Quantized Neural Networks

Nurrachman Liu
rachliu@ucla.edu

Abstract

This work provides a brief overview of recent works in the quantization of neural networks, and introduces the related quantization concepts. It then explores using the Tensorflow Quantization-Aware Training methodology to quantize AlexNet, VGG16, and ResNet50 neural networks at various bitwidths. Activation quantization bitwidth is observed to play an important role in the quantizibility of neural networks at low bitwidths, especially for complex networks. We observed that enabling low-bit activation quantization, while not impacting simpler networks such as AlexNet and VGG16, dramatically impacted a complex network's ability to quantize its weights: ResNet50's minimum reachable weight quantization of 2-bits worsened to 7-bits, when activation quantization was unfixed from a constant of 8-bits and varied to below 8-bits as well.

1 Introduction

Quantized neural networks have recently been of interest in the past 6-7 years. These refer to neural networks that have had their various parameters quantized. Many types of quantized neural networks are possible, varying by different criteria such as quantization level (e.g., binary or k-bit); the parameters that are quantized (weights, activations, gradients); or how they are quantized: for example, in a single-iteration after training, versus quantizing 'in-situ' during the training iterations.

The most notable use for quantized neural networks is in the mobile domain. A quantized neural network is compact and thus saves storage and enables low-power use cases. On-device neural networks can also ensure user privacy, because data does not need to be sent to the cloud for training. Quantization also allows possibly drastic simplification of operators (in software and/or hardware), enabling smaller hardware and lower power and cost. Quantization can thus allow the device to run deep-learning use-cases in contexts where communication may not be available or reliable.

The rest of this paper is organized as follows. We first present a cursory overview of neural networks and a brief survey of the current literature on quantized neural networks. We then present a basic overview of quantization concepts. This is followed by a discussion of Tensorflow's Quantization-aware-training (QAT) methodology and package. Finally, we present results and discussion of two experiments we ran using Tensorflow QAT on three commonly used neural network models. The goal of these experiments are to observe whether common but non-trivial neural networks can be quantized in terms of weights, activations, and/or gradients, and still remain operable with high performance.

1.1 Neural Networks Overview

Neural networks [1] are networks of nodes where each node can take input from any number of previous nodes and pass its output to some next node(s). The nodes are connected such that the resulting network is a DAG. Every node performs a linear combination of its inputs, summarized as $\mathbf{wx} + \mathbf{b}$, where $\mathbf{w}$ is a vector of weights for each input component of input vector $\mathbf{x}$. $\mathbf{b}$ represents the bias and allows for affine transformations. An activation function, typically non-linear, is then applied to this value: $f(\mathbf{wx} + \mathbf{b})$, thus allowing for non-linearity in the network.

The neural network topology can be organized as layers, where each layer is a set of nodes that take input from nodes of a previous layer, and pass on their output to a set of nodes in the next layer. The first layer is referred to as the *input layer* and the final layer is the *output layer*. The layers in between are called the *hidden layers*. For example, for a neural network used for classification, a typical output layer will perform softmax classification.

Thus, taken together, all the weights of nodes in a layer can be organized as a set of row-vectors; i.e., a matrix of weights $\mathbf{W}$. This matrix thus has number of rows equal to the number of nodes in the current layer, and number of columns equal to the number of inputs it takes. The calculation at a node can thus be denoted $\mathbf{Wx} + \mathbf{b}$, where then it is assumed that every node in the current layer can see all (is connected to all) inputs $\mathbf{x}$ from the previous layer. The output value $\mathbf{Wx} + \mathbf{b}$ is therefore a vector of values with dimension equal to the number of nodes of the current layer; the set of all nodes' output values of a layer then forms the input ($\mathbf{x}$ vector) to the next layer. When the activation function is written as $f(\mathbf{Wx} + \mathbf{b})$, it is assumed to apply to every component (output) of the nodes in a layer.

Neural networks are trained (fitted) by updating the weight parameters to minimize the loss; where the loss is the difference between the network's current output and its expected output, for the given example. The backpropagation algorithm is used to do this, which is an efficient way to compute the gradients of the loss with respect to the weight parameters, in order to update the weight parameters.

Common architectures of neural networks exist, such as the Dense network, Convolutional Neural Network (CNN), Recurrent Neural Network (RNN), and Residual Neural Network (ResNet). The Dense network is the most classic architecture, and is an assembly of layers that see all outputs of the previous layer, and pass all outputs as inputs to the next layer. A typical example of such a Dense network would be the Multi-layer Perceptron (MLP). ResNets are an architecture where some layer outputs can skip over layers to connect to further downstream layers; this helps reduce the impact of vanishing gradients (a phenomena where gradients become so small after many stages, that they fail to cause useful parameter updates). Very specific designs are also made, such as AlexNet, a CNN for image classification.

1.2 Survey of the Literature

The first works in quantizing neural networks began around 2014-2015 with *directly quantized* neural networks, where parameters are simply directly quantized, and without using other more elaborate techniques [2]. For example, directly binarizing a real-valued weight using $sign()$ [3; 4]. Shortly after, most subsequent papers employed further optimization techniques for better quantization.

The existing work thus broadly comprises two classes: direct-quantization, and optimization-based quantization. The latter, optimization-based quantization, can be decomposed further by the type of optimization employed [2] (i.e., different cost functions): minimization of quantization error, learned-quantizers, minimization of network loss, and reduction of gradient error.

An example of such an optimization-based approach would be to approximate the mean of the weights using a single scalar factor α which will be multiplied with the weights in the cost function when minimizing (i.e., $J(\mathbf{B}, \alpha) = ||\mathbf{W} - \alpha\mathbf{B}||^2$ where $\mathbf{B}$ are the binarized weights to approximate the real-valued weights $\mathbf{W}$). Another example would be attempting to approximate a binary basis for the weights; this basis vector would be thus learned through optimization as well.

1.2.1 Directly-Quantized Neural Networks

These are neural networks that directly apply the quantizer on parameters and values, without applying other (optimization-based) techniques. Very few works *only* directly-quantize, with most adding further optimization techniques to jointly learn the parameters or the new parameters they introduce (such as the α in the prior example).

These thus were the initial wave of work. The first work was *BinaryConnect* [3], which directly binarized only the weights, with the premise being to achieve storage reduction. Its successor, *BinaryNet* [4] then binarized the activations in addition to the weights.

The main advantage gained by quantizing activations is the simplification of the operator pipeline (in software and/or hardware), saving energy, power, and cost. Both of these works used the $sign()$ function to binarize values, while keeping the gradients in full-precision. The advantage of quantizing gradients is further simplification in operators, while the disadvantage is loss of the precision required for successful training.

1.2.2 Optimization-Based Quantization: Minimizing the Quantization Error

This class of quantized neural networks employs optimization-based quantization that minimizes the quantization error directly; an example would be the cost function we described earlier: $J(\mathbf{B}, \alpha) = ||\mathbf{W} - \alpha\mathbf{B}||^2$, which seeks to best approximate the real-valued weights using the learned binary weights $\mathbf{B}$ together with this α scaling factor. α is thus a scalar constant multiplier that enhances the range of the binarized weights $\mathbf{B}$. This is the idea behind *XNOR-Net* [5], 2016, one of the first of these optimization-based quantization approaches.

DNN using weights +1,0,-1 [6], 2014, used a 3-level weight quantization, and trains by minimizing the quantization error, using a 2-step approach and iterative optimization. *Ternary-Weight Network (TWN)* [7], 2016, focused on weight compression, using iterative-optimization in the quantization of the 3-level ternary weights. *DoReFaNet* [8], 2017, explores quantizing all three (weights, activations, and gradients) at different combinations of bit settings. *ABC-Net* [9], 2017, estimates the real-valued weight matrix (vectorized as) $\mathbf{W}$ using a linear combination of the binary filters. *XNorNet++* [10], 2019, improved on XNOR-Net by avoiding repeated subcalculations.

1.2.3 Optimization-Based Quantization: Learned Quantizers

These works learn the parameters of their quantizers (for example, by minimizing the quantization error). For example, the ReLU activation function can be upper-bounded, and this upper-bound can be learned to optimize the quantization error.

The above method is the one proposed by *PACT* [11], 2018. It upper-bounds the ReLU activation using a constant, learnable parameter α. *LQ-Nets* [12], 2018, learns a binary basis for its quantizer, noting that the dot-product between two binary-basis quantizers can be simplified into an accumulated bitwise dot-product, with scalar constants determined by the particular learned binary bases. The bases are learned by jointly training with the NN by minimizing the quantization error.

1.2.4 Optimization-Based Quantization: Minimization of the Network Loss

While the previous approaches focused on minimizing the quantization error of layers individually, some other works look at finding the appropriate network loss function to take global context into account for binarization/quantization as well [2].

For example, *Loss-Aware Binarization (LAB)* [13], 2016, minimizes the overall loss associated with binary weights using the quasi-Newton algorithm, using information from the second-order moving average [2].

1.2.5 Optimization-Based Quantization: Reduction of Gradient Error

These works aim to have (better) quantization functions in the forward pass or backward pass, which can reduce the gradient error.

For example, *Half-wave Gaussian Quantization* (HWGQ) [14], 2017, exploits the statistics of activations to provide better approximations in forward and backwards passes.

Table 1 summarizes the main differences amongst these works.

2 Quantization Concepts

This section gives a brief overview of basic quantization concepts: the main dichotomies of quantization types (direct-quantization or technique-based); the main flows necessary to quantize (post-training or in-situ); and the main parameters of interest for quantization (weights, activations, gradients).

2.1 Main Types of Quantization Methods

As seen in the introductory survey of works, quantization methods comprise two general classes: direct and technique-based. We re-summarize them here: *Direct quantization* refers to only the application

Work	**Main Method**	**Quant. Type**	**Quant. Levels**	**Framework**	**Framework: Actively Developed?**	**Publicly Available?**
BinaryNet [4]	Direct	W, A Update: FP G,W,A	1-bit W, A	Theano, Torch7	No	Yes
DNN using -1,0,+1 [6]	Quant. Error	W, A Update: FP G, W	2-bit W k-bit A	Unknown		No
XNOR-Net [5]	Quant, Error	W, A Update: FP G, W	1-bit W, A	Torch7	No	Yes
TWN [7]	Quant. Error	W Update: FP G, A	2-bit W	Caffe	No	Yes
DoReFaNet [8]	Quant. Error	W, A, G	k-bit W m-bit A n-bit G	Tensorflow 1	No	Yes
ABC-Net [9]	Quant. Error	W, A Update: FP G	1-bit W, A 1-bit W, 32-bit A 3-5 binary weight bases	Unknown		No
XNOR-Net++ [10]	Quant. Error	W, A Update: FP G, W	1-bit W, A	Pytorch	Yes	No
PACT [11]	Learned Params	W, A, G	k-bit W, A	Tensorflow 1	No	No
LQ-Nets [12]	Learned Params	W, A	m-bit W n-bit A	Tensorflow 1	No	Yes

Table 1: Summary of prior works. (W: weights; A: activations; G: gradients; FP: full-precision)

of a quantizer function to the value(s) of interest (weight, activation, or gradient). *Technique-based quantization* refers to methods that in addition to direct quantization, employ further optimization techniques as well.

2.2 Quantization flows

Quantization flows refers to the training methods used to actually quantize the neural network. For example, training in full-precision and then quantizing the weights in a single pass. This is referred to as *post-training quantization*. It is highly inaccurate; below 8-bits, much accuracy is lost [8].

Almost all works quantize *in-situ* while training. Values (e.g., weights) are quantized in each iteration during the training itself. This flow is recommended for low-bit quantization since it is much more accurate. Due to inter-weaving the quantizing into the training, it is called *Quantization-aware training (QAT)*. This requires either a specific implementation (e.g., Tensorflow QAT) or built-in customizability (e.g., as offered by Tensorflow and Pytorch).

2.3 Parameters that are Quantized

There can be many possible parameters introduced by optimization-based methods for quantization (e.g., the α scaling factor in the cost function of *XNOR-Net*, $J(\mathbf{B}, \alpha) = ||\mathbf{W} - \alpha\mathbf{B}||^2$). This section, however, discusses the quantization of the three fundamental parameters/values of neural networks: weights, activations, and gradients. Every work describes their method for quantizing these fundamental values (and which ones they did not quantize).

2.3.1 Weights quantization

The main premise for quantizing weights is compression: storage reduction and operator-simplification, both conferring power reduction and cost savings. Quantization of weights thus refers to the quantizing of real-valued weight parameters down to some k-bit representation; the extreme being 1-bit [4].

A commonly used 1-bit quantizer is the sign function [4]:

$$\text{binarize}(r) = \text{sign}(r)$$

A commonly used k-bit quantizer is the k-bit 'rounder' (formally: uniform affine quantizer)[15]:

$$x_q = \text{quantize}_k(x) = \text{round}(\frac{x}{\Delta}) + z$$

2.3.2 Activations quantization

The second major focal point of most works is in discussing how they quantize the output of the non-linear activation function (i.e., which quantizer function used, how many bits of quantization, and the accuracy trade-offs between these two choices). Activation quantization is just the application of a quantizer function to the real-valued activation value, and so except for the quantizer, requires no other changes to the neural network. A major advantage to quantizing the activation value is that it can drastically simplify the operators in software and/or hardware.

For example, the activated-value of a typical layer is $f(\mathbf{Wx} + \mathbf{b})$; this is quantized by running it through a quantization function [8]:

$$q^{(i)} = \text{quantize}_k(f(\mathbf{W^b x^b} + \mathbf{b^b})^{(i)})$$

The superscript **b** denotes a quantized parameter. Note that it is not necessary to use the quantized values for the weights $\mathbf{W}$, inputs $\mathbf{x}$, or biases $\mathbf{b}$ in calculations. For example, not all works use the quantized parameters for the backpropagation calculations, to have higher accuracy.

The same as for the weight parameter, possible quantizers for the activation values are the sign function (1-bit) and uniform affine quantizer (k-bit).

2.3.3 Gradient Quantization

The third major focus of many works is on how (or even, *if*) they quantize the gradient. The major advantage of quantizing the gradient is that it can also simplify operators as well.

However, the gradient is the most difficult value to quantize. This is because its values are typically very small; quantizing them can thus make them become *too small*: that is, smaller than the step-size Δ of the quantizer. If this occurs, then no useful parameter updates can occur, as no change in their value is seen.

Therefore, some existing works [4; 5] use full-precision gradients for calculating updates during backpropagation (i.e., just using the framework's provided gradient directly). They may calculate the updates using this full-precision gradient but using the *quantized* version of the weights and activations, as a middle-ground between being fully-quantized and not. This still incurs the cost of having to maintain full-precision operators.

The benefit of using full-precision gradients is that this gives the best accuracy (i.e., of the parameter updates). This is observed by [8], which notes that low-bitwidth gradients (< 4-bits) cause a large accuracy reduction (5-10%), and recommends staying at ≥ 4-bit gradients.

The drawback of using full-precision gradients is that operators must remain full-precision as the upstream gradient and subsequent gradients are in full-precision.

To implement quantized gradients, one needs customizability support from the framework. Tensorflow and Pytorch offer such customizabiility. As far as we know, there is no built-in API for gradient quantization other than fully customizing.

2.4 Modeling the Quantizer in Backpropagation

Backpropagation is an essential part of the neural network training process. This requires that backpropagation through the quantization operator also be modeled. For example, if an activation value is quantized, then backpropagation through it must also be calculated.

However, the direct quantizers all share the same major problem: backpropagation through them is difficult, due to quantizers having hard-thresholds. Such functions have derivatives that are 0 almost everywhere. Thus, backpropagating through them does not allow useful parameter updates.

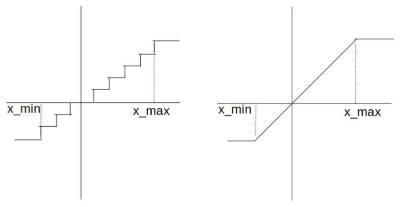
The Straight-Through Estimator (STE) [16] was proposed to solve this issue, and is cited in nearly every work as the estimator they use for modeling backpropagation through the quantizer. However, it is not any particular estimator *per se*; it stands for any estimator that is used to act as a proxy for the gradient of the quantizer, and so is often just some identity-like function to approximate the quantization as just the input signal itself (e.g., derivative of x is 1).

Since modeling backpropagation through the quantizers affects the validity of the quantization method, it is an important point to be addressed. Most works just cite the STE [16], and thus do not provide further theoretical discussion of this point. Typically, then, they just discuss *which* estimator (i.e., function) they choose to approximate the gradient of the quantizer.

2.4.1 Straight-Through Estimator (STE)

The STE is thus any estimator used as a proxy for the gradient of the quantizer; as such, the STE function chosen is often some identity-like function.

For example, the quantization of a ReLU is a staircase function. Instead of taking the quantizer's derivative directly, an STE for it is used: the gradient of the saturated ReLU, called the saturated-STE: $\frac{\partial \text{sat\_ReLU(x)}}{\partial x} = 1_{\{|x|<1\}}$.



(a) Quantized ReLU (b) Approximation of Gradient (STE) of Quantized ReLU

Figure 1: Illustration of STE for quantized-ReLU [15]

As the STE provides justification to using an identity-like function to proxy the gradient, most works thus simply pass the upstream gradient value through the quantizer in the backwards pass. Small modifications may be made to this identity pass-through, such as clipping the value (such as seen above in Figure 1 of the clipped ReLU (saturated ReLU)). Most non-analysis works observe that STEs works well empirically, but do not further elaborate on *why* they work well.

3 Tensorflow Quantization-Aware Training

This section presents a brief technical overview of the Tensorflow Quantization-Aware Training (QAT) package.

The Tensorflow QAT package was first introduced in the 2016 Google Tensorflow paper [17], which provided an open-source package for QAT into Tensorflow 1, in 2016. For Tensorflow 2, the QAT package was moved from *contribs* into the main Tensorflow public API. There have been no major changes in the QAT API between the 2018 to 2022.

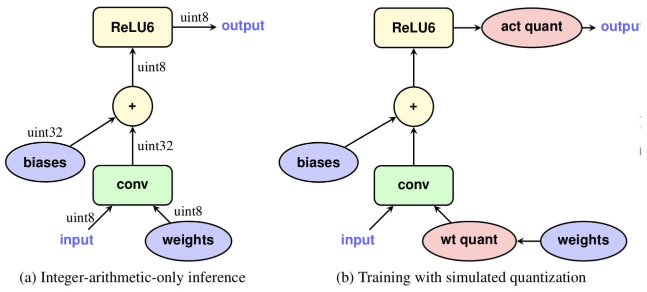


Figure 2: Run-time and Train-time Tensorflow Graphs for QAT [17]

3.1 Goals of QAT [17]

The main goal is to run the inference pipeline using integer-only operations (8-bit preferred). Upcasting to 32-bit integers happens on signals that are the products of 8-bit integer signals or for bias vectors.

QAT modifies the train-time Tensorflow-Graph by inserting the quantization-operators to *simulate* the effects of using only 8/32-bit integer signals at run-time.

At run-time, the actual quantization is simply the direct-use of integer only operators. Thus, this means the QAT package incoporates into Tensorflow, its own 'fused-layer pipeline' software implementation that employs only 8-bit/32-bit operators. Their paper provides a mathematical formulation and justification for the validity of this approach of quantizing using integer-only operations during run-time.

Thus, as the train-time and run-time flows are actually two different processes (Figure 2), these two parallel processes thus need to match as closely as possible for correctness.

3.2 Steps in QAT Flow

The QAT implementation thus follows this flow [17]. Note that this is the flow for the QAT package itself for implementing its in-situ quantization method, not for the end-user who is simply using this package.

1. Create a training graph of the floating-point model.
2. Insert fake quantization Tensorflow operations in locations where tensors will be downcasted to fewer bits during inference.
3. Train in simulated quantized mode until convergence.
4. Create and optimize the inference graph for running in a low bit inference engine.
5. Run inference using the quantized inference graph.

4 Experiments Using Tensorflow QAT

This section introduces how to use Tensorflow QAT as an end-user. It then presents the results and discussion for two experiments ran using Tensorflow QAT to quantize three common neural-networks: AlexNet, VGG16, and ResNet50. Our goal from these experiments is to observe whether common but non-trivial neural networks can be quantized in terms of weights, activations, and/or gradients, and still remain operable with high performance.

4.1 How to use QAT as an end-user

The following steps are officially listed as the way to use QAT in Tensorflow. First, train the full-precision model as usual in Tensorflow. Apply to this trained-model (referred to as the *base_model*), the QAT library's $quantize\_model()$ function: this goes into the model and wraps every neural network layer with a quantization wrapper that specifies how each layer will be quantized. A default 8-bit quantization-scheme is used in the absence of any user-specified custom quantization settings.

The quantized-model thus begins with its weights based on the full-precision model's weights. This model is then trained using Tensorflow $model.fit$; the training is now different than usual, because the layers have been wrapped (i.e., quantization operators have been inserted into their execution graphs).

4.2 Recommended usages

Tensorflow makes the following recommendations. First, to always use a full-precision model as the *base_model*, as above, since it results in the quantized model having the highest accuracies. This flow is thus the official, main flow for using Tensorflow QAT.

While using QAT, the user has some customizable control over exactly which layers they want to quantize. Tensorflow thus recommends quantizing later layers rather than the first layer(s). This matches the observations made in other works, that quantizing the first layers tends to have a noticeabley *negative* effect on accuracy [6]. This is only a recommendation, and the QAT package does not automatically avoid quantizing the first layers for the user.

4.3 Experiment Setup

We use the SVHN dataset. SVHN is an imageset of house numbers from Google Streetview. We quantize three common neural networks: AlexNet, VGG16, and ResNet50. AlexNet (2012) and VGG16 (2014) were important pioneering CNNs for image classification. ResNet (2015) was a ground-breaking design that pioneered skip-connections. We use the Keras *built-in* implementations of VGG16 and ResNet50, calling QAT's $quantize\_model$ on the layers ourselves. Keras is the neural-network Python framework and API that is built into Tensorflow; it essentially acts as *the* interface to Tensorflow. We verified that the Keras official models for VGG16 and ResNet50 were indeed correctly quantized, observing that every layer in the model was wrapped with the quantization-wrappers.

Only a few epochs (< 5) are required for more conventional models such as AlexNet and VGG, with training taking 5-8 (AlexNet) or 8-15 (VGG) minutes per epoch. For complex models such as ResNet, *many more epochs* (50 to 150) were found to be required, with training taking *15-25 minutes per epoch* for SVHN, on a standard Google Colab GPU.

4.4 Experiment Definition

The first experiment keeps the activation quantization constant at the default setting of 8 bits, while varying the bitwidth of the weight quantization. This experiment serves to give intuition on the QAT quantization results, without seeking to change too much (i.e., only the weight bitwidth is varying).

The second experiment performs more rigorous quantization. We quantize both the weights and the activations, at various bitwidths. In addition, we also quantize the outputs $\mathbf{Wx} + \mathbf{b}$ (which are the inputs into the activation functions).

4.5 Experiment 1: Weight quantization varying, Activation quantization constant at 8-bit

This experiment varies weight quantization while keeping activation fixed at 8-bit quantization.

AlexNet and VGG16 were able to quantize successfully from 16 bits down to 5 bits. At and below 4-bits, both of these failed to quantize, meaning that they could not train at all; their accuracy was no better than random. Both networks took about 2-5 epochs to reach stability for quantization.

ResNet50 quantized successfully from 16 bits down to 2 bits, lower than the other two.

Discussion

ResNet50 was much harder to train, requiring 10-20 times more epochs. However, it quantized down to 2 bits, while the other two networks are unable to. This is a surprising result, since ResNet50 is *harder* to train but is able to train to *much lower* quantization bitwidths.

Both AlexNet and VGG16 trained to almost the same precision as in full-precision, even as we varied weight quantization down to 5-bits (below which the two failed to quantize at all). This result is seen in Tables 2 and 3, where accuracies remain the same as full-precision even down to 5-bit quantization, the lowest quantization before the two networks could not train anymore. Furthermore, ResNet50 trained to the same accuracy *regardless of whether its weights are quantized to 16 bits or 2 bits*; this is seen in Figure 3, where the final accuracies for quantizations of 16-bits down to 2-bits remain *the same*. These results seem to imply that weight precision matters little (and so implying that activation quantization may matter more); we test this hypothesis in the next experiment, which varied activation quantization downwards together with the weights.

Acc	Q5	Q6	Q7	Q8	Q12	Q16	FP
Train	96.2	98.9	99.8	99.94	99.96	99.97	99.95
Test	93.5	93.4	94.0	94.2	94.1	94.2	93.9

Table 2: Experiment 1: SVHN AlexNet Quantized versus FP.

Acc	Q5	Q6	Q7	Q8	Q12	Q16	FP
Train	99.93	99.93	99.93	99.93	99.93	99.93	99.93
Test	95.3	95.4	95.5	95.5	95.5	95.3	95.4

Table 3: Experiment 1: SVHN VGG16 Quantized versus FP.

4.6 Experiment 2: Varying weight and activation quantization together

In this experiment, we aggressively quantize by setting QAT to also quantize the outputs $\mathbf{Wx} + \mathbf{b}$, in addition to varying both the weight and activation quantization number-of-bits (k) together.

AlexNet was unable to be quantized below 5-bits. However, AlexNet required only 2-5 epochs to train in QAT, with results being very stable. We see this in Table 4: QAT from 5 to 16 bits achieves the same test accuracies as in full-precision, with a small drop at the last quantization level of 5-bits, below which the network no longer quantized (could not train).

VGG16 was unable to quanize lower than 6-bits, but otherwise we see the same effect as for AlexNet. Table 5 shows test accuracies from 16-bit down to 7-bit quantization achieve parity with full-precision, with a small drop at 6-bit quantization, the final quantization level before the network no longer quantizes (coult not train).

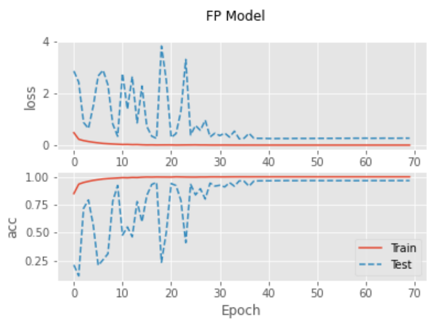
ResNet50 successfully quantizes down to 7-bits. However, at 7-bits, the peak accuracy is only 78%. At 8-16 bits, the peak accuracies range from [86,87]% respectively. These are shown below in Figure 4.

Acc	Q5	Q6	Q7	Q8	Q12	Q16	FP
Train	96.2	99.0	99.8	99.93	99.97	99.95	99.95
Test	93.6	93.2	93.8	93.8	94.3	94.1	93.9

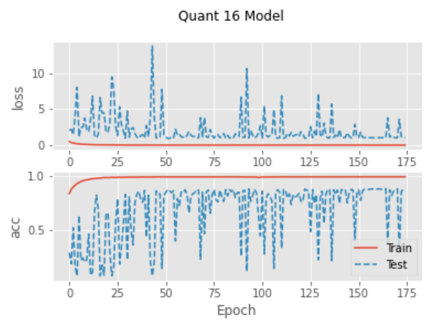
Table 4: Experiment 2: SVHN AlexNet Quantized versus FP.

Acc	Q6	Q7	Q8	Q12	Q16	FP
Train	98.2	99.9	99.93	99.93	99.92	99.93
Test	93.7	95.3	95.4	95.2	95.3	95.4

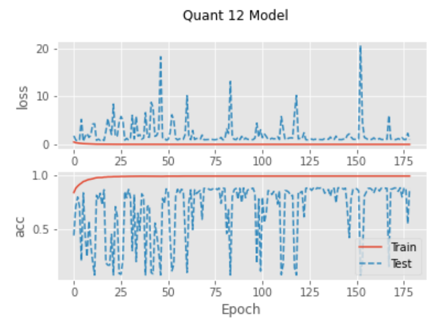
Table 5: Experiment 2: SVHN VGG16 Quantized versus FP



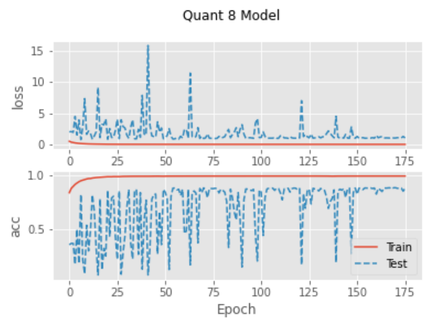
(a) SVHN ResNet50 FP



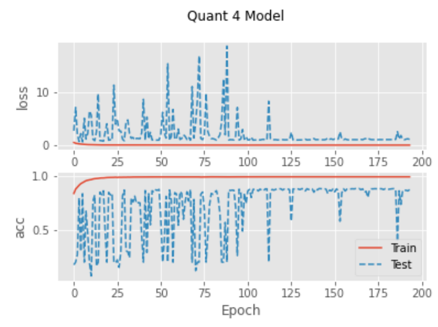
(b) SVHN ResNet50 Q16



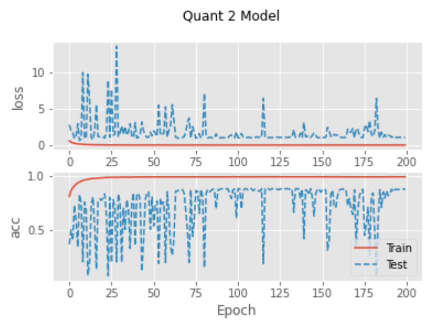
(c) SVHN ResNet50 Q12



(d) SVHN ResNet50 Q8



(e) SVHN ResNet50 Q4



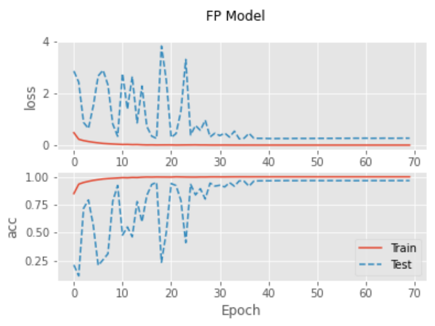
(f) SVHN ResNet50 Q2

Figure 3: Experiment 1: SVHN ResNet50 QAT Results. Varying weight quantization with activations held constant at 8-bit quantization. All quantizations train to an 88% test accuracy; the full-precision trains to 97% test accuracy.

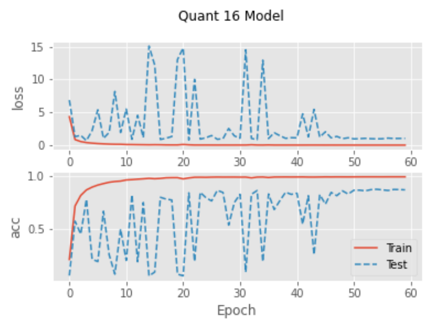
Discussion

Here, we attempted to verify whether activation quantization was a deciding factor for quantizibility; that is, would accuracy degrade significantly as we also vary activation quantization down together with the weight quantization? The results don't seem to indicate otherwise for AlexNet and VGG16, which is surprising. One possible explanation is that these two networks are 'simple', and thus their performance is resilient to quantization to low-bits.

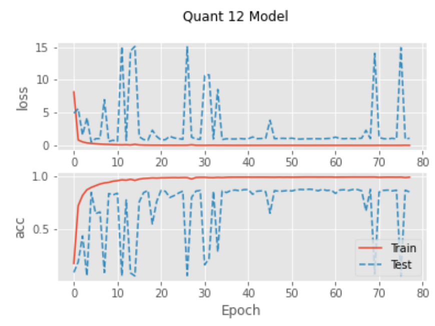
The more complex model, ResNet50, can more clearly test whether activation is a determining factor for quantizibility. ResNet50 quantizes equally well down to lower bitwidths; but now, as we quantize activations downwards, it can only quantize to 7-bit weights instead of quantizing down to 2-bit weights as in experiment 1. This seems to confirm the important role that activations play in quantizing a neural network.



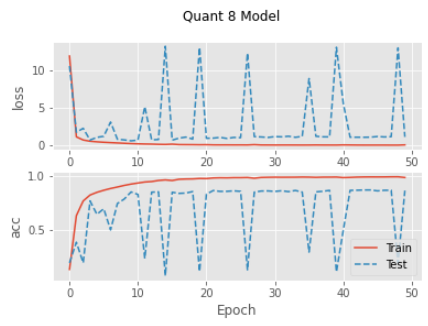
(a) SVHN ResNet50 FP



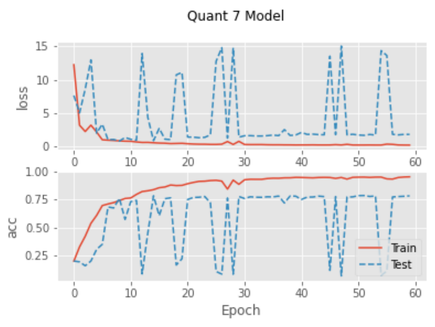
(b) SVHN ResNet50 Q16



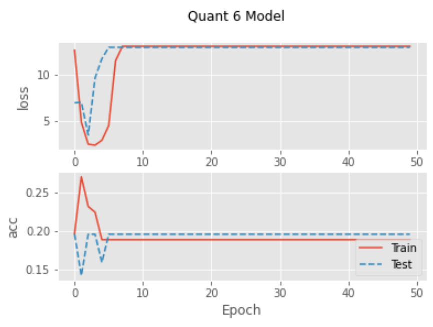
(c) SVHN ResNet50 Q12



(d) SVHN ResNet50 Q8



(e) SVHN ResNet50 Q7



(f) SVHN ResNet50 Q6

Figure 4: Experiment 2: SVHN ResNet50 QAT Results. Varying weight, activation quantizations together.

4.7 Summary of Experiments

It seems that while activation quantization is important, it may not impact simple networks (AlexNet and VGG16) as much as for complex networks (ResNet50).

AlexNet and VGG16 achieved parity between full-precision and quantized accuracies for *both* experiments. That is, these two networks tolerated more stringent activation quantization.

ResNet50 took much longer to train, and reached worse final quantized accuracies in both experiments. It could *not* tolerate more stringent activation quantization, seeing an accuracy drop at 7-bit weight & activation quantization, and unable to quantize further below that, whereas it could quantize previously in experiment 1 down to 2-bit weights. This seems to match our intuitions that activation quantization bitwidth plays an important factor in a neural network's ability to quantize.

Previous works have also noted the significance of activation quantization bitwidth on accuracy [6; 8].

One surprising result that we could not explain, was that experiment 2's training is decidedly far less noisy than experiment 1's, despite experiment 2 being far more rigorous in quantization.

4.8 Conclusion

We have surveyed the quantization of neural networks, and also ran two experiments that used the Tensorflow QAT library to quantize three common neural networks. Our observations seem to match those of prior works, that activation quantization plays an important role in both the quantizibility and quantized performance of neural networks.

These results indicate that simple but effective networks can be aggressively quantized well. In a mobile or constrained use case, complex networks may not be suitable anyways, due to space and energy constraints. More investigation could be made into the bitwidths at which the simple networks failed to quantize (< 5 bits), since it is possible that it is due to some issue in the Tensorflow QAT package rather than an inherent inability to quantize.

Acknowledgements

My thank you to Professor Darwiche for providing suggestions and advice.

References

[1] S. Russell and P. Norvig, "Artificial intelligence: a modern approach," 2002.

[2] H. Qin, R. Gong, X. Liu, X. Bai, J. Song, and N. Sebe, "Binary neural networks: A survey," *Pattern Recognition*, vol. 105, p. 107281, Sep 2020.

[3] M. Courbariaux, Y. Bengio, and J.-P. David, "Binaryconnect: Training deep neural networks with binary weights during propagations," 2016.

[4] M. Courbariaux, I. Hubara, D. Soudry, R. El-Yaniv, and Y. Bengio, "Binarized neural networks: Training deep neural networks with weights and activations constrained to +1 or -1," 2016.

[5] M. Rastegari, V. Ordonez, J. Redmon, and A. Farhadi, "Xnor-net: Imagenet classification using binary convolutional neural networks," 2016.

[6] K. Hwang and W. Sung, "Fixed-point feedforward deep neural network design using weights +1, 0, and 1," in *2014 IEEE Workshop on Signal Processing Systems (SiPS)*, pp. 1–6, 2014.

[7] F. Li, B. Zhang, and B. Liu, "Ternary weight networks," *arXiv preprint arXiv:1605.04711*, 2016.

[8] S. Zhou, Y. Wu, Z. Ni, X. Zhou, H. Wen, and Y. Zou, "Dorefa-net: Training low bitwidth convolutional neural networks with low bitwidth gradients," 2018.

[9] X. Lin, C. Zhao, and W. Pan, "Towards accurate binary convolutional neural network," 2017.

[10] A. Bulat and G. Tzimiropoulos, "Xnor-net++: Improved binary neural networks," 2019.

[11] J. Choi, Z. Wang, S. Venkataramani, P. I.-J. Chuang, V. Srinivasan, and K. Gopalakrishnan, "Pact: Parameterized clipping activation for quantized neural networks," 2018.

[12] D. Zhang, J. Yang, D. Ye, and G. Hua, "Lq-nets: Learned quantization for highly accurate and compact deep neural networks," 2018.

[13] L. Hou, Q. Yao, and J. T. Kwok, "Loss-aware binarization of deep networks," *arXiv preprint arXiv:1611.01600*, 2016.

[14] Z. Cai, X. He, J. Sun, and N. Vasconcelos, "Deep learning with low precision by half-wave gaussian quantization," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 5918–5926, 2017.

[15] R. Krishnamoorthi, "Quantizing deep convolutional networks for efficient inference: A whitepaper," 2018.

[16] Y. Bengio, N. Léonard, and A. Courville, "Estimating or propagating gradients through stochastic neurons for conditional computation," 2013.

[17] B. Jacob, S. Kligys, B. Chen, M. Zhu, M. Tang, A. Howard, H. Adam, and D. Kalenichenko, "Quantization and training of neural networks for efficient integer-arithmetic-only inference," 2017.