Future Generation Computer Systems 132 (2022) 124–135

Contents lists available at ScienceDirect

Future Generation Computer Systems

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

Quantune: Post-training quantization of convolutional neural networks using extreme gradient boosting for fast deployment☆

Jemin Lee *, Misun Yu, Yongin Kwon, Taeho Kim

Artificial Intelligence Research Laboratory, Electronics and Telecommunications Research Institute (ETRI), Daejeon 34129, South Korea

ARTICLE INFO

Article history:
Received 27 June 2021
Received in revised form 26 December 2021
Accepted 10 February 2022
Available online 17 February 2022

Keywords:
Quantization
Neural networks
Model compression
Deep learning compiler

ABSTRACT

To adopt convolutional neural networks (CNN) for a range of resource-constrained targets, it is necessary to compress the CNN models by performing quantization, whereby precision representation is converted to a lower bit representation. To overcome problems such as sensitivity of the training dataset, high computational requirements, and large time consumption, post-training quantization methods that do not require retraining have been proposed. In addition, to compensate for the accuracy drop without retraining, previous studies on post-training quantization have proposed several complementary methods: calibration, schemes, clipping, granularity, and mixed-precision. To generate a quantized model with minimal error, it is necessary to study all possible combinations of the methods because each of them is complementary and the CNN models have different characteristics. However, an exhaustive or a heuristic search is either too time-consuming or suboptimal. To overcome this challenge, we propose an auto-tuner known as Quantune, which builds a gradient tree boosting model to accelerate the search for the configurations of quantization and reduce the quantization error. We evaluate and compare Quantune with the *random*, *grid*, and *genetic* algorithms. The experimental results show that Quantune reduces the search time for quantization by approximately 36.5× with an accuracy loss of 0.07–0.65% across six CNN models, including the fragile ones (MobileNet, SqueezeNet, and ShuffleNet). To support multiple targets and adopt continuously evolving quantization works, Quantune is implemented on a full-fledged compiler for deep learning as an open-sourced project.

© 2022 The Author(s). Published by Elsevier B.V. This is an open access article under the CC BY-NC-ND license (http://creativecommons.org/licenses/by-nc-nd/4.0/).

1. Introduction

In spite of the prevalence of convolutional neural networks (CNNs), the high computational requirements restrict their use in resource-constrained devices [1]. To address this challenge, considerable research is being done on the compression of neural networks. One of the promising compression methods is quantization, which converts high-precision representations (32-bit floating point) into lower bit representations (`int8` fixed point). Quantization can reduce the model size of the CNN, memory footprint, and energy consumption and improve the inference time by utilizing special instructions supported by the hardware platforms.

To compensate for the accuracy drop of the quantized models, most of the quantization methods consider retraining [2–9]. However, this method, which is commonly referred to as quantization-aware training (hereinafter referred to as QAT), is not widely adopted in real-world scenarios because of the following issues. First, a full-size dataset is often unavailable owing to privacy concerns or because it is proprietary information. Second, the retraining process in QAT is time-consuming and resource-hungry because of the long periods of tuning. Third, its hyper-parameter tuning is complicated because it requires considerable expertise to develop the architecture of CNNs. Such limitations prevent us from deploying quantized models in a timely manner.

In practice, post-training quantization (hereinafter referred to as PTQ) methods are widely utilized owing to their good applicability [2,10–17]. To recover the accuracy drop, previous studies on PTQ have proposed the following diverse complementary methods: novel schemes for mapping [2,10], calibration for activation of quantization [2,10], granularity for sharing quantization parameters among tensor elements [2,10], clipping [11–13,15,16], and mixed-precision [17]. In this study, we define *a quantization configuration* as a combination of such complementary methods. Based on our experiments, as shown in Table 1, the quantization configurations vary with the target CNN models to attain optimal results. Nonetheless, possible quantization configurations

☆ This work was supported by Institute of Information & communications Technology Planning & Evaluation (IITP) grant funded by the Korea government(MSIT) (No. 2018-0-00769, Neuromorphic Computing Software Platform for Artificial Intelligence Systems).

* Corresponding author.
E-mail addresses: leejaymin@etri.re.kr (J. Lee), msyu@etri.re.kr (M. Yu), yongin.kwon@etri.re.kr (Y. Kwon), taehokim@etri.re.kr (T. Kim).

https://doi.org/10.1016/j.future.2022.02.005
0167-739X/© 2022 The Author(s). Published by Elsevier B.V. This is an open access article under the CC BY-NC-ND license (http://creativecommons.org/licenses/by-nc-nd/4.0/).

vary depending on the target hardware devices. For example, complicated quantifiers are not tolerated on highly constrained hardware like integer-only accelerators [18,19]. Therefore, it is necessary to perform the configuration search every time depending on the model and the hardware across all the combinations. However, it is daunting to explore all the possible configurations when quantization requests occur.

To overcome this challenge, we propose an auto-tuner for configuration search using a gradient tree boosting model (based on XGBoost [20]) known as Quantune. Quantune generates quantized models without noticeable accuracy degradation and retraining and supports multiple hardware platforms (CPU, GPU, and integer-only accelerator) by implementing them on an open-source deep learning compiler stack. We assume that the features extracted from the CNN models are related to quantization configurations. Based on such an assumption, we build an XGBoost model, considering the features, configurations, and accuracy. We perform our search algorithm to find the quantization configuration. To support multiple targets as a unified quantization model format, we implement Quantune on a compiler stack known as Glow. For a rich configuration, we extend Glow to enable layer-wise mixed precision (*int8* and *fp32*) and integer-only quantization. Quantune enables the generation of quantized models for a range of targets including the CPU, GPU, and integer-only hardware such as an accelerator. For the integer-only hardware, we use Versatile Tensor Accelerator (VTA) [21] as an open-hardware architecture. To support the VTA, the entire inference processes comprise only integer multiplication, addition, and bit-shifting.

We evaluate Quantune, in terms of the efficiency of the proposed search algorithm, accuracy of the quantized models, and end-to-end latency of the embedded CPU, server-side CPU, and GPU. The experimental results show that Quantune is 1.3–36.5 times faster than the *random*, *grid*, and *genetic* algorithms and achieves better quality in terms of quantization for the six models across the CPU, GPU and NPU. To prove the quality of the quantized models, Quantune is compared with mature tools such as TensorRT and TVM, which support the PTQ on the GPU and integer-only accelerator (VTA), respectively. Regarding TensorRT, the experimental results show competitive accuracy of the quantized models in ResNet18, ResNet50, ShuffleNet, and SqueezeNet, and better results in GoogleNet. Quantune precisely quantizes fragile models such as MobileNet, ShuffleNet, and SqueezeNet despite their small representational capacities [22]. In the case of the integer-only accelerator, Quantune achieves a 32.52% improvement in accuracy compared to TVM-VTA [18].

By implementing Quantune on the compiler stack, the quantized models can be performed on the CPU, GPU, and the accelerator. Therefore, the executable binaries for the CPU, GPU, and accelerator are generated from the quantized models. We measure the end-to-end inference time of the quantized models on the edge-side CPU, server-side CPU, and server-side GPU. Quantune achieves speedups of 0.34–1.22, 0.27–2.6, and 0.93–1.57 on ARM-A53, Intel-i7-8700, and NVIDIA 2080ti GPU, respectively, against *fp32* execution. From the experiment, it can be seen that latency is not improved for all the quantized models because the extended compiler does not exploit fast 8-bit multiply-accumulate instruction (ARM-vmlal, Intel VNNI, and NVIDIA-DP4A) provided by hardware vendors while generating the kernel code. This indicates that quantization is not only a method that can reduce the memory footprint as in conventional applications, but also a mandatory step to develop deployable CNN models on the integer-only accelerator. The efficient kernel code generation for the quantized models is an important research direction; however, that is beyond the scope of this paper.

The main contributions of our work are summarized as follows.

- We show that the optimal configurations for quantization are diverse depending on the CNN models. To demonstrate the diversity of the quantization configurations, we conduct entropy analysis. As a result, it is found that the entropy of each complementary method is not the same across all the CNN models. There is no universal configuration that is always applicable regardless of the type of CNN models.
- To efficiently explore all combinations of the quantization model, we propose Quantune, which combines both XGBoost and transfer learning to seek the optimal configuration. Quantune significantly outperforms the *grid*, *random*, and *genetic* algorithms by approximately 36.5× with a 0.07–0.65 accuracy loss across the six CNN models.
- For practical use and as an extension, Quantune was implemented on the open-source compiler stack known as Glow [23], instead of performing a pure algorithm design. The extended Glow provides layer-wise mixed precision and integer-only quantization. Therefore, we generated the binary code of the quantized models for diverse hardware targets ranging from CPU (x86 and ARM) to the integer-only accelerator (VTA). To support the integer-only accelerator, Quantum not only quantized weights, activations, bias, and scales, but also generated a computational graph that is comprised of integer multiplication, addition, and bit shift without any floating-point computation.
- Regarding the quality of the quantized models, Quantune achieves 0.59% better accuracy in GoogleNet slim v4 than TensorRT-7.2.2 on the NVIDIA GPU. Regarding the integer-only quantization, Quantune significantly outperforms the previous result (based on single-scale quantization across the whole layer) by approximately 32.52%. In addition, we directly measure the end-to-end inference time of the quantized models on a real CPU and GPU.

2. Related work

Quantization has attracted significant attention owing to its tangible benefits for model compression. In this section, we categorize previous studies on quantization into post-training quantization and quantization-aware training and describe the novelty of our study in each category by comparing it to the existing tools.

2.1. Quantization-aware training

Quantization-aware training (QAT) methods map high bit precision to low bit precision using training step [2–9]. QAT reduces the accuracy drop from the quantized model by using a retraining procedure that is performed for a few epochs. Owing to retraining, QAT is able to quantize CNN models in low precision representation without noticeable accuracy drop and can even operate at 2 bits. However, QAT has the following limitations: (i) retraining is time-consuming, (ii) the training data are not always accessible by third party services, and (iii) its hyper-parameter tuning is complicated. Even using active and continual learning works does not completely mitigate these limitations [24–27]. Therefore, considering its rapid deployment and practical usage, we focus on post-training quantization that does not require retraining based on the training data.

2.2. Post-training quantization

Post-training quantization (hereafter called PTQ) methods map high precision representation bits to low-precision bits without re-training steps [2,10–17,28,29].

Post-training quantization is widely adopted in practical cases because it is not necessary to access the full training dataset for

Table 1
The best results among all the possible quantized models for six CNN models. Hereafter, we abbreviate MobileNet V2 as "MN", ShuffleNet V1 as "SHN", SqueezeNet V1 as "SQN", GoogleNet Slim V4 as "GN", ResNet18 V1 as "RN18", and ResNet50 V1 as "RN50".

Model name	Precision	# of images for calibration	Granularity	Clipping	Scheme	Accuracy (Error)
MobileNet V2(MN)	int8	1,000	Channel	KL	Asymmetric	71.23(−0.58)%
ShuffleNet V1(SHN)	int8+fp32	1	Channel	Max	Symmetric uint	63.59(−0.37)%
SqueezeNet V1(SQN)	int8	1,000	Channel	KL	Asymmetric	53.15(−0.65)%
GoogleNet Slim V4(GN)	int8	1,000	Tensor	KL	Asymmetric	70.58(+0.19)%
ResNet18 V1(RN18)	int8	1,000	Tensor	KL	Asymmetric	70.25(−0.42)%
ResNet50 V1(RN50)	int8	10,000	Channel	KL	Asymmetric	76.01(−0.07)%

retraining. Post-training quantization remedies the large time-consumption for retraining and the data privacy issue. Therefore, it helps rapid deployment of the CNN models on resource-constrained devices. Typically, PTQ leads to non-trivial accuracy degradation, especially in low precision representations. Owing to the prevalence of *int8* data type support in the many hardware platforms, most of the previous studies on PTQ have focused on `int8` quantization and proposed several methods such as diverse schemes, clippings, and mixed-precision to recover the accuracy drop. However, our experimental result shows that the quantization configurations for the best accuracy are dependent on the CNN models. There is no universal configuration that is always applied to attain the most accurately quantized models. Considering each CNN model, a naive parameter search is time-consuming. We overcome this challenge to seek the best configurations for each CNN model. Empirically, this study is the first work to find the optimal quantization strategy using the machine learning model.

2.3. Deep learning compilers

The increasing demands for efficiency on the deep learning (DL) models has made deep DL compilers prevalent. Deep learning compilers have been proposed in both academia and industry. Most of the DL compilers focus on improving the quantization ability of a single class of hardware platform such as Intel nGraph [30], NVIDIA TensorRT [16], ARM NN,[1] and Xilinx Vitis.[2] Nonetheless, it is difficult to extend such tools to other hardware platforms owing to their proprietaries. On the contrary, there are community-driven DL compilers for multiple hardware platforms: TVM [31] and Glow [23]. Such open-source DL compilers support adequate capabilities to adopt diverse quantization settings on multiple targets. Each of them needs to go through manual search procedures to find the optimal quantization settings because of the lack of an auto-tuner, hindering the rapid deployment of the models. Quantune complements the existing studies on the open DL compilers. Quantune introduces a novel search algorithm that generates the optimal quantized models, considering accuracy. With the integration on Glow, Quantune employs full advantages of its compiler to generate kernel codes on multiple targets.

3. Overview

This section describes the overall procedure for quantization by presenting each module. Fig. 1 shows the overall workflow for the configuration search (Quantune) and code generation of the quantized CNN models (Glow Extension). We implement Quantune onto Glow [23] (an open-source DL compiler). Therefore, our quantization method can support a variety of target devices and bring about rapid deployment. In addition, we release the code of Glow extension as a part of NEST-C[3] and Quantune implemented in R.[4] The two colored components in Fig. 1 represent either the changed or new ones. In the Quantune module part, all the components are fully developed for the configuration search. Considering the Glow, our extension aims to support the integer-only accelerator and provide layer-wise quantization for mixed-precision. The overall process of quantization goes through two phases: calibration and configuration search.

Calibration Phase. The dashed-lines indicate the whole procedure of the calibration phase in Fig. 1. In this phase, the histogram of possible numeric ranges in each layer of the neural network is captured for the activation of the quantization and saved to a calibration cache. First, for the calibration, the Glow compiler takes a pre-trained model and an image as input. The images are selected from the training dataset using the Image Selector. Second, the Glow generates the instrumented codes by moving through the Loader, Graph-IR, and Tensor-IR. Concerning the Graph IR, the Glow performs two kinds of optimizations: target independent and target dependent passes. In Tensor-IR level, the Glow determines a schedule of operators while optimizing the memory usage. Finally, the histogram of the tensor values is generated by observing the execution during the inference to capture the possible numeric ranges of activations in each layer of the neural network.

Search Phase. The solid-lines indicate the whole procedure of the search phase in Fig. 1. In this phase, the quantized models and optimized codes are generated with the configuration. As mentioned earlier, the quantization error varies in a chosen configuration. To quickly find the optimal configuration, Quantune efficiently explores the possible configurations based on the XGBoost models that are trained with the model architecture (e_i), explored configuration (s_i), and measured accuracy (c_i). All the possible configurations are detailed in Section 4. Further details of the search algorithm are described in Section 5.

4. Quantization methodology

In the calibration phase, Quantune collected calibration data and saved them into the calibration cache. The calibration cache containing the distribution of numerical data as a histogram was used to obtain accurate thresholds for each tensor in the original *fp32* model. The quantized models are generated by calculating the scale (a quantization parameter) based on the collected data of the activation tensors. The search space of the possible configuration for quantization is the combination of five complementary methods (calibration, scheme, clipping, granularity, and mixed-precision) that affect the accuracy of the quantized models. Eq. (1) denotes the space of the possible configurations.

$$\begin{aligned} &\textit{Search Space}(96) = \textit{Calibration Cache}(3) \times \\ &\quad \textit{Scheme}(4) \times \textit{Clipping}(2) \\ &\quad \times \textit{Granularity}(2) \times \textit{Mixed Precision}(2) \end{aligned} \tag{1}$$

1 https://github.com/ARM-software/armnn.
2 https://www.xilinx.com/products/design-tools/vitis/vitis-platform.html.
3 https://github.com/etri/nest-compiler.
4 https://github.com/leejaymin/qaunt_xgboost.

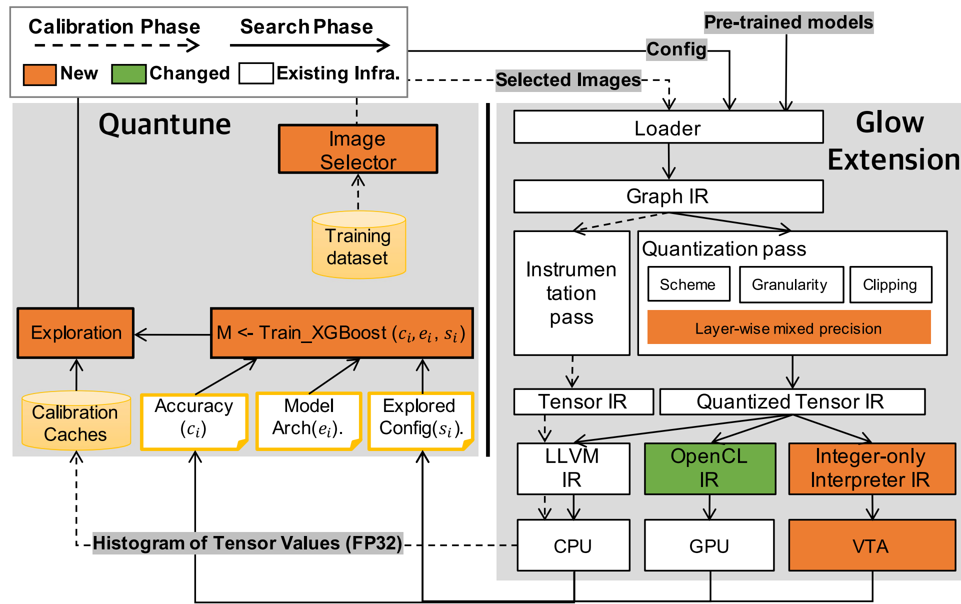


Fig. 1. System overview of the proposed quantization framework. Our framework consists of two modules (Quantune and Glow Extension) and two phases (Calibration and Search).

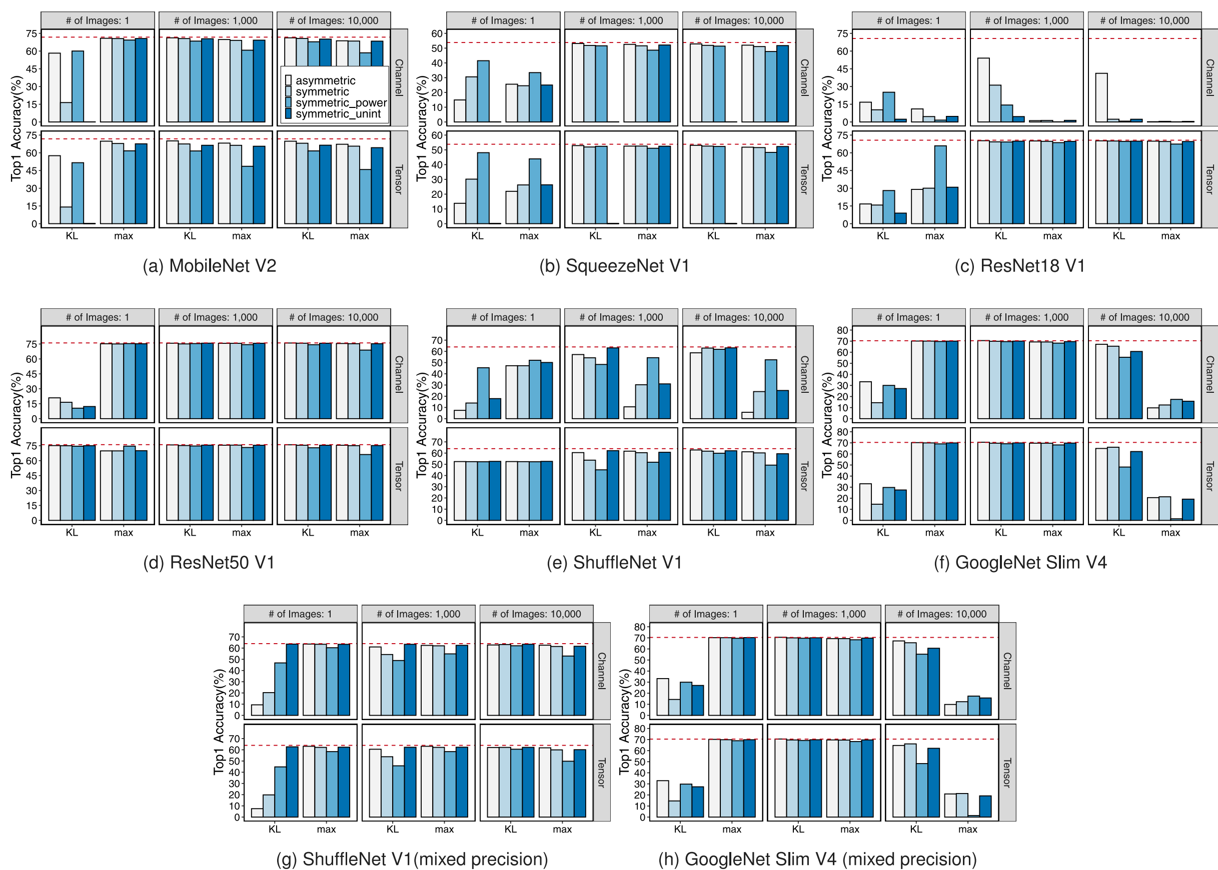


Fig. 2. Accuracy with `int8` quantization of weights and activations. The considered configurations are calibration, schemes, clipping, granularity, and mixed-precision. The dashed-lines indicate full-precision accuracy (`fp32`).

The time cost of the accuracy measurement for a configuration exploration ranges from minutes to hours, depending on hardware platforms. As listed in Table 2, we quantified time costs of measuring accuracy depending on the processors capability of the target devices. Hence, the accuracy measurement across the six models took 0.12–0.58 h on GPU(2080ti), 0.51–12.05 h on CPU(i7-8700), and 10.54–374.15 h on CPU(a53). Even though we restrict the possible search space by applying a configuration to the whole layers, an exhaustive search that covers all the combinations of the configurations may take few days when it

Table 2
Time cost for measuring Top1 accuracy depending on target devices.

Model	Measurement time (h)		
	CPU(a53)	CPU(i7-8700)	GPU(2080ti)
MN	26.14	1.84	0.22
SHN	10.53	0.51	–
SQN	11.56	0.52	0.03
GN	374.15	12.05	0.58
RN18	53.31	2.34	0.12
RN50	126.65	4.87	0.22

is performed on the edge and mobile systems. In the following subsection, we elaborate each quantization configuration.

4.1. Calibration cache

There are many ways to generate a calibration cache. The generated calibration cache depends on the given images during the calibration phase, as shown in Fig. 1. Moreover, the considered images for the calibration phase are selected by the *image selector*. To shirk the search space, the image selector randomly chooses images from the ImageNet training dataset using three parameters: 1, 1000, 10,000. Therefore, three calibration caches were used.

4.2. Scheme

Considering an efficient code generation on multiple hardware platforms, we focus on a uniform integer quantization. Regarding uniform quantization, the schemes to map the real values to the integers are composed of four linear methods: *asymmetric*, *symmetric*, *symmetric with uint8*, and *symmetric power2*.

Asymmetric: This scheme stands for affine. The float range is converted to $[-2^{n-1}, 2^{n-1}-1]$, where *qmin* is -128 and *qmax* is 127. This scheme fully uses the presentation capability of *int8*. The quantizer of this scheme is defined by the following:

$$x_{i8} = Quant(x_{fp32}) = ROUND\left(\frac{x_{fp32}}{scale} + zero\ point\right), \tag{2}$$

where x_{fp32} is a real value (fp32) and x_{i8} is 8 bit-width of a signed integer value. In Eq. (2), *scale* is defined as

$$scale = \frac{max_{fp32} - min_{fp32}}{2^n - 1}, \tag{3}$$

where max_{fp32} and min_{fp32} are maximum and minimum values in the current feature map, respectively. In Eq. (2), *zero point* is defined as:

$$zero\ point = -ROUND(\frac{min_{fp32}}{scale}) - 2^{n-1} \tag{4}$$

The dequantizer of this scheme is defined as follows:

$$x_{fp32} = Dequant(x_{i8}) = scale \cdot x_{i8} - zero\ point \tag{5}$$

Symmetric: The real zero directly maps to the quantized zero. It does not convert the *min* and *max* of the *fp32* range to the quantized range (*int8*). Instead, the absolute maximum value between the min and max of the *fp32* range is used to set the *qmin* and *qmax*. Symmetric scheme is more efficient than that of the asymmetric because it does not use *zero point*. However, the symmetric scheme can result in a severe quantization error when the minimum and maximum values are significantly different. The quantizer of the symmetric scheme is defined as

$$x_{i8} = Quant(x_{fp32}) = ROUND(\frac{x_{fp32}}{scale}) \tag{6}$$

In Eq. (6), *scale* is defined as

$$scale = \frac{MAX(ABS(x_{fp32}))}{2^{(n-1)} - 1}, \tag{7}$$

where $MAX(ABS(x_{fp32}))$ is an absolute maximum value among real values in the current feature map. The dequantizer of this scheme is defined as follows:

$$x_{fp32} = Dequant(x_{i8}) = scale \cdot x_{i8} \tag{8}$$

Symmetric with uint8: This scheme is a combination of asymmetric and symmetric schemes. This scheme adaptively switches the quantization method depending on the distribution of real values. By configuring the offset depending on whether negative values exist, this scheme achieves a computation overhead that is less than or equal to that of asymmetric scheme, and an accuracy higher than or equal to that of symmetric scheme. In this scheme, *zero point* is determined in either zero or -128. The determined *zero point* enables the quantization scheme to be either symmetric(*zero point* $= 0$) or asymmetric (*zero point* $= -128$). In the code level, *uint8* ranges are represented by combining *int8* ranges and the *zero point* of -128. The quantizer of this scheme is defined as follows:

$$x_{i8} = Quant(x_{fp32}) = ROUND(\frac{x_{fp32}}{scale} + zero\ point) \tag{9}$$

In Eq. (9), *scale* is defined as

$$scale = \frac{MAX(ABS(x_{fp32}))}{2^n - 1}, \tag{10}$$

where $MAX(ABS(x_{fp32}))$ is an absolute maximum value among real values in the current feature map. In Eq. (9), *zero point* is defined as

$$zero\ point = \begin{cases} -128, & \text{if } min_{fp32} \geq 0 \\ 0, & \text{otherwise} \end{cases} \tag{11}$$

The dequantizer of this scheme is defined as follows:

$$x_{fp32} = Dequant(x_{i8}) = scale \cdot x_{i8} - zero\ point \tag{12}$$

Symmetric with power of two-scale: This is similar to the symmetric. This scheme represents the quantized ranges by mapping real zero to the quantized zero. In addition to the zero mapping, this scheme converts a real scale to approximately a power of two-scale. By doing that, the power of two-scale quantizer substitutes multiplications by bit-shift operations. The multiplication elimination makes the hardware design simpler and leads to a better performance although it results in a poor representation of the quantized range. In addition, only integer operators are used in the entire inference. For that reason, the quantized model with *symmetric with power two-scale* is deployed on integer-only hardware. The quantization operation is defined by the following:

$$scale = 2^{\left\lceil \log_2 \frac{MAX(ABS(x_{fp32}))}{2^{(n-1)}-1} \right\rceil} \tag{13}$$

In summary, the four schemes constitute a trade-off between inference latency and quantization error. As listed in Table 3, we have classified the advantages and drawbacks into four metrics based on the following questions: (i) How precise is the quantization mapping? (fine-grained mapping); (ii) How well is the skewed distribution handled (robustness to skewness); (iii) How much is the execution overhead of quantization? (low computation); and (iv) Can the quantization computation be solely represented with integer operators? (integer-only hardware). The comparison reveals that the four methods complement each other because each scheme has its advantages and disadvantages; thus, there is no superior among these four schemes.

Table 3
Comparison of quantization schemes. Three symbols ✓, ▲ and, ✗ denote full, partial, and no supports respectively.

Schemes	Fine-grained mapping	Robustness of skewness	Low computation	Integer-only HW
Asymmetric	✓	✓	✗	✗
Symmetric	▲	✗	✓	✗
Symmetric with uint8	▲	▲	▲	✗
Power of two-scale	✗	✗	✓	✓

4.3. Clipping

Without the retraining step, a uniform quantization causes an accuracy drop. Accuracy loss mainly stems from the Gaussian shape of the distributions for weights and activations of the pre-trained neural networks [9,32]. Considering such a characteristic of the distributions, a few weights and activations are sparsely spread as outliers. The outliers in a long tail make a uniform quantizer assign few quantization levels to small values and too many to large ones. This skewness of the distributions leads to significant accuracy degradation [2,13].

To rectify this problem, the Glow compiler allows clipping the range of the distributions for weights and activations. This method chooses a clip threshold which (approximately) minimizes the Kullback–Leibler (KL) divergence between the floating-point and quantized [16].

4.4. Granularity

Considering the quantization, we decided on how far the scale value should be shared among the tensors. We refer to this choice as quantization granularity. We consider two kinds of granularity for the quantization: tensor-wise and channel-wise. Granularity shows the trade-off between accuracy and latency. Fine-grained granularity requires more computation because of the increasing multiplications [17]. Furthermore, a convolution containing a wide range of weight values should consider the channel-wise to compute the scale for quantization. Therefore, granularity is determined by examining the granularity impact on accuracy and latency.

4.5. Mixed-precision

As a part of the extended Glow, we implemented layer-wise mixed precision by extending the Glow compiler. The original Glow compiler does not support the mixed-precision at the layer level. Instead of considering all the layers for the mixed-precision, we only keep the first and last layers of the original precision (*fp32*). This is because an experimental result of a previous study [17] demonstrates that the first and last layers are the most sensitive to the quantization error.

5. Modeling: Parameter search using eXtreme gradient boosting

5.1. Problem definition

The quantization configuration search is performed using historical data previously found in other CNN models regardless of having to search for a new quantization configuration from a random initial point. We hypothesize that the quantization configurations of a CNN model is related to other configurations. Considering the model, a next configuration can be predicted and the result sends feedback to the online training process to update the model.

As described in Eq. (14), our tuning problem can be formulated, considering two kinds of features as demonstrated below:

$$s^*_{opt} = \underset{s \in S_e}{\text{argmax}} \, f(g(e,s)) \tag{14}$$

First is the block expression of the CNN as denoted by e. The CNN block expression e consists of the following operations: the number of layers, convolutions, activation functions, skip-layers, and depth-wise and pointwise convolutions. These kinds of blocks are based on the predefined common structures in the neural architecture search [33,34]. The blocks have been used to reduce the search time and to find a good model in the neural architecture search [33,34].

We generate multiple quantized models that have different accuracies for a given $e \in \mathcal{E}$. We use S_e to denote the space of quantization configurations from e to the quantized models. For instance, if $s \in S_e$ let $x = g(e,s)$ is the generated quantized model, g represents the *Glow extension* that generates the quantized tensor IR from e, s. We aim to maximize $f(x)$, which is the accuracy of the quantized models on the target hardware. Particularly, we verified if an output for $f(x)$ could measure accuracy by running experiments on the hardware. For a given g, e, S_e, f, the accuracy of the quantized models are demonstrated using Eq. (14).

5.2. Auto-tuning algorithm

We propose a machine learning (ML)-based auto-tuning to search the quantization configurations. Fig. 4 shows the components of the auto-tuning and how they interact with one another. To predict the accuracy of quantized model x, the cost model $\hat{f}(x)$ is trained using the historical data. The search engine generates a new quantized model by using the Glow extension and measures its accuracy on the hardware. The accuracy of the quantized model is saved in a database as demonstrated below: $\mathcal{D} = \{(e_i, s_i, c_i)\}$. As shown in Fig. 1, e_i, s_i, and c_i denote the measured accuracy, the extracted model architecture, and the explored configuration, respectively. The collected data could be used to train $\hat{f}$. The following subsections elaborates objective function to train a statistical cost model and the design choices of each component.

5.2.1. Cost model and objective function

XGBoost is an extension and improvement of the gradient tree boosting (GBT) algorithm. The characteristic of XGBoost is scalability, efficiency, sparsity-aware fitting, and well-supported libraries [20]. To this end, the XGBoost is widely adopted to solve real-world problems such as security [27,35], fault detection [36], drug discovery [37], and disease diagnosis [38,39]. In addition to academic fields, XGBoost is the winning algorithm in Kaggle challenges [40].

We selected the XGBoost algorithm to train the data successfully found in the other CNN models and to predict the configuration arising from the most accurate model generation in the next quantization step. With the cost model $\hat{f}(x)$ of XGBoost, we could estimate the accuracy of each quantized model x. Specifically, the cost model $\hat{f}(x)$ of XGBoost represents a tree ensemble model that uses K additive functions defined as

$$\hat{f}(x_i) = \hat{y}_i = \sum_{k=1}^{K} f_k(x_i), f_k \in F \tag{15}$$

For a given quantized model x_i, $\hat{f}(x_i)$ is the predicted accuracy $\hat{y}_i$. The value of the ith instance is $f_k(x_i)$ at the kth tree. Moreover, the values of the space of trees are denoted by the function F.

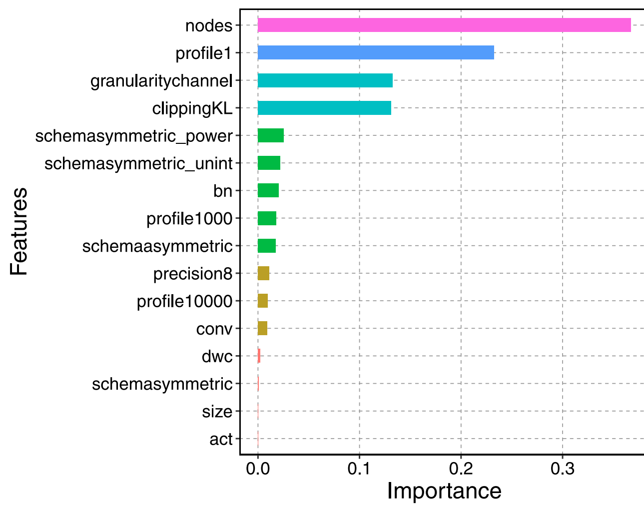


Fig. 3. Ranking of the features used for the modeling.

To train the cost model $\hat{f}(x)$, we follow the regularized objective function and the optimization method introduced in the original paper [20]. The regularized objective function of XGBoost is defined as

$$Obj = \sum_{i=1}^{N} \mathcal{L}(\hat{y}_i, y_i) + \sum_{k=1}^{K} \Omega(f_k) \tag{16}$$

Eq. (16) consists of two parts: the loss function ($\mathcal{L}(\hat{y}_i, y_i)$) and the regularization function ($\Omega(f_k)$). $\mathcal{L}(\hat{y}_i, y_i)$ is a differentiable convex function that computes the difference between the prediction $\hat{y}_i$ and the true label y_i. The differentiable convex functions are mean square error and Logistic loss. The regularization function $\Omega(f_k)$ for the kth tree is then defined as

$$\Omega(f_k) = \gamma T + \frac{1}{2}\lambda \|w\|^2, \tag{17}$$

where T is the number of leaves in a tree, γ lies between 0 and 1 and is multiplied by T to reduce the complexity of each leaf, and λ is a parameter that scales the penalty to avoid the overfitting.

The regularized objective function in Eq. (16) cannot be optimized using traditional methods. Instead, the cost model is trained using an additive method. At the tth step, the $\hat{y}_i^{(t)}$ is optimized as follows:

$$Obj^{(t)} = \sum_{i=1}^{n} \mathcal{L}\left(y_i, \hat{y}_i^{(t-1)} + f_t(x_i)\right) + \Omega(f_t) \tag{18}$$

Each f_t denotes to an independent tree generated by instance i in the tth step. The additive method combines all f_t that maximally improves the objective function in Eq. (16). To reduce the computation cost of the objective function, Eq. (18) is transformed using the second-order Taylor approximation as follows

$$Obj^{(t)} \simeq \sum_{i=1}^{n} \left[\mathcal{L}(y_i, \hat{y}_i^{(t-1)}) + g_i f_t(x_i) + \frac{1}{2} h_i f_t^2(x_i)\right] + \Omega(f_t), \tag{19}$$

where g_i and h_i are defined as

$$\begin{aligned} g_i &= \partial_{\hat{y}_i^{(t-1)}} \mathcal{L}(y_i, \hat{y}_i^{(t-1)}) \\ h_i &= \partial^2_{\hat{y}_i^{(t-1)}} \mathcal{L}(y_i, \hat{y}_i^{(t-1)}) \end{aligned} \tag{20}$$

We could simplify Eq. (20) by removing the constant terms. At step t, the final objective function depends on the first and second-order gradients and is defined as

$$\tilde{Obj}^{(t)} = \sum_{i=1}^{n} \left[g_i f_t(x_i) + \frac{1}{2} h_i f_t^2(x_i)\right] + \Omega(f_t) \tag{21}$$

Finally, with the simplified objective function in Eq. (21), we could iteratively evaluate the model performance after a certain node split in a tree. If the tree model performance is improved after splitting, this change will be accepted; otherwise, the split will be stopped. In this manner, the optimal splitting point for each tree to minimize the objective function is determined; hence, the regularization term remedies overfitting during training. This is the working principle of XGBoost.

Algorithm 1 Quantune: Search for the Optimal Configuration.

Input: Macro-arch. blocks e
Input: Search space S_e
Output: s^*_{opt}
1: $\mathcal{D} \leftarrow \emptyset$ ▷ The collected data will be contained
2: **while** $n_trials < max_n_trials$ **do**
3: $s \leftarrow$ A top candidate in unexplored S_e using $\hat{f}$
4: $c \leftarrow f(g(e, s))$
5: $\mathcal{D} \leftarrow \mathcal{D} \cup \{(e, s, c)\}$
6: update $\hat{f}$ using $\mathcal{D}$
7: $n_trials \leftarrow n_trials + 1$
8: **end while**
9: **return** $s^*_{opt} \leftarrow$ history best quantization config.

5.2.2. Training objective function with hyperparameters

The XGBoost models have hyper-parameters which can be set in order to customize the model for a specific dataset. To train the XGBoost model, we consider several factors including the hyper-parameters, preprocessing, and loss functions.

First, the hyper-parameters were *Eta* and *gamma*. To feed the dataset to the XGBoost model, we considered the preprocessing of two kinds of features: a model-arch (e_i) and a configuration (s_i). Second, the preprocessing can be determined using categorical or one-hot encoding. In this study, we consider one-hot encoding features for the preprocessing because it shows better accuracy than the categorical ones. Third, the possible loss functions for the training are *rank* and *regression*. To apply the *rank* loss function in the training process, it is necessary to add rank the information, and the ranked information can be grouped by a type of the CNN models or whole data. The result of the comparison between the two loss functions shows that regression achieves a better search result with a lower number of trials. Therefore, we consider regression function.

To understand the value of the selected features in the construction of the XGBoost model, we performed the analysis of the feature importance using the XGBoost library in *R*. The significant score of the features is simply calculated using purity (the Gini index) [41]. Fig. 3 shows the result of the importance of the feature. As a result, the number of nodes, calibration (profile), granularity, and clipping is important, considering the predicted accuracy of the XGBoost model.

5.2.3. Search engine

The search engine seeks the optimal configuration of the quantization as described in Algorithm 1. Algorithm 1 takes macro-arch blocks e and search space S_e as its inputs. It produces the optimal configuration for quantization as an output. At each iteration, the engine picks a candidate based on $\hat{f}(x)$ and queries $f(x)$ on the accuracy of quantized model. We enumerate the entire space of S_e and pick the top candidate. The top candidate is not

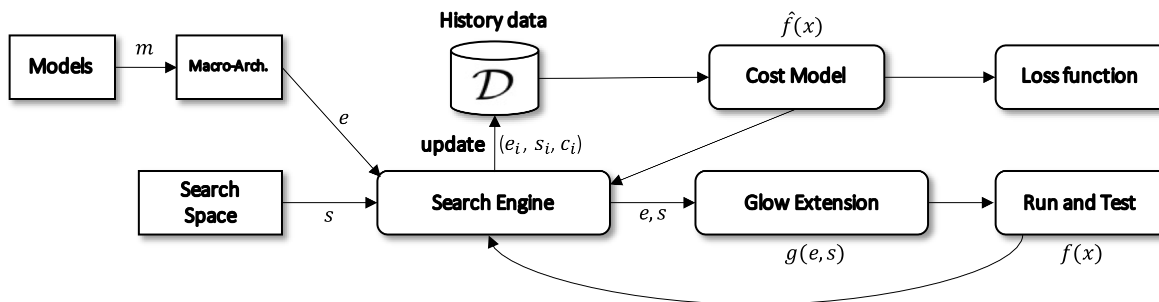


Fig. 4. Auto-tuner to efficiently explore quantization configurations.

explored in the previous step. To accelerate convergence in search step, we apply transfer learning.

The loop iterates over possible optimal configurations. We set the maximum iteration max_n_trials = search space as described in Eq. (1). In Algorithm 1, lines 2–8 describe the steps needed to obtain an optimal configuration from the possible configuration space S_e. At each iteration, the following steps take place sequentially. First, a top candidate is selected using $\hat{f}$, considering diversity. Therefore, we select a candidate from unexplored configurations and try to measure the accuracy of the quantized model on the real target hardware.

6. Evaluation

We conducted three experiments: (1) the variation of accuracy depending on the quantization configurations, (2) effectiveness of the XGBoost based parameter search in search time and accurate model generation, and (3) benefits of the inference time in low precision representation.

6.1. Diversity in quantization configurations

Considering the broad selection of the quantization configurations, it is not obvious to determine which choice mostly reduces the quantization error. To find out the best configuration, we empirically explored various choices for *int8* quantization of the six models. As described in Eq. (1), the search space includes a large number of configurations. We experimentally explored all the combinations that map *fp32* to *int8* to show that there was no universal configuration that was always applied to attain the most accurately quantized models. Fig. 2 shows the Top1 accuracy. The relative error which substrates the quantized accuracy from the baseline ranges from −71.72% to 0.19% across all the combinations. Considering the exploration results of all the configurations, we found the following insights. The accuracy of the quantized models with various configurations varies. Consequently, there is no clear solution to be applied in all the cases. Whether clipping is applied depends on the amount of calibration data. If calibration is performed with a small number of samples, there are relatively few outliers. In this case, the quantization error can be reduced by using full ranges of tensors without clipping. In contrast, clipping reduces the quantization error by increasing the number of sample images during the calibration.

To determine the diversity among the quantization configuration at the accurately quantized models, the results are selected within relative error of 1%. The industrial margin for quantization error is 1% accuracy drop because a de-facto benchmark known as MLPerf [42] allows the quantized model to degrade the accuracy within -1%. Therefore, the following analysis for diversity is based on 1% accuracy drop across all the quantized models.

Table 4
Diversity analysis of quantization configurations. For the analysis, the quantization configurations that achieve accuracy loss within 1% are used.

Precision	Calibration	Granularity	Clipping	Scheme	# of samples
0.50	1.43	0.99	0.98	1.80	71

The Shannon-entropy equation (22) is used to show diversity index in each quantization configuration.

$$H(X) = H(p) = -\sum_{i} p(x_i) \times \log p(x_i) \quad (22)$$

The analysis demonstrates the number of configurations that generate the quantized models that meet the industrial quality and diversity index. Each column in Table 4 shows the diversity index that represents the diversity of the configurations. The diversity indexes for all models within 1% range from 0.50 to 1.80 across the calibration, scheme, clipping, granularity, and mixed-precision. If the entropy is zero, there is no uncertainty. This indicates that there is no obvious configuration to generate the optimal quantized models because all the configurations are not zero entropy. Therefore, it is hard to intuitively and manually select a configuration for the quantization.

6.2. Efficiency of XGBoost based configuration search

From the experiment, the quantization configuration for the most accurately quantized models varies, depending on the type of models. However, the search space for the configuration selection is large as shown in Eq. (1). To reduce the search time, we devise a Quantune that seeks the optimal configuration for the generation of the quantization model based on the XGBoost.

In this section, we show how many trials are required to obtain the optimal quantized model with the optimal configuration. To show the efficiency of the Quantune, we compared it to four algorithms. The *random* search defines a search space as a combination of hyper-parameter values and randomly selects a point in the range. The *grid* search specifies a search space as a grid of the hyper-parameter values and samples of a point in the grid. The *genetic* algorithm (GA) is an optimization method that exploits the idea of evolution by natural selection. GA is based on the hypothesis that a new population will be better than the previous one. To apply genetic algorithm in the quantization configuration search, we implemented the details using the GA package.[5] With the GA package, we defined Top1 accuracy evaluation as the fitness function. Furthermore, we exploited binary encoding for crossover and mutation. In other cases, we used the default settings from the GA package. We apply the XGBoost in two cases: individual learning and transfer learning.

[5] https://cran.r-project.org/web/packages/GA/GA.pdf.

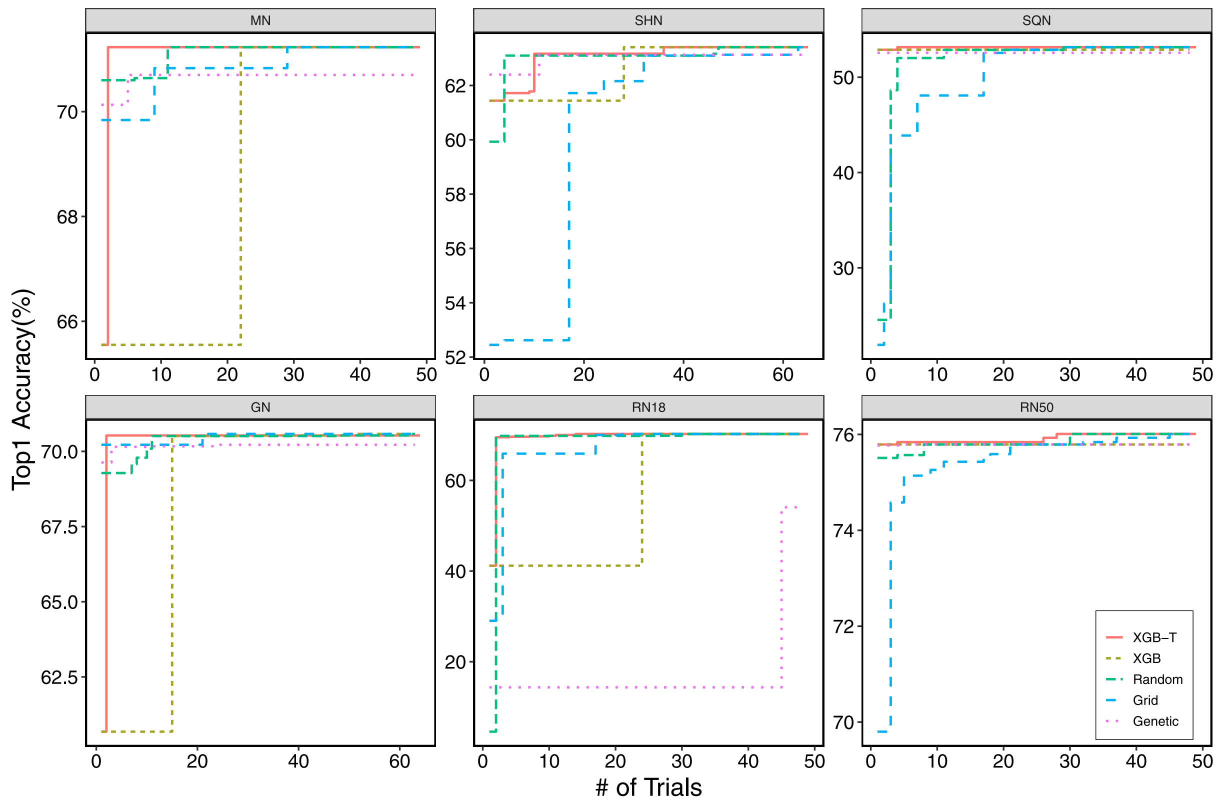


Fig. 5. Comparison of different search algorithms, considering the convergence speed and Top1 accuracy on the six CNN models.

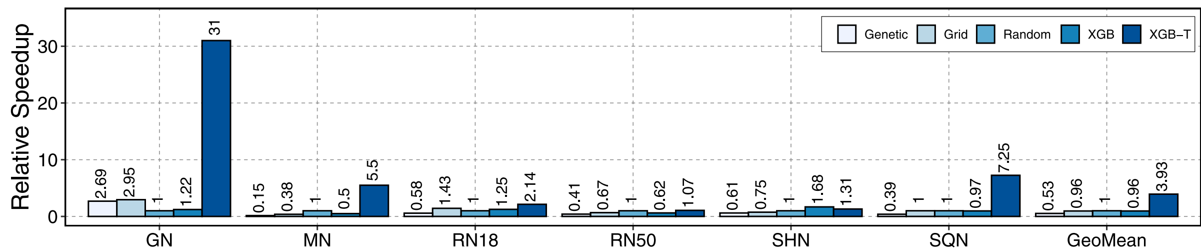


Fig. 6. Relative speedups of convergence over the random search.

The individual XGBoost model is initialized with no historical data that are produced from the other CNN models. It shows that the XGBoost online-learning starts from the base of the current CNN model-tuning. Practically, the Quantune can collect historical data ($\mathcal{D}$) from previous exploration. We combine the XGBoost and transfer learning to effectively use $\mathcal{D}$.

Figs. 5 and 6 show five algorithms that search the quantization configurations of the six CNN models. Our proposed search algorithm (*XGB-T*) is able to guide the configuration search to faster convergence for MN, SHN, SQN, GN, RN18, and RN50 by 5.50×, 1.31×, 7.25×, 31×, 2.14×, and 1.07× for the *random* search and 14.5×, 1.75×, 7.25×, 10.5×, 1.5×, and 1.6× for the *grid* search, respectively. In addition, XGB-T yielded a 2.13× to 36.5× speedup compared with the *genetic* search. All the improvements are due to the cost model and transfer learning. In contrast to XGBoost, the other three algorithms work without cost model. Furthermore, transfer learning allows speedup and attains a higher accuracy. Specifically, transfer learning improves the convergence time of the MN, GN, and RN18 relative to the XGBoost by 11×, 25×, and 1.7×, respectively. Regarding the SQN and RN50, the *XGB-T* achieved +0.27% and +0.22% higher accuracy, respectively, than the *XGB*.

6.3. Accuracy

Through the extended Glow stack, the Quantune supports both of the general-purpose units (CPU or GPU) and the VTA. Considering both targets, we show how accurate the Quantune generates the quantized models compared to the other existing tools.

First, in the general-purpose targets, the Quantune is directly compared to NVIDIA TensorRT7.2.2 (released in 19 Dec. 2020) on the server-side GPU. The TensorRT[6] is a well-supported tool by NVIDIA to speed up an inference of the CNN. For comparison, we develop a test code that generates the quantized models using TensorRT's post-training quantization and releases it as an open source.[7] As shown in Fig. 7, the Quantune achieves 0.59% higher accuracy on GoogleNet slim v4 than the TensorRT. Nevertheless, it attains 0.19–1.36 lower accuracy across the four models than the TensorRT. Considering that the Quantune is a unified open

6 https://developer.nvidia.com/tensorrt.

7 https://github.com/leejaymin/tensorrt_quantization_imagenet.

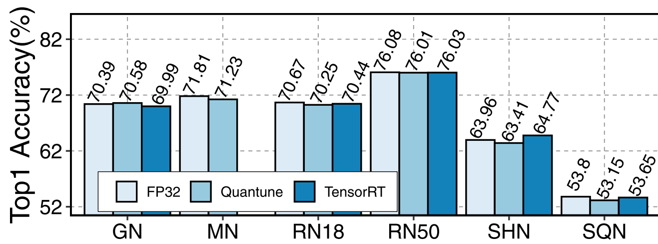


Fig. 7. Quantune vs. TensorRT - Quantune achieves a comparable accuracy of the quantized CNN models against the off-the-shelf compiler (TensorRT) on the NVIDIA GPU.

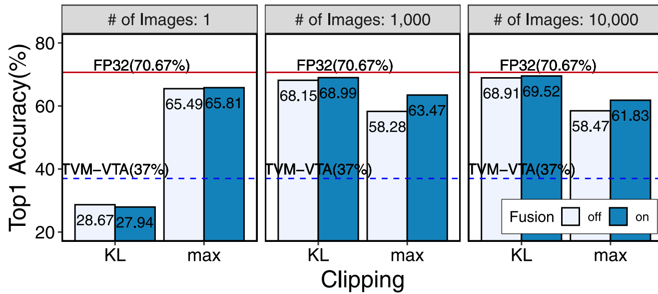


Fig. 8. Quantune vs. TVM-VTA - Quantune leads to the significant improvement in accuracy by approximately 32.52% as against the TVM-VTA [18]. The solid-line indicates full-precision accuracy (*fp32*). The dashed-line indicates the TVM-VTA accuracy.

toolchain to support multiple hardware devices, it shows a competitive accuracy compared to the TensorRT which is used only for the NVIDIA GPUs.

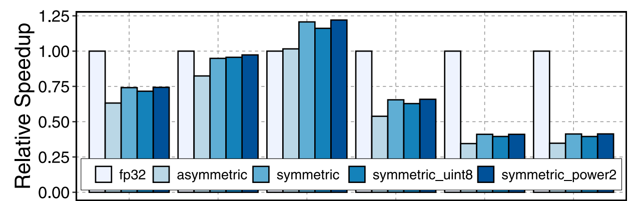
Second, for the VTA, we compare the Quantune to the TVM (released in Jul. 2017) on the VTA. Previous results from the VTA-TVM [18] reported a significant accuracy drop (−33.76%). This drastic accuracy drop in the TVM-VTA stems from the choice of a quantization scale for the whole network because a scale value can be imprecise for small values and truncate large values. Contrary to the VTA-TVM [18], the Quantune selects different scales depending on each layer and traverses all the possible configurations. The search space is different because of the limitations of the VTA hardware. In the VTA, the scheme and granularity only support the power of the two-scale and tensor level. Owing to the architecture support, the fused operator for the convolution and ReLU is executed in consecutive cycles without extra off-chip memory access. There are 12 distinct configurations derived from the following possible combinations.

$$\begin{aligned} &\mathit{Search\ Space}(12) = \mathit{Calibration\ Caches}(3) \times \\ &\quad \mathit{Schemes}(1) \times \mathit{Clipping}(2) \times \\ &\quad \mathit{Granularity}(1) \times \mathit{Fusion}(2) \end{aligned} \tag{23}$$

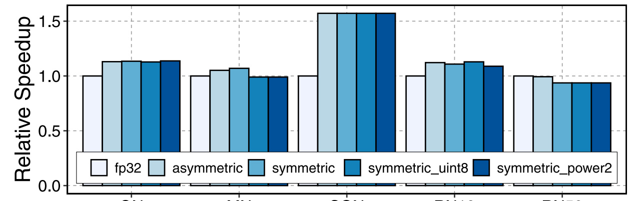
From the results of all configurations, as shown in Fig. 8, Quantune leads to significant improvement in accuracy by approximately 32.52%. We show that even with a limited scheme and granularity, the best result is 0.73% lower than the ones on the general-purpose units.

6.4. Model size

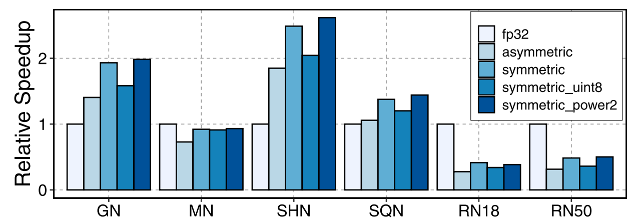
To demonstrate the effectiveness of our model, we measured the size of quantized models depending on the quantization configurations. Among the considered configurations, *granularity* and *mixed-precision* affect model size. As listed in Table 5, the size of quantized models varies for four configurations combined with granularity and mixed-precision. With tensor granularity alone, the compression rate is the highest spanned across six quantized models. In contrast, the combining channel granularity and mixed-precision shows the lowest compression rate. The differences between tensor and channel granularity stem from the number of scale factors for quantization. Intuitively, channel level quantization requires more scale factors than tensor-level. In the case of mixed-precision, the model size is dependent on the number of parameters in the first and last layers.



(a) ARM A53



(b) NVIDIA 2080ti



(c) Intel i7-8700

Fig. 9. Comparison of relative speedup between the original and quantized models on ARM A53, NVIDIA 2080ti, and Intel i7-8700.

Table 5
Comparison of the size of quantized models depending on quantization configuration. Model size means the number of bytes required to save all of the weights in the CNN model.

Model	Original	Tensor	Channel	Tensor+Mixed	Channel+Mixed
MN	13.96 MB	3.54 MB	3.70 MB	7.39 MB	7.54 MB
SHN	5.52 MB	1.42 MB	1.54 MB	3.06 MB	3.17 MB
SQN	4.94 MB	1.25 MB	1.29 MB	1.25 MB	1.29 MB
GN	170.60 MB	42.75 MB	43.01 MB	47.36 MB	47.63 MB
RN18	47.94 MB	12.91 MB	12.95 MB	14.47 MB	14.51 MB
RN50	102.12 MB	25.61 MB	25.84 MB	31.79 MB	32.01 MB

6.5. Latency

Our evaluation has focused on the accuracy of the quantized models. In this section, we measured end-to-end inference time using the Glow's code generation (referred to as *CodeGen*) on a server class GPU (an embedded CPU) and a desktop CPU for both the floating point and quantized models. As shown in Fig. 9, the inference time of all the quantized models is not improved against the *fp32* models. This is because the *CodeGen* in the

Glow is not efficiently implemented at the *int8* quantization level for chosen target platforms. Previous studies have reported that the naive implemented kernels for quantization can be slower than the original models [10,22,43,44]. As mentioned in Table 3, the four schemes are a trade-off between inference latency and quantization error. Therefore, the latency of the quantized models varies for different schemes. In the naive implementation, the increasing latency stems from extra operations such as the scales and offsets in each layer of the quantization. In some cases like ShuffleNet, the improvements of latency are attributed to better cache reuse by reducing the weights and activations.

In a nutshell, the quantization method in DL frameworks preserves the accuracy of the quantized models; however, it is insufficient to provide latency improvement on diverse hardware devices. To overcome this challenge, few studies have been proposed [10,22,43,44]. An efficient kernel code generation for quantized models is an important research direction that is beyond the scope of our study.

7. Conclusion

To enable rapid deployment of quantized models without noticeable accuracy loss, we present Quantune, which can build a model to find optimal configurations for quantization, thereby accelerating the search speed and enhancing the accuracy of the quantized models. The experimental results on real hardware devices show that Quantune achieved 1.3–31 faster convergence time than the three algorithms with 0.07–0.65% accuracy drop across the six CNN models including MobileNet, ShuffleNet, and SqueezeNet, which are light-weight networks. Moreover, Quantune achieved 0.59% higher accuracy for GoogleNet than TensorRT on the NVIDIA GPU. Furthermore, Quantune led to a significant improvement in accuracy by achieving an improvement of 32.52% on the accelerator compared with the published study on VTA. Finally, by extending the DL compiler stack, we reduced the effort needed to efficiently execute the quantized CNN models on diverse hardware devices.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

This work was supported by Institute of Information & communications Technology Planning & Evaluation (IITP) grant funded by the Korea government(MSIT) (No. 2018-0-00769, Neuromorphic Computing Software Platform for Artificial Intelligence Systems).

References

[1] M. Astrid, S.-I. Lee, Deep compression of convolutional neural networks with low-rank approximation, ETRI Journal 40 (4) (2018) 421–434.
[2] R. Krishnamoorthi, Quantizing deep convolutional networks for efficient inference: A whitepaper, 2018, arXiv preprint arXiv:1806.08342.
[3] S.K. Esser, J.L. McKinstry, D. Bablani, R. Appuswamy, D.S. Modha, Learned step size quantization, in: 8th International Conference on Learning Representations (ICLR) 2020, OpenReview.net, 2020, pp. 1–12.
[4] J. Choi, Z. Wang, S. Venkataramani, P.I.-J. Chuang, V. Srinivasan, K. Gopalakrishnan, Pact: Parameterized clipping activation for quantized neural networks, in: 6th International Conference on Learning Representations (ICLR) 2018, OpenReview.net, 2018, https://openreview.net/forum?id=ryQu7f-RZ.
[5] D. Zhang, J. Yang, D. Ye, G. Hua, LQ-Nets: Learned quantization for highly accurate and compact deep neural networks, in: Proceedings of the European Conference on Computer Vision (ECCV), 2018, 2018, pp. 365–382.
[6] S. Jung, C. Son, S. Lee, J. Son, J.-J. Han, Y. Kwak, S.J. Hwang, C. Choi, Learning to quantize deep networks by optimizing quantization intervals with task loss, in Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR), 2019, pp. 4350–4359.
[7] S. Zhou, Y. Wu, Z. Ni, X. Zhou, H. Wen, Y. Zou, Dorefa-net: Training low bitwidth convolutional neural networks with low bitwidth gradients, 2016, arXiv preprint arXiv:1606.06160.
[8] B. Jacob, S. Kligys, B. Chen, M. Zhu, M. Tang, A. Howard, H. Adam, D. Kalenichenko, Quantization and training of neural networks for efficient integer-arithmetic-only inference, in Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, 2018, pp. 2704–2713.
[9] S. Han, H. Mao, W.J. Dally, Deep compression: compressing deep neural network with pruning, trained quantization and huffman coding, in: 4th International Conference on Learning Representations (ICLR), 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings, 2016, URL: http://arxiv.org/abs/1510.00149.
[10] Z. Jiang, A. Jain, A. Liu, J. Fromm, C. Ma, T. Chen, L. Ceze, Automated backend-aware post-training quantization, 2021, arXiv preprint arXiv:2103.14949.
[11] R. Banner, Y. Nahshan, D. Soudry, Post training 4-bit quantization of convolutional networks for rapid-deployment, in: Advances in Neural Information Processing Systems, Vol. 32, Curran Associates, Inc., 2019.
[12] Y. Choukroun, E. Kravchik, F. Yang, P. Kisilev, Low-bit quantization of neural networks for efficient inference, in: ICCV Workshops, 2019, pp. 3009–3018.
[13] R. Zhao, Y. Hu, J. Dotzel, C. De Sa, Z. Zhang, Improving neural network quantization without retraining using outlier channel splitting, in: International Conference on Machine Learning, PMLR, 2019, pp. 7543–7552.
[14] J.H. Lee, S. Ha, S. Choi, W.-J. Lee, S. Lee, Quantization for rapid deployment of deep neural networks, 2018, arXiv preprint arXiv:1810.05488.
[15] A. Goncharenko, A. Denisov, S. Alyamkin, E. Terentev, Fast adjustable threshold for uniform neural network quantization, Int. J. Comput. Inf. Eng. 13 (9) (2019) 495–499.
[16] S. Migacz, 8-Bit inference with tensorrt, in: GPU Technology Conference, Vol. 2, 2017, p. 5.
[17] H. Wu, P. Judd, X. Zhang, M. Isaev, P. Micikevicius, Integer quantization for deep learning inference: Principles and empirical evaluation, 2020, arXiv preprint arXiv:2004.09602.
[18] T. Moreau, T. Chen, L. Ceze, Leveraging the vta-tvm hardware-software stack for fpga acceleration of 8-bit resnet-18 inference, in Proceedings of the 1st on Reproducible Quality-Efficient Systems Tournament on Co-designing Pareto-efficient Deep Learning, 2018, p. 1.
[19] X. Zhao, Y. Wang, X. Cai, C. Liu, L. Zhang, Linear symmetric quantization of neural networks for low-precision integer hardware, in: International Conference on Learning Representations (ICLR), 2019.
[20] T. Chen, C. Guestrin, Xgboost: A scalable tree boosting system, in Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, 2016, pp. 785–794.
[21] T. Moreau, T. Chen, L. Vega, J. Roesch, E. Yan, L. Zheng, J. Fromm, Z. Jiang, L. Ceze, C. Guestrin, et al., A hardware–software blueprint for flexible deep learning specialization, IEEE Micro 39 (5) (2019) 8–16.
[22] B. Jacob, S. Kligys, B. Chen, M. Zhu, M. Tang, A. Howard, H. Adam, D. Kalenichenko, Quantization and training of neural networks for efficient integer-arithmetic-only inference, in Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, 2018, pp. 2704–2713.
[23] N. Rotem, J. Fix, S. Abdulrasool, G. Catron, S. Deng, R. Dzhabarov, N. Gibson, J. Hegeman, M. Lele, R. Levenstein, et al., Glow: Graph lowering compiler techniques for neural networks, 2018, arXiv preprint arXiv:1805.00907.
[24] C.V. Nguyen, Y. Li, T.D. Bui, R.E. Turner, Variational continual learning, 2017, arXiv preprint arXiv:1710.10628.
[25] A.D. Doulamis, N.D. Doulamis, S.D. Kollias, On-line retrainable neural networks: improving the performance of neural networks in image analysis problems, IEEE Trans. Neural Netw. 11 (1) (2000) 137–155.
[26] S. Shin, Y. Boo, W. Sung, Fixed-point optimization of deep neural networks with adaptive step size retraining, in: 2017 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP), IEEE, 2017, pp. 1203–1207.
[27] S.S. Dhaliwal, A.-A. Nahid, R. Abbas, Effective intrusion detection system using XGBoost, Information 9 (7) (2018) 149.
[28] M. Nagel, M.v. Baalen, T. Blankevoort, M. Welling, Data-free quantization through weight equalization and bias correction, in Proceedings of the IEEE/CVF International Conference on Computer Vision, 2019, pp. 1325–1334.
[29] E. Meller, A. Finkelstein, U. Almog, M. Grobman, Same, same but different: recovering neural network quantization error through weight factorization, in: Proceedings of the 36th International Conference on Machine Learning, in: Proceedings of Machine Learning Research, vol. 97, PMLR, 2019, pp. 4486–4495.

[30] S. Cyphers, A.K. Bansal, A. Bhiwandiwalla, J. Bobba, M. Brookhart, A. Chakraborty, W. Constable, C. Convey, L. Cook, O. Kanawi, et al., Intel ngraph: An intermediate representation, compiler, and executor for deep learning, 2018, arXiv preprint arXiv:1801.08058.

[31] T. Chen, T. Moreau, Z. Jiang, L. Zheng, E. Yan, H. Shen, M. Cowan, L. Wang, Y. Hu, L. Ceze, et al., {TVM}: An automated end-to-end optimizing compiler for deep learning, in: 13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18), 2018, pp. 578–594.

[32] D. Lin, S. Talathi, S. Annapureddy, Fixed point quantization of deep convolutional networks, in: International Conference on Machine Learning, PMLR, 2016, pp. 2849–2858.

[33] B. Wu, X. Dai, P. Zhang, Y. Wang, F. Sun, Y. Wu, Y. Tian, P. Vajda, Y. Jia, K. Keutzer, Fbnet: Hardware-aware efficient convnet design via differentiable neural architecture search, in Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition, 2019, pp. 10734–10742.

[34] A. Wan, X. Dai, P. Zhang, Z. He, Y. Tian, S. Xie, B. Wu, M. Yu, T. Xu, K. Chen, et al., Fbnetv2: Differentiable neural architecture search for spatial and channel dimensions, in Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition, 2020, pp. 12965–12974.

[35] Z. Chen, F. Jiang, Y. Cheng, X. Gu, W. Liu, J. Peng, XGBoost classifier for DDoS attack detection and analysis in SDN-based cloud, in: 2018 IEEE International Conference on Big Data and Smart Computing (bigcomp), IEEE, 2018, pp. 251–256.

[36] D. Zhang, L. Qian, B. Mao, C. Huang, B. Huang, Y. Si, A data-driven design for fault detection of wind turbines using random forests and XGboost, IEEE Access 6 (2018) 21020–21031.

[37] X. Ji, W. Tong, Z. Liu, T. Shi, Five-feature model for developing the classifier for synergistic vs. Antagonistic drug combinations built by XGBoost, Front. Genetics 10 (2019) 600.

[38] A. Ogunleye, Q.-G. Wang, Xgboost model for chronic kidney disease diagnosis, IEEE/ACM Trans. Comput. Biol. Bioinform. 17 (6) (2019) 2131–2140.

[39] K. Budholiya, S.K. Shrivastava, V. Sharma, An optimized XGBoost based diagnostic system for effective prediction of heart disease, J. King Saud Univ.-Comput. Inf. Sci. (2020).

[40] O. Sagi, L. Rokach, Ensemble learning: A survey, Wiley Interdiscip. Rev. Data Min. Knowl. Discov. 8 (4) (2018) 1249.

[41] T. Hastie, R. Tibshirani, J. Friedman, The Elements of Statistical Learning: Data Mining, Inference, and Prediction, Springer Science & Business Media, 2009.

[42] V.J. Reddi, C. Cheng, D. Kanter, P. Mattson, G. Schmuelling, C.-J. Wu, B. Anderson, M. Breughe, M. Charlebois, W. Chou, et al., Mlperf inference benchmark, in: 2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA), IEEE, 2020, pp. 446–459.

[43] M. Cowan, T. Moreau, T. Chen, J. Bornholt, L. Ceze, Automatic generation of high-performance quantized machine learning kernels, in Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization, 2020, pp. 305–316.

[44] A. Jain, S. Bhattacharya, M. Masuda, V. Sharma, Y. Wang, Efficient execution of quantized deep learning models: A compiler approach, 2020, arXiv preprint arXiv:2006.10226.

Jemin Lee received the B.S. and Ph.D. degrees in computer science and engineering from Chungnam National University, in 2011 and 2017, respectively. He is a senior researcher at the Electronics and Communications Research Institute (ETRI). He was a post-doctoral researcher at the Korea Advanced Institute of Science and Technology (KAIST), in 2017–2018. His research interests include energy-aware mobile computing and deep learning compiler.

Misun Yu received the M.S. degree from the Department of Computer Science and Engineering at Pohang University of Science and Technology, Rep. of Korea. She is a principal researcher at the Electronics and Communications Research Institute (ETRI), Daejeon, Rep. of Korea. Her main research interests include concurrent program analysis, software testing, deep learning, and embedded systems.

Yongin Kwon received the B.Sc. degree in electrical and electronic engineering from the Korea Advanced Institute of Science and Technology (KAIST), South Korea, in 2008, and the M.S. and Ph.D. degrees in electrical and computer engineering from Seoul National University, South Korea, in 2010 and 2015, respectively. He is currently a senior researcher at the Electronics and Communications Research Institute (ETRI). His research interests include mobile cloud computing, compiler, deep learning, and embedded systems.

Taeho Kim received the B.S. degree from Sungkyunkwan University in 1995 and the M.S. and Ph.D. degrees from the Department of Computer Science, Korea Advanced Institute of Science and Technology (KAIST), in 1997 and 2005, respectively. He is currently Assistant Vice President of AI SoC Research Division, Electronics and Telecommunications Research Institute (ETRI). His research interests are safety-critical and intelligent cyber–physical systems, system software, and software engineering.