Master's Thesis

An Auto-tuner for Quantizing Deep Neural Networks

심층신경망 양자화를 위한 자동 튜너

February 2019

Graduate School of Seoul National University

Computer Science and Engineering

QUAN QUAN

An Auto-tuner for Quantizing Deep Neural Networks

심층신경망 양자화를 위한 자동 튜너

Advisor Jae W. Lee

Submitting a master's thesis of engineering

January 2019

Graduate School of Seoul National University

Computer Science and Engineering

QUAN QUAN

Confirming the master's thesis written by QUAN QUAN

December 2018

Chair	**Jihong Kim**	**(Seal)**
Vice Chair	**Jae W. Lee**	**(Seal)**
Examiner	**Jaejin Lee**	**(Seal)**

Abstract

An Auto-tuner for Quantizing Deep Neural Networks

QUAN QUAN
Department of Computer Science and Engineering
The Graduate School
Seoul National University

With the proliferation of AI-based applications and services, there are strong demands for efficient processing of deep neural networks (DNNs). DNNs are known to be both compute- and memory-intensive as they require a tremendous amount of computation and large memory space. Quantization is a popular technique to boost efficiency of DNNs by representing a number with fewer bits, hence reducing both computational strength and memory footprint. However, it is a difficult task to find an optimal number representation for a DNN due to a combinatorial explosion in feasible number representations with varying bit widths, which is only exacerbated by layer-wise optimization. To address this, an automatic tuner is proposed in this work for DNN quantization. Here, the auto-tuner can efficiently find a compact representation (type, bit width, and bias) for the number that minimizes the user-supplied objective function, while satisfying the accuracy constraint. The evaluation using eleven DNN models on two DNN frameworks targeting an FPGA platform and a bit-serial hardware, demonstrates over 8× (7×) reduction in the parameter size on average when up to 7% (1%) loss of relative accuracy is tolerable, with a maximum reduction of 16×, compared to the baseline using 32-bit floating-point numbers.

Keywords: Auto-tuner, Quantization, Deep neural network, Optimization
Student Number: 2017-21772

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Deep neural networks (DNNs) are the key enabler for emerging AI-based applications and services. For example, convolutional neural networks (CNNs) are widely deployed for computer vision applications, such as image classification [20, 28] and feature detection [46, 54]. For natural language processing, recurrent neural networks (RNNs) [34, 45] and memory networks [49, 55] are used for question answering [44], voice recognition [10], and image captioning [37]. Recently, Deep Q-Network (DQN) has been proposed to combine DNNs with reinforcement learning to achieve near- or super-human performance in playing Atari games [14].

Although these networks can perform complicated tasks, they require a tremendous amount of computation. For example, ResNet-152 [20], one of the deepest CNNs, requires 22.6 giga-operations (GOPs) to process one image in ImageNet [8]. They also require large memory space up to hundreds of megabytes to store network parameters alone [17]. Thus, there are many proposals to exploit algorithmic characteristics and data locality to reduce both computational intensity and memory footprint [25].

Quantization is a popular method to achieve this by using a reduced-precision format to represent layer inputs, weights, or both. Representing a number with fewer bits accelerates computation and reduces memory footprint, hence improving overall energy efficiency. Since DNNs often have a lot of redundancy in network parameters, an 8-bit fixed-point type, instead of the 32-bit full-precision floating-point type, may be sufficient to execute them without noticeable degradation of the output quality [17, 31]. Log2 quantization [35, 50] represents a value with its exponent only (in power of two) to provide a very

large dynamic range for a given number of bits and reduce the computational strength by substituting a floating-point multiply with an integer add. Due to intrinsic trade-offs between accuracy and performance, it is crucial to find an optimal quantization scheme for a given DNN.

However, finding suitable number representations for a given network is difficult due to a combinatorial explosion in feasible number representations with varying bit widths. Furthermore, an optimal representation may vary layer by layer as different layers have different degrees of performance and accuracy sensitivity to representational changes [24]. Existing quantization frameworks for DNNs have various limitations as they fail to support multiple types [6, 11, 23, 31, 35, 38, 56], layer-wise optimization [17, 38], and are bound to a specific hardware and DNN framework [11, 38, 52, 56] (e.g., Caffe on GPU [17, 57]). Besides, the optimization goal is often hard-coded in the tuning algorithm, thus failing to accommodate various deployment scenarios with different accuracy-performance trade-offs.

To address this challenge, this thesis proposes an auto-tuner for quantizing DNNs. Auto-tuner allows the user to specify her optimization goals for a variety of DNN frameworks and hardware platforms. First, applied the Number abstract data type (ADT) in *libnumber* [36] to the DNN frameworks, which subsumes most the popular number representations for DNNs. It encapsulates the internal representation of a number from the user and the user can declare those data she wants to quantize in a layer (e.g., inputs, weights, or both) as `Number` type. Then the auto-tuner takes inputs from the user (for target layers, accuracy constraints, and optimization goals) and the platform engineer (for a list of supported types by the target platform) to find a suitable (type, bit width, bias) tuple for each layer efficiently, while satisfying the accuracy constraints. Using auto-tuner we separate the concern of developing an effective DNN model from the concern of the low-level optimization of number representation.

To evaluate auto-tuner we use nine CNNs and two multilayer perceptrons (MLPs) on two DNN frameworks, Caffe [21] and DarkNet [42], targeting an FPGA platform and bit-serial hardware. Unlike CPU and GPU, whose supported number types are already fixed, FPGA can flexibly accommodate a va-

riety of types and bit widths, and bit-serial hardware whose latency for MAC operation scales linearly to the number of bits, to make them ideal platforms to demonstrate the effectiveness of auto-tuner. According to our evaluation, auto-tuner reduces the size of the network parameters by 8.28× for the eleven models on average with a maximum reduction of 16.00×. While the size of the valid configuration space ranges from 7.40×10^4 to 1.65×10^{29}, the proposed auto-tuner finds a compact representation after navigating only through a tiny fraction of the configuration space (an order of 10^3 configurations in the worst case).

In summary, auto-tuner satisfies all of the following design goals:

- *High coverage* - It supports multiple data types (floating-point, fixed-point, and exponent in particular) with varying bit widths, biases, and layer-wise optimization to maximize bit savings.
- *Search efficiency* - The auto-tuning algorithm efficiently navigates through huge configuration space with a two-pass search to find an effective representation quickly.
- *Ease of use* - The auto-tuner offers a simple Python-based interface for easy reconfiguration.

The rest of thesis is organized as follows. Chapter 2 provides preliminaries for DNNs and motivates this work. Chapter 3 overviews the tuner working flow, followed by the details of API design of auto-tuner (Chapter 4). Chapter 5 presents the evaluation methodology and the results. At last, we briefly survey related work to ours in Chapter 6 and conclude the thesis in Chapter 7.

Chapter 2

Motivation

2.1 Redundancy in Deep Neural Networks

Figure 2.1 shows the structure of Cuda-convnet, which is used for image classification [26]. It takes as input an image file of 32×32 pixels with three input channels (for RGB color components) from CIFAR-10 dataset [27]. The image passes through a pipeline of kernels (layers), composed of three alternating pairs of convolution and pooling layers, followed by a fully-connected (FC) layer at the end. Each layer performs distinct matrix computations to extract more complex features toward the end of the pipeline. Finally, the FC layer is a classifier layer, which computes a vector of the probability for each of the 10 classes. In CNNs the convolution layers are known to be the most compute-intensive (consuming over 90% of total GOPs), while the FC layers account for a disproportionately large share of network parameters due to their full connectivity [26].

Figure 2.2 is a sequential C reference code for the FC layer. This kernel takes a three-dimensional matrix representing 64 4×4 feature maps as input to generate a one-dimensional probability vector for the 10 classes. The operations being performed are multiplying an input activation (`input[j][k][l]`) by a weight (`filter[i][j][k][l]`) and accumulating it into the output vector (`output[i]`). This multiply-accumulate (MAC) computation is the most common operations not only for the FC layers but also for the convolution layers.

However, it is well known that the network parameters in DNNs have a significant amount of redundancy [19] and we can improve computational effi-

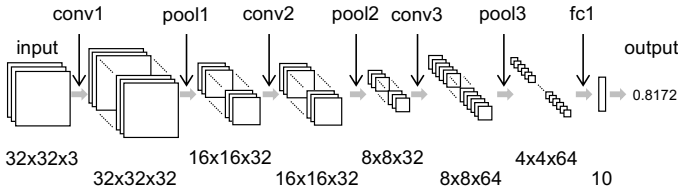


Figure 2.1: Structure of Cuda-convnet

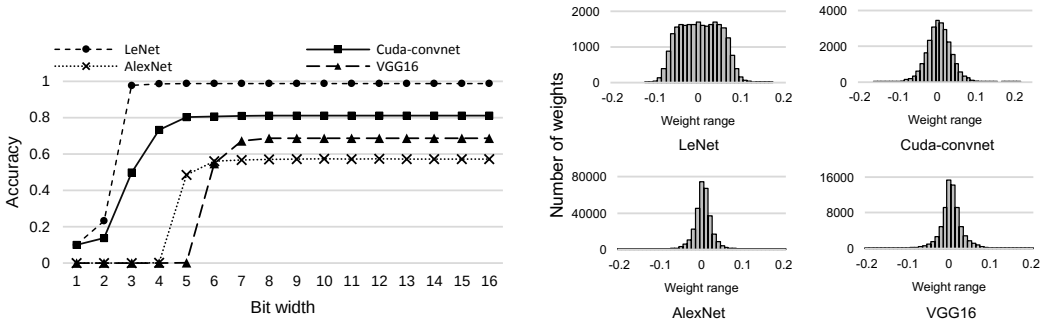
```
float output[out_c];
float input[in_c][h][w];
float filter[out_c][in_c][h][w];

// Computation of fully connected layer
for (int i=0; i<out_c; ++i) {
  for (int j=0; j<in_c; ++j) {
    for (int k=0; k<h; ++k) {
      for (int l=0; l<w; ++l) {
        output[i]+=filter[i][j][k][l]*
                   input[j][k][l];
}}}}
```

Figure 2.2: FC layer reference code (simplified)

ciency by eliminating it. Figure 2.3(a) shows the top-1 accuracy of four popular CNNs with varying fixed-point bit widths from 1 through 16. All of the four CNNs fully recover their accuracy by using 8 or more bits. Moreover, LeNet requires only 3 bits to achieve the full accuracy.

Thus, there are ample opportunities for quantization to compress the network parameters and boost computational efficiency by using lower-precision arithmetic. Figure 2.3(b) shows the distribution of the weights for the four CNNs. The distribution features a relatively short dynamic range of the values. If a wide dynamic range is unnecessary, simpler (and faster) fixed-point computation can replace expensive full-precision floating-point computation. To



(a) Top-1 accuracy-bit width

(b) Distribution of weight parameters

Figure 2.3: Redundancy in CNN weight parameters

exploit these opportunities on a GPU, Nvidia has recently introduced mixed-precision support for their state-of-the-art CUDA 9 [12]. Microsoft's BrainWave deploys a custom 9-bit floating-point format, called *ms-fp9*, to achieve a sub-millisecond latency of a DNN inference task on their FPGA fabric [3].

There are two approaches to DNN quantization: training a quantized network from scratch [6, 7] and applying quantization to a pre-trained network (optionally followed by re-training to recover accuracy) [18, 19]. Although reported some success, training a quantized network is still a difficult problem and demonstrated to work only for relatively small networks [6, 7]. In contrast, the second approach is more attractive to many practitioners with lots of pre-trained models already available in the public domain [18]. Thus, we assume the second approach (i.e., quantizing a pre-trained model) for our work.

2.2 Optimizing Number Representations

Table 2.1 surveys popular number representations used by DNNs. Floating-point, fixed-point, and exponent are the three most widely used types for quantization. (The binary type is a special case of any of the three.) We observe that all of the three types can be represented by the canonical number format based on IEEE 754 Standard as shown in Figure 2.4. The canonical format

Format	Number Type	Bit Width	Network	Quantized Layer	Canonical Form $\langle n_s, n_e, n_f, B \rangle$
FLOAT32	floating-point	32	Default for all DNNs	CONV, FC	$\langle 1, 8, 23, 127 \rangle$
Half-precision		16	CNN, LSTM, DCGAN, DeepSpeech2 [12]	CONV, FC	$\langle 1, 5, 10, 15 \rangle$
MS-fp9, BFP		9	DNNs on Brainwave [5, 16]	CONV, FC	$\langle 1, 5, 3, - \rangle$
FIXED16	fixed-point	16	CaffeNet, VGG16, VGG16-SVD [38]	CONV, FC	$\langle 1, 0, 15, 0 \rangle$
FIXED11		10	CNN [6]	CONV	$\langle 1, 0, 10, - \rangle$
FIXED8		8	MLP, CNN, LSTM [11]	CONV, FC	$\langle 1, 0, 7, - \rangle$
LogQuant4	exponent	4	AlexNet, VGG16 [35]	CONV, FC	$\langle 1, 3, 0, 3 \rangle$
BIN	binary	1	BinarizedNN [7]	CONV, FC	$\langle 1, 0, 0, 0 \rangle$

Table 2.1: Survey of popular number representations for DNNs (− means not disclosed.)

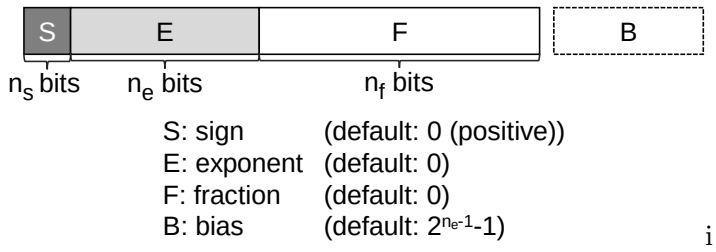


Figure 2.4: Canonical number format based on the IEEE 754 standard

consists of four fields: sign (S), exponent (E), fraction (F) and bias (B) [1]. If the lengths of the S, E, and F fields are denoted by n_s, n_e, and n_f, respectively, a number format can be characterized by a 4-tuple of $\langle n_s, n_e, n_f, B \rangle$. Assuming $E = \sum_{i=0}^{n_e-1} 2^i \cdot e_i$, $F = \sum_{i=0}^{n_f-1} 2^i \cdot f_i$,, where e_i and f_i denote the i-th bit of E and F, respectively, the number can be interpreted as follows:

$$\begin{cases} (-1)^S (1 + F \cdot 2^{-n_f})\ 2^{E-B} & if\ E \neq 0 \\ (-1)^S F \cdot 2^{-n_f} \cdot 2^{1-B} & if\ E = 0 \end{cases} \tag{2.1}$$

More specifically, the three popular number types are mapped to the the

[1] Throughout this paper a *bias* refers to a constant subtracted from an exponent to form a biased exponent in compliance with the IEEE 754 standard. It is different from a constant input (bias) to an artificial neuron.

canonical format as follows. (1) N-bit *floating-point* (`FLOAT(N)`) is represented by a 4-tuple of $\langle 1,\ n_e,\ n_f,\ B\rangle$, where $N = 1 + n_e + n_f$ and $B = 2^{n_e-1}$-1. For example, single-precision (`FLOAT`(32)) and half-precision (`FLOAT`(16)) types correspond to $\langle 1,\ 8,\ 23,\ 127\rangle$ and $\langle 1,\ 5,\ 10,\ 15\rangle$, respectively. (2) N-bit *dynamic fixed-point* (`FIXED(N)`) is represented by a 4-tuple of $\langle 1,\ 0,\ n_f,\ B\rangle$, where B is used to control the position of the radix point (zero by default). Unlike the floating-point type, the fixed-point type has a narrow dynamic range, but is easier to implement using integer arithmetic. (3) N-bit *exponent* (`EXP(N)`) is represented by a 4-tuple of $\langle 1,\ n_e,\ 0,\ B\rangle$, where $B = 2^{n_e-1}$-1 by default. The exponent type has the widest dynamic range at the cost of reduced precision.

In this setup, the task of optimizing number representations is reduced to finding an optimal set of the four parameters (n_s, n_e, n_f, and B) in the 4-tuple. However, this task easily becomes intractable due to a combinatorial explosion in feasible number formats. Suppose an FPGA device that can accommodate the three number types: floating-point (32 and 16 bits), fixed-point (1 through 32 bits), and exponent (2 through 9 bits). For example, if we are to find an optimal representation out of the 42 formats for both activations and weights, and each of the nine convolution layers in DarkNet [42], the search space is as large as $(42^2)^9 = 1.65\times10^{29}$.

To avoid this cost, prior proposals for quantization sacrifice coverage of the search space by either tuning the bit width only for a fixed number type [6, 11, 23, 31, 35, 38, 56] or turning off layer-wise optimization [17, 38]. This can lead to a suboptimal result.

Thus, we need a tuner for DNN quantiozation with broad coverage in optimization space. The auto-tuner should find a suitable number format efficiently for each target layer. Besides, it is highly desirable for the framework to provide an easy-to-use API that can flexibly support a variety of DNN frameworks for tuning.

Chapter 3

Overview

As we mentioned in Chapter 1, it is crucial to find an suitable number representation for a given network. Quantization strategy may vary across different underlying hardwares and different layers have different degrees of performance and accuracy sensitivity to representational changes. Besides, the problem becomes much more tough to be solved with the support of various number types and bit widths. Though, many tuning algorithms are proposed, most of them have framework dependencies. Thus, we proposed an auto-tuner for quantizing DNNs, which has no dependency on any framework. It efficiently navigates through a tiny fraction of the search space to find a compact representation (type, bit width, and bias) for the number, while satisfying the accuracy constraint. In this chapter, we shows the overall operation flow of proposed auto-tuner .

To run the tuner, several inputs should be provided. A pretrained model such as LeNet, an objective function (optimization goal), like minimizing total storage size, and an error margin relative to the full precision accuracy, which are all supplied by DNN developers. And the supported formats (type, bit width range pairs), which is provided by platform developers. With these information, the tuner automatically outputs an optimized configuration for a given network model by applying quantization to target layers as identified by the DNN developer. Figure 3.1 shows the overall operation flow and the tuning algorithm is presented in greater details in Chapter 4. The rest of this section sketches a flow of operations to quantize the activations and weights of each layer with auto-tuner.

Step 1 (Initializing the Auto-tuner). At the core of flow is the auto-

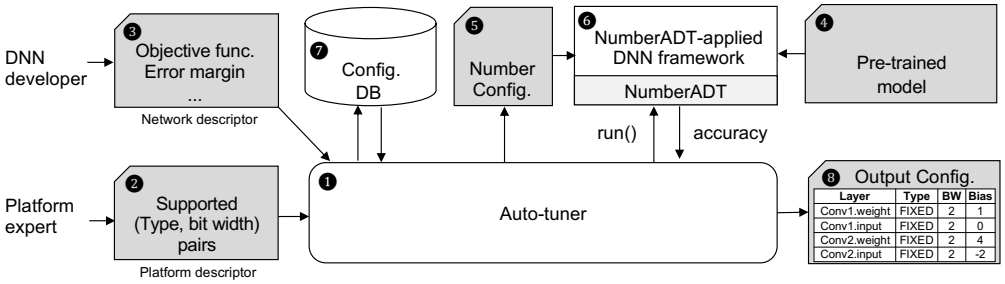


Figure 3.1: Overall operation flow of the tuning a DNN.

tuner ❶ around which multiple components interact with each other. It can be flexibly configured using two descriptor files written in Python. The first one is a *platform descriptor* ❷ containing a key-value pair, where the key is a hardware device name and the value is a list of supported number formats. The second one is a *network descriptor* ❸, which specifies the optimization goal (i.e., accuracy tolerance, objective function) and a list of target layers among others. Using these information the auto-tuner determines search space, and enters the tuning loop to find a configuration that *minimizes* the objective function, denoted by f_{obj}, subject to satisfying the accuracy constraint.

Step 2 (Entering the Tuning Loop). Once initialization is completed, the auto-tuner selects the first configuration to try, and generates a Number configuration file ❺ in text format. This file contains a list of object name-number format pairs (e.g., `conv1.weight FLOAT 32`, `fc1.weight EXP 8 127`). Then the auto-tuner launches an inference job by invoking the NumberADT-applied DNN framework ❻ in which data to be quantized (e.g., layer inputs, weights, or both) are declared as `Number` type. At runtime all `Number` objects in the DNN framework are bound to concrete number formats to quantize pre-trained model ❹ as specified by the Number configuration file.

Step 3 (Assessing the Configuration). When the DNN inference is finished, the auto-tuner first updates the configuration database ❼ with accuracy and the value of $f_{obj}(cfg)$ for the current configuration `cfg`. If a configuration that minimizes f_{obj} is found, the auto-tuner reports the configuration in a plain text file (`opt_conf.txt`) ❽ and exits the loop. Otherwise, it selects the next candidate configuration and repeats the process in Step 2. The auto-tuning

algorithm is presented in greater details in Section 4.2.

Step 4 (Generating DNN Kernels). Once tuner gives the optimal output configuration, generates optimized DNN kernels by applying quantization to the target layers.

Chapter 4

Auto-tuner

4.1 Configuring the Auto-tuner

The auto-tuner takes two descriptor files to configure it: platform and network descriptors. The platform descriptor specifies a `supported_formats` attribute as shown in Figure 4.1. It contains a key-value pair, where the key is a device name and the value is a list of supported number formats. There can be multiple platform descriptor files to support multiple hardware devices. Besides, one can specify different supported types for activations and weights by appending `.activation` and `.weight` suffix to the device name.

The network descriptor file specifies various network parameters as summarized in Table 4.1. We need two command lines (`cmd_smallset` and `cmd_fullset`) to invoke the DNN framework with small and full test sets, respectively, as the tuning algorithm in Section 4.2 uses both. The user specifies an accuracy constraint and optimization goal by setting `err_margin` and `f_obj`. At this point we only use the default objective function, which is a sum of bit width-layer size products for all target layers, to minimize overall storage requirements for activations and weights. However, one can override this function to customize the optimization goal. A regular expression is provided by `get_result_regex`

```
supported_formats = {
    'FPGA':{                    # Platform name
    'FLOAT': [16, 32],          # FLOAT 16 / 32 bits
    'FIXED': range(2, 33),      # FIXED 2 to 32 bits
    'EXP'  : range(2, 10)}}     # EXP   2 to 9  bits
```

Figure 4.1: Example platform descriptor file

Platform descriptor		
Parameter	**Description**	**Example**
supported_formats	Key-value pair of device name and list of supported number formats	Refer to Figure 4.1
Network descriptor		
Parameter	**Description**	**Example**
platform	Device name	"FPGA"
cmd_smallset	Command line to run small test set	"./caffe test -model LeNet.prototxt -weight Lenet.caffemodel -iterations 10"
cmd_fullset	Command line to run full test set	"./caffe test -model LeNet.prototxt -weight Lenet.caffemodel -iterations 100"
err_margin	Error margin relative to baseline accuracy	0.07
f_obj	Objective function to specify optimization goal	"Default" (sum(bit_width[i]*layer_size[i]))
output_type	Specify output type [acc|err]	"acc"
layer_name	Name of target layers in Python list format	["conv1", "conv2"]
sub_layer_name	Name of branches in same layer in Python list format	[[],["branch1", "branch2"...]...]
layerwise_opt	Enable layer-wise optimization [`BOTH`|`WEIGHT-ONLY`|`ACTIVATION-ONLY`|`NONE`]	BOTH
wgt_size	Size of target layers (weight count) in Python list format	[500, 25000]
act_size	Size of target layers (activation count) in Python list format	[784, 2880]
act_max_abs	Maximum absolute values of activations for target layers in Python list format	[0.996094, 5.05664]
wgt_max_abs	Maximum absolute values of weights for target layers in Python list format	[0.5794976, 0.16474459]
get_result_regex	Regular expression to extract accuracy from standard output	"Accuracy : (.*)\n"

Table 4.1: Auto-tuner configuration parameters

to extract the accuracy number from the console output at the end of execution of every run in the tuning loop. The user can also control layer-wise optimization for both activations and weights by setting the `layerwise_opt` field. The remaining parameters are self-explanatory, except that `sub_layer_name` is used to group multiple sublayers using the same format. This feature is useful for optimizing networks with branches such as ResNet [20] and GoogleNet [13].

4.2 Tuning Algorithm

To achieve high quality and efficiency at the same time, the auto-tuner takes a two-pass approach, where each pass consists of two phases. During the first pass, the auto-tuner uses a small test set (specified by `cmd_smallset`) to reduce not only the number of iterations but also the cost of each iteration. Using a smaller dataset can yield an order of magnitude reduction in execution time, hence leading to significant savings in total search time, especially for large

Algorithm 1 Auto-tuning algorithm

Input: Error margin err_margin
Output: Best configuration best_cfg

```
 1: init_cfg ← InitializedConfig()
 2:
 3: test_set ← SmallSet
 4: threshold = GetAccuracy(init_cfg) × (1 - err_margin) // small set
 5:
 6: /* Phase 1 (small set): Coarse-grained search */
 7: cfg ← init_cfg
 8: while GetAccuracy(cfg) ≥ threshold do
 9:     cfg ← CoarseGrainedSearch(cfg)
10: end while
11:
12: /* Phase 2 (small set): Fine-tuning bias, bit width and type */
13: cfg ← FinetuneBiasBitWidthAndType(cfg)
14:
15: test_set ← FullSet
16: threshold = GetAccuracy(init_cfg)×(1 - err_margin) // full set
17:
18: /* Phase 3 (full set): Accuracy recovery */
19: if GetAccuracy(cfg) < threshold then
20:     cfg_set ← GetConfigsWithOneMoreBit(cfg) ∪
21:                 GetConfigsWithDecrementedBias(cfg)
22:     while IsAllBelowThreshold(cfg_set) do
23:         cfg ← SelectBestConfig(cfg_set)
24:         cfg_set ← GetConfigsWithOneMoreBit(cfg) ∪
25:                     GetConfigsWithDecrementedBias(cfg)
26:     end while
27: end if
28:
29: /* Phase 4 (full set): Fine-tuning bias, bit width and type */
30: best_cfg ← FinetuneBiasBitWidthAndType(cfg)
```

and deep networks. Starting from the best configuration of the first-pass, the second pass further tunes (type, bit width, bias) tuples using the full test set (specified by `cmd_fullset`). The rest of this section presents the details of the four phases described in Algorithm 1 and 2 using a real example of auto-tuning LeNet in Figure 4.2.

Phase 1 (Coarse-grained Search). The first phase of the algorithm is described by Line 1-10 in Algorithm 1. This phase performs a coarse-grained search to reduce the bit width starting from the baseline configuration of using `FLOAT(32)` for all layers (Line 1) and a small test set (Line 3). The coarse-

Algorithm 2 Fine-tuning bias, bit width and type

```
Input: Configuration cfg
Output: Best configuration best_cfg
 1: while True do
 2:     cfg_set ← GetConfigsWithIncrementedBias(cfg)
 3:     while ¬ IsAllBelowThreshold(cfg_set) do
 4:         cfg ← SelectBestConfig(cfg_set)
 5:         cfg_set ← GetConfigsWithIncrementedBias(cfg)
 6:     end while
 7:     cfg_set ← GetConfigsWithOneFewerBit(cfg)
 8:     while ¬ IsAllBelowThreshold(cfg_set) do
 9:         cfg ← SelectBestConfig(cfg_set)
10:         cfg_set ← GetConfigsWithOneFewerBit(cfg)
11:     end while
12:     cfg_set ← GetConfigsWithTypeChange(cfg)
13:     if IsAllBelowThreshold(cfg_set) then
14:         break
15:     else
16:         cfg ← SelectBestConfig(cfg_set)
17:     end if
18: end while
19: return cfg
```

grained search procedure (`CoarseGrainedSearch` in Line 9) reduces the bit width by half as long as the accuracy constraint is satisfied. Once the search hits the minimum bit width for `FLOAT`, it switches to the `FIXED` type and continues. If it hits a configuration that violates the accuracy constraint, it selects the configuration in the middle of the last two configurations to perform a binary search. Iterations 1-7 in Figure 4.2 illustrate this phase when tuning the two convolution layers (L1 and L2) of LeNet. As a result, 3-bit fixed-point (I3) is selected as best configuration thus far (Iteration 6). The procedure also attempts to further reduce the bit width using the `EXP` type but fails (Iteration 7). Note that the biases surrounded by parentheses are initialized based on value profiling, as summarized by `act_max_abs` and `wgt_max_abs` parameters in Table 4.1.

Phase 2 (Fine-tuning Bias, Bit Width and Type). The second phase performs a fine-tuning of bias, bit width and type as described in Algorithm 2, starting from the best configuration from Phase 1. First, the algorithm attempts to tune bias for fixed-point and exponent types (Line 2-6), which is crucial to balance the precision and dynamic range of a quantized number. The goal is to

minimize the dynamic range for each layer (i.e., maximizing bias), while satisfying the accuracy constraint. Stripes [22] performs similar tuning to minimize the number of integer bits. In Line 2, we first collect a set of all configurations (`cfg_set`) whose bias is incremented by one from the current configuration at a target layer (for either activations or weights. For example, Iterations 8-11 in Figure 4.2 show the four elements in `cfg_set`. Among them the third one (I3(0)-I3(3) for weights and I3(0)-I3(-3) for activations in Iteration 10) is selected due to highest accuracy, where I3(0) denotes 3-bit fixed-point (I) with bias 0. This process is repeated until no configuration satisfies the accuracy constraint.

Second, the algorithm proceeds to reduce bit width (Line 7-11). In Line 7, it first collects a set of all configurations (`cfg_set`) that use one fewer bit than the current configuration for either activations or weights. In Figure 4.2, if the current configuration is I2(1)-I3(3) for weights and I3(0)-I3(-2) for activations (selected at Iteration 25), the three configurations in Iterations 27-28 represent `cfg_set`. Then the configuration that gives the most benefit (i.e., maximizing $\Delta f_{obj}\,(cfg)$) at the minimum cost of accuracy loss (i.e., minimizing $\Delta accuracy\,(cfg)$) is selected. This process is repeated until no configuration satisfies the accuracy constraint (Line 8-11). For example, the minimum bit widths are found at Iteration 29 in Figure 4.2 (I2(1)-I2(3) for weights and I2(0)-I2(-2) for activations).

Finally, we perform fine-tuning of the number type (Line 12-17). In this step, we first collect a set of all configurations (`cfg_set`) that have only one type different from the current configuration. In Figure 4.2, if the current configuration is I2(1)-I2(3) for weights and I2(0)-I2(-2) for activations (from Iteration 29), the configurations in Iterations 30-32 represent `cfg_set`. If no configuration satisfies the accuracy constraint, the algorithm exits the loop; otherwise, it selects the one with the highest benefit-to-loss ratio as in the previous step, and performs tuning of bias and bit width again. In the example of LeNet, the aforementioned configuration from Iteration 29 is selected as the best one.

Phase 3 (Accuracy Recovery). From this phase the algorithm enters the second pass to use the full test set for evaluating accuracy (Line 15-27 in Algorithm 1). Since we change the test set, there is no guarantee that the current

best configuration still satisfies the accuracy constraint. If the current configuration satisfies the constraint, the algorithm just moves to the next phase; otherwise, it increases the bit width and/or decrements the bias until the constraint is satisfied (Line 22-26). This algorithm is the same as the fine-tuning algorithm for bit width in Phase 2 except that it increments (decrements) the bit width (bias) instead of decrementing (incrementing) it. This loop is repeated until the algorithm finds the first configuration whose accuracy is above the threshold. In the example of LeNet in Figure 4.2, we skip this phase as the accuracy constraint is already satisfied.

Phase 4 (Fine-tuning Bias, Bit Width and Type). Finally, we repeat the same fine-tuning process of bias, bit width and number type as in Phase 2, but using the *full* test set. In Figure 4.2 the algorithm finally outputs the best configuration for LeNet, which is I2(1)-I2(4) for weights and I2(0)-I2(-2) for activations.

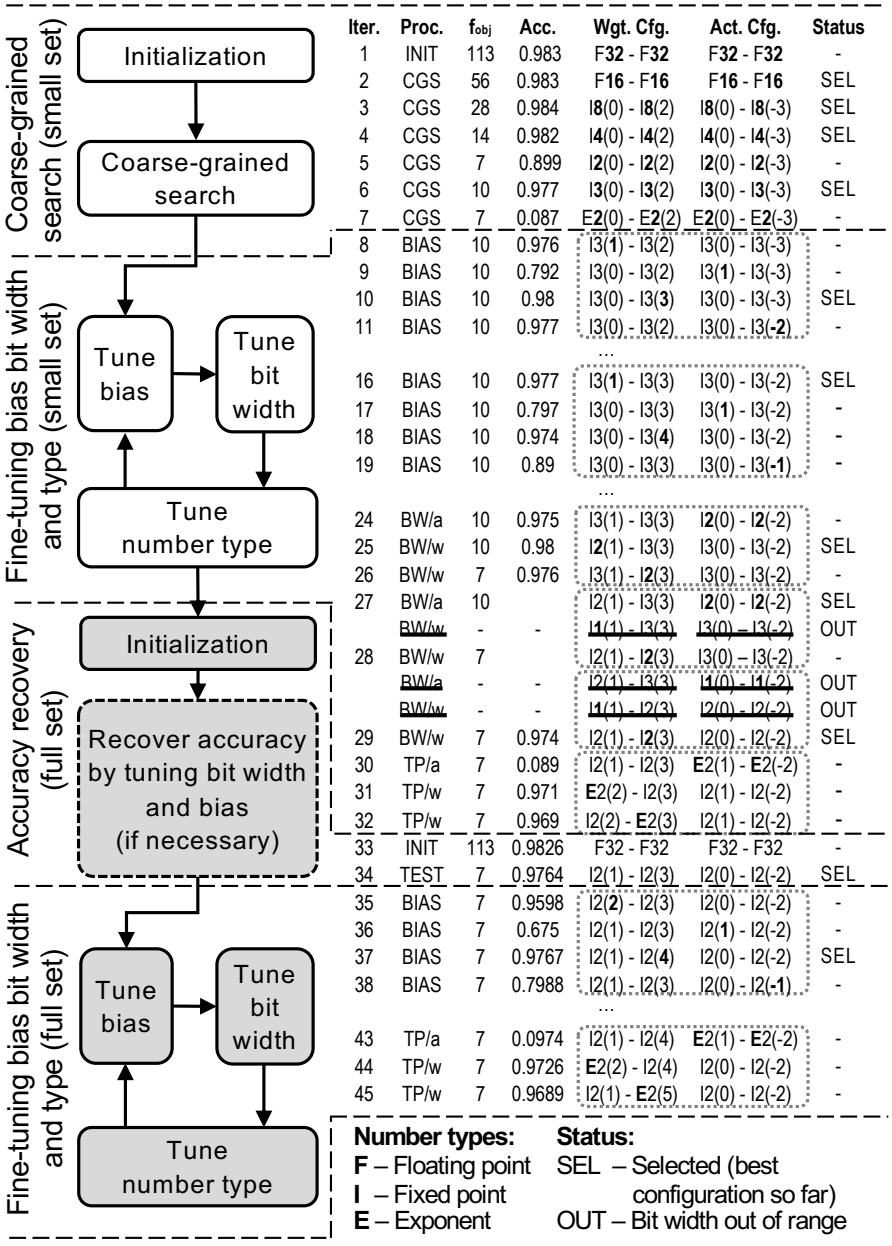


Figure 4.2: Auto-tuning the two CONV layers of LeNet with maximum accuracy loss of 1% (accuracy threshold: 0.9728). Biases (B) are shown in parentheses. Layer-wise optimization is enabled only for weights.

Chapter 5

Evaluation

5.1 Methodology

To evaluate auto-tuner, we use nine CNN and two MLP models as summarized in Table 5.1. We port two popular DNN frameworks to auto-tuner : Caffe [21] and DarkNet [42]. For Tiny YOLO we use a mean-average precision (mAP) metric for quantifying object detection accuracy; for the others image classification accuracy. Following the methodology of Stanford DAWNBench [9], we target 93% relative accuracy of the full-precision 32-bit floating-point type (i.e., `err_margin` = 0.07) by default. We have also tested 99% accuracy to observe similar results, which are omitted due to limited space. As discussed in Section 2.1, we quantize a pre-trained model for the given accuracy constraint [19, 57, 58]. If necessary, one can recover the accuracy by re-training the quantized model.

Framework	**Network**	**Dataset**	**Metric**
Caffe	MLP-M [51]	MNIST [29]	Accuracy
	MLP-C [53]	CIFAR-10 [27]	Accuracy
	LeNet [30]	MNIST [29]	Accuracy
	Cuda-convnet [26]	CIFAR-10 [27]	Accuracy
	AlexNet [28]	ImageNet [8]	Accuracy
	NiN [32]	ImageNet [8]	Accuracy
	VGG-16 [48]	ImageNet [8]	Accuracy
	GoogleNet [13]	ImageNet [8]	Accuracy
	ResNet-50 [20]	ImageNet [8]	Accuracy
DarkNet	DarkNet-ref [42]	ImageNet [8]	Accuracy
	Tiny YOLO [43]	VOC2007 [15]	mAP

Table 5.1: Network models for evaluation

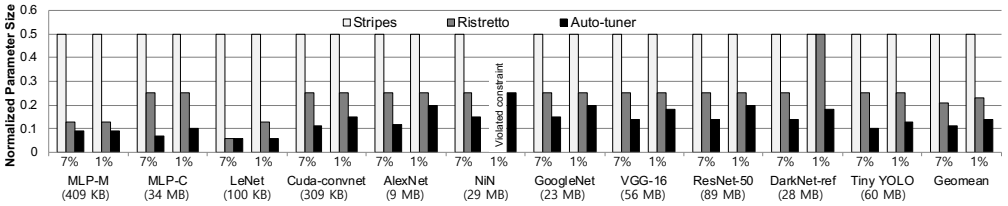


Figure 5.1: Model parameter size normalized to the `FLOAT(32)` baseline with 7% and 1% of accuracy tolerance. Lower is better. Network name is annotated with the baseline parameter size.

We quantize both weights and activations of convolution layers (for CNNs) and fully-connected layers (for MLPs), but layer-wise optimization is applied to weights only. Otherwise, we would have to pay the cost of format conversion across layers, which may nullify performance benefits. We use the default objective function (f_{obj}) for the auto-tuner, which minimizes the total number of bits for both weights and activations, and hence storage requirements as well as computational strength.

We compare the quantization quality of auto-tuner against two well-known frameworks: Ristretto [18] and the Stripes quantizer [22]. We have faithfully reproduced both algorithms and validated them so that they perform at least comparably to or better than the reported results under the same constraints. Note that Stripes quantizes activations only and uses the `FIXED(16)` type for weights. For fair comparison we also run auto-tuner under the same constraints.

5.2 Results

Search Quality. Figure 5.1 compares the size of network parameters at the best configuration for 7% and 1% error tolerance. The results are normalized to the 32-bit floating-point baseline as in [18]. Auto-tuner reduces the parameter size by 8.91× and 7.13× on average and total storage requirements (including activations) by 8.28× and 6.44× for 7% and 1% tolerance, respectively. These numbers translate to 69.6% (38.1%) reduction of storage space over Stripes (Ristretto) for 7% tolerance. Note that Stripes has a constant size of 16 bits as its weight format is fixed to `FIXED(16)` [22]. Both Ristretto and Stripes fail to tune NiN within 1% accuracy loss as NiN requires careful bias tuning

Network	Framework	Per Layer Format (Act/Weights)
MLP-M	Auto-tuner	I3 / E3-E2-E3
	Ristretto	I2 / I4-I4-I4
MLP-C	Auto-tuner	I4 / I2-E3-I3-I4
	Ristretto	I2 / I8-I8-I8-I8
LeNet	Auto-tuner	I2 / E2-E2
	Ristretto	I2 / I2-I2
Cuda-convnet	Auto-tuner	I3 / I4-E3-I4
	Ristretto	I2 / I8-I8-I8
AlexNet	Auto-tuner	I5 / I5-I6-E4-E3-E4
	Ristretto	I4 / I8-I8-I8-I8-I8
NiN	Auto-tuner	I6 / I4-I4-I6-E4-I5-I5-I6-I4-I6-E4-I5-I6
	Ristretto	I8 / I8-I8-I8-I8-I8-I8-I8-I8-I8-I8-I8-I8
VGG-16	Auto-tuner	I8 / I5-I3-I7-I5-I5-I4-E4-I4-I4-I5-I6-I4-I3
	Ristretto	I8 / I8-I8-I8-I8-I8-I8-I8-I8-I8-I8-I8-I8
GoogleNet	Auto-tuner	I6 / I6-I6-I5-I6-I4-I6-I5-I5-I5-I5-I4
	Ristretto	I8 / I8-I8-I8-I8-I8-I8-I8-I8-I8-I8-I8
ResNet-50	Auto-tuner	I7 / I6-I7-I5-I6-I6-I4-I5-I5-I5-I5-I4-I4-I5-I6-I5-I3-I4
	Ristretto	I8 / I8-I8-I8-I8-I8-I8-I8-I8-I8-I8-I8-I8-I8-I8-I8-I8-I8
DarkNet	Auto-tuner	I7 / I8-I7-I8-I6-I6-I6-E4-I5
	Ristretto	I8 / I8-I8-I8-I8-I8-I8-I8-I8
Tiny YOLO	Auto-tuner	I8 / I6-I6-I7-I6-I5-I5-I3-E3-I5
	Ristretto	I8 / I8-I8-I8-I8-I8-I8-I8-I8-I8

Table 5.2: Best configuration: In (`FIXED`), En (`EXP`)

when using fixed-point numbers.[1] The improvement of search quality is largely attributed to careful bias tuning and layer-wise optimization, compared to the prior art. Using multiple types also helps but to a lesser degree.

Our algorithm is a greedy algorithm, which is known to have a problem of local minima, especially when the tuning space is huge. However, this problem is non-existent for some phases. For example, the coarse-grained search phase is free of this problem as it is guaranteed to produce the same outcome with an exhaustive search. It is because the global minimum bit width can be obtained by taking the minimum bit width of all supported types, and once the minimum bit width is found for a type, it is necessary to test only narrower bit widths for the remaining types. The fine-tuning phase may produce a different result if the search order (tuning bias, bit width and number type) is changed. We evaluate different search orders and empirically find the order that yields best results.

Table 5.2 shows the best configurations found by both auto-tuner and

[1]Stripes [22] reports successful quantization of NiN within 1% accuracy loss, but their baseline (`FIXED(16)`) is different from ours (`FLOAT(32)`) having a higher baseline accuracy.

Network	# of target layers	# of iterations		
		Exhaustive	Auto-tuner	Ratio
MLP-M	3	3.11×10^{6}	190	6.11×10^{-5}
MLP-C	4	1.31×10^{8}	359	2.75×10^{-6}
LeNet	2	7.41×10^{4}	74	9.99×10^{-4}
Cuda-convnet	3	3.11×10^{6}	208	6.68×10^{-5}
AlexNet	5	5.49×10^{9}	419	7.63×10^{-8}
NiN	12	1.27×10^{21}	2759	2.18×10^{-18}
VGG-16	13	5.31×10^{22}	3516	6.62×10^{-20}
GoogleNet	11	3.01×10^{19}	1783	5.92×10^{-17}
ResNet-50	17	1.65×10^{29}	5471	3.31×10^{-26}
DarkNet-ref	8	4.07×10^{14}	1130	2.78×10^{-12}
Tiny YOLO	9	1.71×10^{16}	1644	9.63×10^{-14}

Table 5.3: Comparison of search costs

Ristretto. Auto-tuner yields more compact representation to reduce total storage requirements by up to 71.3% for MLP-C with an average of 38.1%. This performance gap is attributed to the difference in the coverage of the configuration space. Ristretto has a much narrower format coverage as it considers only `FIXED(`2^n`)` formats and does not perform layer-wise optimization. For example, the search space of auto-tuner has 1.65×10^{29} configurations for ResNet-50, whereas Ristretto has only 6. Even if we relax the constraints of Ristretto to consider all integers (i.e., `FIXED(n)` with $2 \leq n \leq 32$), instead of `FIXED(`2^n`)`, the parameter compression ratio is still significantly lower than auto-tuner ($6.35\times$ vs. $8.91\times$).

Search Cost. Another important metric is the cost of search. Table 5.3 summarizes this cost for auto-tuner in terms of the number of iterations (i.e., tested configurations), which ranges from 74 to 5471, even if the search space can be as large as 1.65×10^{29}. The auto-tuning algorithm navigates through only a tiny fraction of the search space to find best configuration. Besides, the two-pass algorithm, using both small set and full set, effectively reduces the average cost per iteration. Using the small set yields a maximum speedup of $4.48\times$ for Cuda-convnet with an average speedup of $2.52\times$ over a version of auto-tunerusing the full set only.

There is a tradeoff between search cost and quality. For example, one may reduce search time by using the same configuration for replicated blocks of

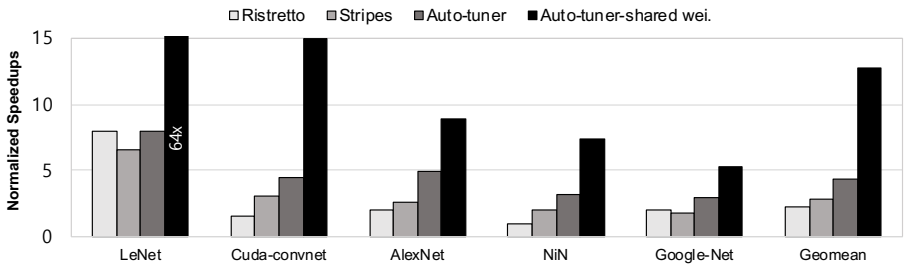


Figure 5.2: Normalized speedups using bit-serial hardware

layers (e.g., inception modules, residual blocks) for models such as ResNet and GoogleNet. This can be easily done in auto-tuner by having corresponding layers to share the same layer name. However, it increases the total parameter size by 82% and 22% over auto-tuner for ResNet-50 and GoogleNet, respectively. Note that this tradeoff can be easily explored using auto-tuner.

Ideal Speedups on Bit-serial Hardware. The benefits of quantization can be best realized by bit-serial hardware, whose latency for a MAC operation scales (almost) linearly to the number of bits [1, 22, 47]. We use simple analytical performance models to estimate ideal performance of a bit-serial accelerator based on Loom [47]. Loom is a serialized version of DaDianNao [4]. Assuming two constraints as Stripes [22] and Loom (i.e., 16-bit fixed-point baseline, quantizing activations for Stripes and quantizing both for Loom only using dynamic fixed-point type, 1% accuracy tolerance), we run auto-tuner to quantize the target layers. Note that layer-wise optimization is only applied to activation only. The corresponding bit widths for Stripes are taken from the original paper, instead of using our version of the Stripes quantizer. Assuming 100% PE utilization and 1-cycle latency for a 1-bit serial operation, the execution time of a kernel on DaDianNao is estimated to be the total GOPs of the MAC operations for the layer divided by the total number of PEs (in ops/cycle). We also assume a linear scaling of the throughput to reduction in bit width (i.e., speedup is $\frac{256}{p_w p_a}$ if p_w and p_a is the bit width of weight and activation).

Figure 5.2 shows ideal speedups of Stripes, Ristretto and auto-tuner with Stripes constraint and auto-tuner with Loom constraint over DaDianNao. We only use a subset of 5 DNN models that are also used for evaluating Stripes [22]

and Loom [47]. Auto-tuner achieves a geomean speedup of 4.19×, which outperforms both Ristretto and the Stripes quantizer by only quantizing activation. However, quantize both weights and activations, which reduces the bit width aggressively, yields a higher 12.7× geomean speedup. In this setup, a reduction in bit width leads to performance boost directly, to demonstrate greater performance gap between auto-tuner and the other two quantizers.

Chapter 6

Related Work

Quantization for DNNs. There are proposals to reduce the precision of data to improve performance of pre-trained DNNs with little degradation of accuracy [7, 12, 23, 24, 31, 33, 35, 41]. However, they have limited coverage by considering only one type of numbers [23, 24, 35] or supporting mixed types in a very limited way [12]. Other techniques employ a table-like structure to extract representative values for quantization. Zhu et al. [58] propose trained ternary quantization, which iteratively trains an DNN using three representative values for each layer. These values are selected and encoded during a training phase. Deep compression [19] uses a value clustering technique to identify representative values. However, these techniques suffer from irregular memory accesses and inefficient computation due to an extensive use of high-precision values. In contrast, auto-tuner selects an suitable representation for each layer by considering both number type and bit width to effectively eliminate redundancy in numbers. Zhou et al.[57] propose a quantization technique that utilizes multiple number types, and demonstrate savings in bit count. However, they have much narrower type coverage by using zero and exponent types only, which can be problematic for complex networks.

Tuning Algorithms and Frameworks. Caffe-Ristretto [17] is a framework designed for evaluating quantization on DNNs. However, their implementation lacks portability to make it hard to use for tuning on another platform beyond Caffe on GPU. As demonstrated in Section 5.2, their framework has a very limited coverage of the number format space with no support for layer-wise optimization. In contrast, our auto-tuner is light-weight and has no platform dependencies, provides much better coverage in supported number formats.

Furthermore, auto-tuner can flexibly support various optimization goals by configuring the auto-tuner using its configuration interface.

Outside the domain of DNNs, auto-tuning has been investigated in the research community for a long time. Recently, auto-tuning techniques have been applied to compiler optimization [2], runtime parallelism adaptation [39, 40], to name a few. While there are some potentials to these frameworks for tuning DNN workloads, their efficiency will be much lower without considering DNNs' algorithmic characteristics. In contrast, auto-tuner employs an efficient tuning algorithm customized for quantizing DNNs.

Chapter 7

Conclusion

This work introduces an automatic quantization tuner that optimizes number representation for each layer of a DNN targeting an FPGA platform or a bit-serial hardware, and separates the concern for developing an effective DNN from the concern of optimizing the number representation at a bit level. Auto-tuner performs an efficient search by navigating through a tiny fraction of the search space, which ranges from 74 to 5471, to find a suitable number representation for each layer of a DNN, even if the search space can be as large as 1.65×10^{29}. More than this, with some flexible Python APIs, user can easily to make a tradeoff between search cost and search quality by sharing the same number representation for a network basic block (e.g., residual block in ResNet-50). We also demonstrate the high-quality search by evaluating eleven DNN models on two DNN frameworks. Under the 7% (1%) loss constraint relative to the baseline 32-bit floating-point accuracy, auto-tuner achieves over 8× (7×) reduction in the parameter size on average, with a maximum reduction of 16× with careful bias tuning and layer-wise optimization.

Bibliography

[1] Jorge Albericio, Alberto Delmás, Patrick Judd, Sayeh Sharify, Gerard O'Leary, Roman Genov, and Andreas Moshovos. Bit-pragmatic deep neural network computing. In *2017 50th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '17, 2017.

[2] Jason Ansel, Shoaib Kamil, Kalyan Veeramachaneni, Jonathan Ragan-Kelley, Jeffrey Bosboom, Una-May O'Reilly, and Saman Amarasinghe. Opentuner: An extensible framework for program autotuning. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, PACT '14, 2014.

[3] Doug Burger. Microsoft unveils Project Brainwave for real-time AI. `https://www.microsoft.com/en-us/research/blog/microsoft-unveils-project-brainwave/`, 2017.

[4] Y. Chen, T. Luo, S. Liu, S. Zhang, L. He, J. Wang, L. Li, T. Chen, Z. Xu, N. Sun, and O. Temam. Dadiannao: A machine-learning supercomputer. In *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '14, 2014.

[5] E. Chung, J. Fowers, K. Ovtcharov, M. Papamichael, A. Caulfield, T. Massengill, M. Liu, D. Lo, S. Alkalay, M. Haselman, M. Abeydeera, L. Adams, H. Angepat, C. Boehn, D. Chiou, O. Firestein, A. Forin, K. S. Gatlin, M. Ghandi, S. Heil, K. Holohan, A. El Husseini, T. Juhasz, K. Kagi, R. Kovvuri, S. Lanka, F. van Megen, D. Mukhortov, P. Patel, B. Perez, A. Rapsang, S. Reinhardt, B. Rouhani, A. Sapek, R. Seera, S. Shekar, B. Sridharan, G. Weisz, L. Woods, P. Yi Xiao, D. Zhang, R. Zhao, and D. Burger. Serving dnns in real time at datacenter scale with project brainwave. *IEEE Micro*, 2018.

[6] Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. Training deep neural networks with low precision multiplications, 2014.

[7] Matthieu Courbariaux, Itay Hubara, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. Binarized neural networks: Training deep neural networks with weights and activations constrained to +1 or -1, 2016.

[8] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *The IEEE Conference on Computer Vision and Pattern Recognition*, CVPR '09, 2009.

[9] Cody A. Coleman et al. Dawnbench: An end-to-end deep learning benchmark and competition. `https://github.com/stanford-futuredata/dawn-bench-entries`, 2017.

[10] G. Hinton et al. Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups. *IEEE Signal Processing Magazine*, 2012.

[11] Norman P. Jouppi et al. In-datacenter performance analysis of a tensor processing unit. 2017.

[12] Paulius Micikevicius et al. Mixed precision training, 2017.

[13] Szegedy et al. Going deeper with convolutions. In *The IEEE Conference on Computer Vision and Pattern Recognition*, CVPR '15, 2015.

[14] Volodymyr Mnih et. al. Human-level control through deep reinforcement learning. *Nature*, 2015.

[15] Mark Everingham, Luc Van Gool, Christopher KI Williams, John Winn, and Andrew Zisserman. The pascal visual object classes (voc) challenge. *International journal of computer vision.*, 2010.

[16] Jeremy Fowers, Kalin Ovtcharov, Michael Papamichael, Todd Massengill, Ming Liu, Daniel Lo, Shlomi Alkalay, Michael Haselman, Logan Adams, Mahdi Ghandi, Stephen Heil, Prerak Patel, Adam Sapek, Gabriel Weisz, Lisa Woods, Sitaram Lanka, Steven K. Reinhardt, Adrian Caulfield, Eric

S. Chung, and Doug Burger. A configurable cloud-scale dnn processor for real-time ai. In *Proceedings of the 45th International Symposium on Computer Architecture*, ISCA '18, 2018.

[17] Philipp Gysel, Mohammad Motamedi, and Soheil Ghiasi. Hardware-oriented approximation of convolutional neural networks. 2016.

[18] Philipp Gysel, Jon Pimentel, Mohammad Motamedi, and Soheil Ghiasi. Ristretto: A framework for empirical study of resource-efficient inference in convolutional neural networks. *IEEE Transactions on Neural Networks and Learning Systems.*, 2018.

[19] Song Han, Huizi Mao, and William J. Dally. Deep compression: Compressing deep neural network with pruning, trained quantization and huffman coding, 2015.

[20] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *The IEEE Conference on Computer Vision and Pattern Recognition*, CVPR '16, 2016.

[21] Yangqing et al. Jia. Caffe: Convolutional architecture for fast feature embedding. In *Proceedings of the 22nd ACM International Conference on Multimedia*, ACMMM '14, 2014.

[22] P. Judd, J. Albericio, T. Hetherington, T. M. Aamodt, and A. Moshovos. Stripes: Bit-serial deep neural network computing. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '16, 2016.

[23] Patrick Judd, Jorge Albericio, Tayler Hetherington, Tor Aamodt, Natalie Enright Jerger, Raquel Urtasun, and Andreas Moshovos. Reduced-precision strategies for bounded memory in deep neural nets. 2015.

[24] Patrick Judd, Jorge Albericio, Tayler Hetherington, Tor M. Aamodt, Natalie Enright Jerger, and Andreas Moshovos. Proteus: Exploiting numerical precision variability in deep neural networks. In *Proceedings of the 2016 International Conference on Supercomputing*, ICS '16, 2016.

[25] Yong-Deok Kim, Eunhyeok Park, Sungjoo Yoo, Taelim Choi, Lu Yang, and Dongjun Shin. Compression of deep convolutional neural networks for fast and low power mobile applications, 2015.

[26] Alex Krizhevsky. cuda-convnet: High-performance c++/cuda implementation of convolutional neural networks. `https://code.google.com/p/cuda-convnet/`, 2012.

[27] Alex Krizhevsky and Geoffrey Hinton. Learning multiple layers of features from tiny images. 2009.

[28] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*. 2012.

[29] Yann LeCun. The mnist database of handwritten digits. *http://yann.lecun.com/exdb/mnist/.*, 1998.

[30] Yann LeCun et al. Lenet-5, convolutional neural networks. 1998.

[31] Darryl Lin, Sachin Talathi, and Sreekanth Annapureddy. Fixed point quantization of deep convolutional networks. In *Proceedings of the 33rd International Conference on Machine Learning*, ICML '16, 2016.

[32] Min Lin, Qiang Chen, and Shuicheng Yan. Network in network, 2013.

[33] Zhouhan Lin, Matthieu Courbariaux, Roland Memisevic, and Yoshua Bengio. Neural networks with few multiplications, 2015.

[34] T. Mikolov, S. Kombrink, L. Burget, J. Cernocky, and S. Khudanpur. Extensions of recurrent neural network language model. In *2011 IEEE International Conference on Acoustics, Speech and Signal Processing*, ICASSP '11, 2011.

[35] Daisuke Miyashita, Edward H. Lee, and Boris Murmann. Convolutional neural networks using logarithmic data representation, 2016.

[36] Young H Oh, Quan Quan, Daeyeon Kim, Seonghak Kim, Jun Heo, Sungjun Jung, Jaeyoung Jang, and Jae W Lee. A portable, automatic data qantizer for deep neural networks. In *Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques*, page 17. ACM, 2018.

[37] Cesc Chunseong Park, Byeongchang Kim, and Gunhee Kim. Attend to you: Personalized image captioning with context sequence memory networks. In *The IEEE Conference on Computer Vision and Pattern Recognition*, CVPR '17, 2017.

[38] Jiantao et al. Qiu. Going deeper with embedded fpga platform for convolutional neural network. In *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '16, 2016.

[39] Arun Raman, Hanjun Kim, Taewook Oh, Jae W. Lee, and David I. August. Parallelism orchestration using dope: the degree of parallelism executive. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, 2011.

[40] Arun Raman, Ayal Zaks, Jae W. Lee, and David I. August. Parcae: a system for flexible parallel execution. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, 2012.

[41] Mohammad Rastegari, Vicente Ordonez, Joseph Redmon, and Ali Farhadi. Xnor-net: Imagenet classification using binary convolutional neural networks, 2016.

[42] Joseph Redmon. Darknet: Open source neural networks in c, 2013-2016. URL `http://pjreddie.com/darknet/`.

[43] Joseph Redmon and Ali Farhadi. Yolo9000: better, faster, stronger. 2016.

[44] Mengye Ren, Ryan Kiros, and Richard Zemel. Exploring models and data for image question answering. In *Advances in neural information processing systems*. 2015.

[45] Hochreiter S. and Schmidhuberm J. Long short-term memory. *Neural Computation*, 1997.

[46] Florian Schroff, Dmitry Kalenichenko, and James Philbin. Facenet: A unified embedding for face recognition and clustering. In *The IEEE Conference on Computer Vision and Pattern Recognition*, CVPR '15, 2015.

[47] Sayeh Sharify, Alberto Delmas Lascorz, Kevin Siu, Patrick Judd, and Andreas Moshovos. Loom: Exploiting weight and activation precisions to accelerate convolutional neural networks. In *Proceedings of the 55th Annual Design Automation Conference*, page 20. ACM, 2018.

[48] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition, 2014.

[49] Sainbayar Sukhbaatar, arthur szlam, Jason Weston, and Rob Fergus. End-to-end memory networks. In *Advances in neural information processing systems*. 2015.

[50] Hokchhay Tann, Soheil Hashemi, R Iris Bahar, and Sherief Reda. Hardware-software codesign of accurate, multiplier-free deep neural networks. In *Design Automation Conference (DAC), 2017 54th ACM/EDAC/IEEE*, pages 1–6. IEEE, 2017.

[51] TensorFlow™. Tensorflow mechanics 101. `https://github.com/tensorflow/tensorflow/tree/r1.2/tensorflow/examples/tutorials/mnist`, 2017.

[52] TensorFlow™. Tensorflow: How to quantize neural networks with tensorflow. `https://www.tensorflow.org/performance/quantization`, 2018.

[53] Rakesh Vasudevan. Cifar-10 classifier. `https://github.com/vrakesh/CIFAR-10-Classifier`, 2017.

[54] Ouyang Wanli and Xiaogang Wang. Joint deep learning for pedestrian detection. In *Proceedings of the IEEE International Conference on Computer Vision*, ICCV '13, 2013.

[55] Jason Weston, Sumit Chopra, and Antoine Bordes. Memory networks. In *Proceedings of the International Conference on Learning Representations*, ICLR '15, 2015.

[56] Ritchie Zhao, Weinan Song, Wentao Zhang, Tianwei Xing, Jeng-Hau Lin, Mani Srivastava, Rajesh Gupta, and Zhiru Zhang. Accelerating binarized convolutional neural networks with software-programmable fpgas. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '17, 2017.

[57] Aojun Zhou, Anbang Yao, Yiwen Guo, Lin Xu, and Yurong Chen. Incremental network quantization: Towards lossless cnns with low-precision weights, 2017.

[58] Chenzhuo Zhu, Song Han, Huizi Mao, and William J. Dally. Trained ternary quantization. 2016.

국문초록

AI 기반 응용 프로그램 및 서비스의 확산으로 심층 신경망 (DNN)의 효율적인 처리에 대한 수요가 크게 증가하고 있다. DNN은 많은 계산량과 메모리 공간을 필요로 하기 때문에 컴퓨팅 및 메모리 집약적인 것으로 알려져 있다. 양자화는 적은 비트 수로 숫자를 표현하여 컴퓨팅 성능과 메모리 공간을 모두 줄이는데 널리 사용되는 방법이다.

그러나 계층별 최적화로 인해 악화되는 다양한 비트 폭을 가진 가능한 숫자 표현의 조합이 수천만가지가 있다, 따라서 DNN에 대한 최적의 숫자 표현을 찾는 것은 어려운 작업이다. 이를 해결하기 위해 본 논문는 DNN 양자화를 위한 자동 튜너를 제안한다. 여기서 자동 튜너는 정확도 제약 조건을 만족시키면서 사용자의 목적 함수를 최소화하여 숫자의 콤팩트한 표현 (숫자유형, 비트 및 바이어스)을 찾아 준다. FPGA 플랫폼과 bit-serial 하드웨어을 응용대상으로 각각 두 DNN 프레임 워크에서 11 개의 DNN 모델을 사용하여 평가 했다. 상대 정확도 최대 7% (1%) 손실이 허용되는 상황에 32 비트 floating-point를 사용하는 baseline 과 비교할 때에 변수 크기가 평균적으로 8배 (7배) 감소되고, 최대로는 16배까지 감소되었다.

주요어: 자동 튜너, 양자화, 심층 신경망, 최적화
학번: 2017-21772

Acknowledgements

I would like to take this opportunity to express my deep gratitude to my supervisor, Dr. Jae W.Lee, a respectable, responsible and excellent scholar, who has offered me constructive guidance for the thesis and encouragement for its completion and improvement. What's more, his invaluable advice on both research and career had a great influence on my life.

Besides, I wish to extend my sincere gratitude to my co-workers Young H. Oh, Daeyeon kim, Seonghak Kim, Jun Heo, Sungjun Jung, Jaeyoung Jang, who had put considerable time for giving a lot of valuable comments and helped me with writing and experiments. I also owe my sincere gratitude to other workmates, who gave me their help and time in helping me work out many problems during my postgraduate two years.

Last my thanks would go to my beloved family. Thanks for encouraging me in all of my pursuits. I am especially grateful to my mother, who supported me emotionally and financially. Her encouragement had always been my motivation and push me forward.