Neural Computing and Applications
https://doi.org/10.1007/s00521-020-04969-6

ORIGINAL ARTICLE

Automated design of error-resilient and hardware-efficient deep neural networks

Christoph Schorn[1,2] · Thomas Elsken[3,4] · Sebastian Vogel[1,2] · Armin Runge[1] · Andre Guntoro[1] · Gerd Ascheid[2]

Received: 17 June 2019 / Accepted: 25 April 2020

Abstract
Applying deep neural networks (DNNs) in mobile and safety-critical systems, such as autonomous vehicles, demands a reliable and efficient execution on hardware. The design of the neural architecture has a large influence on the achievable efficiency and bit error resilience of the network on hardware. Since there are numerous design choices for the architecture of DNNs, with partially opposing effects on the preferred characteristics (such as small error rates at low latency), multi-objective optimization strategies are necessary. In this paper, we develop an evolutionary optimization technique for the automated design of hardware-optimized DNN architectures. For this purpose, we derive a set of inexpensively computable objective functions, which enable the fast evaluation of DNN architectures with respect to their hardware efficiency and error resilience. We observe a strong correlation between predicted error resilience and actual measurements obtained from fault injection simulations. Furthermore, we analyze two different quantization schemes for efficient DNN computation and find one providing a significantly higher error resilience compared to the other. Finally, a comparison of the architectures provided by our algorithm with the popular MobileNetV2 and NASNet-A models reveals an up to seven times improved bit error resilience of our models. We are the first to combine error resilience, efficiency, and performance optimization in a neural architecture search framework.

Keywords Neural network hardware · Error resilience · Hardware faults · Neural architecture search · Multi-objective optimization · AutoML

1 Introduction

The application of deep neural networks (DNNs) in safety-critical perception systems, for example autonomous vehicles (AVs), poses some challenges on the design of the underlying hardware platforms. On the one hand, efficient and fast accelerators are needed, since DNNs for computer vision exhibit massive computational requirements [56]. On the other hand, resilience against random hardware faults has to be ensured. In many driving scenarios, entering a fail-safe state is not sufficient, but fail-operational behavior and fault tolerance are required [48]. However, fault tolerance techniques at the hardware level often entail large redundancy overheads in silicon area, latency, and power consumption. These overheads stand in contrast to the low-power and low-latency requirements of embedded real-time DNN accelerators. Reliability concerns in nanoscale integrated circuits, for instance soft errors in memory and logic, represent an additional challenge for the realization of fault tolerance mechanisms at the hardware level [2, 33, 36, 70, 86]. Moreover, techniques such as near-threshold computing [26] and approximate computing [66] are desirable to meet power constraints, but can further increase error rates.

✉ Christoph Schorn
Christoph.Schorn@de.bosch.com

1 Bosch Corporate Research, Robert Bosch GmbH, Renningen, Germany

2 Institute for Communication Technologies and Embedded Systems, RWTH Aachen University, Aachen, Germany

3 Bosch Center for Artificial Intelligence, Robert Bosch GmbH, Renningen, Germany

4 Department of Computer Science, University of Freiburg, Freiburg im Breisgau, Germany

Published online: 10 June 2020

To overcome these challenges, one option is to exploit error resilience at the algorithm level and allow for a certain degree of inaccuracy at the hardware level. This is referred to as cross-layer resilience [13]. Due to the implicit information redundancy of neural networks, they offer some robustness against random internal perturbations, which can be exploited in a cross-layer resilience approach. Nevertheless, error resilience is strongly influenced by the architectural design of the DNN [85] as well as its internal data representations [53]. These design choices, in turn, also influence hardware efficiency and classification performance of the network. Taking these multiple, partially opposing objectives into account in a manual DNN design procedure is non-trivial and cumbersome.

Our approach to address this challenge in this paper is depicted in Fig. 1. We develop and evaluate an efficient, automated, multi-objective neural architecture search (NAS) technique which holistically takes classification performance as well as hardware-specific objective functions into account. The architectures obtained by our algorithm and some reference methods from the literature are then retrained and quantized using two different quantization strategies. Finally, the error resilience of different architectures as well as quantization methods is compared against each other by performing fault injection experiments.

In detail, our contributions are the following:

1. We derive a set of objective functions for the prediction of error resilience, energy consumption, latency, and required bandwidth of DNNs on hardware, solely based on the topology of their neural architecture, allowing a fast evaluation of these objectives by avoiding the need for expensive simulations or training of the neural network.
2. We integrate these functions in an efficient, evolutionary, multi-objective NAS algorithm that uses (approximate) network morphisms for a fast Pareto-optimization of DNNs.
3. We add a neural network quantization step at the end of NAS in order to obtain models suitable for an efficient inference accelerator. We compare two recently introduced quantization techniques with respect to resulting classification performance and error resilience of the neural networks.
4. We evaluate our methods on two popular image classification benchmarks, namely CIFAR-10 and German Traffic Sign Recognition Benchmark (GTSRB). In particular, we test the predictive performance of our error resilience prediction metric by measuring the correlation to silent data corruption (SDC) rates, employing a bit-flip fault injection framework.
5. We benchmark the solutions obtained by our NAS algorithm against two reference architectures, namely MobileNetV2 [80] and NASNet-A [112], by performing fault injection experiments.

To the best of our knowledge, this is the first paper combining error resilience and hardware efficiency optimization in the context of neural architecture search.

The remainder of this paper is structured as follows: In Sect. 2, we give an overview of related work. In Sect. 3, we introduce our methodology. This includes the derivation of hardware-specific objective functions, neural network quantization techniques, and the multi-objective optimization algorithm used in this paper. In Sect. 4, we evaluate the outcome of our methods on two image classification benchmarks. We analyze the trade-offs between Pareto-optimal solutions, perform fault injections to compare predicted and measured resilience, and evaluate the characteristics of two different DNN quantization methods. Furthermore, we perform a comparison with two other architectures from the recent literature. We close our paper with a summary and conclusions in Sect. 5.

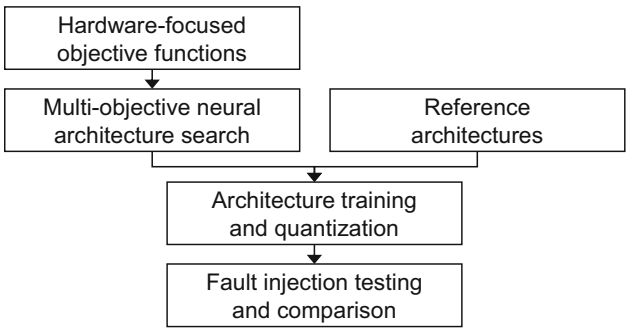


Fig. 1 Overview of our research methodology

2 Background and related work

We now give an overview of related error resilience analysis (Sect. 2.1), resilience optimization techniques for neural networks (Sect. 2.2) as well as preliminaries on multi-objective optimization (Sect. 2.3) and neural architecture search (NAS) (Sect. 2.4).

2.1 Neural network resilience analysis

Understanding a neural network's resilience against erroneous perturbations in its internal computations has been a topic of interest for decades already. Here, we give an overview of the most recent studies that target error

resilience analysis of modern DNNs. An in-depth review of previous literature has been recently given by Mittal [67].

2.1.1 Experimental analysis

The majority of the studies on error resilience in neural networks have been experimental. They range from physical fault induction experiments in real hardware devices [81, 100], over fault injections in (virtual) hardware models [3, 53, 78, 81], to error simulations at the algorithmic behavior level [63, 74, 83]. Behavioral analysis can be connected to realistic hardware faults in a second step, by mapping the effect of these faults to error models in the algorithm domain [72]. For the model-based analysis, stuck-at-zero, stuck-at-one, and random bit-flips of memory cells are commonly used. Stuck-at types are used to model permanent faults (e.g., resulting from manufacturing defects) and bit-flips are typically used to model radiation-induced transient faults that lead to soft errors [94].

In summary, experimental studies found different determinants of neural network resilience, the most important being the number and type of errors, the data representation of the neural network, the DNN type, and the location where the error occurs. However, while experimental evaluation is useful for an accurate a posteriori resilience determination of a given DNN on hardware, it is cumbersome and provides only limited insight into a priori design choices for DNN developers to improve resilience at the algorithm level.

2.1.2 Theoretical analysis

A theory-guided resilience analysis offers the advantage of being more directly interpretable and avoids lengthy fault injection experiments. El Mhamdi and Guerraoui [28] analytically derived easily computable bounds for the forward error propagation of neurons that are stuck-at-zero (crashed neurons) and for neurons that transmit arbitrary values (Byzantine neurons). They found that the choice of activation function and the number of neurons per layer are design choices that affect the forward error propagation. More precisely, an activation function with a low Lipschitz constant as well as a high number of neurons per layer can reduce forward error propagation.

A different analytical technique to derive neuron resilience prediction has been used in the context of approximate neural network computing. Backpropagation of error gradients, comparable to the technique used to determine weight updates during neural network training, has been used to estimate the average output sensitivity to perturbations in individual neurons [97, 109].

Recently, Schorn et al. [83] showed that a technique based on layerwise relevance propagation (LRP) [4] outperforms gradient-based resilience prediction. Contrarily to gradient methods, which determine the sensitivity to small perturbations in neurons, LRP attributes to each neuron its absolute contribution to the DNN output [69], which can be interpreted as layerwise Taylor decomposition [68]. A high neuron relevance, averaged over a training set of input samples, corresponds to a high sensitivity against errors [83].

2.2 Neural network resilience optimization

The optimization of neural network error resilience at the algorithm level is an active field of research. A number of publications simulate the effects of hardware faults during neural network training to improve resilience [22, 45, 57, 104, 106]. Reference [22] considers timing variations, [45, 106] static random-access memory (RAM) supply voltage scaling, and [57, 104] hard defects in memristors and resistive RAM, respectively. The drawback of these approaches is that they complicate the training process, since fault injections have to be performed by placing hardware in the training loop or through realistic fault simulations. Common regularizing techniques, such as dropout [44, 88] and weight decay [50], also improve the general error resilience of neurons [28].

A second approach is to adjust the mapping of the algorithm to hardware for an optimized resilience. A significance-driven mapping of network weight bits to memory cells with different resilience has been suggested in [87]. However, the authors did not follow an analytical approach to determine weight resiliencies, but relied on their experience. In contrast, the LRP-based method in [83] gives a theoretically founded resilience mapping of neurons.

A third approach is to use modifications in hardware that are tailored to exploit the algorithmic resilience properties of neural networks. This can be zero-biased [3] or selectively hardened [53] memory cells, optimized data representations [100], masking techniques [73, 78], anomaly detectors [53, 84], and relaxed versions of classical fault tolerance mechanisms, such as triple modular redundancy (TMR) [62] and (ABFT) checksums [81].

Modifications of the neural architecture to increase resilience have been proposed as well. Dias et al. [24] suggest a resilience optimization procedure by replication of critical neurons and weights. However, they use exhaustive simulation to determine criticality values, which is infeasible for large-scale DNNs. Schorn et al. [85] showed that critical layers can be identified using LRP. Nevertheless, no automated neural architecture design technique that jointly optimizes error resilience as well as other desirable performance and efficiency objectives of DNNs has been introduced so far.

2.3 Multi-objective optimization

In multi-objective optimization (see, e.g., [64]), one tries to optimize K complementary objective functions $f_1, \ldots, f_K$ over a space $\mathcal{N}$ of feasible solutions (in our case: a space of neural network architectures). Usually, there will be no $N^* \in \mathcal{N}$ that minimizes all objectives $f_1, \ldots, f_K$ at the same time (as the objectives are complementary). Instead, there are multiple *Pareto-optimal* solutions meaning that one cannot reduce any f_i without increasing at least one f_j, with $i,j \in \{1, \ldots, K\}$ and $i \neq j$. Formally, we say that architecture N_1 dominates architecture N_2 iff $f_i(N_1) \leq f_i(N_2)$ for all $i \in \{1, \ldots, K\}$ and $f_j(N_1) < f_j(N_2)$ for at least one $j \in \{1, \ldots, K\}$. N^* is called Pareto-optimal iff N^* is not dominated by any other $N \in \mathcal{N}$. The set of Pareto-optimal solutions is the so-called *Pareto-front*. Typically, multi-objective optimization can only determine a subset $\mathcal{P}$ that approximates this Pareto-front.

In order to rate the overall performance of a given neural network $N \in \mathcal{P}$ across all objectives, the distance to the *ideal point* in the Euclidean space $\mathbb{R}^K$ corresponding to the K objectives $f_1, \ldots, f_K$ can be used as a metric [8]. The (approximate) coordinates $(y_1, \ldots, y_K)$ of the ideal point in this space are determined by the minima of the objective functions $f_i(N)$ over the (approximated) Pareto-front $\mathcal{P}$ [27]. Consequently, each coordinate is given by Eq. (1).

$$y_i = \min_{N \in \mathcal{P}} f_i(N), \quad i \in \{1, \ldots, K\} \tag{1}$$

To enhance comparability, a normalized version of the distance to the ideal point can be computed [8]. Therefore, normalized objective functions $\bar{f}_i(N)$ are determined for all $i \in \{1, \ldots, K\}$ using Eq. (2) so that $0 \leq \bar{f}_i(N) \leq 1$.

$$\bar{f}_i(N) = \frac{f_i(N) - \min_{N \in \mathcal{P}} f_i(N)}{\max_{N \in \mathcal{P}} f_i(N) - \min_{N \in \mathcal{P}} f_i(N)} \tag{2}$$

Then, a norm on the vector $\bar{\mathbf{f}}(N) = (\bar{f}_1(N), \ldots, \bar{f}_K(N))$ is computed to measure the distance of N from the ideal point.

Blasco et al. [8] suggest to take the infinity norm for the purpose of trade-off analysis, as defined in Eq. (3).

$$\left\| \bar{\mathbf{f}}(N) \right\|_\infty = \max\left\{ \bar{f}_i(N) \right\}, \quad i \in \{1, \ldots, K\} \tag{3}$$

That way, a score between 0 and 1 is obtained, which supplies information about the worst objective value. For example, a value of 1 means that N has the worst observed performance in at least one of the objectives. We refer to $\left\| \bar{\mathbf{f}}(N) \right\|_\infty$ as *normalized worst objective value* in the remainder of this paper.

2.4 Neural architecture search

One crucial aspect for the success of deep learning in recent years was the design of novel neural network architectures [35, 40, 80, 92]. However, manually designing such architectures is a cumbersome trial-and-error process. To overcome the need for architectural engineering, neural architecture search (NAS)—the process of automatically designing neural network architectures—has arisen as a subfield of automated machine learning [41]. By now, architectures found by NAS have outperformed human-designed architectures on a variety of tasks such as image recognition [76], object detection [112], or dense prediction tasks [17, 77].

We briefly summarize related work here and refer to the survey by Elsken et al. [31] for a more thorough literature overview. Reinforcement learning techniques [5, 110–112] or evolutionary methods [65, 75, 76, 90] are commonly employed to search for well-performing architectures. Another biologically inspired algorithm that has been applied in the context of neural architecture optimization is particle swarm optimization [79, 95]. Furthermore, it has been proposed to select relevant features of neural networks in an unsupervised learning approach via local structure learning [54].

As early work on NAS required vast amount of computational resources, often in the range of hundreds or even thousands of GPU days [76, 111, 112], making NAS more efficient was the focus of many researchers, e.g., by employing network morphisms [9, 10, 29], by sharing weights [7, 71, 82] or by performance prediction [6, 47]. A recent series of work [11, 59, 105] employed a real-valued relaxation of the discrete architecture search space, enabling gradient-based optimization.

While the previously discussed approaches solely optimize for a single objective, namely minimizing some error rate, there has also been some work on *multi-objective* neural architecture search [16, 25, 30, 39, 46, 61, 93, 103], optimizing other objectives such as network size, latency or energy consumption concurrently. [25] extend [58] by considering multiple objectives during the model selection step. [61] employ NSGA-II [21], a well-known multi-objective optimization algorithm, in the context of NAS. Instead of actually solving the multi-objective problem, many researchers use scalarization methods, such as the weighted product or sum method [20], to obtain a single objective. This is then optimized via, e.g., reinforcement learning [39, 93] or differentiable NAS [103]. [12] use multi-objective Bayesian optimization to search for convolutional cells [112]. In this work, we will build up on the multi-objective evolutionary method `LEMONADE` [30] that exploits cheap-to-evaluate objectives to make the search

more efficient. This perfectly fits our application as we will see later as our objectives are solely based on the neural network architecture (and not, e.g., on expensive simulations or trained neural network weights) and thus cheap to compute. We discuss LEMONADE more detailed in Sect. 3.2.1.

3 Hardware-focused neural architecture design

In this section, we introduce our framework for the automated design of error-resilient and hardware-efficient DNN architectures. In a first step, we identify optimization goals that typically appear in embedded DNN hardware applications and derive corresponding objective functions (Sect. 3.1). In the further course of this paper, these functions serve as input to a multi-objective neural architecture search algorithm (Sect. 3.2). Fixed-point quantization is applied as post-processing step after NAS to enable efficient DNN execution on dedicated hardware accelerators (Sect. 3.3).

3.1 Hardware-specific objectives

We consider four different objectives that are commonly desirable in embedded DNN hardware applications, namely high error resilience, low latency, high energy efficiency, and a low bandwidth requirement.

3.1.1 Error resilience

In the context of this paper, error resilience is regarded as robustness of the neural network classifier against perturbations in its neuron activation values. Such perturbations can be the result of random hardware faults, such as radiation-induced bit-flips. We measure the degree of perturbation using bit error rate (BER), i.e., the fraction of flipped bits across all activations of the DNN. We define *architecture sensitivity* at a given BER as probability for the predicted class output to differ, with and without bit errors. In order to maximize error resilience, we want to minimize architecture sensitivity.

Following the approach in [83, 85], we derive an architecture-dependent error sensitivity metric using LRP. A key prerequisite in the mathematical framework of LRP is the relevance conservation principle [69]. It ensures that the total amount of neuron relevance, which is propagated backwards through the network after the forward pass of inference on an input sample, is conserved in each layer. Consequently, for a group of neurons k and its inputs j

$$\sum_j r_j = \sum_j \sum_k r_{j \leftarrow k} = \sum_k \sum_j r_{j \leftarrow k} = \sum_k r_k. \tag{4}$$

In Eq. (4), r_j and r_k are the relevance values attributed to neurons j and k, respectively, and $r_{j \leftarrow k}$ is the amount of relevance propagated backwards from neuron k to neuron j. The conservation principle is motivated by the fact that an output activation of neuron k can be completely decomposed into contributions of its input neurons j.

The relevance distribution among the neurons in each layer depends on their activations and the synaptic weights [69]. For the initial backpropagation step, the final output neuron relevance of the DNN is predetermined by the one-hot encoded target vector belonging to each input sample. This ensures that $\sum_k r_k = 1$ in each layer. Consequently, for a uniformly randomly drawn neuron in a layer l, the expected relevance is

$$\mathrm{E}\left[r_k^{(l)}\right] = \frac{1}{n_{\text{outputs}}^{(l)}}, \quad k \sim \text{unif}\{0, n_{\text{outputs}}^{(l)} - 1\}. \tag{5}$$

In Eq. (5), $n_{\text{outputs}}^{(l)}$ is the total number of neurons in that layer. The observation that a higher average relevance corresponds to a more likely change of the DNN classification output suggests that layers with few neurons are more sensitive to errors [83, 85].

Effect of max-pooling Max-pooling is commonly used in some layers of a DNN, in order to reduce the output dimensions of that layer [51]. A max-pooling stage divides the outputs of a layer into subsets and selects the maximum output value out of each subset. We do not regard max-pooling as a separate layer, but consider it as attachment to a layer. If a layer l has max-pooling, the reduced number of output values after the pooling stage is taken to calculate $n_{\text{outputs}}^{(l)}$.

Additionally, we observed an increased error sensitivity of neurons in layer l if max-pooling is present in the subsequent layer $l+1$. We suppose that this is because information about the input sample is reduced by the pooling stage, but critical errors, which are mostly changes from a low to a high activation value [53], are likely to propagate through. Thus, we obtain an effective error sensitivity of neurons in layer l by multiplication with the pooling factor of layer $l+1$. The pooling factor is the fraction of input to output dimension of the pooling stage and equals 4 for the max-pooling layers that we use throughout our experiments.

Effect of merge layers Skip connections, i.e., concurrent paths through the network, can improve the training of deep architectures and thus have become popular in state-of-the-art DNNs [34]. At some point in the network, the parallel paths have to be merged again, which can be done

by componentwise addition [35] or by feature concatenation [92]. While a concatenation does not affect error propagation, an add layer increases error sensitivity of the DNN. There are two reasons for this. Firstly, an add layer involves additional (error prone) load, accumulates, and stores operations, while concatenation only involves the change of the address range from which data are loaded in the subsequent layers.

Secondly, the fraction of neurons affected by errors is likely to increase through the add operation. If two inputs with an equal and small fraction of erroneous neurons are added, the resulting fraction of erroneous neurons doubles as long as the error locations in the inputs do not coincide. This can be regarded as doubling the effective error sensitivity of the neurons preceding the add operation.

Architecture sensitivity index The aforementioned insights are now used to define a metric that estimates the error sensitivity of a neural network N solely based on the topology of its architecture. We call this metric architecture sensitivity index (ASI). It is defined as sum of the expected error sensitivities over all layers L_N of N,

$$f_{\mathrm{ASI}}(N) = \sum_{l \in L_N} \frac{\lambda^{(l)} \zeta^{(l)}}{n^{(l)}_{\mathrm{outputs}}}. \tag{6}$$

In Eq. (6), $\lambda^{(l)}$ is the max-pooling factor of the succeeding layer $l+1$ (i.e., 1 for no pooling) and $\zeta^{(l)}$ is 2, if l is connected to an add layer, else 1.

Concatenations are not counted as extra layers in this sum, while add layers are. Furthermore, supportive functionalities, such as activation function, pooling and batch normalization [42], are not considered as separate layers, but included in the neuron layers.

We want to emphasize that Eq. (6) can be computed very easily, since it only depends on the network topology and does not require any training or other expensive computations.

3.1.2 Latency

Aside from error resilience, real-time inference with low latency is an additional necessity in many applications. AVs, for instance, should be able to derive driving actions from sensory input in less than 100 ms, in order to surpass human-level perception performance and provide a sufficient level of safety [56]. While low latency can be achieved by employing a parallelized hardware architecture and a high operating frequency, the performance of a DNN accelerator is constrained by manufacturing, power consumption, reliability, and flexibility requirements. Thus, a reduction in computational complexity at the algorithm level is desirable.

The roofline model [102] is commonly used to describe the attainable computational performance of a DNN accelerator [107]. It defines two operational domains, which are entered depending on the computational workload of the accelerator. In the memory-bound domain, latency is determined by the amount of data transfer to memory and the available memory bandwidth. In the compute-bound domain, latency can be regarded as being proportional to the number of operations required by the algorithm.

Being compute bound is preferable over memory-bound operation, since it allows maximum utilization of the available computational resources and highest throughput. Thus, we assume an accelerator, whose memory bandwidth is sufficiently large so that it will predominantly operate in the compute-bound domain for the workloads considered in this paper. We can therefore take the number of operations of the DNN as approximate determinant of latency. Furthermore, we regard the number of operations as being solely dependent on the neural network architecture, i.e., we do not consider any data-dependent operation reductions.

Our objective function for latency reduction is given by

$$f_{\mathrm{latency}}(N) = \sum_{l \in L_N} n^{(l)}_{\mathrm{op}}. \tag{7}$$

In Eq. (7), $n^{(l)}_{\mathrm{op}}$ counts the number of operations of layer l.

3.1.3 Energy efficiency

A further frequent demand on embedded DNN accelerators is a low energy consumption per classification inference. This can have mainly two reasons. Firstly, mobile devices have a limited amount of energy storage capacity, and thus, energy-efficient DNN accelerators are required, for example to extend the battery life and range of AVs. Secondly, embedded devices often have a strict size limitation, which makes it difficult to realize the necessary heat dissipation. As the thermal leakage power of an accelerator directly depends on the number of classifications per second and the energy per classification, energy efficiency is desirable to enable high classification throughput.

Energy consumption of DNN accelerators is dominated by data transfers to and from memory [91]. This is due to the large amount of parameters and intermediate data outputs of typical large-scale DNNs.

According to Horowitz [38], energy consumption for off-chip dynamic RAM access is about two orders of magnitude higher than for internal cache accesses and arithmetic operations. While some hardware designers increase energy efficiency by integrating huge on-chip static RAMs in their DNN accelerator (e.g., [101]), this approach is not feasible in every case. In this paper, we

assume an accelerator with small on-chip buffer (such as [15]), so that a layerwise data transfer to and from off-chip memory is necessary, which dominates energy consumption.

Consequently, to maximize energy efficiency, our objective is to minimize data transfer to and from memory per inference. We neglect the number of operations in this calculation because of its limited influence on energy consumption and since it is already part of the latency minimization objective function. To determine the data transfer of a layer, we assume that each input and weight parameter of the layer is loaded once from external memory and each output is written back once. Furthermore, we assume that the same bit-width is used to represent all activations and parameters of the network.

Our objective function for minimizing energy consumption is thus given by the sum of layerwise input, output, and parameter data word transfers over the whole network,

$$f_{\text{energy}}(N) = \sum_{l \in L_N} (n^{(l)}_{\text{inputs}} + n^{(l)}_{\text{outputs}} + n^{(l)}_{\text{params}}). \tag{8}$$

In Eq. (8), $n^{(l)}_{\text{inputs}}$ and $n^{(l)}_{\text{outputs}}$ count the number of input neurons and output neurons, respectively, and $n^{(l)}_{\text{params}}$ counts the number of parameters of layer l.

3.1.4 Bandwidth requirement

As described in Sect. 3.1.2, we assume the accelerator for which we optimize DNN architectures to operate predominantly in the compute-bound domain of the roofline model. In order to guarantee compute-bound operation, the accelerator has to provide a certain maximum bandwidth to memory. It is desirable to keep this bandwidth requirement within bounds to simplify the accelerator architecture.

The required memory bandwidth can vary for different layers of a DNN. We employ the ratio between data transfers and operations of a layer as estimator for its bandwidth requirement. The intuition behind this is that a low number of operations are related to a short processing time of the layer, and consequently, a high bandwidth is required to be able to perform the necessary data movements in that given time.

We define an overall objective function to optimize neural architectures for a low bandwidth requirement by adding up the data–computation ratios of all layers. Thus, our objective function for minimizing the bandwidth requirement is given by the accumulated data–computation ratio (ADCR), as described in Eq. (9).

$$f_{\text{ADCR}}(N) = \sum_{l \in L_N} \frac{n^{(l)}_{\text{inputs}} + n^{(l)}_{\text{outputs}} + n^{(l)}_{\text{params}}}{n^{(l)}_{\text{op}}} \tag{9}$$

3.2 Multi-objective NAS

In the following, we introduce LEMONADE, a **L**amarckian **E**volutionary algorithm for **M**ulti-**O**bjective **N**eural **A**rchitecture **DE**sign [30], that we will use in our later experiments to automatically design well-performing, error-resilient, and hardware-efficient architectures.

3.2.1 LEMONADE

We continue to use the notation introduced in Sect. 2.3. LEMONADE maintains a population $\mathcal{P}$ of neural networks $N \in \mathcal{N}$, where $\mathcal{N}$ denotes a suitable space of network architectures that are defined in Sect. 3.2.2. This population is improved over the course of the algorithm with respect to the multi-objective optimization problem $\min_{N \in \mathcal{N}} \mathbf{f}(N)$ with objective function

$$\mathbf{f}(N) = (\mathbf{f}_{\text{exp}}(N), \mathbf{f}_{\text{cheap}}(N)) \in \mathbb{R}^K = \mathbb{R}^U \times \mathbb{R}^V. \tag{10}$$

The objective function in Eq. (10) is composed of a vector of expensive-to-evaluate objectives $\mathbf{f}_{\text{exp}}(N) \in \mathbb{R}^U$ (in our case: the validation error, only obtainable by expensive training) and a vector of cheap-to-evaluate objectives $\mathbf{f}_{\text{cheap}}(N) \in \mathbb{R}^V$ (in our case: the objectives defined in Sect. 3.1). The population $\mathcal{P}$ is chosen to comprise all non-dominated networks with respect to $\mathbf{f}$, i.e., the population approximates the Pareto-front. LEMONADE exploits that $\mathbf{f}_{\text{cheap}}$ is cheap to evaluate in order to bias the sampling of children toward areas of the Pareto-front of $\mathbf{f}_{\text{cheap}}$ that are sparsely populated. While $\mathbf{f}_{\text{cheap}}$ is evaluated many times in LEMONADE, $\mathbf{f}_{\text{exp}}$ is evaluated only a few times for promising networks that are likely to improve the approximation of the Pareto-front.

In every iteration of LEMONADE, firstly parent networks are sampled with respect to some probability distribution (discussed later) that is solely based on the cheap objectives. By applying mutations to the parents (such as adding or removing a layer, see Sect. 3.2.2 for a detailed description), children are generated. In a second sampling stage, a subset of all generated children is selected, again solely based on cheap objectives, and solely this subset is evaluated on the expensive objectives $\mathbf{f}_{\text{exp}}$. Lastly, LEMONADE computes the Pareto-front from the current generation and the subset of generated children, yielding the next generation. The described procedure is repeated for a prespecified number of iterations.

The sampling distribution The sampling distribution is designed to only depend on the cheap objectives and to guide the search toward sparsely crowded regions in the current Pareto-front. In order to achieve this, LEMONADE computes a kernel density estimator p_{KDE} on the cheap

objective values $\{\mathbf{f}_{\text{cheap}}(N)|N \in \mathcal{P}\}$ of the current population. Then, for both sampling stages (i.e., (i) the probability for choosing a network N as a parent as well as (ii) the probability of a generated child N being part of the subset), LEMONADE uses a sampling distribution anti-proportional to p_{KDE}:

$$p(N) = \frac{c}{p_{\text{KDE}}(\mathbf{f}_{\text{cheap}}(N))}, \tag{11}$$

In Eq. (11), c is a proper normalizing constant. Therefore, networks in sparsely populated regions of the Pareto-front are more likely to be chosen as parents and generated children lying in sparsely populated regions of the Pareto-front are more likely to be evaluated on $\mathbf{f}$. The motivation behind also choosing parents in less crowded regions is that mutations do not change the network drastically; hence, children are expected to have similar objective values as their parents. By this sampling distribution and the two-staged sampling strategy, LEMONADE generates and evaluates more children that are more likely to improve the current approximation of the Pareto-front rather than just evaluating the cheap objective $\mathbf{f}_{\text{exp}}(N)$ for all children, making it more efficient than off-the-shelf multi-objective optimization algorithms. We highlight that all objectives from Sect. 3.1 are cheap to evaluate as they all solely depend on the neural network architecture and not, e.g., on the weights of the network only obtainable by expensive training. Hence, LEMONADE is a perfect fit for our purpose. For more details, we refer the reader to the original work [30].

3.2.2 Search space and mutations within LEMONADE

In this work, we focus on NAS for image classification tasks. Convolutional neural networks (CNNs) are the predominantly used type of DNN in this domain [51]. However, in the recent years, the number of variations and design choices for CNN architectures has significantly grown (see, e.g., [34] for an overview). We limit the search space of LEMONADE to a number of predefined building blocks, hyperparameters and allowed mutations for two reasons. Firstly, support for a limited set of building blocks requires less flexibility of the underlying hardware. This enables the use of more efficient dedicated DNN accelerators instead of general purpose hardware. Secondly, the space of feasible architectures $\mathcal{N}$ rapidly grows with each additional variation that is allowed. This combinatorial explosion slows down the convergence of NAS, which is why a reasonable limitation of the search space has to be chosen.

We now describe the set of mutations that are used by LEMONADE in our experiments to generate child networks.

1. Insert a convolutional layer with batch normalization [42] and (ReLU) activation [32]. The layer is inserted at a random position, and its number of filters is chosen to match the number of filters of the preceding layer. The kernel height h and width w of the convolutional filter are randomly sampled: $(h, w) \in \{(3,3), (5,5), (7,7), (9,9)\}$.
2. Increase the number of filters of a randomly chosen convolution by a randomly chosen factor $\in \{2, 4\}$. A maximum of 1100 filters is allowed.
3. Add a skip connection. We allow skip connection either by concatenation [92] or by addition [35].
4. Remove a randomly chosen layer or a skip connection.
5. Prune a randomly chosen convolutional layer (i.e., remove 1/2 or 1/4 of its filters). A minimum of 15 filters is allowed.
6. Replace a randomly chosen convolution by a depthwise separable convolution [18].

Note that by random we always mean uniformly at random. We highlight that the first three operations in general increase objectives such as network's size or energy consumption, but likely also decrease objectives such as the error, while the last three operations in general decrease the firstly mentioned objectives, but increase the lastly objectives. Consequently, these mutations are suitable for multiple, opposing objectives.

To further speed up NAS, the authors of LEMONADE propose to apply these mutations as *network morphisms* [14, 99]. Network morphisms are function-preserving operators on neural networks, i.e., a network morphism maps a neural network N^w with weights w to another neural network $\tilde{N}^{\tilde{w}}$ with weights $\tilde{w}$ so that for every input x to the network $N^w(x) = \tilde{N}^{\tilde{w}}(x)$. Effectively this means that, when utilizing network morphisms as mutations to generate children, children *do not need to be trained from scratch* but rather just fine-tuned as children *by design* have the same error as their parent. This can be interpreted as Lamarckian inheritance in the context of evolutionary algorithms, where Lamarckism refers to a mechanism which allows passing skills acquired during an individual's lifetime (e.g., by means of learning), on to children by means of inheritance. The equality $N^w(x) = \tilde{N}^{\tilde{w}}(x)$ can be achieved by properly choosing $\tilde{w}$. For example, if one wants to insert a linear layer at an arbitrary position in a network, equality can be achieved by simply initializing the linear layer as an identity mapping. Mutations 1–3 from above can all be formulated as a network morphism (see [30] for details). Mutations 4–6, on the other hand, cannot be framed as network morphisms, as they all generally decrease the network's capacity and equality cannot be guaranteed. Instead, Elsken et al. [30] propose *approximate* network morphisms to find proper

initialization for these cases. Approximate network morphisms essentially copy the weights of layers not affected by structural changes and train affected layers via knowledge distillation [37].

3.3 Fixed-point quantization

Neural network training algorithms usually rely on data representations and computations with high numerical precision, for example a 32-bit floating-point format, typically used in graphics processing units (GPUs). However, after training, a reduced precision number format can be used for inference on a dedicated DNN accelerator to reduce energy consumption and bandwidth [55]. In this context, an 8-bit fixed-point format is a common choice in embedded and mobile devices [43]. Hence, to deploy a DNN on an embedded device after training on a GPU, weights, biases, and activations need to be transformed from a floating-point to a fixed-point number format. This procedure is denoted by network *quantization*. We apply network quantization as post-processing step after neural architecture search with `LEMONADE`.

The quantization of a real value χ to a signed fixed-point value χ_q using B bit is given by

$$\chi_q = \text{clip}\left(\text{round}\left(\frac{\chi}{\Delta}\right), -2^{B-1}, 2^{B-1}\right)\Delta. \tag{12}$$

In Eq. (12), $\text{clip}(x, a, b) = \min(\max(x, a), b)$, the function $\text{round}(x)$ denotes rounding x to an integer number, and Δ denotes the step size, i. e., the smallest distance between two quantization sampling points of χ. In other words, Δ corresponds to the value of the least significant bit (LSB).

In [96], a simple method to find a suitable step size for a given data distribution in DNNs with sigmoid activations has been introduced. It determines the step size Δ based on the maximum range of a distribution according to

$$\Delta = \frac{\max(|\chi|)}{2^{N-1} - 1}. \tag{13}$$

In the following, we refer to this quantization method as *MaxRange*.

However, modern DNNs commonly use unbounded activation functions, such as ReLU, and thus may entail data distributions with far outliers. Since the quantization range is adapted to the maximum value, the step size Δ is maximal and consequently leads to a coarse sampling of smaller values. Moreover, as data distributions in DNNs typically follow a Gaussian distribution, Eq. (13) leads to a coarse sampling of a large number of values.

A quantization method which specifically targets this problem has been introduced in [98]. Here, parameters and activations are quantized by minimizing the effect of the quantization error $\delta = \chi - \chi_q$ in the network. In a neural network, the output value y of a neuron with a rectifying unit $\Phi(\cdot)$, bias b, weights w and input values x is determined by

$$y = \Phi\left(b + \sum wx\right). \tag{14}$$

For the purpose of measuring the influence of the quantization error of inputs (δ_x), weights (δ_w) and biases (δ_b), we define $\tilde{y}$ as the resulting neuron output when quantities of Eq. (14) are quantized. More precisely, $\tilde{y}_w$ is defined as the neuron output determined with quantized weights w_q where activations and biases remain in a 32 bit floating-point number format. $\tilde{y}_x$ and $\tilde{y}_b$ are defined accordingly. The optimal quantization step sizes $\Delta^{*(l)}$ are then individually determined for each layer $l \in L_N$ by

$$\begin{aligned}
\Delta_w^{*(l)} &= \arg\min_{\Delta_w^{(l)} \in \mathcal{D}} \left\| \mathbf{y}^{(l)} - \tilde{\mathbf{y}}_w^{(l)}(\Delta_w^{(l)}) \right\|^2, \\
\Delta_x^{*(l)} &= \arg\min_{\Delta_x^{(l)} \in \mathcal{D}} \left\| \mathbf{y}^{(l)} - \tilde{\mathbf{y}}_x^{(l)}(\Delta_x^{(l)}) \right\|^2, \text{and} \\
\Delta_b^{*(l)} &= \arg\min_{\Delta_b^{(l)} \in \mathcal{D}} \left\| \mathbf{y}^{(l)} - \tilde{\mathbf{y}}_b^{(l)}(\Delta_b^{(l)}) \right\|^2.
\end{aligned} \tag{15}$$

In Eq. (15), the vectors $\mathbf{y}^{(l)}$, $\tilde{\mathbf{y}}_w^{(l)}$, $\tilde{\mathbf{y}}_x^{(l)}$, and $\tilde{\mathbf{y}}_b^{(l)}$ are composed of the individual y, $\tilde{y}_w$, $\tilde{y}_x$, and $\tilde{y}_b$ of all neurons in layer l, respectively. Additionally, the set of allowed step sizes is constrained to power-of-two values, i.e., $\mathcal{D} = \{2^z \,|\, z \in \mathbb{Z}\}$. This enables a direct fixed-point operation in a hardware accelerator. In the rest of the paper, this quantization method is referred to as *minimal propagated quantization error (MinPQE)*.

4 Experiments

4.1 Experimental setup

To evaluate our methods, we use two common image classification benchmarks. Firstly, CIFAR-10 [49] is used, which consists of 32×32 pixel RGB images divided into ten distinct classes. The samples are divided into 50000 training and 10000 test samples. Out of the training set, 5000 samples are used for validation during neural architecture search.

Secondly, GTSRB [89] is used, which contains RGB images of 43 different types of traffic signs. The images of this benchmark are scaled to a resolution of 48×48 pixels before they are fed into the classifier. The dataset has 39,210 training samples, out of which 4010 are separated for classification validation. An additional set of 12,630 images is used for measuring final test error rates.

Unless otherwise noted, we use the same hyperparameter setup for both benchmarks. We run LEMONADE for 300 evolutionary iterations. The algorithm is initialized with a population of 15 manually chosen trivial network architectures with different numbers of convolutional layers and kernel shapes. For DNN training, we use stochastic gradient descent (SGD) with cosine annealing [60], momentum of 0.9, and a weight decay of 0.0005. The learning rate for each training phase during architecture search is initialized with 0.01. The training batch size is set to 64 throughout our experiments. Furthermore, we apply commonly used data augmentations during training [60]. However, we leave out horizontal image flips for GTSRB, since they would change the meaning of some traffic signs. In addition, we use mixup [108] and cutout [23] for further training data augmentation.

The final population sizes of the CIFAR-10 and GTSRB models are 439 and 238, respectively. From each of these, the 50 architectures with best validation error rates are selected and each of these is trained from scratch on the set of training and validation images for 200 epochs. The learning rate is initialized with 0.025 in this case, and all other hyperparameters stay the same. Classification error is evaluated on the separate test set after the training. Subsequently, we quantize the networks' weights and activations to an 8-bit fixed-point representation using the MaxRange and MinPQE methods described in Sect. 3.3 for further evaluations.

4.2 Error simulations

Random bit-flip error simulations are used to evaluate the actual resilience of the obtained set of neural networks. For this purpose, we use the fault simulation framework that has been previously described in [85]. The framework builds up on the Keras [19] DNN library with TensorFlow back-end [1]. This allows for performing fast bit-level fault injections in the neuron activation outputs (feature maps) of a CNN. Most of the computation workload required for the simulation can be efficiently computed on a GPU. The framework automatically adds some operations behind each neuron output stage of a given CNN, which emulate a fixed-point format and allow for a bit-wise fault injection in the neuron output memory by applying a definable Boolean fault mask (Fig. 2).

4.3 Results

4.3.1 Trade-off analysis between objectives

Table 1 lists the properties of certain DNN architectures N obtained for both benchmarks, CIFAR-10 and GTSRB. The selected models are the ones that minimize each an individual objective function $f_i(N)$ (BestASI, BestValErr, BestEfficiency, and BestADCR), the model with maximum error sensitivity (WorstASI) as well as the model with lowest normalized worst objective value (Sect. 2.3) $\left\|\bar{\mathbf{f}}(N)\right\|_\infty$, i.e., the balanced optimizer of all objectives (BalOpt). Data transfer and accumulated data–computation ratio are calculated taking 8-bit fixed-point quantization of activations and weights into account. It can be observed that the BestEfficiency models actually minimize both $f_{\text{latency}}(N)$ (i.e., operations) and $f_{\text{energy}}(N)$ (i.e., data transfer). This indicates a correlation between the two quantities. The respective models are also the smallest in terms of weight parameters.

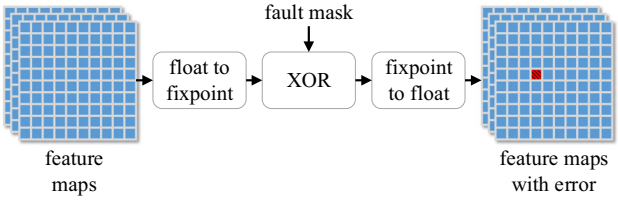


Fig. 2 Steps performed by fault injection framework between the computation of two neural network layers [85]

Furthermore, Table 1 shows that choosing a DNN with minimal cost in one objective often leads to the outcome that at least one other objective is close to its worst value. This is especially the case for CIFAR-10, where $\left\|\bar{\mathbf{f}}(N)\right\|_\infty$ is 1 or close to 1 for all single objective optimizers, BestASI, BestValErr, BestEfficiency, and BestADCR. The optimal trade-off models (BalOpt), however, come quite close to the ideal point, with normalized distances of 0.371 (CIFAR-10) and 0.267 (GTSRB).

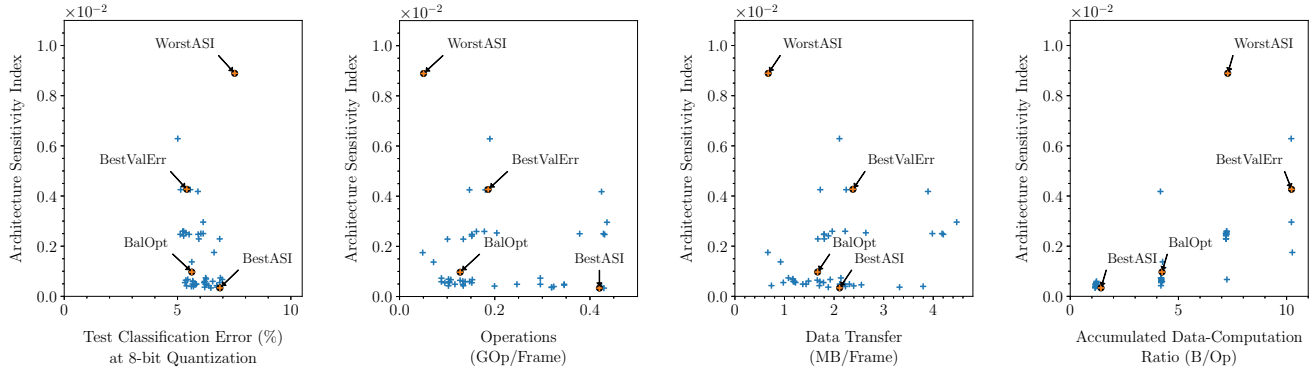
Another aspect visible in Table 1 is that 8-bit quantization does not significantly increase test set classification error rates of the models in comparison with the 32-bit float case (in some cases the error is even smaller after quantization). The differences between the MaxRange and MinPQE quantization methods with respect to test error rate are marginal.

The resulting distributions of objective values for all 50 models that were selected after the optimization with LEMONADE are shown in Figs. 3 and 4 for CIFAR-10 and GTSRB, respectively. The sub-figures (a)–(d) each depict the ASI given by Eq. (6) versus each of the other objective functions. It can be seen that the WorstASI models have comparatively few operations and data transfers. However, the reverse is not always true, since there are models with few operations and data transfers as well as low ASI. In other words, it is possible to have high efficiency and high error resilience at the same time.

Another interesting aspect visible in Figs. 3d and 4d is a correlation between ADCR and ASI. Consequently, a low ratio of data transfers to operations is not only beneficial

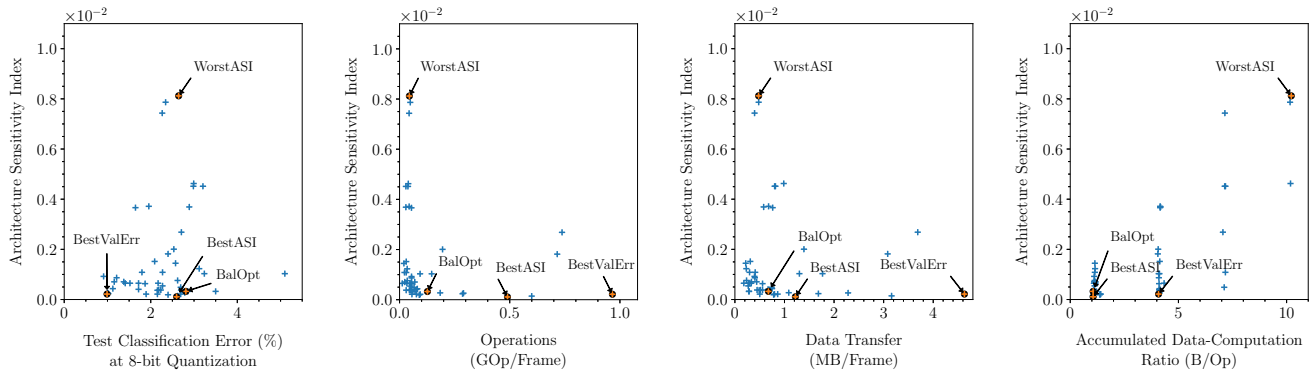
Table 1 Properties of certain architectures with minimal or maximal value in some objective. Bold numbers indicate minimal values among the 50 obtained architectures for each dataset

Model	Optimized quantities					Other quantities				
	Architecture sensitivity index ($\times 10^{-3}$)	Validation set error rate (%)	Operations (GOp/Frame)	Data transfer (MB/Frame)	Acc. data–computation ratio (B/Op)	Normalized worst objective value	Number of parameters ($\times 10^{6}$)	Test set error rate (%) (32b float)	Test set error rate (%) (8b MaxRange)	Test set error rate (%) (8b MinPQE)
CIFAR-10										
WorstASI	8.891	9.20	0.050	0.672	7.279	1.000	0.344	7.31	7.58	7.52
BestASI	**0.336**	9.16	0.420	2.112	1.422	0.959	1.645	6.95	6.91	6.87
BestValErr	4.267	**6.52**	0.186	2.381	10.230	0.996	1.489	5.48	5.33	5.41
BestEfficiency	1.750	9.18	**0.049**	**0.665**	10.264	1.000	**0.337**	6.54	6.68	6.61
BestADCR	0.336	9.30	0.429	2.122	**1.150**	0.993	1.654	6.42	6.57	6.47
BalOpt	0.970	7.56	0.127	1.668	4.241	**0.371**	1.330	5.72	5.66	5.63
GTSRB										
WorstASI	8.120	0.45	0.045	0.478	10.218	1.000	0.101	2.53	2.66	2.64
BestASI	**0.109**	0.30	0.490	1.220	1.058	0.501	0.865	2.60	2.64	2.60
BestValErr	0.217	**0.00**	0.966	4.629	4.081	1.000	3.200	0.90	1.08	0.99
BestEfficiency	0.651	0.45	**0.012**	**0.181**	1.166	0.600	**0.041**	1.32	1.41	1.41
BestADCR	0.145	0.12	0.600	3.161	**1.048**	0.670	2.833	2.50	2.61	2.62
BalOpt	0.326	0.20	0.126	0.676	1.057	**0.267**	0.513	2.78	2.84	2.81



(a) ASI vs. Test Error **(b)** ASI vs. Operations **(c)** ASI vs. Data Transfer **(d)** ASI vs. ADCR

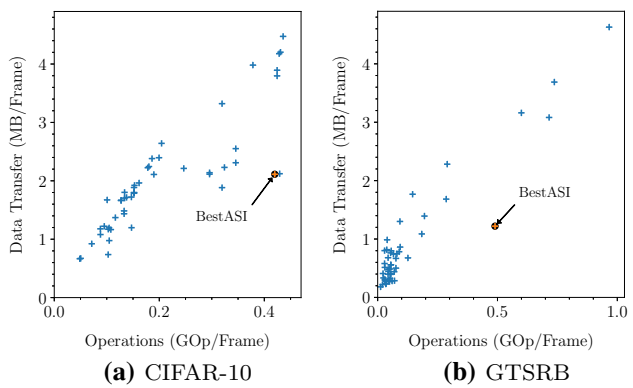
Fig. 3 Pairwise comparison of ASI with each of the other objective function outcomes for 50 Pareto-optimal architectures on CIFAR-10



(a) ASI vs. Test Error **(b)** ASI vs. Operations **(c)** ASI vs. Data Transfer **(d)** ASI vs. ADCR

Fig. 4 Pairwise comparison of ASI with each of the other objective function outcomes for 50 Pareto-optimal architectures on GTSRB

for limiting the required bandwidth of the DNN accelerator, but also helps to reduce error sensitivity. This aspect becomes also apparent in Fig. 5. It can be seen that models with more operations typically also require more data transfers. However, the BestASI models have a relatively high number of operations in comparison with their data transfers, as they are located offside the main trend in the scatter plot.



(a) CIFAR-10 **(b)** GTSRB

Fig. 5 Data transfer versus number of operations for Pareto-optimal architectures. BestASI models are located offside the main trend

4.3.2 Evaluation of resilience prediction

We now evaluate the predictive performance of our ASI metric Eq. (6) by performing bit-flip fault injections using the framework described in Sect. 4.2. Bit-flips are randomly injected in all convolutional layer feature map outputs (after ReLU activation and pooling, where applicable) that are written to memory. MinPQE quantization with 8 bits is used, except where otherwise specified. The value of each bit in the feature map outputs is toggled with a probability given by a defined BER. To get statistically meaningful results [52], random fault locations are sampled $n = 200$ times, and for each trial the effect on the classification output of the network is measured using the complete test set of the respective benchmark. For this purpose, the (CCR), i.e., the fraction of images in the test set that are classified differently after the fault injection, is calculated. In the ideal case of no corruption of the network output, CCR would be zero. The sample mean of CCR over

all $n = 200$ trials is reported. This can be interpreted as expected probability of SDC at the given BER.

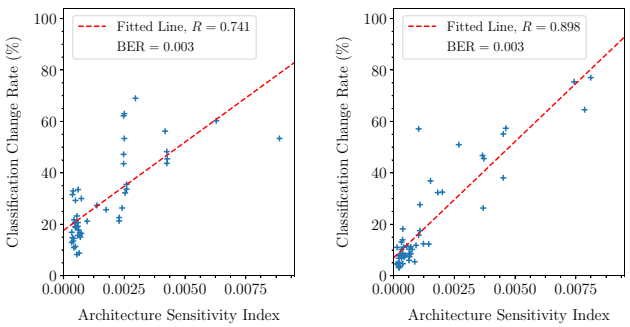
The results of a linear least-squares regression on the ASI versus CCR value pairs of the 50 optimized models for each benchmark are shown in Fig. 6. A BER of 0.003 was used for bit-flip injections. A correlation coefficient $R = 0.741$ is achieved for CIFAR-10 and $R = 0.898$ for GTSRB. While this indicates that our ASI metric Eq. (6) does not explain the variation in CCR completely, the correlation is relatively strong. This is especially surprising, considering the fact that Eq. (6) is completely determined by the architecture of the neural network and does not require any cumbersome measurements based on test data or weight parameters. Thus, we argue that ASI is an efficient and useful heuristic to guide NAS toward more resilient DNN architectures.

We also evaluate CCRs for varying BERs for a subset of models. The results for CIFAR-10 and GTSRB are plotted in Figs. 7 and 8, respectively. An approximately linear dependency between BER and CCR can be observed at very low bit error rates. At higher BERs, a transition first to a rapid growth of CCR (note the log scales) is visible, and then, the value saturates at a value corresponding to chance probability of choosing the same label after fault injection.

An interesting finding observable in Figs. 7 and 8 is that the BestValErr models exhibit an unexpectedly low CCR at low BERs, while they degrade less gracefully (much steeper increase CCR) at high BERs. In the case of GTSRB, BestValErr is actually, despite its higher ASI, much more resilient than BestASI at low BERs. An explanation might be that a good baseline classification performance adds an extra degree of error resilience, which is not captured by Eq. (6). The steeper increase, on the other hand, could be due to an overfitting to the task (i.e., weaker ability for generalization).

4.3.3 Comparison of quantization methods

We now compare the MaxRange and MinPQE quantization methods (Sect. 3.3), with respect to resulting CCRs after bit-flip fault injections with a BER of 0.005. Results are



(a) CIFAR-10 **(b)** GTSRB

Fig. 6 Correlation of ASI and CCR. A correlation coefficient $R = 0.741$ is achieved for CIFAR-10 and $R = 0.898$ for GTSRB

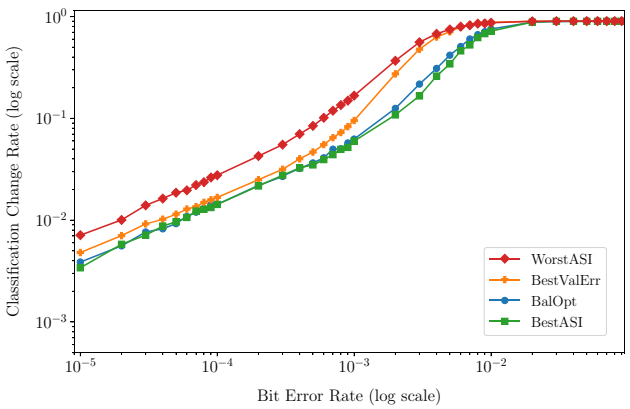


Fig. 7 Resulting CCR for different obtained optimizers on CIFAR-10 over a range of BERs

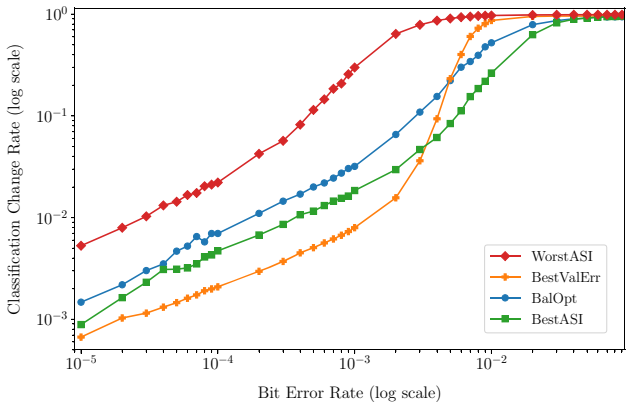


Fig. 8 Resulting CCR for different obtained optimizers on GTSRB over a range of BERs

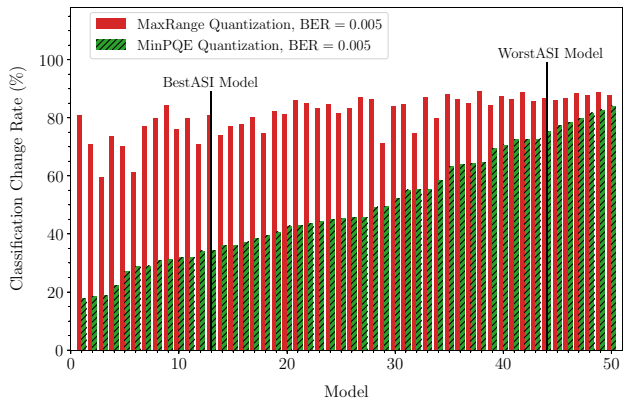


Fig. 9 Comparison of CCR at bit error rate 0.005 for CIFAR-10 models quantized with the MaxRange and MinPQE quantization methods. Models sorted after CCR observed with MinPQE quantization

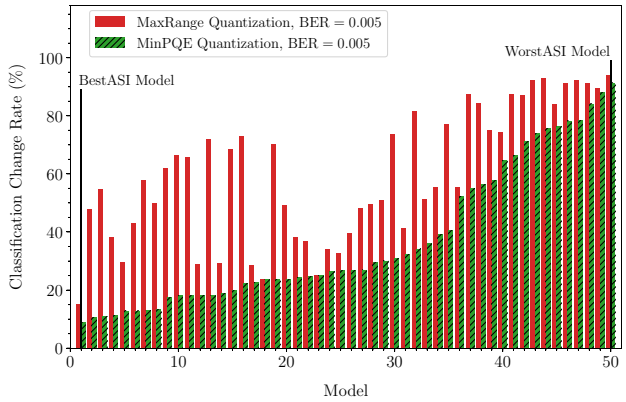


Fig. 10 Comparison of CCR at bit error rate 0.005 for GTSRB models quantized with the MaxRange and MinPQE quantization methods. Models sorted after CCR observed with MinPQE quantization

shown in Figs. 9 and 10. The models are sorted in ascending order of CCR after MinPQE quantization in these figures.

It can be seen that MaxRange results in a significantly worse CCR in most of the cases. This can be explained by the fact that MaxRange tends to quantize values to a larger range, which is determined by far outliers, while these outliers are ignored (i.e., clipped) by MinPQE. Consequently, MaxRange leads to a weaker *signal-to-noise ratio* compared to MinPQE in the case of bit-flip errors. We thus argue that MinPQE is the preferable method, since it achieves both, low baseline classification error rates as well as high error resilience.

4.3.4 Comparison with differently optimized networks

To highlight the advantage of taking error resilience and other hardware-oriented objective functions into account during the neural architecture search process, we now compare the models found by our multi-objective optimization algorithm with two well-known reference networks. Firstly, we consider MobileNetV2 [80], which has been derived in a manual neural architecture optimization process. The authors of MobileNetV2 point out that their goal was to find an architecture with low classification error, while keeping the number of weights and operations of the network small. They target mobile, performance-constrained compute platforms.

Secondly, we compare our results with NASNet-A [112], an architecture found by NAS using a reinforcement learning approach. With 2000 GPU-days, the optimization process in [112] requires about 100 times more computational resources than our work. The authors of this method solely incorporated a single objective, namely classification error rate, to be minimized by their search algorithm.

Our algorithm yields a set of Pareto-optimal architectures, while the references only come with a single base architecture. To account for the trade-off possibility of model size against classification performance, we evaluate differently scaled versions of both reference architectures. As suggested by the authors of MobileNetV2 [80], we use a width multiplier α with which the numbers of filters in all convolutional layers except the very last are scaled. We choose $\alpha \in \{0.2, 0.25, 0.5, 0.75, 1.0, 1.25\}$. For NASNet-A, we use scaled versions of the CIFAR-10 architecture following the construction rule defined in the paper [112]. The architectures are described using the notation $(n \,@\, p)$, where n is the number of cells and p is the number of penultimate convolutional filters of the architecture. We evaluated nine different parametrizations in the range of $(1 \,@\, 192)$ up to $(4 \,@\, 768)$.

For a fair comparison, we train all reference architectures from scratch using exactly the same training setup and hyperparameters as used in the final retraining step for the architectures found by our method (Sect. 4.1). Furthermore, we quantize the reference architectures in the same way using the MinPQE method (Sect. 3.3) with 8-bit resolution.

For the following evaluation, we select the top-10 of our architectures with lowest obtained ASI scores found for the CIFAR-10 benchmark and compare them with the scaled variants of the two reference architectures also trained on CIFAR-10. The plot in Fig. 11 compares the resulting data transfer (MB/Frame), classification error rate, and hardware error resilience, measured by CCR for a given BER of 0.005 in the feature map outputs. As before, mean CCRs over 200 evaluations are reported. The construction of the upscaled architecture will be explained further below.

A first interesting observation is that all our models outperform the reference models in terms of error resilience, since they achieve significantly lower CCRs. Put in numbers, at least 75% of the MobileNetV2 classification

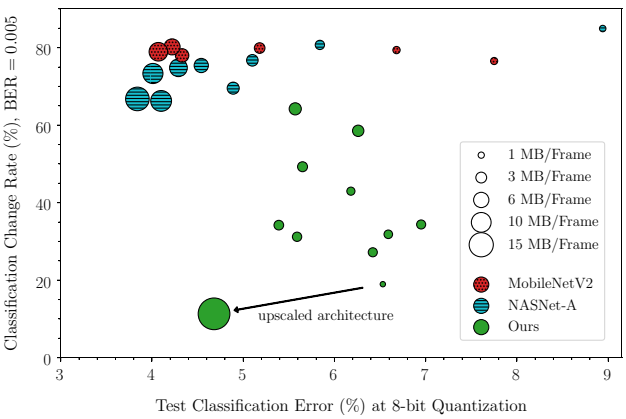


Fig. 11 Comparison of CCR, test error, and data transfer of our models and competing architectures on CIFAR-10

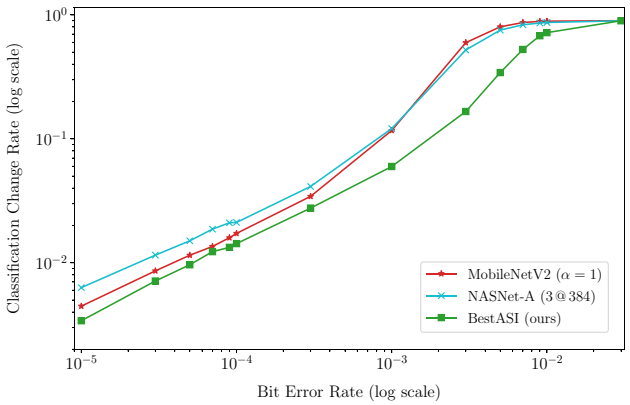


Fig. 12 Comparison of our BestASI model and references on CIFAR-10 based on CCR over BER

outputs and at least 65% of the NASNet-A classification outputs are corrupted after fault injection, while our best architecture achieves a CCR of less than 20% at the given BER. As visible in Fig. 12, we not only outperform the reference models at this BER, but over a broad range of BERs in terms of error resilience (remember that BestASI not even has the lowest CCR, see Fig. 9). These results indicate that incorporating our ASI heuristic Eq. (6) as objective function into the architecture search process helps finding significantly more resilient architectures.

At the same time, the influence of energy efficiency optimization by the objective function defined in Eq. (8) is visible in Fig. 11, since the amount of data transfer per frame is comparatively low for our architectures (except the upscaled version). In contrast, the largest evaluated NASNet-A architecture requires a data transfer of about 14.2 MB per image classification. This can be explained by the large number of 170 convolutional layers in this model (ours have six or less). Thus, energy efficiency of this model in a hardware accelerator with layerwise data transfer between compute arithmetic and external memory would be very low. Moreover, since the number of operations for each individual layer is low, the ADCR of the largest NASNet-A model is at a very high value of 217.9 (the worst ADCR among our models is 10.26, see Fig. 3d). For a layerwise hardware accelerator, this means that either throughput is restricted by memory bandwidth, or a large bandwidth is required to operate in a compute-bound domain of the roofline model (Sect. 3.1.2). This example illustrates the importance of not focusing solely on the number of operations for evaluating the efficiency of a neural architecture.

While our top-10 ASI models shown in Fig. 11 achieve low CCR, their baseline test error rate is slightly behind the larger MobileNetV2 and NASNet-A versions, although they are still competitive with respect to their small model size. Therefore, we evaluate if we can improve the classification performance of our models by making them larger. We select the model with lowest CCR and upscale it by multiplying its number of filters in each convolutional layer by a factor of eight. The resulting upscaled model is located in the bottom left part of Fig. 11. It can be seen that upscaling comes at the price of a largely increased data transfer. Nevertheless, an improvement by almost two percentage points in test classification error rate can be achieved. At the same time, CCR can be further lowered to a remarkable value of about 11.3%. This is about a 6× to 7× lower data corruption rate compared to the MobileNetV2 and NASNet-A models. A lower CCR is also expected based on Eq. (6). Thus, our method enables us to construct highly error-resilient model architectures with close to state-of-the-art classification performance.

5 Conclusions

We have introduced a multi-objective optimization method for hardware-focused neural architecture design. Our method incorporates a novel heuristic, called ASI, which predicts the hardware error sensitivity of a neural architecture. We are the first to jointly consider error resilience, efficiency, and performance optimization in a neural architecture search framework.

Utilizing the CIFAR-10 and GTSRB image classification benchmarks, we have demonstrated that our ASI objective performs well in an automated NAS framework and is able to find neural networks with significantly increased error resilience. In comparison with the popular MobileNetV2 and NASNet-A models, the most resilient architecture found by our method achieves about a 6× to 7× lower data corruption rate at 0.5% bit error rate in the feature maps of the network. We have evaluated the predictive performance of ASI by performing linear regression between predicted and measured output corruption rates with a resulting correlation coefficient of up to 0.898.

We have complemented ASI with further hardware-focused objective functions that focus on the efficiency optimization of neural architectures for resource constrained neural accelerators. Since our new objective functions only require topology information of the neural network, they enable an efficient architecture search process. We have utilized the `LEMONADE` algorithm which performs an optimized sampling of the architecture search space in order to obtain Pareto-optimal solutions over a wide range of objective values. This allows the choice of an optimal solution based on the requirements of a given application. Finally, our findings about the influence of different quantization techniques on DNN error resilience highlight the importance of choosing an optimization

technique that fosters a high signal-to-noise ratio to limit the influence of bit-flip errors.

Compliance with ethical standards

Conflict of interest The authors declare that they have no conflict of interest.

References

1. Abadi M, Agarwal A, Barham P, Brevdo E, Chen Z, Citro C, Corrado GS, Davis A, Dean J, Devin M, Ghemawat S, Goodfellow I, Harp A, Irving G, Isard M, Jia Y, Jozefowicz R, Kaiser L, Kudlur M, Levenberg J, Mane D, Monga R, Moore S, Murray D, Olah C, Schuster M, Shlens J, Steiner B, Sutskever I, Talwar K, Tucker P, Vanhoucke V, Vasudevan V, Viegas F, Vinyals O, Warden P, Wattenberg M, Wicke M, Yu Y, Zheng X (2015) Tensorflow: large-scale machine learning on heterogeneous distributed systems. https://www.tensorflow.org/
2. Aitken R, Cannon EH, Pant M, Tahoori MB (2015) Resiliency challenges in sub-10nm technologies. In: IEEE 33rd VLSI Test Symposium (VTS), pp 1–4. https://doi.org/10.1109/VTS.2015.7116281
3. Azizimazreah A, Gu Y, Gu X, Chen L (2018) Tolerating soft errors in deep learning accelerators with reliable on-chip memory designs. In: IEEE international conference on networking, architecture and storage (NAS), pp 1–10. https://doi.org/10.1109/NAS.2018.8515692
4. Bach S, Binder A, Montavon G, Klauschen F, Müller KR, Samek W (2015) On pixel-wise explanations for non-linear classifier decisions by layer-wise relevance propagation. PLoS ONE. https://doi.org/10.1371/journal.pone.0130140
5. Baker B, Gupta O, Naik N, Raskar R (2017) Designing neural network architectures using reinforcement learning. In: International conference on learning representations (ICLR)
6. Baker B, Gupta O, Raskar R, Naik N (2017) Accelerating neural architecture search using performance prediction. In: NIPS workshop on meta-learning
7. Bender G, Kindermans PJ, Zoph B, Vasudevan V, Le Q (2018) Understanding and simplifying one-shot architecture search. In: International conference on machine learning (ICML)
8. Blasco X, Herrero JM, Sanchis J, Martínez M (2008) A new graphical visualization of n-dimensional Pareto front for decision-making in multiobjective optimization. Inf Sci 178(20):3908–3924. https://doi.org/10.1016/j.ins.2008.06.010
9. Cai H, Chen T, Zhang W, Yu Y, Wang J (2018) Efficient architecture search by network transformation. In: AAAI
10. Cai H, Yang J, Zhang W, Han S, Yu Y (2018) Path-level network transformation for efficient architecture search. In: International conference on machine learning (ICML)
11. Cai H, Zhu L, Han S (2019) ProxylessNAS: direct neural architecture search on target task and hardware. In: International conference on learning representations (ICLR)
12. Cai L, Barneche AM, Herbout A, Sheng Foo C, Lin J, Ramaseshan Chandrasekhar V, Sabry M (2019) TEA-DNN: the quest for time-energy-accuracy co-optimized deep neural networks. In: International symposium on low power electronics and design (ISLPED). https://doi.org/10.1109/ISLPED.2019.8824934
13. Carter NP, Naeimi H, Gardner DS (2010) Design techniques for cross-layer resilience. In: Design, automation & test in Europe conference & exhibition (DATE), pp 1023–1028. https://doi.org/10.1109/DATE.2010.5456960
14. Chen T, Goodfellow IJ, Shlens J (2016) Net2Net: accelerating learning via knowledge transfer. In: International conference on learning representations (ICLR)
15. Chen YH, Krishna T, Emer JS, Sze V (2017) Eyeriss: an energy-efficient reconfigurable accelerator for deep convolutional neural networks. IEEE J Solid-State Circuits 52(1):127–138. https://doi.org/10.1109/JSSC.2016.2616357
16. Cheng AC, Dong JD, Hsu CH, Chang SH, Sun M, Chang SC, Pan JY, Chen YT, Wei W, Juan DC (2018) Searching toward pareto-optimal device-aware neural architectures. In: Proceedings of the international conference on computer-aided design (ICCAD), ICCAD '18. https://doi.org/10.1145/3240765.3243494
17. Chenxi L, Liang Chieh C, Florian S, Hartwig A, Wei H, Alan L Y, Li FF (2019) Auto-deeplab: hierarchical neural architecture search for semantic image segmentation. In: Conference on computer vision and pattern recognition (CVPR). https://doi.org/10.1109/CVPR.2019.00017
18. Chollet F (2017) Xception: deep learning with depthwise separable convolutions. In: IEEE conference on computer vision and pattern recognition (CVPR), pp 1800–1807. https://doi.org/10.1109/CVPR.2017.195
19. Chollet F et al (2015) Keras. https://keras.io
20. Deb K, Kalyanmoy D (2001) Multi-objective optimization using evolutionary algorithms. Wiley, New York. https://doi.org/10.5555/559152
21. Deb K, Agrawal S, Pratap A, Meyarivan T (2000) A fast elitist non-dominated sorting genetic algorithm for multi-objective optimization: NSGA-II. In: Schoenauer M, Deb K, Rudolph G, Yao X, Lutton E, Merelo JJ, Schwefel HP (eds) Parallel problem solving from nature PPSN VI. Springer, Heidelberg, pp 849–858
22. Deng J, Fang Y, Du Z, Wang Y, Li H, Temam O, Ienne P, Novo D, Li X, Chen Y, Wu C (2015) Retraining-based timing error mitigation for hardware neural networks. In: Design, automation and test in Europe conference and exhibition (DATE), pp 593–596
23. DeVries T, Taylor GW (2017) Improved regularization of convolutional neural networks with cutout. eprint arXiv:1708.04552
24. Dias FM, Borralho R, Fontes P, Antunes A (2010) FTSET: a software tool for fault tolerance evaluation and improvement. Neural Comput Appl 19(5):701–712. https://doi.org/10.1007/s00521-009-0329-0
25. Dong JD, Cheng AC, Juan DC, Wei W, Sun M (2018) Dpp-net: Device-aware progressive search for pareto-optimal neural architectures. In: Ferrari V, Hebert M, Sminchisescu C, Weiss Y (eds) 15th European conference on computer vision (ECCV). https://doi.org/10.1007/978-3-030-01252-6_32
26. Dreslinski RG, Wieckowski M, Blaauw D, Sylvester D, Mudge T (2010) Near-threshold computing: reclaiming Moore's law through energy efficient integrated circuits. Proc IEEE 98(2):253–266. https://doi.org/10.1109/JPROC.2009.2034764
27. Ehrgott M, Tenfelde-Podehl D (2003) Computation of ideal and Nadir values and implications for their use in MCDM methods. Eur J Oper Res 151(1):119–139. https://doi.org/10.1016/S0377-2217(02)00595-7
28. El Mhamdi EM, Guerraoui R (2017) When neurons fail. In: IEEE international parallel and distributed processing symposium (IPDPS), pp 1028–1037. https://doi.org/10.1109/IPDPS.2017.66
29. Elsken T, Metzen JH, Hutter F (2017) Simple and efficient architecture search for convolutional neural networks. In: NIPS workshop on meta-learning

30. Elsken T, Metzen JH, Hutter F (2019) Efficient multi-objective neural architecture search via Lamarckian evolution. In: International conference on learning representations (ICLR)
31. Elsken T, Metzen JH, Hutter F (2019) Neural architecture search: a survey. J Mach Learn Res 20(55):1–21
32. Glorot X, Bordes A, Bengio Y (2011) Deep sparse rectifier neural networks. In: International conference on artificial intelligence and statistics (AISTATS), vol 15
33. Gomez LB, Cappello F, Carro L, DeBardeleben N, Fang B, Gurumurthi S, Pattabiraman K, Rech P, Reorda MS (2014) GPGPUs: how to combine high computational power with high reliability. In: Design, automation and test in Europe conference and exhibition (DATE). https://doi.org/10.7873/DATE.2014.354
34. Gu J, Wang Z, Kuen J, Ma L, Shahroudy A, Shuai B, Liu T, Wang X, Wang G, Cai J, Chen T (2018) Recent advances in convolutional neural networks. Pattern Recognit 77:354–377. https://doi.org/10.1016/j.patcog.2017.10.013
35. He K, Zhang X, Ren S, Sun J (2016) Deep residual learning for image recognition. In: IEEE conference on computer vision and pattern recognition (CVPR), pp 770–778. https://doi.org/10.1109/CVPR.2016.90
36. Henkel J, Bauer L, Dutt N, Gupta P, Nassif S, Shafique M, Tahoori M, Wehn N (2013) Reliable on-chip systems in the nano-era. In: 50th annual design automation conference (DAC), pp 695–704. https://doi.org/10.1145/2463209.2488857
37. Hinton G, Vinyals O, Dean J (2015) Distilling the knowledge in a neural network. arXiv preprint arxiv: 1503.02531
38. Horowitz M (2014) Computing's energy problem (and what we can do about it). In: IEEE international solid-state circuits conference (ISSCC), pp 10–14. https://doi.org/10.1109/ISSCC.2014.6757323
39. Hsu CH, Chang SH, Juan DC, Pan JY, Chen YT, Wei W, Chang SC (2018) MONAS: multi-objective neural architecture search. arXiv preprint
40. Huang G, Liu Z, van der Maaten L, Weinberger KQ (2017) Densely connected convolutional networks. In: IEEE conference on computer vision and pattern recognition (CVPR), pp 4700–4708. https://doi.org/10.1109/CVPR.2017.243
41. Hutter F, Kotthoff L, Vanschoren J (eds) (2019) Automated machine learning: methods, systems, challenges. Springer, Berlin. https://doi.org/10.1007/978-3-030-05318-5
42. Ioffe S, Szegedy C (2015) Batch normalization: accelerating deep network training by reducing internal covariate shift. In: International conference on machine learning (ICML)
43. Jacob B, Kligys S, Chen B, Zhu M, Tang M, Howard AG, Adam H, Kalenichenko D (2018) Quantization and training of neural networks for efficient integer-arithmetic-only inference. In: IEEE conference on computer vision and pattern recognition (CVPR). https://doi.org/10.1109/CVPR.2018.00286
44. Kerlirzin P, Vallet F (1993) Robustness in multilayer perceptrons. Neural Comput 5(3):473–482. https://doi.org/10.1162/neco.1993.5.3.473
45. Kim S, Howe P, Moreau T, Alaghi A, Ceze L, Visvesh S (2018) MATIC: Learning around errors for efficient low-voltage neural network accelerators. In: Design, automation and test in Europe conference and exhibition (DATE). https://doi.org/10.23919/DATE.2018.8341970
46. Kim YH, Reddy B, Yun S, Seo C (2017) NEMO: neuro-evolution with multiobjective optimization of deep neural network for speed and accuracy. In: ICML'17 AutoML workshop
47. Klein A, Falkner S, Springenberg JT, Hutter F (2017) Learning curve prediction with Bayesian neural networks. In: International conference on learning representations (ICLR)
48. Koopman P, Wagner M (2016) Challenges in autonomous vehicle testing and validation. SAE Int J Transp Saf 4(1):15–24. https://doi.org/10.4271/2016-01-0128
49. Krizhevsky A (2009) Learning multiple layers of features from tiny images. Master Thesis, University of Toronto
50. Krogh A, Hertz JA (1991) A simple weight decay can improve generalization. In: Advances in neural information processing systems
51. LeCun Y, Bengio Y, Hinton G (2015) Deep learning. Nature 521(7553):436–444. https://doi.org/10.1038/nature14539
52. Leveugle R, Calvez A, Maistri P, Vanhauwaert P (2009) Statistical fault injection: quantified error and confidence. In: Design, automation and test in Europe conference and exhibition (DATE), pp 502–506. https://doi.org/10.1109/DATE.2009.5090716
53. Li G, Hari SKS, Sullivan M, Tsai T, Pattabiraman K, Emer J, Keckler SW (2017) Understanding error propagation in deep learning neural network (DNN) accelerators and applications. In: Proceedings of the international conference for high performance computing, networking, storage and analysis. https://doi.org/10.1145/3126908.3126964
54. Li J, Wen G, Gan J, Zhang L, Zhang S (2019) Sparse nonlinear feature selection algorithm via local structure learning. Emerg Sci J. https://doi.org/10.28991/esj-2019-01175
55. Lin DD, Talathi SS, Annapureddy VS (2016) Fixed point quantization of deep convolutional networks. In: International conference on machine learning (ICML), vol 48, pp 2849–2858
56. Lin SC, Zhang Y, Hsu CH, Skach M, Haque ME, Tang L, Mars J (2018) The architectural implications of autonomous driving: constraints and acceleration. In: International conference on architectural support for programming languages and operating systems, pp 751–766. https://doi.org/10.1145/3173162.3173191
57. Liu C, Hu M, Strachan JP, Li H (2017) Rescuing memristor-based neuromorphic design with high defects. In: 54th annual design automation conference (DAC), pp 1–6. https://doi.org/10.1145/3061639.3062310
58. Liu C, Zoph B, Neumann M, Shlens J, Hua W, Li LJ, Fei-Fei L, Yuille A, Huang J, Murphy K (2018) Progressive neural architecture search. In: 15th European conference on computer vision (ECCV). https://doi.org/10.1007/978-3-030-01246-5_2
59. Liu H, Simonyan K, Yang Y (2019) DARTS: differentiable architecture search. In: International conference on learning representations (ICLR)
60. Loshchilov I, Hutter F (2017) SGDR: stochastic gradient descent with warm restarts. In: International conference on learning representations (ICLR)
61. Lu Z, Whalen I, Boddeti V, Dhebar Y, Deb K, Goodman E, Banzhaf W (2019) NSGA-net: a multi-objective genetic algorithm for neural architecture search. In: Genetic and evolutionary computation conference (GECCO). https://doi.org/10.1145/3321707.3321729
62. Mahdiani HR, Fakhraie SM, Lucas C (2012) Relaxed fault-tolerant hardware implementation of neural networks in the presence of multiple transient errors. IEEE Trans Neural Netw Learn Syst 23(8):1215–1228. https://doi.org/10.1109/TNNLS.2012.2199517
63. Marques J, Andrade J, Falcao G (2017) Unreliable memory operation on a convolutional neural network processor. In: IEEE international workshop on signal processing systems (SiPS). https://doi.org/10.1109/SiPS.2017.8110024
64. Miettinen K (1999) Nonlinear multiobjective optimization. Springer, Berlin
65. Miikkulainen R, Liang J, Meyerson E, Rawal A, Fink D, Francon O, Raju B, Shahrzad H, Navruzyan A, Duffy N, Hodjat B (2017) Evolving deep neural networks. arXiv:1703.00548

66. Mittal S (2016) A survey of techniques for approximate computing. ACM Comput Surv 48(4):1–33. https://doi.org/10.1145/2893356
67. Mittal S (2020) A survey on modeling and improving reliability of DNN algorithms and accelerators. J Syst Archit. https://doi.org/10.1016/j.sysarc.2019.101689
68. Montavon G, Lapuschkin S, Binder A, Samek W, Müller KR (2017) Explaining nonlinear classification decisions with deep Taylor decomposition. Pattern Recogn 65:211–222. https://doi.org/10.1016/j.patcog.2016.11.008
69. Montavon G, Samek W, Müller KR (2018) Methods for interpreting and understanding deep neural networks. Digit Signal Proc 73:1–15. https://doi.org/10.1016/j.dsp.2017.10.011
70. Mutlu O (2017) The Row–Hammer problem and other issues we may face as memory becomes denser. In: Design, automation and test in Europe conference and exhibition (DATE). https://doi.org/10.23919/DATE.2017.7927156
71. Pham H, Guan MY, Zoph B, Le QV, Dean J (2018) Efficient neural architecture search via parameter sharing. In: International conference on machine learning (ICML)
72. Piuri V (2001) Analysis of fault tolerance in artificial neural networks. J Parallel Distrib Comput 61(1):18–48. https://doi.org/10.1006/jpdc.2000.1663
73. Reagen B, Whatmough P, Adolf R, Rama S, Lee H, Lee SK, Hernandez-Lobato JM, Wei GY, Brooks D (2016) Minerva: enabling low-power, highly-accurate deep neural network accelerators. In: ACM/IEEE 43rd annual international symposium on computer architecture (ISCA), pp 267–278. https://doi.org/10.1109/ISCA.2016.32
74. Reagen B, Gupta U, Pentecost L, Whatmough P, Lee SK, Mulholland N, Brooks D, Wei GY (2018) Ares: a framework for quantifying the resilience of deep neural networks. In: 55th annual design automation conference (DAC). https://doi.org/10.1109/DAC.2018.8465834
75. Real E, Moore S, Selle A, Saxena S, Suematsu YL, Tan J, Le QV, Kurakin A (2017) Large-scale evolution of image classifiers. In: Precup D, Teh YW (eds) International conference on machine learning (ICML), PMLR, International Convention Centre, Sydney, Australia, proceedings of machine learning research, vol 70, pp 2902–2911
76. Real E, Aggarwal A, Huang Y, Le QV (2019) Regulraized evolution for image classifier architecture search. In: AAAI
77. Saikia T, Marrakchi Y, Zela A, Hutter F, Brox T (2019) AutoDispNet: improving disparity estimation with AutoML
78. Salami B, Unsal OS, Kestelman AC (2018) On the resilience of RTL NN accelerators: fault characterization and mitigation. In: 30th international symposium on computer architecture and high performance computing (SBAC-PAD), pp 322–329. https://doi.org/10.1109/CAHPC.2018.8645906
79. Saljoughi AS, Mehvarz M, Mirvaziri H (2017) Attacks and intrusion detection in cloud computing using neural networks and particle swarm optimization algorithms. Emerg Sci J. https://doi.org/10.28991/ijse-01120
80. Sandler M, Howard A, Zhu M, Zhmoginov A, Chen LC (2018) MobileNetV2: Inverted residuals and linear bottlenecks. In: IEEE conference on computer vision and pattern recognition (CVPR). https://doi.org/10.1109/CVPR.2018.00474
81. Santos FF, Pimenta PF, Lunardi C, Draghetti L, Carro L, Kaeli D, Rech P (2019) Analyzing and increasing the reliability of convolutional neural networks on GPUs. IEEE Trans Reliab 68(2):663–677. https://doi.org/10.1109/TR.2018.2878387
82. Saxena S, Verbeek J (2016) Convolutional neural fabrics. In: Lee DD, Sugiyama M, Luxburg UV, Guyon I, Garnett R (eds) Conference on neural information processing systems (NIPS). Curran Associates Inc., Red Hook, pp 4053–4061
83. Schorn C, Guntoro A, Ascheid G (2018) Accurate neuron resilience prediction for a flexible reliability management in neural network accelerators. In: Design, automation and test in Europe conference and exhibition (DATE). https://doi.org/10.23919/DATE.2018.8342151
84. Schorn C, Guntoro A, Ascheid G (2018) Efficient on-line error detection and mitigation for deep neural network accelerators. In: Gallina B, Skavhaug A, Bitsch F (eds) Computer safety, reliability, and security (SAFECOMP), LNCS, vol 11093. Springer, Berlin. https://doi.org/10.1007/978-3-319-99130-6_14
85. Schorn C, Guntoro A, Ascheid G (2019) An efficient bit-flip resilience optimization method for deep neural networks. In: Design, automation and test in Europe conference and exhibition (DATE), pp 1486–1491. https://doi.org/10.23919/DATE.2019.8714885
86. Sridharan V, DeBardeleben N, Blanchard S, Ferreira KB, Stearley J, Shalf J, Gurumurthi S (2015) Memory errors in modern systems: the good, the bad, and the ugly. In: Twentieth international conference on architectural support for programming languages and operating systems (ASPLOS), pp 297–310. https://doi.org/10.1145/2694344.2694348
87. Srinivasan G, Wijesinghe P, Sarwar SS, Jaiswal A, Roy K (2016) Significance driven hybrid 8T-6T SRAM for energy-efficient synaptic storage in artificial neural networks. In: Design, automation and test in Europe conference and exhibition (DATE)
88. Srivastava N, Hinton GE, Krizhevsky A, Sutskever I, Salakhutdinov RR (2014) Dropout: a simple way to prevent neural networks from overfitting. J Mach Learn Res 15:1929–1958
89. Stallkamp J, Schlipsing M, Salmen J, Igel C (2012) Man vs. computer: benchmarking machine learning algorithms for traffic sign recognition. Neural Netw 32:323–332. https://doi.org/10.1016/j.neunet.2012.02.016
90. Stanley KO, Miikkulainen R (2002) Evolving neural networks through augmenting topologies. Evol Comput 10:99–127. https://doi.org/10.1162/106365602320169811
91. Sze V, Chen YH, Yang TJ, Emer JS (2017) Efficient processing of deep neural networks: a tutorial and survey. Proc IEEE 105(12):2295–2329. https://doi.org/10.1109/JPROC.2017.2761740
92. Szegedy C, Vanhoucke V, Ioffe S, Shlens J, Wojna Z (2016) Rethinking the inception architecture for computer vision. In: IEEE conference on computer vision and pattern recognition (CVPR), pp 2818–2826. https://doi.org/10.1109/CVPR.2016.308
93. Tan M, Chen B, Pang R, Vasudevan V, Le QV (2019) MnasNet: platform-aware neural architecture search for mobile. In: IEEE/CVF conference on computer vision and pattern recognition (CVPR). https://doi.org/10.1109/CVPR.2019.00293
94. Torres-Huitzil C, Girau B (2017) Fault and error tolerance in neural networks: a review. IEEE Access 5:17322–17341. https://doi.org/10.1109/ACCESS.2017.2742698
95. Vaezi Nejad SM, Marandi SM, Salajegheh E (2019) A hybrid of artificial neural networks and particle swarm optimization algorithm for inverse modeling of leakage in earth dams. Civ Eng J. https://doi.org/10.28991/cej-2019-03091392
96. Vanhoucke V, Senior A, Mao MZ (2011) Improving the speed of neural networks on CPUs. In: Deep learning and unsupervised feature learning workshop, NIPS 2011
97. Venkataramani S, Ranjan A, Roy K, Raghunathan A (2014) AxNN: energy-efficient neuromorphic systems using approximate computing. In: IEEE/ACM international symposium on low power electronics and design (ISLPED), pp 27–32. https://doi.org/10.1145/2627369.2627613

98. Vogel S, Springer J, Guntoro A, Ascheid G (2019) Self-supervised quantization of pre-trained neural networks for multiplierless acceleration. In: Design, automation and test in Europe conference and exhibition (DATE), pp 1088–1093. https://doi.org/10.23919/DATE.2019.8714901
99. Wei T, Wang C, Rui Y, Chen CW (2016) Network morphism. In: Balcan MF, Weinberger KQ (eds) International conference on machine learning (ICML), PMLR, New York, New York, USA, Proceedings of machine learning research, vol 48, pp 564–572
100. Whatmough PN, Lee SK, Brooks D, Wei GY (2018) Dnn engine: a 28-nm timing-error tolerant sparse deep neural network processor for IoT applications. IEEE J Solid-State Circuits 53(9):2722–2731. https://doi.org/10.1109/JSSC.2018.2841824
101. WikiChip (2019) FSD Chip—Tesla. https://en.wikichip.org/wiki/fsd_chip
102. Williams S, Waterman A, Patterson D (2009) Roofline: an insightful visual performance model for multicore architectures. Commun ACM 52(4):65–76. https://doi.org/10.1145/1498765.1498785
103. Wu B, Dai X, Zhang P, Wang Y, Sun F, Wu Y, Tian Y, Vajda P, Jia Y, Keutzer K (2019) FBNet: hardware-aware efficient convnet design via differentiable neural architecture search. In: IEEE/CVF conference on computer vision and pattern recognition (CVPR). https://doi.org/10.1109/CVPR.2019.01099
104. Xia L, Liu M, Ning X, Chakrabarty K, Wang Y (2017) Fault-tolerant training with on-line fault detection for RRAM-based neural computing systems. In: 54th annual design automation conference (DAC). https://doi.org/10.1145/3061639.3062248
105. Xie S, Zheng H, Liu C, Lin L (2019) SNAS: stochastic neural architecture search. In: International conference on learning representations (ICLR)
106. Yang L, Murmann B (2017) SRAM voltage scaling for energy-efficient convolutional neural networks. In: 18th international symposium on quality electronic design (ISQED), pp 7–12. https://doi.org/10.1109/ISQED.2017.7918284
107. Zhang C, Sun G, Fang Z, Zhou P, Pan P, Cong J (2018) Caffeine: towards uniformed representation and acceleration for deep convolutional neural networks. IEEE Trans Comput Aid Des Integr Circuits Syst. https://doi.org/10.1109/TCAD.2017.2785257
108. Zhang H, Cisse M, Dauphin YN, Lopez-Paz D (2018) mixup: beyond empirical risk minimization. In: International conference on learning representations (ICLR)
109. Zhang Q, Wang T, Tian Y, Yuan F, Xu Q (2015) ApproxANN: an approximate computing framework for artificial neural network. In: Design, automation and test in Europe conference and exhibition (DATE), pp 701–706
110. Zhong Z, Yang Z, Deng B, Yan J, Wu W, Shao J, Liu CL (2018) BlockQNN: efficient block-wise neural network architecture generation. arXiv preprint
111. Zoph B, Le QV (2017) Neural architecture search with reinforcement learning. In: International conference on learning representations (ICLR)
112. Zoph B, Vasudevan V, Shlens J, Le QV (2018) Learning transferable architectures for scalable image recognition. In: Conference on computer vision and pattern recognition (CVPR). https://doi.org/10.1109/CVPR.2018.00907