Neural Network Quantization for Efficient Inference: A Survey

Olivia Weng
Dept. of Computer Science and Engineering
University of California, San Diego
oweng@ucsd.edu

ABSTRACT

As neural networks have become more powerful, there has been a rising desire to deploy them in the real world; however, the power and accuracy of neural networks is largely due to their depth and complexity, making them difficult to deploy, especially in resource-constrained devices. Neural network quantization has recently arisen to meet this demand of reducing the size and complexity of neural networks by reducing the precision of a network. With smaller and simpler networks, it becomes possible to run neural networks within the constraints of their target hardware. This paper surveys the many neural network quantization techniques that have been developed in the last decade. Based on this survey and comparison of neural network quantization techniques, we propose future directions of research in the area.

KEYWORDS

neural network, quantization, edge computing

ACM Reference Format:
Olivia Weng. 2021. Neural Network Quantization for Efficient Inference: A Survey. In *Proceedings of N/A (Research Exam).* ACM, San Diego, CA, USA, 13 pages. https://doi.org/10.1145/nnn.nnn

1 INTRODUCTION

Neural Networks (NNs) are powerful tools for completing high-accuracy pattern recognition tasks; however, to achieve high levels of accuracy, NNs are often over-parameterized [9], growing to such size and depth that make it prohibitively expensive to deploy in resource-constrained environments, e.g., at the edge. Enabling neural network inference in resource-constrained settings is important so that NNs can solve problems like speech recognition, autonomous driving, and image classification in IoT devices, vehicles, and more. To realize this, neural network inference must achieve 1) real-time latency, 2) low energy consumption, and 3) high accuracy [14].

It is difficult to achieve these three goals because NNs are designed in software to reach high accuracy by having tens to hundreds of millions of parameters. Moreover, when training NNs (which usually occurs on GPUs), these parameters are represented using 32-bit floating point numbers because of their high precision. When it comes time to deploy neural networks in hardware, however, we are left with massive NNs that are up to hundreds of megabytes —at times, even several gigabytes—in size and require billions of floating point operations. The two main issues with deploying NNs are that they:

- are too big for hardware and
- use expensive datatypes.

NNs are oftentimes too big for hardware because the target edge devices do not have enough memory to store and run the model. NNs often use 32-bit floating point datatypes to represent their parameters as well as all of the computations involved with inference, meaning they require expensive floating point units to run. These large sizes and intensive computation requirements present substantial obstacles to achieving fast and efficient inference.

Therefore, the size and complexity of NNs are major problems that inhibit real-time, efficient inference. To address these issues, several research avenues have emerged, such as

(1) designing/searching for novel small NNs,
(2) knowledge distillation,
(3) pruning/sparsification, and
(4) quantization.

All of these approaches have a common goal of reducing the size and/or complexity of NNs while maintaining high accuracy.

This paper focuses on NN quantization and surveys the numerous techniques that been developed over the years to address the associated challenges that have arisen in the space. Quantization is defined as reducing the precision used to represent neural network parameters, usually from n bits to m bits, where $n > m$. There is unique opportunity to achieving efficient inference with quantization because hardware such as Field-Programmable Gate Arrays (FPGAs) and Application-Specific Integrated Circuits (ASICs) can be configured to use any kind of numerical representation such as floating point, fixed point, integer, and even custom datatypes.Thus, the goal with NN quantization involves not only reducing the number of bits but also converting the original numerical representation to a less precise datatype that is cheaper to implement in hardware. Doing so makes it is possible to achieve efficient, real-time inference. For example, a common case in NN quantization involves starting with 32-bit floating point representation (since that is generally what is used to train NNs on GPUs) and converting it to 8-bit integer representation. This is desirable because integer arithmetic is less complex than floating point arithmetic and thus faster to compute. With fewer bits to compute and a cheaper numerical representation, executing an 8-bit integer NN significantly lowers latency, energy consumption, and resource utilization.

Research Exam, Fall 2021, San Diego, USA

https://doi.org/10.1145/nnn.nnn

The primary challenge with quantization, however, is maintaining the NN's high accuracy post quantization. When reducing the precision of the network, the NN is effectively losing information it learned during training. This presents the main trade-off involved with quantization: *precision vs. accuracy.* A reasonable expectation is that we must trade accuracy for lower precision to achieve smaller models that can fit in hardware; however, as is often the case with NNs, there are no hard and fast rules.

There exists a wrinkle in the problem of NN quantization: NNs are often over-parameterized and can thus afford to lose precision with minimal to no loss in accuracy [14]. Since one of the primary overarching goals in NN quantization is maintaining accuracy, we are not particularly wedded to quantizing weights such that their quantized values are as close to their floating point counterparts as permitted by the quantization scheme. We want to quantize the weights in a way that maintains the network's overall classification accuracy. In fact, trying to minimize the distance between the floating point and quantized weight representations does not directly translate to maintaining classification accuracy because of the over-parameterization of neural networks [18, 30]. Therefore, the over-parameterization wrinkle presents opportunities to reduce precision in clever ways without any cost to accuracy—at times, quantizing a NN even improves its accuracy.

NN quantization emerged as a field of study in the 1990s during a resurgence of neural network research [19, 22, 23]. At the time, one of the main inhibitors to neural network adoption in the real world was that training was too slow on the machines available then. As such, the primary motivation behind NN quantization then was to reduce the bitwidths/precision as a way of speeding up training times. In 2012, AlexNet, a convolutional neural network (CNN), won the ImageNet Large Scale Visual Recognition Challenge, demonstrating that deep and complex networks could be trained efficiently on GPUs to achieve high accuracy. This popularized using GPUs to train NNs, overcoming the training bottleneck of the past and leading to an exponential increase in NN research. With so many new kinds of NNs being developed, companies and scientists have been itching to deploy them in hardware, such as edge devices, leading to a renewal in NN quantization efforts in the last decade.

This paper surveys the NN quantization research that has surfaced in the last decade. It does not capture the full spectrum of NN quantization, for the field is massive. Therefore, we survey some of the most popular quantization techniques, namely integer/fixed point, binary, ternary, and mixed precision quantization. In our survey, we make the assumption that the reader has a basic understanding of NNs and their components, e.g., that NNs are made up of a series of weight and activation layers and trained with forwards and backward passes and weight updates via Stochastic Gradient Descent.

NNs can be quantized to make either training or inference more efficient (or both). In addition to quantizing a NN's parameters (namely its weights), quantized NN training quantizes gradients, whereas quantized NN inference quantizes activations. In this paper, since we are interested in the implications of quantizing NN for efficient hardware deployment, we limit our scope to inference, i.e., quantizing weights and activations, and only touch on training when it leads to hardware efficiency at inference time.

Paper Layout: Section 2 gives an overview of how numbers are represented as datatypes on machines. Section 3 provides an example of quantization. Section 4 describes the two main procedures used to quantize a network, namely Quantization-Aware Training and Post-Training Quantization. Sections 6, 5, 7 review work on four popular quantization schemes: integer/fixed point, binary, ternary, and mixed precision quantization. Section 8 concludes the paper with possible future avenues of research in NN quantization.

2 REPRESENTING NUMBERS ON MACHINES

In this section, we review how datatypes are used to represent data, and more specifically real numbers, on machines. We detail how floating point and fixed point datatypes work because a common quantization scenario involves quantizing floating point to fixed point. It is important to go into detail on the inner workings of floating point and fixed point to understand the costs and benefits of using them with respect to hardware.

Since data on computers is represented in binary, the machine needs to know how to interpret the binary, i.e, as a character, string, integer, etc. This is the primary function of datatypes. Several standards have been developed to dictate how we should use binary code to represent different kinds of data. For instance, the ASCII standard defines a correspondence between natural numbers and individual characters, like letters, numbers, punctuation, etc. The same needs to be done for representing real numbers.

While integers have a straightforward correspondence to binary values, representing real numbers in binary is much more complex. There are several considerations to be made. For instance, how precise do we want our real number representations to be, i.e., how many decimal places should be accounted for. It is so complex that there is an IEEE standard for how real numbers should be represented as well as how arithmetic on this real number representation should work—IEEE Standard 754, also known as IEEE floating point. Since there are infinitely many real numbers and only so many bits that can be allocated for representing each number on machines, we can actually view representing real numbers on computers as a quantization problem itself because we are reducing the precision of the reals.

There are two main ways of representing real numbers: floating point and fixed point.

2.1 Floating Point

The idea behind floating point is to represent numbers in scientific notation, $n \times 2^m$, wherein we need only keep track of n and m in our representation [4]. IEEE floating point defines a floating point number n as

$$n = (-1)^s \times m \times 2^E \tag{1}$$

where

- s is the *sign* bit, 0 for positive and 1 for negative,
- m is the *mantissa*, also called the significand, which defines the precision, and
- E is the *exponent*, which determines the range.

A floating point number encodes the sign bit, mantissa, and exponent in its binary representation. In single precision floating

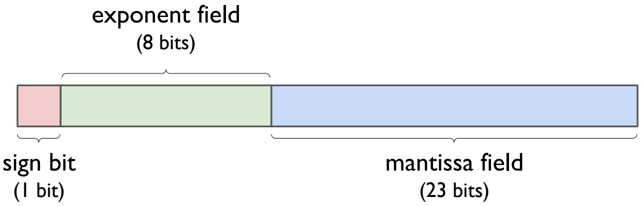

Figure 1: IEEE single-precision floating point bit fields. The binary of a floating point value is broken up, from least significant to most significant bit, into 23 bits for the mantissa field, 8 bits for the exponent field, and 1 bit for the sign bit.

point—the most common precision used for NN training on GPUs—, 32 total bits are given, in which 8 bits are allocated for the exponent, 23 bits are allocated for the mantissa, and 1 bit is allocated for the sign bit, as seen in Figure 1. Depending on the value in the exponent field, there are three ways to interpret the values encoded in the exponent and mantissa bit fields (the sign bit is always interpreted as 0 or 1): 1) normalized values, 2) denormalized values, and 3) special values.

2.1.1 Normalized Values. When the exponent field E is neither 0 (all zeroes) nor 255 (all ones), we are working with normalized values. In this case, the bit fields are interpreted as follows:

- $E = x - Bias$, where x is the unsigned integer actually stored in the exponent bit field, and $Bias$ is $2^8 - 1 = 127$ (since we are given 8 exponent bits). Thus, $-126 \leq E \leq 127$.
- $m = 1 + f$, where f is the fractional binary value actually stored in the mantissa bit field. By fractional, we mean $0 \leq f < 1$. Essentially, f is the fractional number to the right of the radix point. Thus, $1 \leq m < 2$. We implicitly add 1 to gain an extra free bit of precision.

Based on these definitions, we compute normalized values n from the bit fields as

$$n = (-1)^s \times (1 + f) \times 2^{x-127} \tag{2}$$

As seen in Equation (2), the floating point number is effectively scaled based on the value x stored in the exponent field. This allows the radix point to "float" to various positions in the binary value as needed, providing floating point with a large range and thus more precision.

2.1.2 Denormalized Values. When the exponent field E is all zeroes, we are working with denormalized values. In this case, the bit fields are interpreted as follows:

- $E = 1 - Bias$, where $Bias$ is still $2^8 - 1 = 127$, so $E = -126$.
- $m = f$, where f is the fractional binary value stored in the mantissa bit field, as in the normalized value case.

Based on these settings, we compute denormalized values m from the bit fields as

$$n = (-1)^s \times f \times 2^{-126} \tag{3}$$

2.1.3 Special Values. When the exponent field E is all ones, we represent special non-numerical values. For completeness, we include these special values. The value is determined by the bits residing in the mantissa bit field:

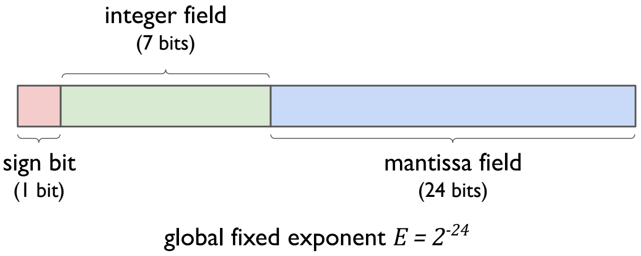

Figure 2: Example 32-bit fixed point format, where, from least significant to most significant bit, 24 bits are chosen for the mantissa, 7 bits are chosen for the integer, and 1 bit is for the sign bit. The value of the global fixed exponent depends on the number of bits allocated for the mantissa. In this case, there are 24 mantissa bits, implying that the global fixed exponent is 2^{-24}.

- $m = 0$. Depending on the sign bit, this value is $-\infty$ or $+\infty$.
- $m \neq 0$. This is NaN, also known as Not a Number.

Based on Equations (2) and (3), we see that there is extra arithmetic on fractional binary values involved with floating point values. Even though much of this arithmetic has been optimized, there is still an overhead cost that needs to be paid when using floating point values; case in point, a floating point unit is required, making floats significantly more expensive to compute in hardware than say integers, which have no such overhead. Moreover, there are as many normalized values ($n \geq 1$) as there are denormalized values ($0 \leq n < 1$), meaning floating point numbers are non-uniformly spaced. There are many more values represented between 0 and 1 such that floating point has higher precision for values that lie in this range. This has many implications for quantization when quantizing from floating point to integer. For instance, if a network has many high-precision values between 0 and 1 and relies on this precision, this presents an extra challenge for quantizing to a less precise datatype like integers.

2.2 Fixed Point

Fixed point is similar to floating point in that it is also encoded using a sign bit and a mantissa, but it uses a single global fixed exponent value that is shared across all fixed point values. Since the exponent is fixed, there is no need to store it in the binary, and the remaining bits are allocated as the integer field. Figure 2 depicts an example 32-bit fixed point value that is broken up into 24 mantissa bits, 7 integer bits, and 1 sign bit. The global exponent effectively places the radix point at a fixed position; hence, "fixed point." The shared exponent is also referred to as a global scaling factor and is typically a power of 2, so that the scaling multiplication can be implemented using bit shifts [7]. Based on the fixed point definition in Figure 2, given 24 mantissa bits, the global scaling factor is 2^{-24}.

Depending on the application, we might want more or less precision in our fixed point values. For instance, Vivado High-Level Synthesis defines a fixed point datatype for its FPGA synthesis called `ap_fixed`$< T, I >$ that allows users to choose the fixed point format, where T is the total number of bits allocated for the fixed point number and I is how many integer bits are allocated [2]. Since the global exponent is fixed and is based on how many bits are allocated to the mantissa, there is a trade-off involved in choosing T and I. Fewer integer bits means there is more bits left for the mantissa, implying high precision but a small range. More integer bits means there are fewer bits left for the mantissa, leading to low precision but a larger range.

Fixed point is typically used in embedded systems that cannot afford to have a floating point unit but still need some precision in their computations [7]. Without a floating point unit, the fixed point bit fields are all interpreted as integers (as opposed to fractional binary values like the floating point mantissa is), which are much cheaper to compute on. To explain how this works, let us walk through an example. Based on the fixed point format depicted in Figure 2, let us encode $\frac{1}{3}$. Since our value is a fraction, we do not need any integer bits, so this field is all zeroes. For the mantissa field, given 24 bits and a global exponent of 2^{-24}, we store the integer 5,592,405 in the mantissa field because $5592405 \times 2^{-24} \approx \frac{1}{3}$. This can be derived for any real number fraction r and m mantissa bits, where the corresponding fixed point mantissa value is $\lfloor r \cdot 2^m \rfloor$ [12]. As such, to determine based on the sign bit, integer field, mantissa field, and global exponent the fixed point value n, we compute

$$n = (-1)^s \times (I + m \times E) \tag{4}$$

where s is the sign bit, I is the integer encoded value, m is the mantissa encoded value, and E is the global exponent. Although this equation looks similar to the floating point equations, note that all of the values are integers and E is typically a power of 2, so all the computation can be performed using integer arithmetic and bit shifts, which is significantly cheaper than floating point arithmetic.

3 QUANTIZATION EXAMPLE

To quantize a value, we must follow a quantization function that systematically maps high precision values to low precision values.

3.1 Quantization

A common quantization function $Q(r)$ maps a floating point value r to an integer by way of

$$Q(r) = Int(r/S) - Z \tag{5}$$

where S is a floating point scaling factor and Z is an integer zero point [14]. By zero point, we mean Z is the integer value that represents 0 in our quantization scheme, which could be 0 or another value. In our scope of quantizing NN for efficient inference, r is either a weight or activation. The $Int(\cdot)$ function assigns the scaled r to an integer, typically via rounding. The rounding function can be as simple as round-to-nearest or something more complex, as seen in [25, 30]. In this case, $Q(r)$ is an example of *uniform quantization* because all of our r input values are scaled by the same value S, implying that the distance between quantized values is equally spaced and thus uniform. It is possible to define non-uniform distances between quantized values, which is known as *non-uniform quantization*; however, this is out of scope.

3.1.1 Choosing a Scaling Factor. Choosing a scaling factor in a way that minimizes accuracy loss is non-trivial. The scaling factor plays a large role in quantization because, as previously discussed, the scaling factor determines the distance between quantized values, i.e., the step size. The scaling factor is defined as

$$S = \frac{\beta - \alpha}{2^b - 1} \tag{6}$$

where β is the upper bound and α is the lower bound of the range of quantized values, and b is the quantization bitwidth. $[\alpha, \beta]$ is also known as the *clipping range*.

Determining the clipping range is known as *calibration*. When $\alpha = -\beta$, we are employing *symmetric quantization*. Using symmetric quantization means the zero point Z is 0, which is computationally less expensive at inference time, especially since the quantization function is now merely $Q(r) = Int(r/S$. While symmetric quantization is more inexpensive, it can occur at the cost of accuracy, in particular for NNs that have an imbalance in their weights or activations. This is apparent in NNs that use ReLU activations, a common activation function, that results in only non-negative activation values. In response, we can select a clipping range such that $\alpha \neq -\beta$ that better reflects the imbalance of our weights and/or activations. This is known as *asymmetric quantization*. More information on calibrating clipping ranges can be found in [14].

3.2 Dequantization

To go from the quantized value to its original real value, we use a dequantization function, which does the reverse of the quantization function. Dequantization is defined as

$$\hat{r} = S \cdot (Q(r) + Z) \tag{7}$$

Note that $\hat{r}$ is not guaranteed to equal the original value r because $Q(r)$ employs a rounding function. The bias introduced by the rounding function introduces some error that is lost and cannot be recovered by the dequantization function.[1]

4 QUANTIZATION PROCEDURES

While it is possible to train a quantized NN from scratch, the majority of research has shown that starting with a pre-trained model to be more effective at minimizing accuracy loss when quantizing a NN. There exists two main methods of quantizing pre-trained NNs: quantization-aware training (QAT) and post-training quantization (PTQ). QAT involves quantizing a NN and then retraining it so that it has a chance to adjust and learn according to the newly quantized values. Sometimes a sufficient amount of training data is not readily available, so we use PTQ, which entails quantizing a NN without any extra training. Figure 3 gives an overview of QAT and PTQ.

4.1 Quantization-Aware Training

Quantization-Aware Training involves retraining a model with quantized parameters so that it can learn and correct any quantization bias that often results from rounding errors. To perform QAT, we must

(1) quantize the weights,

[1] Some quantization work has focused on accounting for this rounding error by introducing bias into the NN's parameters [13, 40].

Table 1: A qualitative comparison of the two main quantization procedures

Quantization Procedure	Accuracy Loss	Training Time	Achievable Precision
Quantization-Aware Training	Negligible	High	≥ 1 bit
Post-Training Quantization	Moderate	Low	≥ 4 bits

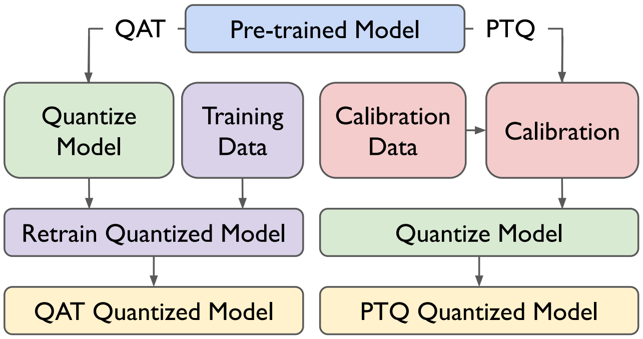

Figure 3: How to quantize a pre-trained model via either Quantization-Aware Training (QAT) or Post-Training Quantizatino (PTQ) [14]. Calibration data can be either a subset of training data or a small set of unlabelled input data. Refer to Section 3 to review calibration.

(2) perform a forward training pass through the model using floating point inputs and activations,
(3) perform a backward pass through the quantized model using floating point gradients,
(4) update the weights using the floating point gradients, and proceed back to step (1).

until the NN converges.

Regarding step (3), we note that the weights are quantized using a quantization function like the one defined in Equation (5), which is non-differentiable. Thus, it is necessary to approximate the gradient. A popular heuristic used for this approximation is called the Straight-Through Estimator (STE) [3]. The STE approximates the non-differentiable parts of the quantization function using the identity function. While it is not clear why the STE works, it has been empirically shown to be effective, except for extreme bitwidths (e.g., quantizing to 1-bit binary values).

In addition to correcting the quantization bias in weights, other quantization parameters can be learned during QAT, such as the clipping range α and β as defined in Equation (6). For instance, Parameterized Clipping Activation (PACT) [6] learns the activation clipping ranges for each activation layer during QAT.

The main trade-off of the high accuracy afforded by QAT is the long training time, e.g., hundreds of epochs, and the associated computational retraining cost. Moreover, a sufficient amount of training is required to prevent ove-fitting. Nevertheless, this long training time is often worth it for models that will be deployed for long periods of time, wherein the hardware and energy efficiency gains more than make up for the retraining cost.

4.2 Post-Training Quantization

QAT not possible when the training data is too sensitive or unavailable at time of deployment. At times like these, Post-Training Quantization is an attractive option for fast quantization. Oftentimes, PTQ uses a small set of calibration data, such as unlabelled input data, to help with choosing the best quantization parameters, e.g., scaling factor. In the past, Post-Training Quantization penalized all quantization errors equally, which is less than ideal because some quantization errors contribute more to altering classification than others. As a result, researchers turned towards QAT to fix the error introduced by quantization by retraining the entire model, as previously discussed. Recently, however, people have been revisiting PTQ in an attempt to quantize in a smarter way when there is limited data available.

To apply PTQ to a NN, we typically

(1) calibrate the quantization parameters based on any available calibration data, and then
(2) quantize the model.

When there is a lack of training data and fast quantization is needed, PTQ is usually the best option. This is often at the cost of achievable precision in that at least 4 bits are required [31]. Even with 4 or more bits, PTQ tends to be less accurate than QAT, so it is the less popular option because with the same bit precision, QAT achieves higher accuracy. Table 1 summarizes these trade-offs.

5 INTEGER/FIXED POINT QUANTIZATION

As described in Section 2, fixed point values are computed using integers, so they cost relatively the same to execute as plain integers do, give or take a few extra bit shifts/multiplications to take care of the global scaling factor. To this end, we consolidate integer and fixed point quantization in the same section.

Since the goal if NN quantization is to reduce the precision of NN implementations, a natural first step is to see if lower bitwidth integer or fixed point values are sufficient for accurate classification because they are cheaper and faster in hardware than floating point is. Quantizing NNs from floating point values to integer was studied as a early as in the 1990s [19, 22, 23]. [23] showed that for the simpler networks at the time, it was possible to quantize to 8-bit integer with minimal accuracy loss. We have seen a continuation of that result in more recent work [26, 42], with researchers now pushing the envelope and quantizing to as low as 2-bit integers [13, 31, 42].

In the early 2010's, [7, 18] explored training NNs with fixed point datatypes. While these works were focused on training, it is worth noting that [7] experimented with training fixed point as well as *dynamic fixed point* [37]. Dynamic fixed point attempts to meet floating point and fixed point in the middle. Recall from Section 2 that floating point encodes a unique exponent for each value, whereas fixed point shares a global exponent across all values.

Dynamic fixed point employs several scaling factors that are shared among a group of values, and these scaling factors are dynamically updated as the statistics of the group changes. On Maxout networks [17], the authors show that 10 bits were sufficient for minimal accuracy loss when using dynamic fixed point, compared with the 20 bits that were necessary with regular fixed point. Although they were able to reduce the bit precision to 10 bits, dynamic fixed point is quite expensive to implement in hardware because the shared exponents must all be updated every so often. This extra dynamic overhead is similar to that found in *dynamic quantization* and does not lend itself well to hardware-efficient inference. Therefore, plain fixed point/integer is preferred.

[26] provides a QAT framework for integer NN quantization and is the basis for TensorFlow Lite [1], which targets NN inference on mobile and edge devices. During the normal training procedure in floating point, the quantization function is applied to the weights and activations to simulate quantized NN inference. Using this method, they quantize the weights to 8 bits, whereas the biases are quantized to 32 bits. For each array of weights, they learn the quantization scaling factor (Equation 6) and zero point (Equation 5) during QAT. Based on these learned quantization parameters, they infer what the best quantization parameters are for the activation arrays. They also make use of "folding" the batch normalization (BN) parameters (when they are present) into the weights before applying the quantization function during training. The idea is to combine the BN parameters with the weight parameters, consolidating the two arrays into one parameter array to reduce multiplications. This way the input data is multiplied with the weights and the batch normalization parameters all at once rather than with each of the two arrays separately. To compute a folded batch normalization weight w_{BN}, do

$$w_{BN} = \frac{\gamma w}{\sqrt{\sigma_B^2 + \epsilon}} \tag{8}$$

where w is the original weight and γ, σ_B, and ϵ are the BN parameters. Doing so, in effect, "pre-computes" BN, reducing the number of multiplications at inference time. Therefore, the weights' quantization parameters reflect the effects of BN folding for more efficient inference.

Similar to [26], [16] also fuses BN with the weights, taking the approach further by fusing BN with the biases, if they are used. A BN-fused bias b_{BN} is computed as

$$b_{BN} = (\frac{\gamma}{\sqrt{\sigma_B^2 + \epsilon}})(b - \mu) + \beta \tag{9}$$

where μ and β are additional BN parameters affecting the bias. A major difference with [26], however, is that [16] is a PTQ technique. They use calibration data to compute the quantization scaling factors for the weights, activations, and biases. By passing the calibration data through the network, the authors record what the maximum absolute value MAX of each array was. The scaling factor S is then set to

$$S = \frac{MAX}{2^b - 1} \tag{10}$$

where b is the desired bit precision. The quantize the weights and activations so that they fit in 8-bit integers and biases so that they fit in 32-bit integers. Since they assume ReLU activations, which are non-negative, they choose $b = 8$ for activations, $b = 7$ for weights, and $b = 31$ for biases—weights and biases to save one bit for the sign bit. They report accuracy losses of less than 1%.

Another integer PTQ procedure called Loss-Aware Post-Training Quantization (LAPQ) [31] reduces the achievable bitwidth further to 4 bits. The authors identify a problem with prior PTQ methods: choosing the scaling factor for each tensor *independently* when there is a quantization error dependency between layers. In essence, the quantization noise of one layer affects the next one and the one after, snow balling to the end of the network. This is particularly detrimental at lower bitwidths (2-4 bits). Therefore, LAPQ optimizes the quantization parameters of a layer with the other layers in mind. The authors view quantization as a multivariate optimization problem. By combining traditional layer-by-layer PTQ with their multivariate optimization problem, LAPQ achieves better accuracy results for 4-bit weight and activation quantization than prior work at the time, though LAPQ's 8-bit weights and 4-bit activations are better at maintaining accuracy (around 2% accuracy loss). Note that they do not quantize the first and last layers whatsoever, which is a major limitation, especially for target hardware that lack floating point units.

[30] is another PTQ method, dubbed AdaRound, that quantizes NNs to 4-bit integers with around 1% accuracy loss. In this work, the authors show that for PTQ, rounding-to-nearest is not optimal and is a primary cause for the propagation of quantization/rounding errors through the network. As such, they pose rounding from floating point to integer as an optimization problem that, similar to LAPQ, takes into account the quantization errors from the preceding layers. This way AdaRound minimizes the accumulation of quantization errors that is especially apparent and incurs high accuracy loss in deeper networks. Thus, AdaRound adapts the rounding for each weight layer to the statistics of its input data with only a small amount of unlabelled calibration data; however, AdaRound only quantizes weights, when the remaining network parameters need to be quantized as well for hardware-efficient deployment.

Given the limitations of PTQ, in particular the lack of access to training data, Quant-Noise [13] seeks to achieve 8-bit and 4-bit integer quantization via QAT; in fact, they incur minimal accuracy losses when training from a quantized model from scratch, i.e., they do not need to start with a pre-trained model. Their unique QAT procedure involves choosing a quantization method and applying it to a random *subset* of weights during each forward pass of training so that that some gradients that are unbiased by the STE are allowed to update the weights. In effect, they are introducing the noise produced by the chosen quantization method randomly and incrementally during training. This way the network is allowed to adjust to the quantization noise, minimizing the quantization error associated with the use of STE during QAT, which becomes especially apparent when quantizing to fewer than 8 bits.

The idea of incremental quantization is taken even further in [42], in which they apply QAT in several stages. The authors introduce three methods of quantizing weights and activations:

(1) *Two-Stage Quantization*: First, quantize the weights only, training them until they reach a sufficient accuracy. Then, do the same for the activations.

(2) *Progressive Quantization*: Progressively decrease the bitwidths for each QAT run. For instance, to quantize 32-bit floating point to 2-bit integer, first apply QAT to the 32-bit floating point model to quantize it to 16-bit integer. Then apply QAT to the 16-bit integer model to quantize it to 8-bit integer, continuing progressively by each lower power of 2 until the network is quantized to 2 bits.

(3) *Guided Quantization*: Jointly train the full-precision model with the low-precision model based on knowledge distillation [21], a (re)training process wherein both models learn from each other.

These methods can be used independently or in combination. In fact, when they are all used together, 4-bit integer AlexNet and ResNet50 on ImageNet marginally outperform their 32-bit floating point counterparts. Even more, their 2-bit integer AlexNet on CIFAR-100 incurs less than 1% accuracy loss.

6 BINARY AND TERNARY QUANTIZATION

In the previous section, we reviewed [42], which showed results for quantizing NNs to 2-bit integers. When a network reaches such low bitwidths, researchers realized that they could limit the representable values to take advantage of bit-wise operations, e.g., XNOR, which are extremely cheap to compute in hardware. NNs quantized in this manner fall into the category of binary and ternary quantization. Binary quantization involves 1 bit of precision, where the two representable values are typically $\{-1, +1\}$. Given these values, it is possible to compute nearly all of a NN's computations using addition, subtraction, logical bitshifts, and simple bit-wise operations. Such low precision often incurs high accuracy loss though, so researchers developed ternary networks, introducing a second bit of representation. With 2 bits of precision, it becomes possible to also represent 0, so a ternary network's values belong to $\{-1, 0, +1\}$, helping to recover some of the accuracy loss. Although it is possible to represent four values with ternary networks, this limited range allows for the use of bit-wise operations, which at the hardware-level is even lower-cost to execute than integer arithmetic.

6.1 Binary Quantization

BinaryConnect [8] brought the area of binary NNs to the forefront in 2015. The idea behind BinaryConnect is to constrain the weights to $\{-1, +1\}$ during the forward and backward training passes. [2] The weights are stochastically binarized, in which weight w is binarized to w_b via

$$w_b = \begin{cases} +1 & \text{with probability } p = \sigma(w) \\ -1 & \text{with probability } 1 - p \end{cases} \tag{11}$$

where σ is the "hard sigmoid" function:

$$\sigma(x) = clip(\frac{x+1}{2}, 0, 1) = max(0, min(1, \frac{x+1}{2})) \tag{12}$$

Their experiments found stochastic binarization to be more effective than deterministic binarization, where w_b is +1 if $w \geq 0$ and -1 otherwise. It is not immediately clear why binarization works, though the authors suggest that binarization acts as a regularizer by adding noise while maintaining the expected value of what the original weight would be at the end. Regularizers are typically used in NN training to reduce over-fitting and improve generalization. Although the goal of BinaryConnect is to speed up training using specialized hardware, it has the added benefit that the resulting binary NN has is more efficient during inference because a large share of the multiply-accumulate operations can be done via simple additions and subtractions; though, the remaining non-binarized parts of the network, such as the activations, still require floating point operations.

[24] introduces BinaryNet, which binarizes activations in addition to the weights to $\{-1, +1\}$ according to Equation 11. To simplify training-time computation, they use a deterministic binarization for the weights because they found it to have negligible impact on accuracy. Given real weight w, we binarize it to w_b via the following the deterministic binarization function:

$$w_b = Sign(w) = \begin{cases} +1 & \text{if } w \geq 0 \\ -1 & \text{otherwise} \end{cases} \tag{13}$$

With binarized weights and activations, the network can replace multiplications with XNOR and bit-counting operations in the multiply-accumulates as follows

$$a'_b \mathrel{+}= popcount(XNOR(a_b, w_b)) \tag{14}$$

where a_b is the binarized input activation, w_b is the binarized weight, and a'_b is the output activation. The $popcount(\cdot)$ function counts the number of 1-bits. This equation is functionally equivalent to $a'_b + = w_b \cdot a_b$, when the weights and activations are constrained to -1 and +1, while being significantly cheaper to compute than with multiplier units. In spite of these hardware-efficiency gains, they do not binarize the first layer. They argue that the first layer is often smaller than the other layers in the network, so paying this floating point cost is reasonable; however, as we have previously noted, this is not possible on hardware that lack floating point units, as is the case with many embedded systems.

[32] also applies the XNOR-popcount multiply-accumulate idea to their binary networks they call XNOR-Networks. They also introduce Binary Weight Networks (BWN), which are in contrast with XNOR-Nets in that their activations are not binarized, meaning they cannot use XNOR-popcount but use simple additions instead like BinaryConnect. BWN and XNOR-Nets differ from [8, 24] because they use a different binarization function, which introduces a real-valued scaling factor. Both BWN and XNOR-Nets are binarized such that a real-valued weight w is approximated as follows

$$w \approx \alpha w_b \tag{15}$$

where $\alpha \in \mathbb{R}^+$ and $w_b \in \{-1, +1\}$, effectively binarizing the network to $\{-\alpha, +\alpha\}$. They compute w_b using the simple $Sign(\cdot)$ function defined in Equation 13 and α by taking the average of the absolute values of the weights in a given weight matrix. Since they use a real floating point scaling factor, they must multiply the scaling factor to the result the XNOR-popcount computations for each weight matrix. XNOR-Nets achieve better accuracy results

[2] The weight updates are still performed using floating point gradients to allow Stochastic Gradient Descent to work properly and minimize accuracy loss.

than BWNs, BinaryConnect, and BinaryNets do because they use a scaling factor and a non-standard layer ordering.[3]

6.2 Ternary Quantization

Despite all of these inroads made in binary NNs, they still suffer from high accuracy loss (around 10-20%) on challenging image classification tasks like ImageNet. As such, researchers have turned towards ternary NNs.

Ternary Weight Networks (TWNs) [27] build on BWNs and XNOR-Nets, as they also using a scaling factor, but this time using 2 bits of precision, limiting their weights to $\{-1, 0, +1\}$. To determine if a weight should be -1, 0, or +1, they set a threshold Δ. They frame finding the threshold as an optimization problem that aims to minimize the Euclidean distance between the ternary weights and the original floating point weights, striking a balance between binary NNs and floating point NNs. The approximate the solution to the threshold optimization problem based on the assumption that the weights are normally distributed, yielding $\Delta \approx 0.7 \cdot \mathrm{E}(|W|) \approx \frac{0.7}{n} \sum_{i+1}^{n} |w_i|$, where W is a weight matrix, and n is the size of the matrix. Details of the optimization problem can be found in the paper. Based on this threshold, they convert a floating point weight w to its ternary counterpart w_t via

$$w_t = \begin{cases} +1 & \text{if } w > \Delta \\ 0 & \text{if } |w| \leq \Delta \\ -1 & \text{if } w < -\Delta \end{cases} \tag{16}$$

They compute the scaling factor α based on the threshold, in which α is the average of the absolute values of the weights greater than the threshold. The network thus uses ternary values $\{-\alpha, 0, +\alpha\}$. Based on this ternary quantization scheme, TWNs outperform binary NNs, incurring only 4% accuracy loss with a TWN ResNet18 on ImageNet.

Rather than approximating the threshold for quantizing weights to ternary precision, Trained Ternary Quantization (TTQ) [41] learns the ternary values and threshold for ternary assignment during training. Instead of choosing one scaling factor per weight matrix (as seen in BWN, XNOR-Nets, TWNs), TTQ is free to choose two different scaling factors; one for the lower bound and one for the upper bound. This means TTQ quantizes a NN to ternary values $\{-W_l^n, 0, +W_l^p\}$ for each layer l, in which $-W_l^n$ and $+W_l^p$ are floating point scaling factors that are learned during training. Since $-W_l^n \neq +W_l^p$, TTQ is an example of asymmetric quantization. The TTQ method involves

(1) normalizing the full-precision weights to $[-1, +1]$,
(2) quantizing weights to $\{-1, 0, +1\}$ (Equation 16), and
(3) training the network.

While TWN approximates the threshold for quantizing the weights, TTQ learns the threshold during retraining, in addition to $-W_l^n$ and $+W_l^p$. By using asymmetric $-W_l^n$ and $+W_l^p$ scaling factors, TTQ NNs are more flexible and have more model capacity. Using asymmetric quantization, however, comes at the cost of performing two separate multiplications per activation, rather than say a global element-wise multiplication. To account for this cost, it is possible to design custom hardware to have these asymmetric scaling factors pre-computed for the activations. TTQ is quite impressive because their TTQ ResNets outperform floating point ResNets on CIFAR-10 by $< 0.5\%$.

7 MIXED PRECISION QUANTIZATION

The main issue with many of the previously discussed quantization schemes is that all the NN's parameters must be quantized to a single uniform precision. Although we saw some work in Section 5 choose different bitwidths for weights, biases, and activations, the bitwidths were still uniform in each type of parameter. On the one hand, with integer/fixed point quantization, the precision of the weights, for instance, was held back by the most sensitive layer that needed the most bits when there may have been less sensitive layers that did not need as many. This reduces the achievable compression rate—we could have a smaller network with the same accuracy if we allocated fewer bits to the less sensitive layers. On the other hand, with binary and ternary quantization, the NNs were limited to 1 and 2 bits of precision, causing some NNs to suffer from high accuracy loss when they could have paid a few more bits to reduce accuracy loss.

Mixed precision attempts to address these problems by combining the previous quantization schemes together. Mixed precision takes the idea of using different bitwidths for weights, biases, and activations even further, lowering the quantization bitwidth granularity to the layer-level. In effect, each layer has a tailored bitwidth and precision because mixed precision recognizes that some layers may benefit from more bits whereas others can afford to use fewer bits. Layers that are less sensitive to low-precision quantization are allocated fewer bits than the more sensitive layers. As a result, we meet integer/fixed point, binary, and ternary quantization in the middle, reducing accuracy loss at higher compression rates and more efficient inference.

The primary challenge of mixed precision is determining which bitwidths are optimal for each layer because the search space is exponential in the number of layers. Brute-force searching this space is impractical, especially for deep networks. For example, quantizing ResNet50 to mixed precision bit settings where the possible bitwidths are $\{1, 2, 4, 8\}$ has a search space of $4^{50} \approx 1.3 \times 10^{30}$ [10]. Hence, the principal challenge mixed precision research addresses is how to search this space efficiently or provide a principled framework that circumvents searching such a large space.

In [28], they use PTQ to quantize NNs to various fixed point schemes, in which they choose the optimal number of fractional bits that each fixed point precision has per layer. Recall from Section 2.2 where we discuss how the `ap_fixed`$< T, I >$ fixed point type, where T is the total number of bits and I is the number of integer bits. In [28], they are choosing the number of $T - I$ bits to use, which is the number of bits to allocate for the fractional part of the value, in effect tailoring the global fixed exponent that is used for each layer. To choose the optimal number of fractional bits, they frame finding the optimal bitwidths as an optimization problem, in which they attempt to maximize what they define as the "signal-to-quantization-noise-ratio" (SQNR). After finding the best bitwidth, they compute the fractional bits needed. SQNR is based on the distance between quantized values and their original floating

[3] They use BN-Activation-Convolution-Pool layer ordering as opposed to the standard Convolution-BN-Activation-Pool order.

point values. They assume that more quantization noise (i.e., the further away the quantized values are from their original floating point values), the more the classification accuracy will degrade. After running a sufficient amount of calibration data through their network, they collect SQNR results to determine the bitwidth and compute the number of fractional bits needed. Their results show that mixed precision outperforms their uniform precision models.

Hessian-Aware Quantization (HAWQ) [11] also provides a systematic way during QAT to determine the precision of each layer's weights and activations while maintaining or improving current state-of-the-art quantization accuracy results. They use second-order information (the second derivative, or in this case specifically the second-order partial derivative), which for matrices is called the Hessian matrix (a matrix of the second derivatives), to determine how sensitive the weights and activations are. Based on this information, they determine the minimum bitwidth each layer needs to maintain overall network accuracy. The key observation is that layers with *higher Hessian spectrum* (larger eigenvalues) have a more volatile loss. These layers are prone to more fluctuations in the loss when even a small amount of quantization noise is introduced (e.g., by rounding errors). Thus, they are more sensitive to quantization need a higher bitwidth. Layers with lower eigenvalues mean their loss is rather flat, even when larger amounts of quantization noise are introduced. With this in mind, these layers are less sensitive to quantization and can afford to have fewer bits. The idea is that a flat loss magnifies noise, e.g., quantization noise, significantly less than a region with sharper curvature in their loss. Based on this Hessian information, they manually select the bitwidths for each layer. Another key insight from HAWQ is that the order in which layers are quantized is important and affects accuracy loss. They elect to quantize layers with higher Hessian values and a larger number of parameters, which means these layers are more sensitive to noise compared with the rest, and retrain them first before quantizing the remaining less sensitive layers. They argue that quantizing and retraining the less sensitive layers first is not very effective. The less sensitive layers adjust well to the introduction of quantization noise, so it is better to “lock in” the quantized values of the more sensitive layers first, allowing the less sensitive layers to recalibrate during this time. Even if this recalibration causes the parameters to stray further away from their original floating point values, their robustness allows them to still be quantized at this point with little effect on the network's overall accuracy.

HAWQ sees two follow-ups in HAWQ-V2 [10] and HAWQ-V3 [39]. The first follow-up HAWQ-V2 improved on HAWQ by using a better sensitivity metric and automatically selecting the bitwidths for each layer. Instead of using the top Hessian eigenvalue as HAWQ does, HAWQ-V2 takes the average of all the Hessian eigenvalues of say a weight matrix to better capture how sensitive the layer is, rather than making decisions based on the layer's most sensitive parameter, i.e., its top eigenvalue. Based on this information, they take a Pareto frontier approach to automatically select bitwidth settings for each layer. Based on the average Hessian traces, they constrain the mixed precision search space and sort the candidate bitwidth settings based on their total second-order perturbation. This metric is defined as

$$\Omega = \sum_{i=1}^{L} \Omega_i = \sum_{i=1}^{L} \overline{Tr}(H_i) \cdot ||Q(W_i) - W_i||_2^2 \tag{17}$$

where L is the number of layers, $\overline{Tr}(H_i)$ is the average Hessian trace, $||Q(W_i) - W_i||_2^2$ is the L_2 norm of the distance between quantized weights $Q(W_i)$ and floating point weights W_i. They argue that the bitwidth setting with minimum total second-order perturbation will generalize to the task at hand better, thus incurring low accuracy loss. This is not the optimal bitwidth setting, but it outperforms state-of-the-art manually selected bitwidth settings.

The second HAWQ follow-up HAWQ-V3 [39] makes significant improvements in that they eliminate all floating point operations in their Hessian-aware quantization scheme. In many quantization schemes that we have previously seen, including HAWQ and HAWQ-V2, floating point scaling factors were used. In HAWQ-V3, they propose to use *dyadic numbers* as the scaling factors. Dyadic numbers are real numbers that can be represented as $b/2^c$, where b and c are both integers. With dyadic scaling factors, the scaling factor multiplications and divisions can be done via integer multiplication and bit shifting, completely eliminating the need to support floating point numbers. Moreover, they further augment their Hessian-Aware quantization by making it more hardware-aware. They use Integer Linear Programming to find a mixed precision scheme that minimizes a NN's second-order perturbation (Equation (17)) subject to limits on the model size, number of binary operations, and latency. This makes the resulting models more practical for deployment on edge devices.

AdaQuant [25] is a PTQ procedure that also uses ILP to determine their mixed precision quantization scheme, though they do not use the average Hessian trace in their optimization setup. With a small calibration dataset, they instead use the following objective to quantize each layer to its optimal precision:

$$(\hat{\Delta}_w, \hat{\Delta}_x, \hat{V}) = argmin||WX - Q_{\Delta_w}(W')Q_{\Delta_x}(X)||^2 \tag{18}$$

In this equation, $\hat{\Delta}_w$ and $\hat{\Delta}_x$ are the step sizes for the weights and activations respectively, which determine their quantization scaling factors. W is the given layer's weights, X is the layer's input activations, and $Q(\cdot)$ is the quantization function. $W' = W + V$ where V is a continuous variable to give the network some leeway during training, in which the quantized values need not be close to the original floating point values. As such, the quantized weights are defined as $W_q = Q_{\hat{\Delta}_w}(W + \hat{V})$, giving the weights some space to account for quantization rounding errors. This equation can be run in parallel for each layer. To further correct the bias introduced by quantization, they run knowledge distillation using their calibration data. They also note that the common practice of fusing the BN layers with their predecessor weight layers (Equation (2)) *before* applying PTQ is problematic, as seen in [16]. Before quantization, the BN parameters are reflecting the internal statistics of the *floating point* model, not the quantized model. Therefore, they introduce “Para-Normalization,” a method to update the BN statistics according to the newly quantized model. They run a few forward passes of the calibration data through the quantized model, collecting new BN parameter statistics, and then re-fuse these new parameters

Table 2: A qualitative comparison of binary, ternary, integer/fixed point, and mixed precision quantization schemes

Quantization Scheme	Accuracy Loss	Advantages	Disadvantages
Binary	High	**Low cost.** All arithmetic done via binary operations. 32× size compression rate.	**High accuracy loss.** Binary networks often incur around 10% accuracy reductions.
Ternary	Low - Moderate	**High compression rate.** Multiplications done via binary operations or capped at two multiplications per activation if using asymmetric scaling factors. 16× compression rate.	**Floating point arithmetic**. For negligible accuracy loss, ternary networks use asymmetric floating point scaling factors, so they need to perform two floating point multiplications per activation.
Integer/Fixed Point	Low	**Integer arithmetic.** All arithmetic done via integer arithmetic, which is much cheaper than floating point arithmetic.	**Uniform precision.** For minimal accuracy loss, the networks are limited to the bitwidth of most sensitive layer, which is often 8 bits, so the compression rates are at most 4×.
Mixed Precision	Low	**Custom precision.** Quantization scheme for each layer or even row of weights is tailored to their precision sensitivity, reaping the benefits of binary, ternary, and integer quantization.	**Large search space.** The search space for which quantization scheme to use for each layer or weight row is exponential in the number of layers or weight rows, respectively.

into the weights, biases, and $\hat{\Delta}_w$ to adjust the weights' quantization scaling factor accordingly.

While the previously discussed work [10, 11, 25, 28, 39] takes a more systematic approach, others [15, 35, 36, 38] leverage machine learning to address the challenge of mixed precision's large search space.

[15, 36] are more heavy-handed in their approaches. Hardware-Aware Quantization (HAQ) [36] uses reinforcement learning that takes hardware simulator results on latency and energy into account to satisfy the given resource constraints to find the optimal mixed precision bitwidth. [15] searches for the optimal combination of NN architecture and quantization scheme by adding quantization as a search parameter during Neural Architecture Search (NAS). NAS involves automating the creation and search for new NN topological structures that outperform hand-designed ones. In this work, the authors claim that the optimal bitwidths should be correlated with the architectures, so they should be search in conjunction to find more accurate and energy-efficient models. The search starts with MobileNetV2 [33] as a base model and $\{2, 4, 6, 8\}$ as the possible bitwidths. They also use an energy simulator to obtain energy metrics for the NN models designed during NAS to guide their search. Their search algorithm finds that their mixed precision models achieve lower energy, lower latency, and lower accuracy loss than uniform precision quantization does. They also show that keeping NAS and quantization as separate processes yields models that are perform worse than their combined NN+Quantization search with respect to accuracy, model size, and energy efficiency.

[35, 38] take a more traditional QAT approach when finding the best mixed precision schemes by learning the best mixed precision quantization parameters during QAT. [35] claims that learning the quantization function's parameters is possible if a good parameterization is chosen during training. They ascertain that a good parameterization to learn during training include the step size and dynamic range of the the quantizer, whereas learning the bitwidth itself performed worse. Instead, the bitwidth is inferred from learned step size and dynamic range. This method learns the step size and dynamic range for each layer, leading to a mixed precision quantization. The authors also note that starting with a pre-trained floating point model outperforms starting from scratch with a random weight initialization. Bit-level Sparsity Quantization (BSQ) [38] also uses a more standard QAT framework; however, the lower the granularity of quantizing at the layer level to the bit level. In their work, the authors treat each bit used to represent each weight as an independent variable, forcing some bits to 0 to induce sparsity and lower bitwidths. BSQ induces negligible accuracy loss while achieving higher compression rates compared to previous mixed precision quantization methods, such as HAWQ [11], though they keep the precision of the first and last layers fixed at 8 bits.

8 FUTURE DIRECTIONS

Plenty of work has been done in quantization, as we have surveyed in this paper. As seen in Table 2, the quantization schemes we have discussed each have their advantages and disadvantages. Additionally, Tables 3 and 4 present overviews of the accuracy and

Table 3: Summary of various quantization methods on the CIFAR-10 dataset. In Precision (w/a), w is the number of weight bits and a is the number of activation bits. Bit-wise ops = Bit-wise operations. MP = Mixed precision.

Quantization	Method	Model	Acc. (%)	Precision (w/a)	QAT/PTQ?	Bit-wise ops.	Int Arith.	Float Arith.
Int/Fixed Point	[7]	Maxout	84.02	20/20	QAT		✓	
Binary & Ternary	BinaryConnect [8]	CNN	91.73	1/float32	QAT			✓
	BNN [24]	CNN	88.6	1/1	QAT	✓		✓
	TWN [27]	VGG-7	92.56	2/float32	QAT	✓		✓
	TTQ [41]	ResNet56	93.56	2/float32	QAT	✓		✓
Mixed Precision	[28]	AlexNet	93.18	8/16	QAT		✓	
	HAWQ [11]	ResNet20	92.22	MP/4	QAT		✓	

Table 4: Summary of various quantization methods on the ImageNet dataset. In Precision (w/a), w is the number of weight bits and a is the number of activation bits. Bit. ops = Bit-wise operations. MP = Mixed precision. * means this model was designed via NAS.

Quantization	Method	Model	Acc. (%)	Precision (w/a)	QAT/PTQ?	Bit. ops.	Int Arith.	Float Arith.
Int/Fixed Point	[26]	ResNet50	74.9	8/8	QAT		✓	
	Quant-Noise [13]	EffNet-B3	79.8	8/8	QAT		✓	
	[42]	AlexNet	58	4/4	QAT		✓	
		ResNet50	75.7	4/4	QAT		✓	
	[16]	ResNet50	71.84	8/8	PTQ		✓	
		Inception-V3	75.31	8/8	PTQ		✓	
	LAPQ [31]	ResNet18	68.8	8/4	PTQ		✓	
		ResNet50	74.8	8/4	PTQ		✓	
		ResNet101	73.6	8/4	PTQ		✓	
		Inception-V3	75.1	8/4	PTQ		✓	
	AdaRound [30]	ResNet18	68.55	4/8	PTQ		✓	
		ResNet50	75.01	4/8	PTQ		✓	
		Inception-V3	75.76	4/8	PTQ		✓	
		MobilenetV2	69.89	4/8	PTQ		✓	
Binary & Ternary	BWN [32]	ResNet18	60.8	1/float32	QAT			✓
	XNOR-Net [32]	ResNet18	51.2	1/1	QAT	✓		
	TTN [27]	ResNet18	61.8	2/float32	QAT	✓		✓
	TTQ [41]	AlexNet	57.5	2/float32	QAT	✓		✓
		ResNet18	66.6	2/float32	QAT	✓		✓
Mixed Precision	AdaQuant [25]	ResNet18	67.4	MP	PTQ		✓	
		ResNet50	73.7	MP	PTQ		✓	
		Inception-V3	72.6	MP	PTQ		✓	
	[28]	AlexNet	80	MP	QAT		✓	
	HAWQ [11]	Inception-V3	75.52	MP	QAT		✓	✓
		ResNet50	75.48	MP	QAT		✓	✓
	HAWQ-V2 [10]	Inception-V3	75.68	MP	QAT		✓	✓
		ResNet50	75.76	MP	QAT		✓	✓
	HAWQ-V3	ResNet18	70.5	MP	QAT		✓	
		ResNet50	75.95	MP	QAT		✓	
	[15]	MobilenetV2*	71.77	MP	QAT		✓	
	HAQ [36]	MobilenetV2	71.89	MP	QAT		✓	
	[35]	MobilenetV2	70.59	MP	QAT		✓	
		ResNet18	70.66	MP	QAT		✓	
	BSQ [38]	ResNet50	75.29	MP	QAT		✓	
		Inception-V3	76.6	MP	QAT		✓	
Mixed Scheme	[5]	ResNet18	70.27	4/4	QAT	✓	✓	
		MobilenetV2	71.31	4/4	QAT	✓	✓	

arithmetic the hardware is required to support. Based on this information, researchers can decide which quantization schemes best suit their deployment requirements.

Nevertheless, there is plenty of opportunity for improvement to make NNs more easily deployed to hardware. Not a lot of work has focused on fixed point quantization and finding the optimal number of fractional bits to use for each layer [28]. This might be because the fixed point datatype is mainly used in the embedded systems space and few other fields of computer science. This is one sub-field of mixed precision quantization that could be further studied, especially considering executing fixed point datatypes in hardware is on par in terms of hardware-efficiency with executing integers. The main challenge is that the search space for how many fractional bits to use is large and also exponential in number of layers if we want to tailor it to each network. This challenge is the same challenge of mixed precision quantization, so it would be fruitful to apply methods of searching the mixed precision space to searching the fixed point precision space.

Additionally, little work has been done on combining the approaches together. While mixed precision does assign different bitwidths for each layer, they are usually still all of the same datatype. With custom hardware, it is possible to group layers together based on the best quantization scheme for them for more efficient hardware deployment. The key challenge here is to balance the complexity of the hybridized quantization schemes with the complexity it would take to implement them in hardware. It would be less than ideal to formulate a highly optimized hybrid quantization scheme whose implementation overhead outweighs the potential efficiency gained in hardware. [5] has shown results for combining their novel hardware-friendly quantization scheme called *sum-of-power-of-2* with fixed point quantization—both these schemes are efficient to implement using FPGA Digital Signal Processing units (DSPs). Since they claim to be the first paper to combine quantization schemes, they dub this process *mixed scheme quantization*. The optimal combination is learned during training. Thus, [5] shows promise for this research direction.

In addition to combining quantization schemes together, there is also ample opportunity to explore combining quantization with other NN compression techniques, such as NAS, knowledge distillation, and pruning. Deep Compression [20], Quantization-Aware Pruning [34], and TTQ [41] have shown results in combining fixed point and ternary quantization with pruning. Distillation-assisted quantization is also an emerging field [29, 39]. But, there are more quantization schemes that could benefit from combining with other compression techniques. Moreover, there is little work on what the optimal combination of compression techniques is.

NN quantization is a well-studied topic with many degrees of freedom with plenty of future directions for the field. The greatest advantage of quantization is the efficiency manifested in hardware. With more models that become deployable on hardware and edge devices, the greater the impact of NNs will be and the more humanity will reap its benefits.

REFERENCES

[1] [n.d.]. TensorFlow Lite. https://www.tensorflow.org/lite

[2] 2012. Vivado design suite user guide. https://www.xilinx.com/support/documentation/sw_manuals/xilinx2012_2/ug902-vivado-high-level-synthesis.pdf

[3] Yoshua Bengio, Nicholas Léonard, and Aaron Courville. 2013. Estimating or Propagating Gradients Through Stochastic Neurons for Conditional Computation. *arXiv:1308.3432 [cs]* (Aug. 2013). http://arxiv.org/abs/1308.3432 arXiv: 1308.3432.

[4] Randal E. Bryant and David R. O'Hallaron. 2011. *Computer systems: a programmer's perspective* (2nd ed ed.). Prentice Hall, Boston. OCLC: ocn457156657.

[5] Sung-En Chang, Yanyu Li, Mengshu Sun, Runbin Shi, Hayden K.-H. So, Xuehai Qian, Yanzhi Wang, and Xue Lin. 2021. Mix and Match: A Novel FPGA-Centric Deep Neural Network Quantization Framework. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, Seoul, Korea (South), 208–220. https://doi.org/10.1109/HPCA51647.2021.00027

[6] Jungwook Choi, Zhuo Wang, Swagath Venkataramani, Pierce I.-Jen Chuang, Vijayalakshmi Srinivasan, and Kailash Gopalakrishnan. 2018. PACT: Parameterized Clipping Activation for Quantized Neural Networks. *arXiv:1805.06085 [cs]* (July 2018). http://arxiv.org/abs/1805.06085 arXiv: 1805.06085.

[7] Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. 2014. Training deep neural networks with low precision multiplications. *arXiv preprint arXiv:1412.7024* (2014).

[8] Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. 2015. Binaryconnect: Training deep neural networks with binary weights during propagations. In *Advances in neural information processing systems*. 3123–3131.

[9] Misha Denil, Babak Shakibi, Laurent Dinh, Marc'Aurelio Ranzato, and Nando de Freitas. 2014. Predicting Parameters in Deep Learning. *arXiv:1306.0543 [cs, stat]* (Oct. 2014). http://arxiv.org/abs/1306.0543 arXiv: 1306.0543.

[10] Zhen Dong, Zhewei Yao, Yaohui Cai, Daiyaan Arfeen, Amir Gholami, Michael W. Mahoney, and Kurt Keutzer. 2019. HAWQ-V2: Hessian Aware trace-Weighted Quantization of Neural Networks. *arXiv:1911.03852 [cs]* (Nov. 2019). http://arxiv.org/abs/1911.03852 arXiv: 1911.03852.

[11] Zhen Dong, Zhewei Yao, Amir Gholami, Michael Mahoney, and Kurt Keutzer. 2019. HAWQ: Hessian AWare Quantization of Neural Networks With Mixed-Precision. In *2019 IEEE/CVF International Conference on Computer Vision (ICCV)*. IEEE, Seoul, Korea (South), 293–302. https://doi.org/10.1109/ICCV.2019.00038

[12] Christer Ericson. 2005. Chapter 11 - Numerical Robustness. In *Real-Time Collision Detection*, Christer Ericson (Ed.). Morgan Kaufmann, San Francisco, 427–463. https://doi.org/10.1016/B978-1-55860-732-3.50016-4

[13] Angela Fan, Pierre Stock, Benjamin Graham, Edouard Grave, Remi Gribonval, Herve Jegou, and Armand Joulin. 2021. Training with Quantization Noise for Extreme Model Compression. *arXiv:2004.07320 [cs, stat]* (Feb. 2021). http://arxiv.org/abs/2004.07320 arXiv: 2004.07320.

[14] Amir Gholami, Sehoon Kim, Zhen Dong, Zhewei Yao, Michael W. Mahoney, and Kurt Keutzer. 2021. A Survey of Quantization Methods for Efficient Neural Network Inference. *arXiv:2103.13630 [cs]* (June 2021). http://arxiv.org/abs/2103.13630 arXiv: 2103.13630.

[15] Chengyue Gong, Zixuan Jiang, Dilin Wang, Yibo Lin, Qiang Liu, and David Z. Pan. 2019. Mixed Precision Neural Architecture Search for Energy Efficient Deep Learning. In *2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, Westminster, CO, USA, 1–7. https://doi.org/10.1109/ICCAD45719.2019.8942147

[16] Jiong Gong, Haihao Shen, Guoming Zhang, Xiaoli Liu, Shane Li, Ge Jin, Niharika Maheshwari, Evarist Fomenko, and Eden Segal. 2018. Highly Efficient 8-bit Low Precision Inference of Convolutional Neural Networks with IntelCaffe. In *Proceedings of the 1st on Reproducible Quality-Efficient Systems Tournament on Co-designing Pareto-efficient Deep Learning*. ACM, Williamsburg VA USA, 1. https://doi.org/10.1145/3229762.3229763

[17] Ian Goodfellow, David Warde-Farley, Mehdi Mirza, Aaron Courville, and Yoshua Bengio. 2013. Maxout networks. In *International conference on machine learning*. PMLR, 1319–1327.

[18] Suyog Gupta, Ankur Agrawal, Kailash Gopalakrishnan, and Pritish Narayanan. 2015. Deep learning with limited numerical precision. In *International conference on machine learning*. PMLR, 1737–1746.

[19] D. Hammerstrom. 1990. A VLSI architecture for high-performance, low-cost, on-chip learning. In *1990 IJCNN International Joint Conference on Neural Networks*. IEEE, San Diego, CA, USA, 537–544 vol.2. https://doi.org/10.1109/IJCNN.1990.137621

[20] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2015. Deep Residual Learning for Image Recognition. *arXiv:1512.03385 [cs]* (Dec. 2015). http://arxiv.org/abs/1512.03385 arXiv: 1512.03385.

[21] Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. 2015. Distilling the Knowledge in a Neural Network. *arXiv:1503.02531 [cs, stat]* (March 2015). http://arxiv.org/abs/1503.02531 arXiv: 1503.02531.

[22] M. Hoehfeld and S.E. Fahlman. 1992. Learning with limited numerical precision using the cascade-correlation algorithm. *IEEE Transactions on Neural Networks* 3, 4 (July 1992), 602–611. https://doi.org/10.1109/72.143374

[23] J.L. Holt and J.-N. Hwang. 1993. Finite precision error analysis of neural network hardware implementations. *IEEE Trans. Comput.* 42, 3 (March 1993), 281–290. https://doi.org/10.1109/12.210171

[24] Itay Hubara, Matthieu Courbariaux, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. 2016. Binarized Neural Networks. *30th Conference on Neural Information*

Processing Systems (2016), 9.

[25] Itay Hubara, Yury Nahshan, Yair Hanani, Ron Banner, and Daniel Soudry. 2021. Accurate Post Training Quantization With Small Calibration Sets. In *Proceedings of the 38th International Conference on Machine Learning*. 10.

[26] Benoit Jacob, Skirmantas Kligys, Bo Chen, Menglong Zhu, Matthew Tang, Andrew Howard, Hartwig Adam, and Dmitry Kalenichenko. 2018. Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference. In *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*. IEEE, Salt Lake City, UT, 2704–2713. https://doi.org/10.1109/CVPR.2018.00286

[27] Fengfu Li, Bo Zhang, and Bin Liu. 2016. Ternary Weight Networks. *arXiv:1605.04711 [cs]* (Nov. 2016). http://arxiv.org/abs/1605.04711 arXiv: 1605.04711.

[28] Darryl D Lin, Sachin S Talathi, and V Sreekanth Annapureddy. 2016. Fixed Point Quantization of Deep Convolutional Networks. In *Proceedings of the 33rd International Conference on Machine Learning*. 10.

[29] Asit Mishra and Debbie Marr. 2017. Apprentice: Using Knowledge Distillation Techniques To Improve Low-Precision Network Accuracy. *arXiv:1711.05852 [cs]* (Nov. 2017). http://arxiv.org/abs/1711.05852 arXiv: 1711.05852.

[30] Markus Nagel and Rana Ali Amjad. 2020. Up or Down? Adaptive Rounding for Post-Training Quantization. In *Proceedings of the 37th International Conference on Machine Learning*. 10.

[31] Yury Nahshan, Brian Chmiel, Chaim Baskin, Evgenii Zheltonozhskii, Ron Banner, Alex M. Bronstein, and Avi Mendelson. 2020. Loss Aware Post-training Quantization. *arXiv:1911.07190 [cs]* (March 2020). http://arxiv.org/abs/1911.07190 arXiv: 1911.07190.

[32] Mohammad Rastegari, Vicente Ordonez, Joseph Redmon, and Ali Farhadi. 2016. XNOR-Net: ImageNet Classification Using Binary Convolutional Neural Networks. In *Computer Vision – ECCV 2016*, Bastian Leibe, Jiri Matas, Nicu Sebe, and Max Welling (Eds.). Vol. 9908. Springer International Publishing, Cham, 525–542. https://doi.org/10.1007/978-3-319-46493-0_32 Series Title: Lecture Notes in Computer Science.

[33] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. 2018. Mobilenetv2: Inverted residuals and linear bottlenecks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 4510–4520.

[34] Nhan Tran, Benjamin Hawks, Javier M Duarte, Nicholas J Fraser, Alessandro Pappalardo, and Yaman Umuroglu. 2021. Ps and Qs: Quantization-aware pruning for efficient low latency neural network inference. *Frontiers in Artificial Intelligence* 4 (2021), 94.

[35] Stefan Uhlich, Lukas Mauch, Fabien Cardinaux, Kazuki Yoshiyama, Javier Alonso García, Stephen Tiedemann, Thomas Kemp, and Akira Nakamura. 2020. MIXED PRECISION DNNS: ALL YOU NEED IS A GOOD PARAMETRIZATION. (2020), 21.

[36] Kuan Wang, Zhijian Liu, Yujun Lin, Ji Lin, and Song Han. 2019. HAQ: Hardware-Aware Automated Quantization With Mixed Precision. In *2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. IEEE, Long Beach, CA, USA, 8604–8612. https://doi.org/10.1109/CVPR.2019.00881

[37] Darrell Williamson. 1991. Dynamically scaled fixed point arithmetic. In *[1991] IEEE Pacific Rim Conference on Communications, Computers and Signal Processing Conference Proceedings*. IEEE, 315–318.

[38] Huanrui Yang, Lin Duan, Yiran Chen, and Hai Li. 2021. BSQ: Exploring Bit-Level Sparsity for Mixed-Precision Neural Network Quantization. *arXiv:2102.10462 [cs]* (Feb. 2021). http://arxiv.org/abs/2102.10462 arXiv: 2102.10462.

[39] Zhewei Yao, Zhen Dong, Zhangcheng Zheng, Amir Gholami, Jiali Yu, Eric Tan, Leyuan Wang, Qijing Huang, Yida Wang, Michael Mahoney, et al. 2021. Hawq-v3: Dyadic neural network quantization. In *International Conference on Machine Learning*. PMLR, 11875–11886.

[40] Shuchang Zhou, Yuxin Wu, Zekun Ni, Xinyu Zhou, He Wen, and Yuheng Zou. 2018. DoReFa-Net: Training Low Bitwidth Convolutional Neural Networks with Low Bitwidth Gradients. *arXiv:1606.06160 [cs]* (Feb. 2018). http://arxiv.org/abs/1606.06160 arXiv: 1606.06160.

[41] Chenzhuo Zhu, Song Han, Huizi Mao, and William J. Dally. 2017. Trained Ternary Quantization. *arXiv:1612.01064 [cs]* (Feb. 2017). http://arxiv.org/abs/1612.01064 arXiv: 1612.01064.

[42] Bohan Zhuang, Chunhua Shen, Mingkui Tan, Lingqiao Liu, and Ian Reid. 2018. Towards Effective Low-Bitwidth Convolutional Neural Networks. In *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*. IEEE, Salt Lake City, UT, 7920–7928. https://doi.org/10.1109/CVPR.2018.00826