FTT-NAS: Discovering Fault-Tolerant Neural Architecture

Wenshuo Li[1,2*], Xuefei Ning[1,2*], Guangjun Ge[1,2], Xiaoming Chen[3], Yu Wang[1,2], Huazhong Yang[1,2]
[1]Department of Electronic Engineering, Tsinghua University, Beijing, China
[2]Beijing National Research Center for Information Science and Technology, Beijing, China
[3]State Key Laboratory of Computer Architecture, Institute of Computing Technology, CAS, Beijing, China
[1]e-mail: yu-wang@tsinghua.edu.cn

Abstract—With the fast evolvement of deep-learning specific embedded computing systems, applications powered by deep learning are moving from the cloud to the edge. When deploying NNs onto the edge devices under complex environments, there are various types of possible faults: soft errors caused by atmospheric neutrons and radioactive impurities, voltage instability, aging, temperature variations, and malicious attackers. Thus the safety risk of deploying neural networks at edge computing devices in safety-critic applications is now drawing much attention. In this paper, we implement the random bit-flip, Gaussian, and Salt-and-Pepper fault models and establish a multi-objective fault-tolerant neural architecture search framework. On top of the NAS framework, we propose Fault-Tolerant Neural Architecture Search (FT-NAS) to automatically discover convolutional neural network (CNN) architectures that are reliable to various faults in nowadays edge devices. Then we incorporate fault-tolerant training (FTT) in the search process to achieve better results, which we called FTT-NAS. Experiments show that the discovered architecture FT-NAS-Net and FTT-NAS-Net outperform other hand-designed baseline architectures (58.1%/86.6% VS. 10.0%/52.2%), with comparable FLOPs and less parameters. What is more, the architectures trained under a single fault model can also defend against other faults. By inspecting the discovered architecture, we find that there are redundant connections learned to protect the sensitive paths. This insight can guide future fault-tolerant neural architecture design, and we verify it by a modification on ResNet-20 — ResNet-M.

I. Introduction

Convolution Neural Networks (CNN) have achieved breakthroughs in various tasks, including classification [1], detection [2] and segmentation [3], etc. Due to its promising performance, CNNs have been utilized in various safety-critic applications, such as autonomous driving, intelligent surveillance, and identification. In recent years, driven by academic and industrial efforts, the embedded neural network accelerators [4, 5] have been rapidly evolving. With the help of model compression and quantization techniques [6, 7], CNNs are becoming more light-weighted, thus more suitable to be deployed onto edge computing devices. For hardware-aware neural architecture design, researchers have also designed neural architectures that are efficient on mobile phones [8].

Despite the promising performance, the safety and reliability related characteristics of neural networks are widely doubted, thus are attracting more efforts into addressing these issues. In safety-critic applications, there are a lot of safety risks: With the down-scaling of CMOS techniques, circuits become more sensitive to soft errors, coming from atmospheric neutrons and radioactive impurities [9]. Voltage instability, aging and temperature variations are also common problems for edge computing devices, which could lead to an error. Moreover, malicious attackers may influence the results by embedding hardware Trojans, manipulating back-doors, and do memory injection. As the decisions made by neural network lack interpretability and accountability, it is difficult to conduct risk aversion, especially in the complex edge computing scenarios.

Recently, some studies [10, 11] are devoted to analyzing the sensitivity of the model. They proposed to predict whether a neuron is sensitive to faults and then protected the sensitive ones. For fault tolerance, a straightforward way is to introduce redundancy in the hardware. Triple Modular Redundancy (TMR) is a commonly used but expensive method to ensure that the system works normally with a single fault. For increasing the fault tolerance capability from the model perspective, fault-tolerant training (FTT) is usually used [12, 13, 14], in which faults are injected in the training process.

Although redesigning the hardware for reliability is effective, it inevitably introduces large overhead. It would be better if the issue could be mitigated as far as possible from the algorithmic perspective. Existing methods mainly concerned about designing training methods and analysing the sensitivity. Intuitively, the neural architecture is also important for fault tolerance [15], since it determines the "path" of fault propagation. To verify this intuition, we show the accuracy of baselines under random bit-flip fault model[1] with different fault probabilities in Table I. This preliminary exploration showed that the fault tolerance characteristics vary between neural architectures, which motivates the employment of the NAS technique into the designing of a fault-tolerant neural architecture.

In this paper, we employ Neural Architecture Search (NAS) to discover a fault-tolerant neural network architecture, and demonstrate the effectiveness by experiments. The main contributions of this paper are as follows.

*Both authors contributed equally to this work.

[1]The random bit-flip fault model is formalized in Sec III.

978-1-7281-4123-7/20/$31.00

- We establish a multi-objective fault-tolerant neural architecture search framework. On top of the framework, we propose two methods to discover neural architectures with better reliability: **FT-NAS** (NAS with a fault-tolerant multi-objective), and **FTT-NAS** (NAS with a fault-tolerant multi-objective and fault-tolerant training).
- The discovered architectures FT-NAS-Net and FTT-NAS-Net achieve 58.1% and 86.6% accuracy on CIFAR-10 at fault ratio 5%, which is much better than the baselines without/with fault-tolerant training (10%/52.2%). The abilities of our models to defense against different fault models are also illustrated by experiments.
- Draw insights and verify them from the inspection of discovered FTT-NAS-Net: There are redundant connections learned in the architecture to protect these sensitive paths. This insight can guide future fault-tolerant neural architecture design, and we verify it by a modification on ResNet-20. The modified version ResNet-M outperforms the original ResNet-20.

TABLE I
INFORMATION OF THE BASELINE MODELS ON CIFAR-10

Model	Acc.(0/0.5%/1%)	#Params	#FLOPs
ResNet-20	93.2/84.0/65.4	11.2M	1110M
VGG-16[1]	91.5/76.7/55.0	14.7M	626M
MobileNet-V2[2]	92.5/34.1/15.3	8.06M	675M

II. Related work

A. Fault Resilience

There are some related studies on the hazard of faults in neural network computing systems. [9] revealed advanced nanotechnology might make circuits more sensitive to electronic noise, which causes soft errors. [16, 15] explored how the Single Event Upset (SEU) faults impact the FPGA-based CNN computation system. SEU is a phenomenon in which a single particle strikes an electronic device to cause a change in state. [17, 18] provided possible attack methods to interfere with the output of the neural network. SEUs and attacks could be simulated with random bit-flips in the computation. [10, 11] analyzed the robustness of neural networks from the view of algorithms with Stuck-at-Fault (SaF) model.

The methods to improve the fault resilience ability of neural computation system mainly fall into two categories: fault tolerance and fault detection. As for fault tolerance, [19, 20] proposed to protect partial sensitive parameters or layers to balance the overhead and effectiveness. [12, 13, 14] showed that fault-tolerant training is useful to improve the reliability of neural networks. For the fault detection regime, Error Correction Code (ECC) is a powerful method to detect faults. [21] detected faults by checking the code after computation and then tried to repair it. There are also platform-specific detection and fault-tolerant methods proposed [12].

[1]For simplicity, we only keep one fully-connected layer of VGG-16.
[2]For fair comparison, we double the channels of MobileNet-v2.

B. Neural Architecture Search

Neural Architecture Search (NAS), as an automatic neural network architecture design method, has been recently applied to design model architectures for image classification and language models [22, 23, 24]. The architectures found by the NAS techniques have demonstrated surpassing performance than the hand-designed ones. [22] proposed NASNet in 2017. They used a recurrent neural network (RNN) controller to sample architectures, trained them, and use the final validation accuracy to instruct the learning of the controller. Instead of using RL-learned RNN as the controller, [24, 25] used a relaxed differentiable formulation of the neural architecture search problem, and applied gradient-based optimizer for optimizing the architecture parameters; [26, 27] used evolutionary-based methods for sampling new architectures. Although NASNet [22] is powerful, the searching process is extremely slow and computationally expensive. To address this pitfall, a lot of methods are proposed to speed up the performance estimation in NAS. [28] incorporated learning curve extrapolation to predict the performance after a few epochs of training; [26, 27] sampled architectures using mutation on existing trained models and initialized the weights of the sampled architectures by inheriting from the parent model; [23] shared the weights among different sampled architectures, and using the shared weights to evaluate every sampled architecture.

NAS is also used with various objectives other than accuracy. [25, 29] used multiple objectives to learn architectures with better accuracy and lower latency simultaneously.

III. Fault Model

Denoting the input, output, weights and bias of i-th layer as x_i, y_i, W_i, b_i, the computation of a convolution layer is:

$$y_i = f(W_i \circledast x_i + b_i) \tag{1}$$

where $f(\cdot)$ represents the activation functions, for which ReLU $f(x) = \max(x, 0)$ is the most-common choice.

Currently, fixed-point computations are used by most edge devices, thus quantization is usually applied before the model is deployed onto the edge devices [5]. Thus, our simulation incorporates 8-bit quantization for the weights and activations, to keep consistent with the actual deploying scenario.

Our fault model used in training and search is random bit-flip, which is an approximation of SEUs. Denoting the dimension of the output feature map as (C, H, W) (channel, height, and width, respectively), the computation of a convolution layer under this fault model could be written as

$$\begin{aligned} y_i &= f(W_i \circledast x_i + b_i + \theta_i \cdot 2^{\alpha_i} \cdot (-1)^{\beta_i}) \\ \theta_i &\sim \text{Bernoulli}(p)^{C\times H\times W} \\ \alpha_i &\sim U\{0, \cdots, Q-1\}^{C\times H\times W} \\ \beta_i &\sim U\{0, 1\}^{C\times H\times W} \end{aligned} \tag{2}$$

where Q represents the fixed-point width, θ_i is the mask indicating whether some fault is triggered at each position, α_i

represents which bit is flipped, β_i represent the flip orientation (positive or negative). Note that we implement bit-flip as a bias added to the feature map for efficient simulation.

Besides random bit-flips, we implement other fault models including Gaussian noise and Salt-and-Pepper noise. Gaussian noise can simulate faults such as thermal noise [13]. Salt-and-Pepper noise is a common noise model in the area of computer vision. We test the defense ability of our models trained under the random bit-flip fault model to all these fault models.

IV. Fault-Tolerant NAS

A. Framework Overview

There are multiple components in our Neural Architecture Searching (NAS) framework: A **controller** that samples different architecture rollouts from the **search space**; A shared weights based **evaluator** that evaluate the performance of different rollouts on the CIFAR10 **dataset**, using fault-tolerant **objectives**. And the shared weights are trained using fault-tolerant training in FTT-NAS. The overall NAS framework is illustrated in Fig 1.

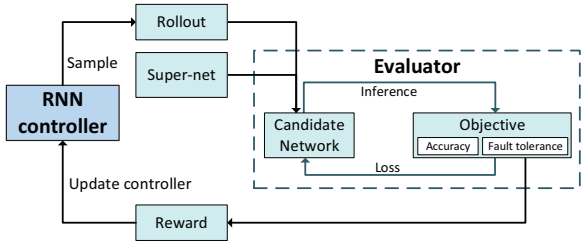


Fig. 1. Illustration of the NAS framework.

B. Search Space

The design of the search space is as follows: We use a cell-based macro architecture, similar to the one used in [23, 24]. There are two types of cells: normal cell, and reduction cell with stride 2. The layout and connections between cells are illustrated in Fig 2. In every cell, there are B nodes, node 1 and 2 are treated as the cell's inputs, which are the outputs of the two previous cells. The following preprocess operations refer to a ReLU-Conv-BN block used to adjust channels. For each of the other $B-2$ nodes, two in-coming connection will be selected and element-wise added, and for each connection, the 7 possible operations are: none; skip connect; 3x3 average (avg.) pool; 3x3 max pool; 1x1 Conv; 3x3 ReLU-Conv-BN block; 5x5 ReLU-Conv-BN block. We selected these operations according to a simple evaluation of hand-designed models.

The complexity of the search space can be estimated. For each cell type, there are $(7^{(B-2)} \times (B-1)!)^2$ possible choices. As there are two independent cell types, there are $(7^{(B-2)} \times (B-1)!)^4$ possible architecture in the search space, which is roughly 6.9×10^{21} with $B = 6$ in our experiments.

C. Sampling and Assembling Architectures

In our experiments, the controller is a recurrent neural network (RNN), and the performance evaluation strategy is based on a super network with shared weights, as used by [23].

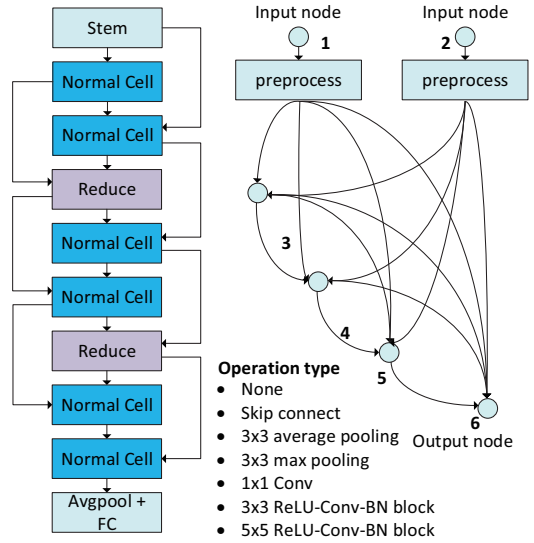


Fig. 2. Illustration of the search space design. Left: The layout and connections between cells. Right: The possible connections in each cell, and the possible operation types on every connection.

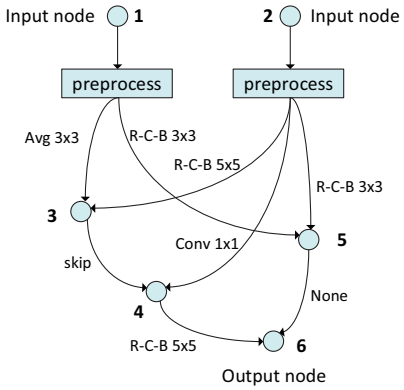


Fig. 3. An example of the sampled cell architecture.

An example of the sampled cell architecture is illustrated in Fig 3. Specifically, to sample a cell architecture, the controller RNN samples $B-2$ blocks of decisions, one for each node $3, \cdots, B$. In the decision block for node i, $M = 2$ input nodes are sampled from $1, \cdots, i-1$, to be connected with node i. Then M operations are sampled from the 7 basic operation primitives, one for each in the M connections. Note that the two sampled input nodes can be the same node j, which will result in two independent connections from node j to node i.

The architecture assembling process using the shared-weights super network is straightforward [23]: Just take out the weights from the super network corresponding to the connections and operation types of the sampled architecture.

D. Searching for Fault-Tolerant Architecture

To search for a fault-tolerant architecture, a weighted sum of the clean accuracy and the accuracy with fault injection is used as the reward to instruct the training of the controller:

$$R = (1 - \alpha_r) * \text{acc}_c + \alpha_r * \text{acc}_f \tag{3}$$

For optimization of the controller, we employ the Adam optimizer [30] to optimize the REINFORCE [31] objective, to-

gether with an entropy encouraging regularization.

In every epoch of the search process, we alternatively train the shared weights in the super network and the controller on separate data splits D_t and D_v, respectively. For the training of the shared weights, we carried out experiments under two different settings: without/with fault-tolerant training. When training with fault-tolerant training, we use a weighted sum of the clean cross entropy loss CE_c and the cross entropy loss with fault injection CE_f to train the shared weights:

$$L = (1 - \alpha_l) * \text{CE}_c + \alpha_l * \text{CE}_f \tag{4}$$

For each step of training the shared weights, we sample architecture a using the current controller parameterized by θ, then use the loss objective (with/without fault injection) to update the parameters.

$$\begin{aligned} &a \sim \pi(a; \theta) \\ &x_t, y_t \sim D_t \\ &w = w - \eta_w \nabla_w L(x_t, y_t, \text{Net}(a; w)) \end{aligned} \tag{5}$$

For each step of training the controller, we sample architecture from the controller, assemble this architecture using the shared weights, get the reward R on one batch of data in the validation data split. Finally, this reward is used to update the controller by applying the REINFORCE technique.

$$\begin{aligned} &x_v, y_v \sim D_v \\ &\text{optimize } E_{a \sim \pi(a;\theta)}[R(x_v, y_v, \text{Net}(a; w))] \end{aligned} \tag{6}$$

The result algorithm without FTT is called FT-NAS, and the one with FTT is called FTT-NAS.

V. Experiments

A. Setup

Our experiments are carried out on CIFAR-10 [32] dataset. CIFAR-10 is one of the most commonly used computer vision datasets and contains 60000 32×32 RGB images. Three hand-designed architecture VGG-16, ResNet-20 and MobileNet-V2 are chosen as the baselines. 8-bit quantization is used throughout the search and training process.

For the neural architecture searching process, we split the training dataset into two subsets. 80% of the training data is used to train the shared weights and the remaining 20% is used to train the controller. The super network is an 8-cell network, with all the possible connections and operations. The channel number of the first cell is set to 20 during the search process, the channel number increases by 2 upon every reduction cell. The controller network is an RNN with one hidden layer of size 100, the learning rate of training the controller is 1e-3. The reward baseline is updated using moving average with momentum 0.99. To encourage exploration, we add an entropy encouraging regularization to the controller's REINFORCE objective, with a coefficient 0.01. For training the shared weights, we use an SGD optimizer with momentum 0.9 and weight decay 1e-4, the learning rate is scheduled by a cosine annealing scheduler [33], started from 0.01. Note that all these settings are typical settings similar to [23].

B. Results of FT-NAS and FTT-NAS

As described in Sec IV, we conduct neural architecture searching without/with fault-tolerant training (i.e. FT-NAS and FTT-NAS, correspondingly). The injection probability p used in the search process is 0.1. The reward coefficients α_r in Eq 3 is set to 0.5. The loss coefficient α_l in FTT-NAS is also 0.5.

As the baselines for FT-NAS and FTT-NAS, we train ResNet-20, VGG-16, MobileNet-V2 with both normal training and fault-tolerant training. For each model trained with FTT, we successively try fault injection probability in $\{10\%, 5\%, 1\%\}$, and use the largest injection probability with which the model could achieve a clean accuracy above 60%.

Table II shows that, basic fault-tolerant training could improve the reliability of the baseline architectures, but will result in a large degradation in the normal accuracy. At various fault ratios, variants of FT-NAS-Net and FTT-NAS-Net outperform the baselines significantly, while keeping the FLOPs and parameter number at the same order of magnitude. Interestingly, the FT-NAS-Net discovered by FT-NAS trained without FTT is much more fault-tolerant than the baseline architectures trained with FTT when the fault ratio $\leq 5\%$.

We apply model augmentation and reduction to the found architectures, to explore the performance of the model at different scales. We can see that, even with much smaller FLOPs and parameter number, FTT-NAS-Net-10 (base of channel number $= 10$) achieves much better accuracy than the baselines, e.g. 80.5% VS. 52.2% (ResNet-20) at fault ratio 5%.

C. Ability to Defense Other Fault Models

To investigate whether the model FTT-trained under the random bit-flip fault model can tolerate other faults, we evaluate the reliability of FTT-NAS-Net under the Gaussian Noise (e.g. thermal noise) and the Salt-and-Pepper fault models (e.g. impulse noise). As shown in Fig 4 (b)(c), models trained under the random bit-flip fault model can defense against other fault models, and FTT-NAS-Net still outperforms all the baseline architectures consistently at all noise levels.

D. RNN Controller v.s. Random Sample

To demonstrate the effectiveness of the learned controller, we random sampled 5 architectures from the search space, and trained them with FTT for 50 epochs, using fault injection probability of 10%.

As shown in Table III, the reliability performance of different architectures in the search space varies a lot, and the architecture sampled by the learned controller FTT-NAS-Net outperforms all the random sampled architectures.

E. Inspection of the Discovered Architecture

The discovered cell architectures are shown in Fig 5. The controller chooses to establish double connections between some pairs of nodes. It seems that the controller identifies the sensitive connections in the neural architecture and then add redundant paths to protect these nodes. Inspired by this

TABLE II
COMPARISON OF DIFFERENT ARCHITECTURES

Arch	Training*	Clean accuracy	Accuracy with random bit-flips (%)						#FLOPs	#Params
			0.5	1	3	5	8	10		
ResNet-20	clean	93.2	84.0	65.4	13.6	10.0	10.0	10.0	1110M	11.16M
VGG-16	clean	91.5	76.7	55.0	18.0	11.2	10.0	10.0	616M	14.65M
MobileNet-V2	clean	92.5	34.1	15.3	10.0	10.0	10.0	10.0	675M	8.06M
FT-NAS-Net	clean	**93.3**	**91.7**	**90.4**	**78.2**	**58.1**	**27.2**	**19.2**	750M	**2.65M**
ResNet-20	5% fault	69.0	69.2	70.3	65.0	52.2	29.8	23.4	1110M	11.16M
VGG-16	5% fault	76.4	75.3	74.3	66.3	52.8	32.0	23.7	616M	14.65M
MobileNet-V2	1% fault	91.4	89.0	84.5	10.1	10.2	10.0	10.0	675M	8.06M
ResNet-M†	5% fault	89.4	88.6	87.8	81.9	72.7	47.1	35.2	862M	8.65M
ResNet-M†	10% fault	67.4	66.8	66.5	66.6	65.2	57.2	50.9	862M	8.65M
FTT-NAS-Net-10‡	10% fault	84.1	83.8	83.7	82.8	80.5	68.4	54.6	**176M**	**0.63M**
FTT-NAS-Net-20‡	10% fault	90.4	90.0	89.7	88.6	86.6	81.0	73.3	704M	5.00M
FTT-NAS-Net-48‡	10% fault	**93.4**	**93.2**	**93.0**	**92.7**	**92.0**	**89.8**	**87.7**	4054M	12.16M

∗: As also noted in the main text, for all the FTT trained models, we successively try fault injection probability in $\{10\%, 5\%, 1\%\}$, and use the largest injection probability with which the model could achieve a clean accuracy above 60%.
†: Described in Sec E.
‡: The "-N" suffix represent the base of the channel number is N.

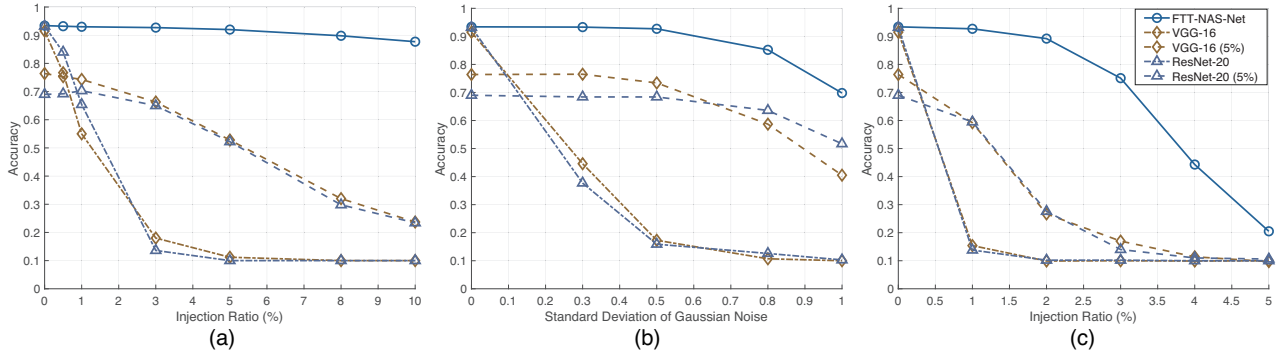


Fig. 4. Accuracy curve under different fault models. Left: Random bit-flips. Medium: Gaussian noise. Right: Salt-and-Pepper noise.

TABLE III
RNN CONTROLLER V.S. RANDOM SAMPLE

	clean acc	10% faults acc	#FLOPs	#Params
sample1	56.2	42.5	210M	0.75M
sample2	20.7	24.0	372M	1.29M
sample3	65.1	45.8	392M	1.40M
sample4	46.8	33.3	290M	1.08M
sample5	38.0	18.7	330M	1.01M
Ours	**86.8**	**74.2**	704M	2.53M

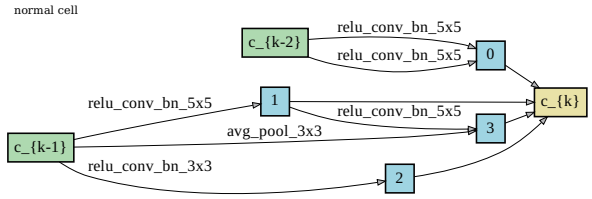
"double connection" structure, we design a modified version of ResNet-20, called ResNet-Modified (abbr. ResNet-M). The basic block of ResNet-M is shown in Fig 6. For keeping roughly the same FLOPs and parameter number as the original ResNet-20, the base of channels is reduced from 64 to 40.

After fault-tolerant training, ResNet-M achieved better performance than ResNet-20, as shown in Table II. The improvement is not only from the redundancy in the forwarding process, but from the architecture too. To verify that, we conduct a simple experiment: For every convolution operation in ResNet-20, we conduct the convolution operation twice, and use the average of the results as the output. This naive "Forwarding Redundancy" trick achieved 57.4% accuracy under 5% random bit-flip, which is slightly better than ResNet-20 but worse than ResNet-M. Moreover, our FTT-NAS-Net is still better than ResNet-M, indicating that FTT-NAS is effective in identifying the sensitive connections in a neural network.

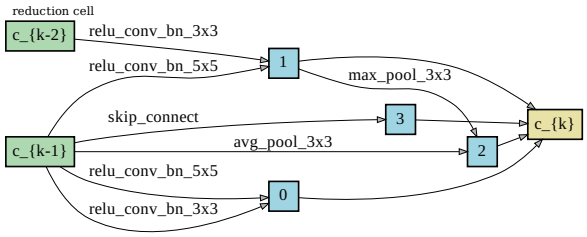
VI. CONCLUSIONS AND FUTURE WORK

In this paper, we propose the multi-objective Fault-Tolerant NAS (FT-NAS) method and the multi-objective Fault-Tolerant Training NAS (FTT-NAS) method, to search for the fault-tolerant convolutional neural network architectures. In FTT-NAS, the Neural Architecture Searching technique (NAS) is employed in conjunction with the Fault-Tolerant Training (FTT). The discovered architectures FT-NAS-Net and FTT-NAS-Net outperform multiple hand-designed architecture baselines in reliability significantly. And the fault tolerance capability of FTT-NAS-Net, trained under the random bit-flip fault model, can defense other fault models. Finally, we draw insights from the discovered structure, which can be used to guide future architecture design. Utilizing these insights, we designed a modified version of the ResNet model, ResNet-M, and verified that the modified ResNet-M can outperform the original ResNet in unreliable settings.

For future work, the transferability of the discovered archi-



(a) Normal Cell



(b) Reduction Cell

Fig. 5. The discovered cell architectures.

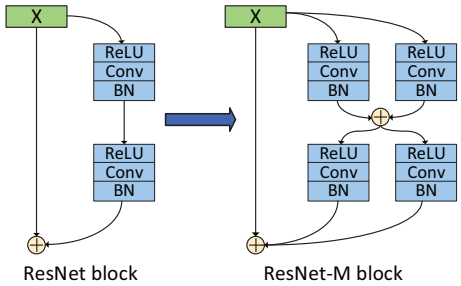


Fig. 6. ResNet-M's modification on ResNet blocks.

tecture FTT-NAS-Net to larger datasets (e.g. ImageNet) should be evaluated. Also, taking the hardware implementation into consideration, more realistic fault model for different types of errors should be formalized. Another interesting direction is to include other hardware-platform-aware objectives in the FTT-NAS framework, to enable discovering fault-tolerant neural network architecture with low latency or energy consumption.

Acknowledgments

The work of Y. Wang and H. Yang was supported in part by National Key R&D Program of China (No. 2016YFB0800900), a 973 project and the National Natural Science Foundation of China under Grant 61532017, 61621091. X. Chen's work was supported by the Beijing Academy of Artificial Intelligence under Grant BAAI2019QN0402. The authors gratefully acknowledge the support from TOYOTA.

References

[1] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *CVPR*, 2016.

[2] W. Liu *et al.*, "Ssd: Single shot multibox detector," in *ECCV*, 2016.

[3] J. Long *et al.*, "Fully convolutional networks for semantic segmentation," in *CVPR*, 2015.

[4] T. Chen *et al.*, "Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning," in *ACM Sigplan Notices*, 2014.

[5] J. Qiu *et al.*, "Going deeper with embedded fpga platform for convolutional neural network," in *FPGA*. ACM, 2016, pp. 26–35.

[6] S. Han *et al.*, "Learning both weights and connections for efficient neural network," in *NIPS*, 2015, pp. 1135–1143.

[7] I. Hubara *et al.*, "Quantized neural networks: Training neural networks with low precision weights and activations," *JMLR*, 2017.

[8] M. Sandler *et al.*, "Mobilenetv2: Inverted residuals and linear bottlenecks," in *CVPR*, 2018, pp. 4510–4520.

[9] J. Henkel *et al.*, "Reliable on-chip systems in the nano-era: Lessons learnt and future trends," in *DAC*, 2013.

[10] J.-C. Vialatte and F. Leduc-Primeau, "A study of deep learning robustness against computation failures," *arXiv:1704.05396*, 2017.

[11] A. Bosio, P. Bernardi, A. Ruospo, and E. Sanchez, "A reliability analysis of a deep neural network," in *LATS*, 2019.

[12] L. Xia *et al.*, "Fault-tolerant training with on-line fault detection for rram-based neural computing systems," in *DAC*, 2017.

[13] Z. He *et al.*, "Noise injection adaption: End-to-end reram crossbar non-ideal effect adaption for neural network mapping," in *DAC*, 2019.

[14] G. B. Hacene *et al.*, "Training modern deep neural networks for memory-fault robustness," in *ISCAS*. IEEE, 2019.

[15] A. P. Arechiga and A. J. Michaels, "The robustness of modern deep learning architectures against single event upset errors," in *HPEC*, 2018.

[16] F. Libano *et al.*, "Selective hardening for neural networks in fpgas," *IEEE Transactions on Nuclear Science*, 2018.

[17] J. Breier *et al.*, "Practical fault attack on deep neural networks," in *CCS*, 2018.

[18] Y. Zhao *et al.*, "Memory trojan attack on neural network accelerators," in *DATE*, 2019.

[19] X. She and N. Li, "Reducing critical configuration bits via partial tmr for seu mitigation in fpgas," *IEEE Transactions on Nuclear Science*, 2017.

[20] A. G. Christoph Schorn and G. Ascheid, "Accurate neuron resilience prediction for a flexible reliability management in neural network accelerators," in *DATE*, 2018.

[21] T. Liu *et al.*, "A fault-tolerant neural network architecture," in *DAC*, 2019, pp. 55:1–55:6.

[22] B. Zoph and Q. Le, "Neural architecture search with reinforcement learning," *ICLR*, 2017.

[23] H. Pham *et al.*, "Efficient neural architecture search via parameter sharing," in *ICML*, 2018.

[24] H. Liu, K. Simonyan, and Y. Yang, "Darts: Differentiable architecture search," *arXiv preprint arXiv:1806.09055*, 2018.

[25] B. Wu *et al.*, "Fbnet: Hardware-aware efficient convnet design via differentiable neural architecture search," *arXiv preprint arXiv:1812.03443*, 2018.

[26] E. Real, A. Aggarwal, Y. Huang, and Q. Le, "Aging evolution for image classifier architecture search," in *AAAI*, 2019.

[27] T. Elsken, J. H. Metzen, and F. Hutter, "Efficient multi-objective neural architecture search via lamarckian evolution," *ICLR*, 2019.

[28] B. Baker *et al.*, "Accelerating neural architecture search using performance prediction," *arXiv preprint arXiv:1705.10823*, 2017.

[29] M. Tan *et al.*, "Mnasnet: Platform-aware neural architecture search for mobile," in *CVPR*, 2019, pp. 2820–2828.

[30] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *ICLR*, 2015.

[31] R. J. Williams, "Simple statistical gradient-following algorithms for connectionist reinforcement learning," *Machine learning*, 1992.

[32] A. Krizhevsky and G. Hinton, "Learning multiple layers of features from tiny images," 2009.

[33] I. Loshchilov and F. Hutter, "Sgdr: Stochastic gradient descent with warm restarts," *ICLR*, 2017.