residual module. Red layers can use the approximate multipliers, whereas only accurate multipliers are supported in the grey ones (Color figure online)

$$t_o = t_1 + t_2 \iff t_o^{ijk} = t_1^{ijk} + t_2^{ijk} \text{ for } \begin{array}{l} i = 0, ..., c-1 \\ j = 0, ..., h-1 \\ k = 0, ..., w-1. \end{array} \quad (1)$$

It has to be ensured that the height and width of both the tensors are identical. If it is not so, the common max-pooling technique is applied to the 'bigger' tensor to unify these dimensions. The problem with unmatched number of channels is resolved by zero padding applied to the 'smaller' tensor, i.e., $shape(t_o) = (h_o, w_o, c_o)$, where $h_o = min(h_1, h_2)$, $w_o = min(w_1, w_2)$ and $c_o = max(c_1, c_2)$.

The *residual module* contains a sequence of $N \times N$ and $M \times M$ convolutions that can be skipped, which is implemented by the addition layer. The residual module, shown in Fig. 2a, consists of two convolutional layers with the filters $N \times N$ and $M \times M$, both either preceded or followed by batch normalization and ReLU activation layers based on whether we use pre-activation version or not. In parallel, one convolution with filter 1×1 is computed. Results of $M \times M$ and 1×1 convolutions are added together to form a result. Convolutional layers with filters $N \times N$ and 1×1 operate with stride n to ensure that the inputs to the addition layer have the same dimensions.

The *bottleneck residual module* shown in Fig. 2b, is a variant of the residual module, which comprises of one convolutional layer with filter $N \times N$, which applies batch normalization and ReLU activation to its input and output. This convolutional layer is surrounded by two 1×1 convolutional layers, the first reducing the number of channels by a factor f and the second one restoring the number of channels to the original number. The idea behind this approach is to reduce the number of channels

for the $N \times N$ convolution (and thus make it less computationally intensive) and then increase the number of channels to their original value. In parallel to this, another 1×1 convolutional layer is employed to ensure that the inputs to the addition layer have the same dimensions. The parallel $N \times N$ convolution layer and 1×1 convolution layers operate with stride n, whereas all other convolutional layers use stride 1.

We also support the *inverted bottleneck* variant of the residual module, shown in Fig. 2c. Compared to the bottleneck residual module, this variant uses the depthwise convolutional layer, which applies only one filter to each of its inputs. As the name suggests, this module uses the opposite approach to the bottleneck residual module. The first 1×1 convolutional layer increases the number of the channels by the factor f for the following $N \times N$ depthwise convolutional layer with stride n, which applies batch normalization and ReLU activation to its input and output. The second 1×1 convolutional layer reduces the number of channels to its original value. In parallel to this, another 1×1 convolutional layer with stride n is employed to ensure that the inputs to the addition layer have the same dimensions.

3.2 Genetic operators

The pseudo-code of the proposed mutation operator is listed in Alg 1. During the mutation of an individual (I), an active node (N) is randomly selected and its function is changed with probability p_{funct} and connection is changed with probability p_{conn}. Functions responsible for these changes are *mutate_function*(N') and *mutate_connection*(N'), respectively. When the node's function undergoes the mutation, it is replaced by a function randomly chosen from the set of functions specified in Table 1. Parameters of this function are randomly selected from available values. On the other hand, connection mutation is implemented in such a way that the node is reconnected to a randomly chosen node which is placed in up to L previous columns, where L is a user-defined parameter (the so-called L-back parameter in CGP [39]). Finally, the resulting offspring M is obtained from the individual I by substituting node N by node N'. Whenever any parameter has to be randomly determined in this algorithm, all its potential values have the same probability of selection. Like most search algorithms based on CGP, no crossover operator is utilized.

Algorithm 1 Mutation

Input
  I Individual to mutate
  p_{funct} Probability of function mutation
  p_{conn} Probability of connection mutation
Output
  M Offspring
procedure MUTATE(I, p_{funct}, p_{conn})
  $N \leftarrow select\_random\_active\_node(I)$
  $P_{funct} \leftarrow$ UNIFORM(0,1)
  $P_{conn} \leftarrow$ UNIFORM(0,1)
  $N' \leftarrow N$
  if $P_{funct} \leq p_{funct}$ **then**
    $N' \leftarrow mutate\_function(N')$
  if $P_{conn} \leq p_{conn}$ **then**
    $N' \leftarrow mutate\_connection(N')$
  $M \leftarrow I$[replace $N$with N']
  return M

3.3 Fitness functions

The objectives are to maximize the CNN accuracy and to minimize the CNN complexity (expressed as the number of parameters) and power needed to perform all multiplications in the convolutional layers. As this study is devoted to analyzing the impact of utilizing approximate multipliers in CNNs, we only consider the power associated with multiplication. Reporting the overall energy needed by the accelerator for one inference would require employing a significantly more complex model of the underlying hardware, which is out of the scope of this study.

Hence, the particular fitness functions are defined as follows. The fitness function expressing the accuracy of a candidate network x (evaluated using a data set D), is calculated using TensorFlow as $f_1(x, D) = \text{Top-1-Accuracy}(x, D)$. The number of parameters in the entire CNN x is captured by fitness function $f_2(x)$.

Power consumption of convolutional layers utilizing approximate multipliers is estimated as

$$f_3(x) = \sum_{i}^{N_{CL}} N_{mult}(i) \cdot P_{mult}(i), \tag{2}$$

where N_{CL} is the number of convolutional layers of CNN x, N_{mult} is the number of multiplications executed during inference in convolutional layer i and $P_{mult}(i)$ is the power consumption of the approximate multiplier used in the i-th layer.

3.4 Search algorithm

The search algorithm (see Algorithm 2) is constructed as a multi-objective evolutionary algorithm inspired in CGP-based NAS [34] and NSGA-II [36].

The initial population is randomly generated as in the standard CGP, i.e., each gene of the chromosome is initialized with a randomly selected value from a set of eligible values for the particular position. Functions and their parameters are taken from Table 1. Possible values for connections are determined by the CGP parameters n_c, n_r, n_i, n_o, and L.

Training of a CNN is always followed by testing to obtain fitness values $f_1(x)$, $f_2(x)$, and $f_3(x)$. Training is conducted for E_{train} epochs on complete training data set D_{train}. The accuracy of the candidate CNN (i.e., f_1) is determined using the entire test data set D_{test} (Algorithm 1, line 2). To overcome the overfitting problem during the training, the cutout data augmentation [41], L1 and L2 regularizations were employed [42]. To further speed up the exploration process, additional enhancements, in the form of early stopping techniques, have been implemented. They help to reduce the evaluation time of candidates that are significantly underperforming (e.g., due to the poor architecture design). If a candidate solution represents a CNN model, which does not achieve some meaningful accuracy (15%, 55%, and 65%, respectively) after some epochs of training (1, 3, and 7, respectively), then its training is prematurely stopped, and a low fitness score is assigned.

The offspring population (O) is created by applying the mutation operator on each individual of the parental population P. The offspring population is evaluated in the same way as the parental population (Algorithm 1, line 5). Populations P and O are joined to form an auxiliary population R (line 6). The new population is constructed by selecting non-dominated individuals from Pareto fronts (PF) established in R (lines 8–10). If any front must be split, a crowding distance is used for the selection of individuals to P (lines 12–13) [36]. The search terminates after evaluating a given number of CNNs.

As the proposed algorithm is multi-objective, the result of a single CGP run is a set of non-dominated solutions. At the end of evolution, the best-performing individuals from this set are re-trained (fine-tuned) for $E_{retrain}$ epochs on the complete training data set D_{train} and the final accuracy is reported on the complete test data set D_{test}.

Algorithm 2 Neuroevolution

```
P ← initial_population(); g ← 0
training_evaluation(P, E_train, D_train, D_test)
repeat
    O ← mutate(P')
    training_evaluation(O, E_train, D_train, D_test)
    R ← P ∪ O; P ← ∅
    while |P| ≠ population_size do
        PF ← non_dominated(R)
        if |P ∪ PF| ≤ population_size then
            P ← P ∪ PF
        else
            n ← |PF ∪ P| − population_size
            P ← P ∪ crowding_reduce(PF, n)
        R ← R \ PF
    g ← g + 1
until stop_criteria_satisfied()
training_evaluation(P, E_retrain, D_train, D_test)
return (P)
```

3.5 NAS with approximate multipliers

In order to find the most suitable approximate multipliers for convolutional layers and modules of a CNN architecture, every node has the *approx_mul_file* parameter specifying a particular implementation of the used multiplier in each layer (see Table 1). The search algorithm thus optimizes N_{CL} additional parameters.

In addition to 8-bit approximate multipliers, we also support the non-standard 8 × N-bit approximate multipliers. While the first input of the multiplier has 8 bits, the second input has 7, 6, 5, or 4 bits to represent the weight in a simplified form. These non-standard multipliers can contribute not only in reducing the overall energy of the multiplication operations conducted in CNNs, but also in reducing the memory usage and power consumption as fewer bits are needed to store the weights. Accurate versions of these multipliers can also be utilized [15]. For each bit width, we selected several implementations of approximate multipliers from the EvoApproxLib.[1] library [18] These approximate multipliers show different trade-offs between power consumption and error. By allowing the CGP to choose the most suitable bit width of the operand, the quantization problem is thus automatically solved.

While adopting approximate multipliers in ordinary layers, such as convolutional layers or depthwise convolutional layers, is straightforward, utilizing approximate multipliers in residual modules is more complicated. Figure 2 determines the layers of the residual modules which can employ the approximate multipliers (red) and which always use the accurate implementation (grey).

Before a candidate CNN is sent to TensorFlow for training or testing, the standard multipliers used in individual layers or modules are replaced with the approximate

[1] http://www.fit.vutbr.cz/research/groups/ehw/approxlib/

Table 2 Parameters of the experiment

Parameter	Value	Description
n_r	10	Number of rows in the CGP grid.
n_c	30	Number of columns in the CGP grid.
L	4	L-back parameter.
pop_size	8	Number of individuals in the population.
G	10	Maximum number of generations.
D_{train}	50,000	CIFAR-10 training data set size.
	73,257	SVHN training data set size.
D_{test}	10,000	CIFAR-10 test data set size.
	26,032	SVHN test data set size.
E_{train}	10	Number of epochs (during the evolution).
$E_{retrain}$	150	Number of epochs (for re-training).
batch_size	128	Batch size.
p_{funct}	1.0	Probability of mutation of the function.
p_{conn}	1.0	Probability of mutation of the connection.
N_{mut}	1	Number of nodes for mutation.

multipliers specified by the *approx_mul_file* parameter. TensorFlow, with the help of TFApprox, performs all multiplications in the convolution computations in the forward pass of the learning algorithm with the approximate multipliers. In contrast, all computations in the backward pass remain with the standard floating-point multiplications.

4 Results

This section specifies and justifies the experimental setup utilized to evaluate the EvoApproxNAS method. Detailed experimental results obtained by EvoApproxNAS are presented in several subsections, in which we mainly analyze the impact of using approximate multipliers and compare the obtained results with similar methods.

4.1 Setup

EvoApproxNAS is primarily evaluated on the automated design of CNNs for the CIFAR-10 image classification data set [43]. The CIFAR-10 data set consists of 60,000 32×32-pixel color images in 10 classes, with 6000 images per class. There are 50,000 training images and 10,000 test images. The second benchmark is Google's Street View House Number (SVHN) digit classification data set [44]. The goal of CGP is to provide CNN architectures showing good trade-offs between the classification accuracy, size, and power consumption (of multipliers used in convolutional layers).

EvoApproxNAS was implemented in Python with the help of TensorFlow and TFApprox [20]. The fitness functions are defined according to Sect. 3.3. The weights are obtained using Adam optimization algorithm implemented in TensorFlow.

Table 2 summarizes all parameters of CGP and the learning method used in our experiments. These parameters were experimentally selected based on a few trial runs. Because of limited computational resources, we could generate and evaluate only $pop\_size + G \times pop\_size = 88$ candidate CNNs in each run.

Implementations of approximate multipliers were taken from EvoApproxLib, which also provided us with the pre-computed metrics for each circuit (the mean absolute error and power consumption). In total, CGP can exploit 35 multipliers—the accurate 8×8, 8×7, 8×6, 8×5, and 8×4-bit multipliers, ten 8×8-bit approximate multipliers, and five $8 \times N$-bit approximate multipliers, $N \in \{7, 6, 5, 4\}$. Inspired in ALWANN [21], we selected from EvoApproxLib such approximate multipliers that show diverse trade-offs between the error and power consumption; however, none of them consumes less than 0.100 mW as such multipliers are very inaccurate for our purposes.

Experiments were performed on a machine with two 12-core CPUs Intel Skylake Gold 6126, 2.6 GHz, 192 GB, equipped with four GPU accelerators NVIDIA Tesla V100-SXM2. A single CGP run with CNNs utilizing approximate multipliers can take up to 53 GPU hours (including the final re-training); a complete run of all scenarios requires around 130 GPU hours on average. When TFApprox emulates approximate multipliers, the average time needed for all inferences in ResNet-8 on CIFAR-10 is 1.7 s (initialization) + 1.5 s (data processing) = 3.2 s. If TensorFlow performs the same task in the 32-bit FP arithmetic, the time is 1.8 s + 0.2 s = 2.0 s. Hence, the time overhead coming with approximate operations is 37.5%.

4.2 The role of approximate multipliers in NAS

In the following descriptions, symbols A and E will denote Accuracy and Energy. We consider four scenarios to analyze the role of approximate multipliers in NAS:

- (S1) CNN architecture is co-optimized with the 8×8-bit approximate multipliers under fitness functions f_1 and f_3 (denoted 'EvoApproxNAS + 8×8-bit-approx-mult-A/E' in the following figures);
- (S2) CNN architecture is co-optimized with the $8 \times N$-bit approximate multipliers under fitness functions f_1 and f_3 (denoted 'EvoApproxNAS + $8 \times$ N-bit-approx-mult-A/E');
- (S3) The 8-bit exact multiplier is always used in NAS (denoted 'EvoApproxNAS + 8×8-bit-accurate-mult-A/E').
- (S4) The $8 \times N$-bit exact multipliers are always used in NAS (denoted 'EvoApproxNAS + $8 \times$ N-bit-accurate-mult-A/E').

Figure 3 plots a typical progress of a CGP run for individual scenarios on CIFAR-10 data set. The lighter points represent the candidate solutions in earlier generations. The stars indicate the Pareto fronts for each scenario. As the best trade-offs (Accuracy vs. Energy) are moving to the top-left corner of the figure, we observe that CGP can improve candidate solutions despite only ten populations being generated.

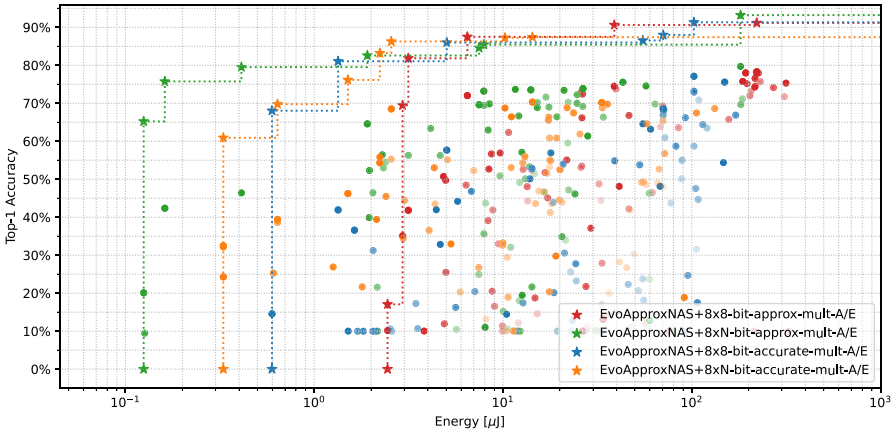


Fig. 3 The progress of evolution on the CIFAR-10 data set. The lighter points represent the candidate solutions in earlier generations. The stars indicate the final Pareto fronts for each scenario

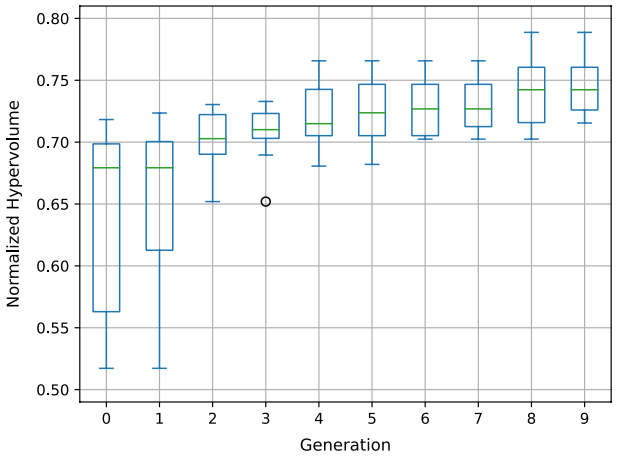


Fig. 4 The hypervolume performance metrics in the course of ten independent CGP runs in scenario S2 on the CIFAR-10 benchmark

For the purpose of statistical analysis of the CGP behavior, we employ the hypervolume (HV) performance metric, which gives the dominated area (hypervolume in the general case) from a set of evolved CNNs to a reference point, the so-called nadir point, which is estimated using a vector concatenating worst objective values of the Pareto front [45]. Hence, the higher the HV measures, the better solutions are obtained. Figure 4 shows, using box plots calculated from ten independent runs in scenario S2, that HV is continually improved and the multi-objective CGP thus converges.

The best trade-offs between the accuracy (estimated in the fitness function) and the total energy needed for performing all multiplications in convolutional layers

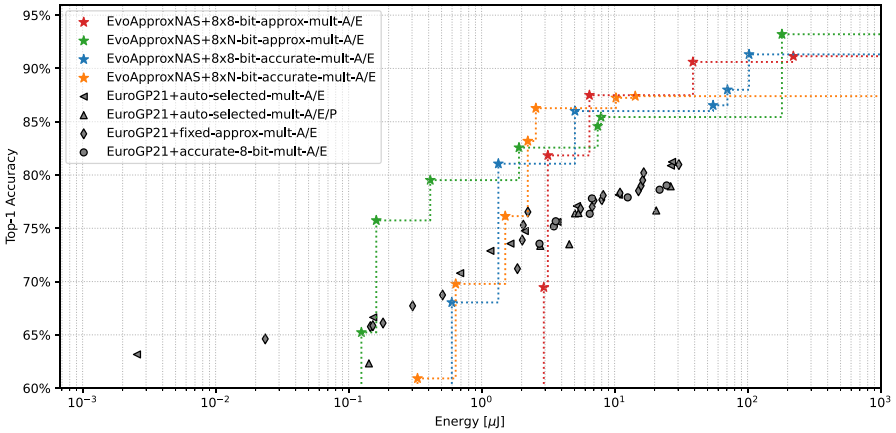


Fig. 5 Trade-offs between the accuracy and the total energy needed for performing all multiplications in convolutional layers during one inference for CIFAR-10. Design scenarios for EvoApproxNAS are shown in color. The results of our initial study presented at EuroGP21 [17] are in grey (Color figure online)

during inference are shown in Fig. 5. Different scenarios clearly offer different trade-offs. It is worth mentioning scenario S2 (green), particularly because the CNNs it produces offer both the least energy consumption and the highest accuracy ($\sim$ 93%). CNNs provided in scenario S3 (orange) offer the best compromise between energy consumption and accuracy. CNNs from scenarios S1 and S4 then offer very different compromises. Figure 5 also contains parameters of CNNs obtained in our initial study [17] (gray points). As can be observed, almost all solutions generated by EvoApproxNAS dominate most of the solutions produced by the former implementation.

4.3 Comparison with other similar methods

The results are compared with human-created ResNet networks of similar complexity as evolved CNNs. The ResNet networks were further optimized using the ALWANN method [21]. ALWANN tries to identify the best possible assignment of an approximate multiplier to each layer of ResNet (i.e., different approximate multipliers can be assigned to different layers). Figure 6 shows that EvoApproxNAS produces CNNs, which dominate the ResNet CNNs (denoted by AxResNet-8, AxResNet-14 and AxResNet-50) optimized by the ALWANN method.

Figure 7 compares the accuracy and the size (the number of trainable parameters) of CNNs created by EvoApproxNAS and various NAS methods from the literature. We report the number of trainable parameters as this number is usually available, contrasted with the energy requirements. EvoApproxNAS offers solutions comparable to some of these models, especially when the number of parameters is below 1 million. It is also worth noting that most of the CNNs, which dominate our solutions, have not been optimized in terms of the energy consumption but rather in

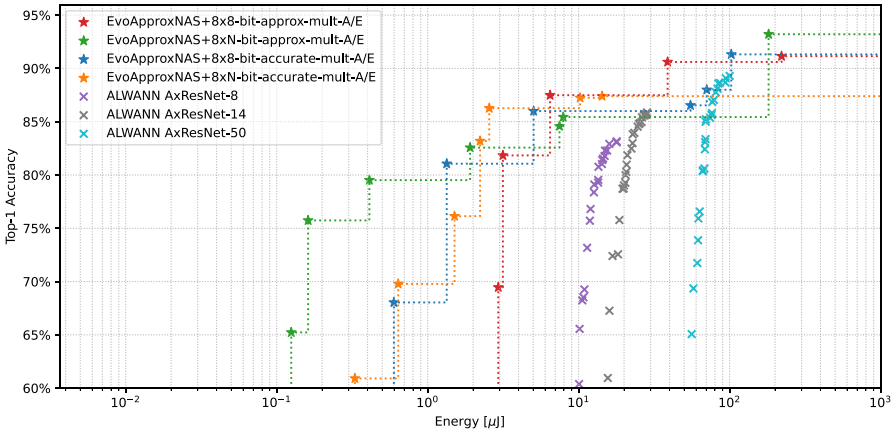


Fig. 6 Pareto fronts obtained by EvoApproxNAS for four proposed optimization scenarios compared with various ResNet networks optimized with the ALWANN method [21] (crosses) on the CIFAR-10 benchmark

terms of the number of parameters, and the NAS methods could generate more candidate designs than EvoApproxNAS .

Wistuba et al. [52] report 31 CNNs generated by various NAS methods and five CNNs designed by human experts. On CIFAR-10, the error is between 2.08% and 8.69%, the number of parameters is between 1.7 and 39.8 million, and the design time is between 0.3 and 22,400 GPU days. Our results are far from all these numbers as we address much smaller networks (operating with 8-bit multipliers) that must, in principle, show higher errors. Tann et al. [53] report several human-created CNN hardware accelerators with the classification accuracy 80.77–81.53% on CIFAR-10, and with the total energy consumption $34.2 - 335.7\mu$J (energy of multiplications is not reported separately). These numbers are quite comparable with our results even under the conservative assumption that multiplication requires only 20% of the total power of the accelerator. It has to be noted that they also used a different implementation technology.

Table 3 lists the key parameters of selected CNNs (the final test accuracy, the energy needed for all multiplications in convolutional layers, the number of parameters, and the number of MAC operations performed in convolutional layers), obtained in the individual scenarios for CIFAR-10. For comparison, the table also presents the most accurate CNNs produced by the ALWANN method (AxResNet-8, AxResNet-14, and AxResNet-50).

To demonstrate that the proposed method is applicable to other image classification tasks of similar complexity, we report in Table 4 the parameters of evolved CNNs for the SVHN benchmark. Our results are comparable to those published in the study [54], in which the parameters of the CNNs for the SVHN data set were in the range 0.27M–1.53M and the accuracy ranged from 95.40% to 96.34%.

Table 3 Parameters of two selected CNNs obtained with EvoApproxNAS in each scenario on the CIFAR-10 benchmark

Method		Cifar-10			
		Acc	Params	Energy	MAC
Name	Objs	[%]	[$\times 10^6$]	[μJ]	[$\times 10^6$]
8×8 Approx	A/E	91.14	2.07	220.9	564.7
		90.60	0.82	38.8	84.0
$8 \times n$ Approx	A/E	93.20	1.11	181.2	458.2
		85.43	0.93	7.9	35.3
8×8 Accurate	A/E	91.32	0.84	102.5	183.4
		87.99	0.25	70.5	126.2
$8 \times n$ Accurate	A/E	87.40	0.45	14.3	59.7
		87.23	0.25	10.2	37.4
8×8 Approx	A/E/P	87.55	0.35	11.3	26.9
		84.76	0.36	3.0	7.0
$8 \times n$ Approx	A/E/P	86.72	0.74	67.6	192.6
		85.47	0.31	7.1	44.1
8×8 Accurate	A/E/P	90.60	1.77	101.9	182.2
		91.54	1.33	111.7	199.8
$8 \times n$ Accurate	A/E/P	89.67	0.34	16.9	36.5
		86.61	0.11	2.4	9.2
AxResNet-8 [21]	A/E	83.16	0.07	17.8	21.1
AxResNet-14 [21]	A/E	85.87	0.17	28.3	35.3
AxResNet-50 [21]	A/E	89.30	0.75	100.1	120.2

Parameters of CNNs optimized by the ALWANN [21] are listed for the comparison. Symbols: accuracy (A), energy (E), and the number of parameters (P)

Table 4 Parameters of two selected CNNs obtained with EvoApproxNAS in each scenario on the SVHN benchmark

Method		SVHN			
		Acc	Params	Energy	MAC
Name	Objs	[%]	[$\times 10^6$]	[μJ]	[$\times 10^6$]
8×8 Approx	A/E	96.48	0.32	11.0	24.8
		96.12	0.45	6.3	19.6
$8 \times n$ Approx	A/E	95.72	1.17	22.4	93.1
		94.19	0.27	8.7	26.1
8×8 Accurate	A/E	96.91	0.90	138.2	247.3
		96.33	0.58	54.4	97.4
$8 \times n$ Accurate	A/E	95.75	0.26	3.1	7.4
		95.68	0.39	1.5	6.4

Symbols: accuracy (A), energy (E)

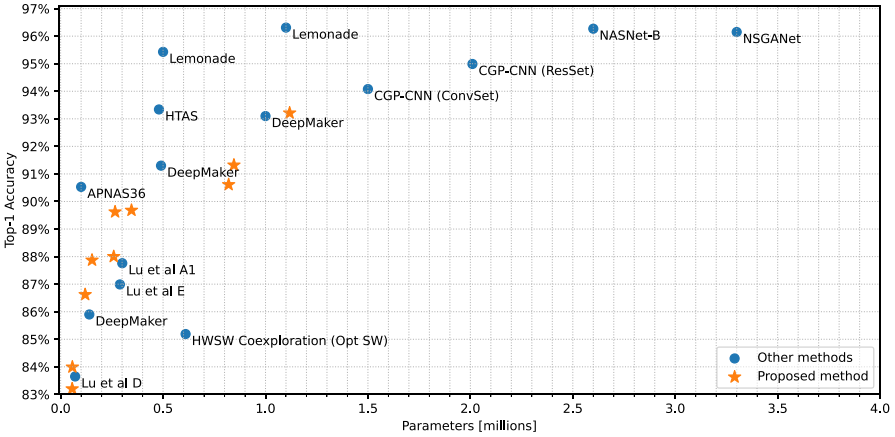


Fig. 7 The Top-1 accuracy on CIFAR-10 and the number of trainable parameters of CNNs generated by the proposed method (orange stars) and various state-of-the art NAS methods (blue points): Lu et al [37], HWSW Coexploration [46], DeepMaker [47], APNAS [48], HTAS [49], Lemonade [50], CGP-CNN [34], NASNet [51], and NSGANet (V1) [10] (Color figure online)

4.4 Discussion

By analyzing the evolved CNNs, we observed that they have diverse architectures, i.e., the method is not biased towards some set of structures. Examples of highly linear as well as wide architectures are shown in Fig. 8. EvoApproxNAS also effectively utilizes approximate multipliers with different bit widths. The most accurate CNN evolved by EvoApproxNAS was obtained when the 8 × N-bit approximate multipliers were enabled in the function set.

Compared to [17], the top-1 accuracy on CIFAR-10 was improved from 83.98 to 93.2% for the same number of evaluations per CGP run. This significant improvement is primarily attributed to a proper implementation of CGP and using data augmentation techniques. In order to statistically evaluate our search-based CNN design method and possibly find its setting leading to better results, more experiments have to be conducted. Obtaining this type of result will be one of our future tasks when more computational resources are available. A possible approach enabling to reduce the computational overhead is the use of neural architecture transfer, super nets, or surrogate models [55, 14].

5 Conclusions

We developed a multi-objective evolutionary design method capable of automated design of CNN topology and selection of approximate multipliers. This is a challenging problem not addressed in the literature. On the standard CIFAR-10 classification benchmark, the CNNs co-optimized with approximate multipliers show very good trade-offs between the classification accuracy and energy needed for multiplication in convolutional layers when compared with common

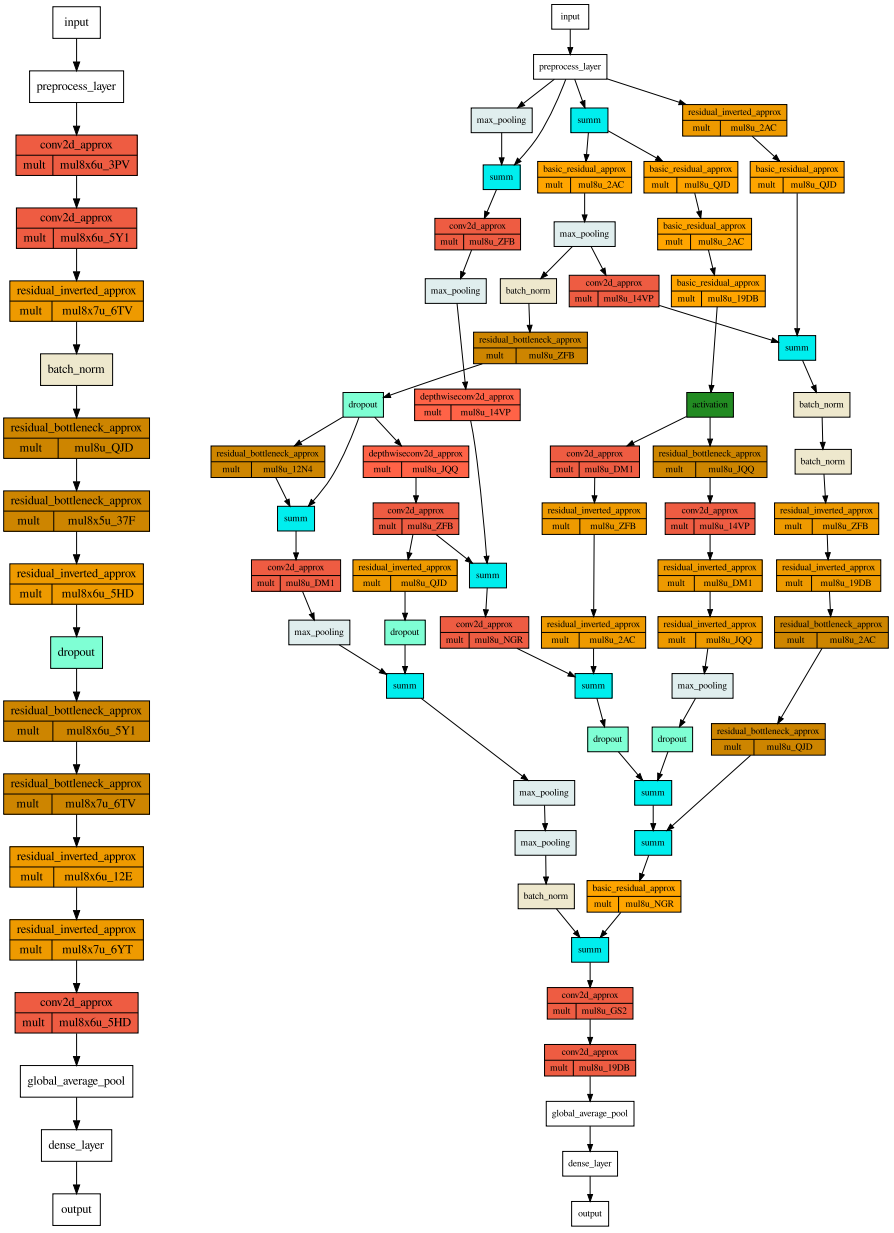


Fig. 8 Diagrams of two evolved CNNs for CIFAR-10. Left: The most accurate CNN obtained in scenario S2, with the accuracy of 93.2%, 1.11M parameters, and power consumption of 181.2 μJ in the multiplications in the convolutional layers. Right: The most accurate CNN obtained in scenario S1, with the accuracy of 91.14%, 2.07M parameters, and the power consumption of 220.9 μJ in the multiplications in the convolutional layers. Approximate multipliers 'mult' are denoted according to their codes in EvoApproxLib (Color figure online)

ResNet CNNs utilizing 8-bit multipliers and CNNs optimized with the ALWANN method. Despite very limited computational resources, resulting CNNs also exhibit comparable accuracy and the number of parameters with CNNs developed with other NAS methods. These results and additional experiments with the SVHN data set demonstrated that it makes sense to co-optimize CNN architecture with approximate arithmetic operations in a fully automated way.

Our future work will be devoted to extending the proposed method. We plan to consider other hardware parameters of candidate CNNs to optimize the overall energy and latency. Other approximation techniques such as memory subsystem optimization and dataflow organization will be integrated into the proposed method. More computational resources will be employed to show the effectiveness of EvoApproxNAS on more complex problems instances.

Acknowledgements This work was supported by the Czech Science Foundation Project 21-13001S. The computational experiments were supported by The Ministry of Education, Youth and Sports from the Large Infrastructures for Research, Experimental Development and Innovations project "e-Infrastructure CZ - LM2018140".

References

1. I. Goodfellow, Y. Bengio, A. Courville, *Deep Learning* (MIT Press, New York, 2016)
2. V. Sze, Y. Chen, T. Yang, J.S. Emer, Efficient processing of deep neural networks: a tutorial and survey. Proc. IEEE **105**(12), 2295–2329 (2017)
3. S. Mittal, A survey of techniques for approximate computing. ACM Comput. Surv. **48**(4), 1–33 (2016)
4. P. Panda, A. Sengupta, S.S. Sarwar, G. Srinivasan, S. Venkataramani, A. Raghunathan, K. Roy, Invited—cross-layer approximations for neuromorphic computing: From devices to circuits and systems. In: 53nd Design Automation Conference, pp. 1–6. IEEE (2016). https://doi.org/10.1145/2897937.2905009
5. S. Venkataramani et al., Efficient AI system design with cross-layer approximate computing. Proc. IEEE **108**(12), 2232–2250 (2020). https://doi.org/10.1109/JPROC.2020.3029453
6. T. Elsken, J.H. Metzen, F. Hutter, Neural architecture search: a survey. J. Mach. Learn. Res. **20**(55), 1–21 (2019)
7. P. Ren, Y. Xiao, X. Chang, P.Y. Huang, Z. Li, X. Chen, X. Wang, A comprehensive survey of neural architecture search: challenges and solutions. ACM Comput. Surv. **54**, 4 (2021). https://doi.org/10.1145/3447582
8. B. Zoph, Q.V. Le, Neural architecture search with reinforcement learning (2016). http://arxiv.org/abs/1611.01578
9. X. Yao, Evolving artificial neural networks. Proc. IEEE **87**(9), 1423–1447 (1999)
10. Z. Lu, I. Whalen, V. Boddeti, Y.D. Dhebar, K. Deb, E.D. Goodman, W. Banzhaf, NSGA-Net: neural architecture search using multi-objective genetic algorithm. In: Proceedings of the Genetic and Evolutionary Computation Conference, pp. 419–427. ACM (2019)
11. K.O. Stanley, J. Clune, J. Lehman, R. Miikkulainen, Designing neural networks through neuroevolution. Nat. Mach. Intell. **1**, 24–35 (2019)
12. H. Cai, C. Gan, T. Wang, Z. Zhang, S. Han, Once-for-all: Train one network and specialize it for efficient deployment. In: International Conference on Learning Representations (2020)
13. H. Cai, L. Zhu, S. Han, ProxylessNAS: Direct neural architecture search on target task and hardware. In: International Conference on Learning Representations (2019)
14. L. Sekanina, Neural architecture search and hardware accelerator co-search: a survey. IEEE Access **9**, 151337–151362 (2021). https://doi.org/10.1109/ACCESS.2021.3126685

15. V. Mrazek, L. Sekanina, Z. Vasicek, Libraries of approximate circuits: Automated design and application in CNN accelerators. IEEE J. Emerg. Sel. Topics Circuits Syst. **10**(4), 406–418 (2020). https://doi.org/10.1109/JETCAS.2020.3032495
16. S.S. Sarwar, S. Venkataramani, A. Ankit, A. Raghunathan, K. Roy, Energy-efficient neural computing with approximate multipliers. J. Emerg. Technol. Comput. Syst. **14**(2), 1–23 (2018)
17. M. Pinos, V. Mrazek, L. Sekanina, Evolutionary neural architecture search supporting approximate multipliers. In: Genetic Programming—24th European Conference, EuroGP 2021, LNCS, vol. 12691, pp. 82–97. Springer (2021). https://doi.org/10.1007/978-3-030-72812-0_6
18. V. Mrazek, R. Hrbacek, et al., Evoapprox8b: Library of approximate adders and multipliers for circuit design and benchmarking of approximation methods. In: Proceedings of DATE'17, pp. 258–261 (2017)
19. M. Abadi, A. Agarwal, et al., TensorFlow: Large-scale machine learning on heterogeneous systems (2015). https://www.tensorflow.org/. Software available from tensorflow.org
20. F. Vaverka, V. Mrazek, Z. Vasicek, L. Sekanina, TFApprox: Towards a Fast Emulation of DNN Approximate Hardware Accelerators on GPU. In: Design, Automation and Test in Europe, pp. 1–4 (2020)
21. V. Mrazek, Z. Vasicek, L. Sekanina, A.M. Hanif, M. Shafique, ALWANN: Automatic layer-wise approximation of deep neural network accelerators without retraining. In: Proceedings of the IEEE/ACM International Conference on Computer-Aided Design, pp. 1–8. IEEE (2019)
22. K. He, X. Zhang, S. Ren, J. Sun, Deep residual learning for image recognition. CoRR **abs/1512.03385** (2015). http://arxiv.org/abs/1512.03385
23. Intel Movidius vision processing units (VPUs) (2021). https://www.intel.com/content/www/us/en/products/details/processors/movidius-vpu.html
24. Y. Chen, T. Luo, S. Liu, S. Zhang, L. He, J. Wang, L. Li, T. Chen, Z. Xu, N. Sun, O. Temam, DaDianNao: a machine-learning supercomputer. In: 2014 47th Annual IEEE/ACM International Symposium on Microarchitecture, pp. 609–622 (2014). https://doi.org/10.1109/MICRO.2014.58
25. Y. Chen, T. Yang, J. Emer, V. Sze, Eyeriss v2: a flexible accelerator for emerging deep neural networks on mobile devices. IEEE J. Emerg. Select. Topics Circuits Syst. **9**(2), 292–308 (2019)
26. N.P. Jouppi, C. Young, N. Patil, D. Patterson, A domain-specific architecture for deep neural networks. Commun. ACM **61**(9), 50–59 (2018)
27. S. Mittal, A survey of FPGA-based accelerators for convolutional neural networks. Neural Comput. Appl. **32**(32), 1109–1139 (2020)
28. A. Garofalo, G. Tagliavini, F. Conti, D. Rossi, L. Benini, Xpulpnn: accelerating quantized neural networks on RISC-V processors through ISA extensions. In: 2020 Design, Automation Test in Europe Conference Exhibition (DATE), pp. 186–191 (2020). https://doi.org/10.23919/DATE48585.2020.9116529
29. E. Wang, J.J. Davis, R. Zhao, H.C. Ng, X. Niu, W. Luk, P.Y.K. Cheung, G.A. Constantinides, Deep neural network approximation for custom hardware: where we've been, where we're going. ACM Comput. Surv. **52**, 2 (2019). https://doi.org/10.1145/3309551
30. W. Jiang, L. Yang, S. Dasgupta, J. Hu, Y. Shi, Standing on the shoulders of giants: hardware and neural architecture co-search with hot start (2020). https://arxiv.org/abs/2007.09087
31. P. Gysel, J. Pimentel, M. Motamedi, S. Ghiasi, Ristretto: a framework for empirical study of resource-efficient inference in convolutional neural networks. IEEE Trans. Neural Netw. Learn. Syst. **29**(11), 5784–5789 (2018)
32. K.O. Stanley, R. Miikkulainen, Evolving neural networks through augmenting topologies. Evol. Comput. **10**(2), 99–127 (2002)
33. E. Real, S. Moore, A. Selle, S. Saxena, Y.L. Suematsu, J. Tan, Q. Le, A. Kurakin, Large-Scale Evolution of Image Classifiers. arXiv e-prints arXiv:1703.01041 (2017)
34. M. Suganuma, S. Shirakawa, T. Nagao, A genetic programming approach to designing convolutional neural network architectures. In: Proceedings of the Genetic and Evolutionary Computation Conference, GECCO '17, pp. 497–504. ACM (2017)
35. Z. Lu, K. Deb, E.Goodman, W. Banzhaf, V.N. Boddeti, NSGANetV2: Evolutionary multi-objective surrogate-assisted neural architecture search. In: Computer Vision—ECCV 2020, pp. 35–51. Springer, Cham (2020)
36. K. Deb, A. Pratap, S. Agarwal, T. Meyarivan, A fast and elitist multiobjective genetic algorithm: NSGA-II. IEEE Trans. Evol. Comput. **6**(2), 182–197 (2002)
37. Q. Lu, W. Jiang, X. Xu, Y. Shi, J. Hu, On neural architecture search for resource-constrained hardware platforms (2019). http://arxiv.org/abs/1911.00105

38. Y. Lin, M. Yang, S. Han, NAAS: Neural accelerator architecture search. In: 2021 58th ACM/ESDA/IEEE Design Automation Conference (DAC) (2021)
39. J.F. Miller, *Cartesian Genetic Programming* (Springer, Berlin, 2011)
40. K. He, X. Zhang, S. Ren, J. Sun, Identity mappings in deep residual networks. In: Computer Vision—ECCV 2016, pp. 630–645. Springer (2016)
41. T. Devries, G.W. Taylor, Improved regularization of convolutional neural networks with cutout. CoRR **abs/1708.04552** (2017). http://arxiv.org/abs/1708.04552
42. C. Shorten, T. Khoshgoftaar, A survey on image data augmentation for deep learning. J. Big Data **6**, 1–48 (2019)
43. , A. Krizhevsky, V. Nair, G. Hinton, CIFAR-10 (Canadian Institute for Advanced Research) http://www.cs.toronto.edu/~kriz/cifar.html
44. Y. Netzer, T. Wang, A. Coates, A. Bissacco, B. Wu, A.Y. Ng, Reading digits in natural images with unsupervised feature learning. In: NIPS Workshop on Deep Learning and Unsupervised Feature Learning 2011 (2011). http://ufldl.stanford.edu/housenumbers/nips2011_housenumbers.pdf
45. M. Fleischer, The measure of pareto optima applications to multi-objective metaheuristics, in *Evolutionary Multi-criterion Optimization*. (Springer, Berlin, 2003), pp. 519–533
46. W. Jiang, L. Yang, E.H.M. Sha, Q. Zhuge, S. Gu, S. Dasgupta, Y. Shi, J. Hu, Hardware/software co-exploration of neural architectures. IEEE Trans. Comput.-Aid. Des. Integr. Circuits Syst. **39**(12), 4805–4815 (2020). https://doi.org/10.1109/TCAD.2020.2986127
47. M. Loni, S. Sinaei, A. Zoljodi, M. Daneshtalab, M. Sjödin, DeepMaker: a multi-objective optimization framework for deep neural networks in embedded systems. Microprocess. Microsyst. **73**, 102989 (2020). https://doi.org/10.1016/j.micpro.2020.102989
48. P. Achararit, M.A. Hanif, R.V.W. Putra, M. Shafique, Y. Hara-Azumi, APNAS: accuracy-and-performance-aware neural architecture search for neural hardware accelerators. IEEE Access **8**, 165319–165334 (2020). https://doi.org/10.1109/ACCESS.2020.3022327
49. Y. Jiang, X. Wang, W. Zhu, Hardware-aware transformable architecture search with efficient search space. In: 2020 IEEE International Conference on Multimedia and Expo (ICME), pp. 1–6 (2020). https://doi.org/10.1109/ICME46284.2020.9102721
50. T. Elsken, J.H. Metzen, F. Hutter, Efficient multi-objective neural architecture search via Lamarckian evolution. In: 7th International Conference on Learning Representations, ICLR 2019. OpenReview.net (2019)
51. B. Zoph, V. Vasudevan, J. Shlens, Q.V. Le, Learning transferable architectures for scalable image recognition. In: 2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition, pp. 8697–8710 (2018). https://doi.org/10.1109/CVPR.2018.00907
52. M. Wistuba, A. Rawat, T. Pedapati, A survey on neural architecture search. CoRR **abs/1905.01392** (2019). http://arxiv.org/abs/1905.01392
53. H. Tann, S. Hashemi, S. Reda, *Lightweight Deep Neural Network Accelerators Using Approximate SW/HW Techniques* (Springer, Berlin, 2019), pp. 289–305
54. H. Lee, E. Hyung, S.J. Hwang, Rapid neural architecture search by learning to generate graphs from datasets. In: International Conference on Learning Representations (2021). https://openreview.net/forum?id=rkQuFUmUOg3
55. Z. Lu, G. Sreekumar, E. Goodman, W. Banzhaf, K. Deb, V.N. Boddeti, Neural architecture transfer. IEEE Trans. Pattern Anal. Mach. Intell. **43**(9), 2971–2989 (2021). https://doi.org/10.1109/TPAMI.2021.3052758