Technische Universität Berlin

Big Data Engineering (DAMS)
Fakultät IV
Ernst-Reuter-Platz 7
10587 Berlin
https://www.tu.berlin/dams

Thesis

[Choose yours: Bachelor or Master's]

Learned Quantization Schemes for Data-centric ML Pipelines

Anuun Chinbat

Matriculation Number: 0463111
20.01.2025

Supervised by
Prof. Dr. Matthias Boehm
M.Sc. Sebastian Baunsgaard

Hereby I declare that I wrote this thesis myself with the help of no more than the mentioned literature and auxiliary means.

Berlin, 01.01.2024

. . . .

(Signature) [your name]

Abstract

Machine learning (ML) models are notoriously resource-intensive. Given their widespread application across every-day edge devices, the need to reduce their memory and computing requirements is becoming ever more pressing. Despite the said resource-intensiveness of ML models, at the same time they offer the main ingredient for the remedy to the malady - redundancy - which can be exploited to reduce their memory usage. While the redundancy exploitation of ML models is already a common technique that comes in different forms, starting from weight pruning [28] [30][13] and ending with knowledge distillation [14] [32], quantization presents itself as a promising area especially in the sense of learned quantization - the process of making ML models learn their optimal quantization on their own. Hence, this work introduces two techniques designed to address the primary challenge of learned quantization: achieving effective model compression without incurring significant accuracy degradation. While the first technique involves custom loss functions that directly take into account quantization goals, the second, more novel approach incorporates a custom scaling factor gradient calculation that takes into account the gradient of the parameters that are being quantized. As a result, a memory usage reduction of up to $8\times$ is achieved for the parameters of networks trained on MNIST, CIFAR-10, and Imagenette.

Zusammenfassung

This is a placeholder for the german abstract (Kurzfassung) which should follow the same structure as the abstract. Just testing

Contents

1 Introduction

Such is the life of a modern human being that not a single day passes without a machine learning model toiling away in the background. From unlocking one's phone with Face ID in the morning to receiving a curated recommendation feed on Netflix in the evening — all is ML — but at what cost?

If we consider GPT-3 as an example, its 175 billion parameters need a whopping 700 gigabytes of storage in total — 4 bytes for each parameter represented in single-precision floating-point format (FP32). This costliness of modern ML models has revitalized interest in the research area of *quantization of Neural Networks* (NNs) which aims to reduce model size by developing methods that directly or indirectly decrease the amount of memory needed to store parameters numbering in the millions or billions. Going back to the GPT-3 example, by directly clamping its FP32 parameters to an 8-bit integer (INT8) representation, we can reduce its storage requirement from 700 to just 175 gigabytes.

But what costs does quantization itself entail? The natural answer to this question would be a reduction in model performance, as a decrease in precision logically implies a decrease in accuracy. However, there is a somewhat counterintuitive phenomenon where quantization improves accuracy by introducing noise, which act like a form of regularization, forcing the model to generalize better [4]. No matter whether quantization results in better or a slightly worse performance, the assumptions is that for each model there is an optimal way to quantize it within reasonable degradation ranges. And if the model is able to learn its optimal parameters, it is most likely also able to learn its optimal quantization parameters.

The fact that, indeed, models can learn their optimal quantization parameters has been proven many times in the past. But is there a way to make them do it better - is a question that will always remain and to which this thesis aims to contribute. In that sense, the current work will explore novel ways to tackle the two main problems that the process of learning optimal quantization parameters poses. First, how to overcome the issue of non-differentiability of rounding operations in the back-propagation. Second, how to guide the model to quantize selectively where necessary and adaptively relax the quantization when a certain threshold is reached.

While there is a number of methods dealing with the issue of non-differentiability of rounding operations in quantization, the most popular of them are the Straight-Through Estimator and other similar approaches that employ continuous relaxations of discrete functions to enable gradient-based optimization. Besides these approximation methods, other more aggressive techniques — in a sense — avoid non-differentiability altogether and perform quantization based on strictly defined constraints. Binary Connect [4], for example, binarizes weights during both forward pass and backpropagation, with the real values of weights used only during parameter update. However, these aggressive methods, while effective in simpler scenarios, often struggle on more complex datasets. The more effective methods seem to combine both constraints and gradient approximations — just like XNOR-Net enhances BinaryConnect with a gradient computation formula designed for binary weights [36].

Despite the abundance of different methods, there is still room for improvement with regard to the second problem mentioned earlier, namely, not only how to quantize, but also where to do so and when to stop. Therefore, in this thesis, we try to fill this room with the following contributions:

- We introduce a method that directly considers the gradient-to-parameter ratio, which conveys how much the parameter is being adjusted relative to its current value. Based on this ratio, the model learns its quantizer scaling factors, applied at different granularities, alongside the standard trainable parameters.
- We also provide a modular framework that can be easily integrated into a wide range of applications and layers with minimal adjustments, ensuring flexibility and usability.
- We also provide appliccable ranges for this ratio for effective quantizaiton for dense and convolutional layers.

2 Background

This chapter addresses the theoretical and contextual background necessary to understand the key concepts and methodologies that form the foundation of the current research. The first section will discuss the basics of deep NNs, upon which the technical setup of this thesis is based. The next section aims to provide a broader context for the term quantization, followed by a final section that explains common techniques of *learned quantization*, as well as the trade-offs and challenges they present.

2.1 Fundamentals of Deep Learning

This section introduces the fundamental concepts of deep learning, beginning with the most basic NN architecture components 2.1.1 and progressing to loss functions with regularization 2.1.2. The concepts of the forward pass and backpropagation will be explained in the last subsection 2.1.3.

2.1.1 Dense and Convolutional Layers

NNs can be considered a mathematical abstraction of the human decision-making process. Consider a scenario where, given an image, you need to say aloud what you see. The two eyes can be regarded as input nodes that receive the initial data, the brain can be seen as a set of *hidden layers* that process this data, and your mouth — the output node that provides the final answer.

A hidden layer, which typically consists of many neurons, is where the magic — or the transformation of data — happens. In its simplest form, within the classic *Multilayer Perceptron* (MLP) model, each hidden layer neuron performs a weighted operation:

$$output = f(w \cdot input + b)$$

where:

- *input* refers to the outputs from the previous layer (or the initial data from input nodes) that are fed into a specific neuron in the hidden layer.
- w (weights) is a vector of parameters associated with that specific neuron, defining the importance of each input received by this neuron.
- b (bias) is an additional scalar parameter specific to the neuron, which shifts the result of the weighted sum, allowing for more flexibility.

- f is the *activation function*, a nonlinear function applied to the weighted sum of inputs and bias in that specific neuron, allowing for more complexity.
- *output* is the result produced by the neuron, which will then be passed on to the next hidden layer (or to the final output layer).

Hidden layers where each neuron is connected to every neuron in the previous layer and every neuron in the next layer are called *dense layers*.

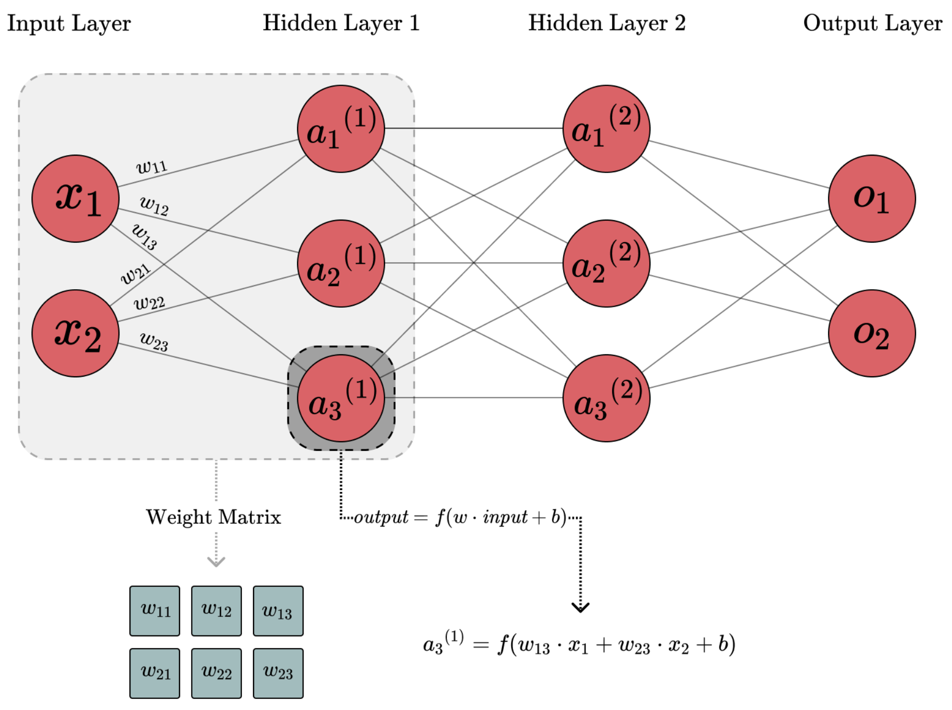


Figure 2.1: An example of a NN with two hidden dense layers, showing the connections between neurons in adjacent layers.

Mathematically, dense layers can be represented as:

$$a = f(W \cdot x + b)$$

where:

- x is the input vector, representing outputs from the previous layer (or initial input data for the first layer).
- W is the *weight matrix*, with each row corresponding to the weight vector w of a specific neuron.

- b represents the bias vector, where each scalar element corresponds to the bias of a specific neuron.
- f is the *activation function* that is applied element-wise.
- a refers to the output vector, representing the activations of all neurons.

This interconnectedness of dense layers introduces the inherent redundancy, or the over-parameterizedness of NNs [9]. It is particularly true in models with a large number of neurons, where W results in a vast number of parameters, which do not contribute to the model accuracy equally [18].

Convolutional layers are another type of hidden layers that involve a *convolution* operation on the input. Intuitively, a standard convolution is a process of sliding a small grid over an input to find patterns. The figure below, for example, shows the application of the Sobel kernel that detects edges on the input image.

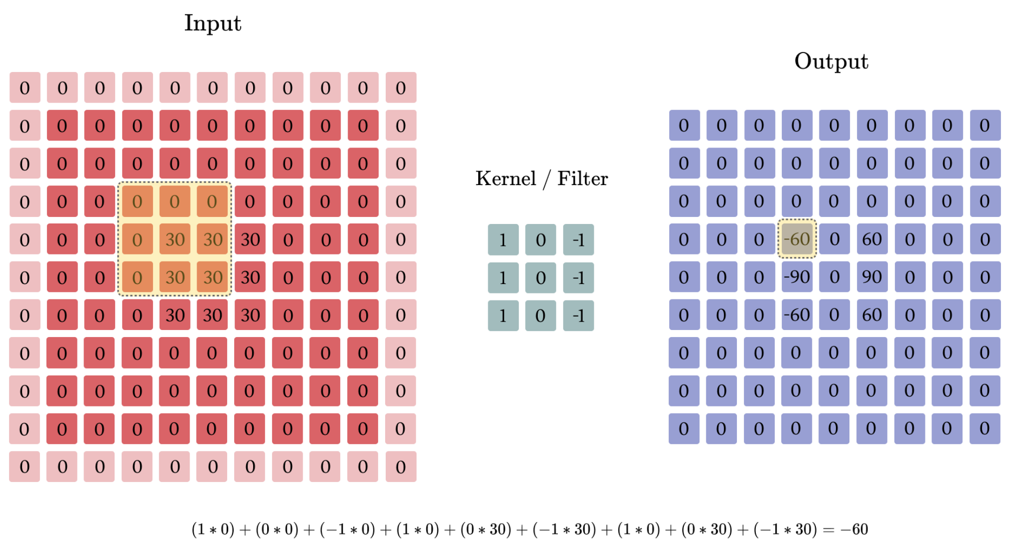


Figure 2.2: A 3×3 kernel (filter) sliding over a padded input matrix to compute the output feature map, demonstrating the interaction between the kernel weights and input values at a specific position.

For multi-channeled inputs, like RGB images, the convolution operation uses a multi-channeled kernel, as shown in Figure 2.3, producing a single-channeled feature map that combines weighted contributions from all input channels. A convolutional layer typically includes multiple such kernels, generating feature maps equal to the number of kernels. After the convolution operation generates the feature maps, a bias term is added to each map, and the activation function is applied element-wise — just like in dense layers.

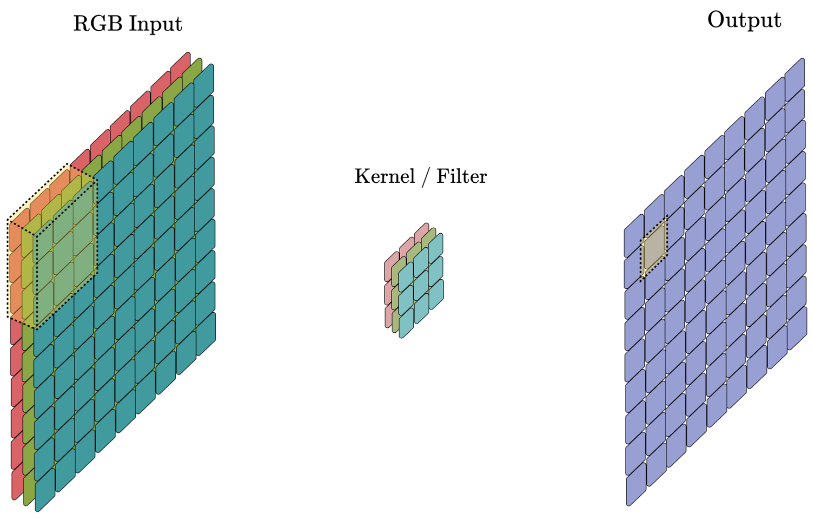


Figure 2.3: A 3×3x3 kernel (filter) sliding over an RGB input matrix to produce a single-channeled output feature map.

Mathematically, a convolutional layer can be represented as:

$$y_{i,j,k} = \phi\left(\sum_{m=1}^{M}\sum_{p=1}^{P}\sum_{q=1}^{Q} x_{i+p-1,j+q-1,m} \cdot w_{p,q,m,k} + b_k\right)$$

where:

- P, Q are the height and width of the filter, respectively.
- M is the number of input channels.
- $y_{i,j,k}$ denotes the output at position (i, j) for the k-th filter.
- $x_{i+p-1,j+q-1,m}$ is the input at position $(i + p - 1, j + q - 1)$ for the m-th input channel.
- $w_{p,q,m,k}$ represents the weight of the filter at position (p, q) for the m-th input channel and k-th filter.
- b_k is the bias for the k-th filter.
- $\phi(\cdot)$ is the activation function.

In simpler terms, a convolutional layer applies filter weights as it slides over rows (p), columns (q), and channels (m), sums the results, adds bias (b_k), and repeats this for all positions (i, j) and filters (k).

Although convolutional layers often have fewer weight parameters than dense layers in typical architectures, they still contain redundancies [17], presenting an opportunity for quantization. Thus, both dense and convolutional layers will be the focus of this work.

2.1.2 Loss Functions and Regularization

The weights and biases are usually *learnable parameters* that the model adjusts during *training*. The training process of NNs is similar to how our brains learn from mistakes. Given the ground truth, a NN adjusts its learnable parameters using a specific function that compares the ground truth with the output generated by the network, essentially measuring the magnitude of the network's errors.

This function is called a *loss function*, and depending on the type of question the network aims to answer, it can take many different forms. For example, for the MLP described in Figure 2.1 that generates a binary classification, we would use the *log loss* function. Since the datasets used in this thesis involve multi-class classification, the *sparse categorical cross-entropy* (SCCE) loss function will be used, which measures the difference between the predicted class probabilities and the true labels for each class in the dataset.

Often the loss function alone is not enough for a NN to perform well, as it may lead to overfitting or fail to capture desired generalization properties. This is why a *regularization term* that penalizes unwanted behaviours is added to the loss function.

A typical regularization term is L_2, which penalizes large weights by adding the sum of the squared weights to the loss. The modified loss function is then expressed as:

$$\mathcal{L}_{\text{total}} = \mathcal{L}_{\text{data}} + \lambda \sum_i w_i^2$$

where:

- $\mathcal{L}_{\text{data}}$ is the original loss function (in our case, the SCCE loss function).
- λ is a scalar parameter that controls the strength of the regularization.
- w_i represents each individual weight value in the model.

The current work employs multiple custom regularization terms that encourage specific behaviors in the models while discouraging others. These terms will be discussed in detail in the Experimental Setup section 4.1.

2.1.3 Forward-Pass and Back-Propagation

The repetition of the mathematical operations described earlier in the Dense and Convolutional Layers subsection 2.1.1 during model training constitutes the *forward pass*. It is the process where input data is passed through the network layer by layer, with each layer applying its learned weights and biases to produce a final output.

As mentioned in the previous subsection, this output is then compared with the ground truth by the loss function that produces an error. This error is used to update the learnable parameters in W and b during a process called *back-propagation*.

In other words, back-propagation is the method by which the network adjusts its parameters to minimize the error. It calculates the gradient of the loss function with respect to each parameter using the chain rule. W and b are typically updated as follows:

$$W = W - \eta \frac{\partial L}{\partial W}, \quad b = b - \eta \frac{\partial L}{\partial b}$$

where L is the loss function, and η is the learning rate.

For example, consider the weight $w_{1,1}$ represented as the line between x_1 and the hidden layer node ${a_1}^{(1)}$ in Figure 2.1. The gradient of this weight with respect to the loss is calculated using the chain rule:

$$\frac{\partial L}{\partial w_{1,1}} = \frac{\partial L}{\partial o_1} \cdot \frac{\partial o_1}{\partial {a_1}^{(1)}} \cdot \frac{\partial {a_1}^{(1)}}{\partial w_{1,1}}$$

Where:

- $\frac{\partial L}{\partial o_1}$ is the gradient of the loss with respect to o_1.
- $\frac{\partial o_1}{\partial {a_1}^{(1)}}$ is the gradient of o_1 with respect to the output of ${a_1}^{(1)}$.
- $\frac{\partial {a_1}^{(1)}}{\partial w_{1,1}}$ is the value of x_1, since ${a_1}^{(1)}$ is a weighted sum of the inputs.

This shows how each weight contributes to the final error during back-propagation.

2.2 Basics of Quantization

This section aims to answer the *why* question with respect to quantization and further provides a broader understanding of the term regarding its types.

2.2.1 Purpose and Definition

As we become increasingly dependent on deep learning models disguised as everyday tools, the need for these models to function in a resource- and time-efficient manner is more imperative than ever. The focus on resource efficiency is particularly important, with the research community expressing concerns regarding the environmental effects of large models, the exponential size growth of which continues to significantly outpace that of system hardware [41]. In this regard, studies have examined quantization within the context of Green AI as a method to reduce the carbon footprint of ML models [37].

Aside from the environmental considerations, the mere need to reduce the computational cost and speed of predictive models comes as an apparent business requirement. This requirement is essential when — quite ironically — embedded systems, famous for their compactness, meet ML models, infamous for their complexity. Microcontrollers, for instance, usually are not able to perform floating-point operations, which must therefore be emulated in software, introducing significant overhead. This is why quantization, the process which reduces the memory footprint of a model, is also extensively covered in the realm of embedded systems that inherently prefer integer arithmetic, as well as bitwise operations [36] [43] [1][31].

Another motivation for quantization — although somewhat controversial — is the fact that reducing the bit-width of ML models makes them robust to adversial attacks in certain cases [29]. This holds significant value in fields, such as autonomous driving, where model vulnerability may result in fatal outcomes. Interestingly enough, the use cases where such robustness is required also demand fast inference, as they rely on real-time predictions. Consider healthcare diagnostics needed for emergency scenarios or military defense mechanisms designed for immediate action.

The list of reasons why quantization is useful may go on for a while, but regardless of the motivation, the essence of the term itself — rooted in the early 20th century — remains unchanged: quantization refers to the division of a quantity into a discrete number of small parts [11]. With regard to ML models, it describes the process of dividing higher bit-width numbers into a discrete number of lower bit-width representations without causing significant degradation in performance [9].

Since ML models are generally considered redundant or over-parameterized, there are multiple points where quantization can be applied. Specifically, in this thesis, we apply quantization to the weights and biases of dense layers, as well as the kernels and biases of convolutional layers. Other applications include, but are not limited to, layer activation and layer input quantization (two sides of the same coin), as well as gradient quantization. The bottom line is that wherever there is an opportunity for arithmetic or memory optimization, there is room for quantization.

2.2.2 Core Quantization Approaches

There is a multitude of ways to classify NN quantization methods, a broader overview of which will be covered in the Related Work chapter 5. For now, we will focus on a few basic approaches from the general categories of both *data-driven* and *data-free* methods

[42] to provide a basic understanding of the NN quantization process.

The simplest form of data-free quantization, or *post-training quantization* [20], involves converting already trained parameters from FP32 to a lower bit-width format without using the initial training data. A common approach is to apply *uniform quantization* that maps real values to a set number of *bins*. The general formula can be written as:

$$Q(r) = round(\frac{r}{S})$$

Where:

- $Q(\cdot)$ denotes the quantization operation.
- r is the real value of a given model parameter in higher bit-width representation.
- $round(\cdot)$ is some rounding operation, such as a simple $floor(\cdot)$.
- S is a scaling factor.

As a result, we essentially end up with a discrete number of values in lower bit precision, instead of an almost continuous range of real numbers as shown in Figure 2.4.

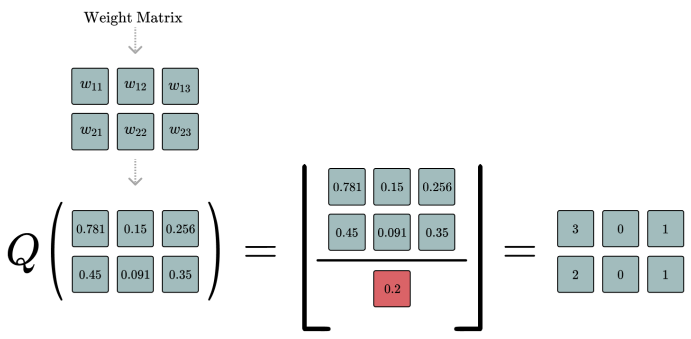
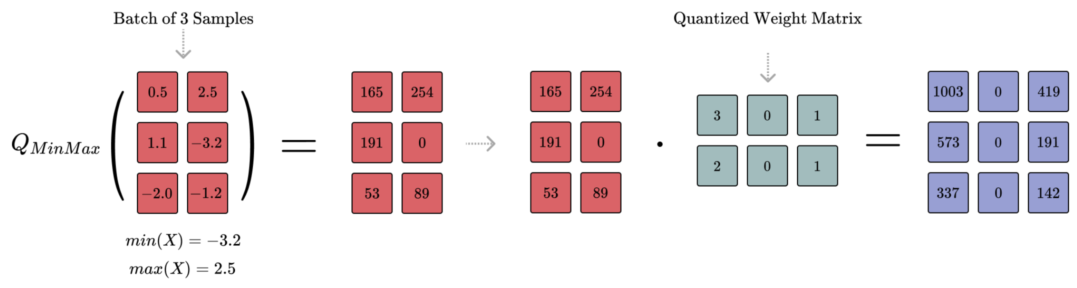


Figure 2.4: An example illustrating the quantization operation on the weight matrix from Figure 2.1, with arbitrary values for demonstration purposes.

Unlike data-free quantization — as the name suggests — data-driven quantization typically involves retraining the model using the initial data. An example of this approach is the Ristretto framework [12], which, similar to data-free methods, first analyzes the trained model to select suitable lower bit-width number formats for its weights. Then, using a portion of the original dataset, the framework determines appropriate formats for layer inputs and outputs. As a next step, based on the validation data, Ristretto adjusts the quantization settings to achieve optimal performance under the given constraints. Finally, the quantized model is fine-tuned using the training data.

A much simpler example of data-driven quantization could be the min-max quantization on input data as shown in Figure 2.5. This method can also be used in a data-free scenario to quantize learned model parameters and is internally used as one of the default techniques in popular ML frameworks like Tensorflow and PyTorch.



$$Q_{MinMax}(x) = \lfloor \frac{x - min(X)}{s} \rfloor$$

$$s = \frac{max(X) - min(X)}{2^8 - 1} = \frac{2.5 - (-3.2)}{255} \approx 0.0224$$

Figure 2.5: An example illustrating min-max quantization of input data to 8 bits, followed by matrix multiplication with the quantized weight matrix from Figure 2.4. Input data has arbitrary values for demonstration purposes.

In the previous subsection, we discussed *where* quantization could be applied in a model, mentioning weights, kernels, and biases as the focus of this thesis. Figures 2.4 and 2.5 show examples of quantization using a scalar scaling factor. However, scaling factors could be applied at varying levels of detail, and this is where the concept of *quantization granularity* comes into play.

Granularity refers to the level of detail at which scaling factors are applied, ranging from a single factor for an entire kernel (coarse granularity) to separate factors for individual spatial locations, channels, or filters (fine granularity). For instance, Figure 2.6 illustrates various possible granularities for the kernels of convolutional layers. Despite this wide range of possibilities, channel-wise quantization is currently the standard for convolutional layers [9], as it helps parallel processing capabilities of accelerators that compute channel outputs independently. For dense layers, row-wise quantization (one scaling factor for weights used by a single output neuron) is more prevalent because it aligns with matrix-vector multiplication, which then can be carried out by specialized linear algebra libraries in an optimized way [23].

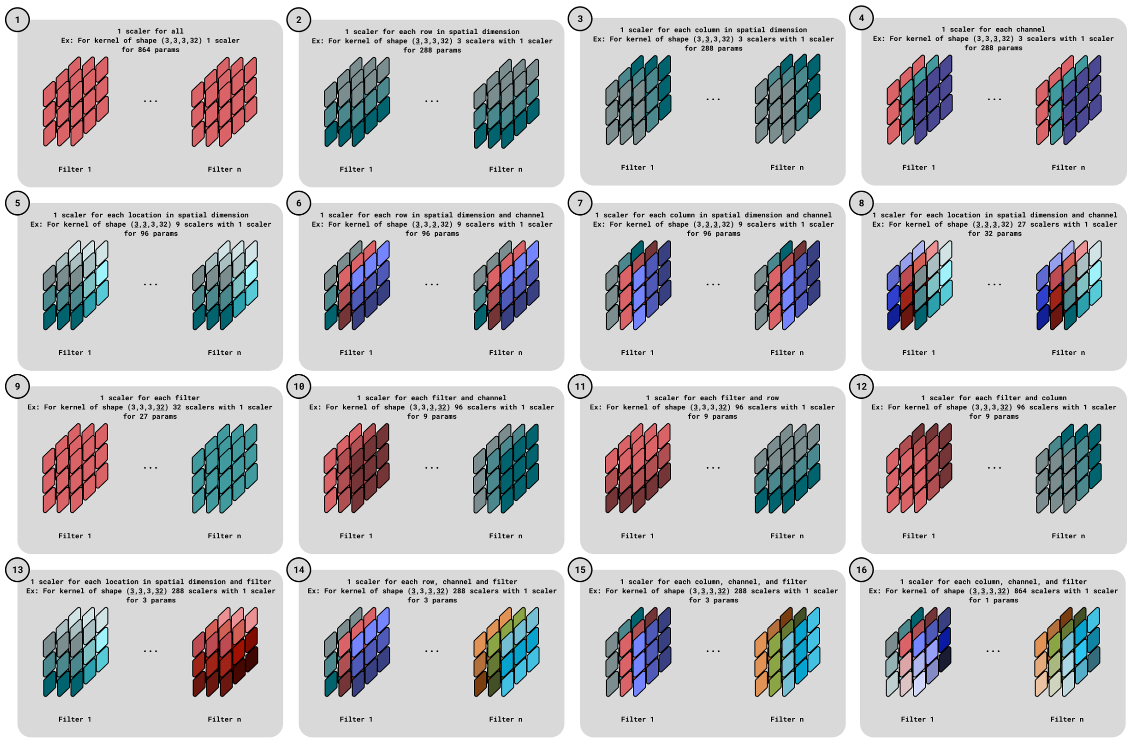

1
1 scaler for all
Ex: For kernel of shape (3,3,3,32) 1 scaler
for 864 params
Filter 1
...
Filter n
2
1 scaler for each row in spatial dimension
Ex: For kernel of shape (3,3,3,32) 3 scalers with 1 scaler
for 288 params
Filter 1
Filter n
3
1 scaler for each column in spatial dimension
Ex: For kernel of shape (3,3,3,32) 3 scalers with 1 scaler
for 288 params
Filter 1
Filter n
4
1 scaler for each channel
Ex: For kernel of shape (3,3,3,32) 3 scalers with 1 scaler
for 288 params
Filter 1
Filter n
5
1 scaler for each location in spatial dimension
Ex: For kernel of shape (3,3,3,32) 9 scalers with 1 scaler
for 96 params
Filter 1
Filter n
6
1 scaler for each row in spatial dimension and channel
Ex: For kernel of shape (3,3,3,32) 9 scalers with 1 scaler
for 96 params
Filter 1
Filter n
7
1 scaler for each column in spatial dimension and channel
Ex: For kernel of shape (3,3,3,32) 9 scalers with 1 scaler
for 96 params
Filter 1
Filter n
8
1 scaler for each location in spatial dimension and channel
Ex: For kernel of shape (3,3,3,32) 27 scalers with 1 scaler
for 32 params
Filter 1
Filter n
9
1 scaler for each filter
Ex: For kernel of shape (3,3,3,32) 32 scalers with 1 scaler
for 27 params
Filter 1
Filter n
10
1 scaler for each filter and channel
Ex: For kernel of shape (3,3,3,32) 96 scalers with 1 scaler
for 9 params
Filter 1
Filter n
11
1 scaler for each filter and row
Ex: For kernel of shape (3,3,3,32) 96 scalers with 1 scaler
for 9 params
Filter 1
Filter n
12
1 scaler for each filter and column
Ex: For kernel of shape (3,3,3,32) 96 scalers with 1 scaler
for 9 params
Filter 1
Filter n
13
1 scaler for each location in spatial dimension and filter
Ex: For kernel of shape (3,3,3,32) 288 scalers with 1 scaler
for 3 params
Filter 1
Filter n
14
1 scaler for each row, channel and filter
Ex: For kernel of shape (3,3,3,32) 288 scalers with 1 scaler
for 3 params
Filter 1
Filter n
15
1 scaler for each column, channel, and filter
Ex: For kernel of shape (3,3,3,32) 288 scalers with 1 scaler
for 3 params
Filter 1
Filter n
16
1 scaler for each column, channel, and filter
Ex: For kernel of shape (3,3,3,32) 864 scalers with 1 scaler
for 1 params
Filter 1
Filter n

Figure 2.6: A demonstration of the varying application of scaling factors, ranging from a single scalar applied to the entire kernel (1) to separate scalars assigned to spatial dimensions (e.g., 1, 2), channels (4), filters (9), and other granular configurations.

2.3 Learned Quantization

Now that the fundamentals of quantization have been covered, this section introduces key concepts commonly encountered in learned quantization, including its challenges, trade-offs, and the popular techniques used to overcome them.

2.3.1 Trade-offs and Challenges

The inherent — or rather the generally accepted — characteristic of quantization is that it negatively influences performance. This is referred to as the trade-off between quantization and generalization, which reflects the question of how much accuracy — or whatever performance metric is being used — we are willing to sacrifice to gain a reduction in computational cost, memory usage, or inference time. However, the truth is that we usually cannot afford sacrificing anything. This, coupled with the lack of a guarantee that pre-defined *quantizers* can yield optimal results [43] [7], has paved the way for the burgeoning field of learned quantization, which aims to *learn* how to quantize the model in a manner that mitigates performance loss.

Learned quantization is, however, a double-edged sword in the sense that, despite producing compact results, the cost to achieve them is higher [34]. The obvious reason is the additional computational overhead introduced by learnable quantizers. Thus, it is important to strike a balance between learning optimal quantization and keeping the training process manageable — which explains the prevailing emphasis on simplicity in most learned quantization research.

The main issue in achieving this simplicity is posed by the fact that discretization, in its essence, is non-differentiable — meaning it is challenging to integrate any kind of discretizing operations into gradient-based optimization methods, upon which ML models rely. Using the chain rule back-propagation example from subsection 2.1.3, let's consider a simple flooring operation introduced into the process to better understand the problem.

Suppose we want to quantize activations and apply $floor(\cdot)$ to hidden layer outputs:

$$a_{1,q}{}^{(1)} = floor(a_1{}^{(1)})$$

As a result, the chain rule becomes:

$$\frac{\partial L}{\partial w_{1,1}} = \frac{\partial L}{\partial a_{1,q}{}^{(1)}} \cdot \frac{\partial a_{1,q}{}^{(1)}}{\partial a_1{}^{(1)}} \cdot \frac{\partial a_1{}^{(1)}}{\partial w_{1,1}}$$

where $\frac{\partial a_{1,q}{}^{(1)}}{\partial a_1{}^{(1)}}$ presents a challenge. Since $a_{1,q}{}^{(1)} = floor(a_1{}^{(1)})$, the derivative is:

$$\frac{\partial a_{1,q}^{(1)}}{\partial a_1^{(1)}} = \begin{cases} 0 & \text{if } a_1^{(1)} \notin \mathbb{Z}, \\ \text{undefined} & \text{if } a_1^{(1)} \in \mathbb{Z}. \end{cases}$$

This means that for most values of $a_1{}^{(1)}$, which are non-integer, the gradient becomes 0, resulting in $w_{1,1}$ not receiving any updates. For integer values of $a_1{}^{(1)}$, the backpropagation process fails altogether.

In essence, circumventing the issue of non-differentiability is the fundamental problem that learned quantization aims to solve, all while managing the aforementioned trade-offs to produce a model that is compact, not overly complex to train, and highly performant.

2.3.2 Common Methods

The most common method to address the issue of non-differentibility is to approximate the gradient of quantization operators using the Straight-Through Estimator (STE) [2] [8]. This workaround applies the quantizion operation as is during the forward-pass, but replaces the gradient of the piece-wise discontinuous function with that of a continuous identity function. Returning to the example from the previous subsection, if we apply the STE to the problematic gradient, instead of:

$$\frac{\partial a_{1,q}^{(1)}}{\partial a_1^{(1)}} = \begin{cases} 0 & \text{if } a_1^{(1)} \notin \mathbb{Z}, \\ \text{undefined} & \text{if } a_1^{(1)} \in \mathbb{Z}. \end{cases}$$

we approximate it as:

$$\frac{\partial a_{1,q}^{(1)}}{\partial a_1^{(1)}} \approx 1$$

This enables gradient flow, allowing model parameters to receive updates without being hindered by the non-differentiability of the quantization step.

The STE — and other estimators [3] — are the cornerstone of quantization-aware training (QAT) [19], a subfield that falls under the broader umbrella of learned quantization. QAT, however, does not necessarily focus on learning quantization parameters. Instead, it focuses on helping the model adapt to the loss caused by the quantization process during the forward pass, whether or not trainable quantizers are involved.

More often than not QAT and trainable quantization parameters are combined. An example is quantization-interval-learning (QIL) [21], which uses three trainable quantization parameters (the center of the interval, the distance to the center, and a parameter that controls the scaling itself) with piecewise differentiable operations and relies on STE for gradient updates of quantized weights and activations. Similarly, the learned step size quantization (LSQ) approach [7] defines a single trainable quantization parameter (step size) with an explicit gradient formula and also uses STE for standard parameter updates. LQ-Nets [43] depend on STE too, but — unlike the two previous techniques — directly incroporate a quantization error minimazation algorithm to calculate binary encodings and the associated bases of model parameters given a predefined bit size.

INSERT STUFF ABOUT QUANTIZATION ENCOURAGING REGULARIZATION

In this thesis, we will utilize the STE but introduce a custom gradient calculation for the scale factor gradients based on a specific type of gradient sensitivity in the model parameters. Additionally, we will explore custom loss regularization terms that encourage quantization and systematically compare them across different datasets.

3 Learned Quantization Schemes

This chapter introduces two custom learned quantization schemes — approaches that allow models to learn to quantize themselves with adjustable aggressiveness. The first one, a custom quantization layer featuring a threshold for gradient based scale updates, will be discussed in the first section. The second scheme, which focuses on custom regularization terms with a configurable penalty rate, will be covered second.

3.1 Nested Quantization Layer

To separate the quantization logic from the usual structure of NN layers, we define a nested quantization layer that can be used within a standard one. This approach provides usability, making it easy to extend the functionality to other types of layers beyond dense and convolutional ones. We will first explain the core logic of the nested quantization layer and how it integrates into a model, followed by a detailed explanation of how the trainable scaling factors are updated.

3.1.1 Core Logic and Structure

As the name "nested quantization layer" suggests — this layer is implemented in a way that it is initialized from within a model layer itself.

To explain, let P be the parameter of a layer (weights, bias or kernel). This P is used as the input for the nested quantization layer, where it undergoes quantization based on a scaling factor s. In turn, s is the only trainable parameter of the nested layer itself.

The forward pass of the nested layer performs the following operation:

$$P_{reconstructed} = P_{quantized} \cdot s$$

where $P_{quantized}$ is the quantized integer form of P and is defined as:

$$P_{quantized} = floor(\frac{P}{s})$$

During backpropagation, the nested layer receives the upstream gradient of $P_{reconstructed}$ and passes it downward as is. This follows the standard STE behaviour used with non-differentiable discretizers, such as $floor(\cdot)$ in our case, as discussed in Subsection 2.2.2.

For updating the scale factor s — its own trainable parameter — the nested layer utilizes the upstream gradient information of $P_{\text{reconstructed}}$ and applies a custom logic, which will be detailed in the next subsection. Essentially, this approach solves two problems with one tool — updating both P and s using the same gradient, although with different logic.

Conceptually, the resulting structure with one or more nested quantization layers is illustrated in Figure 3.1 on the example of a convolutional layer.

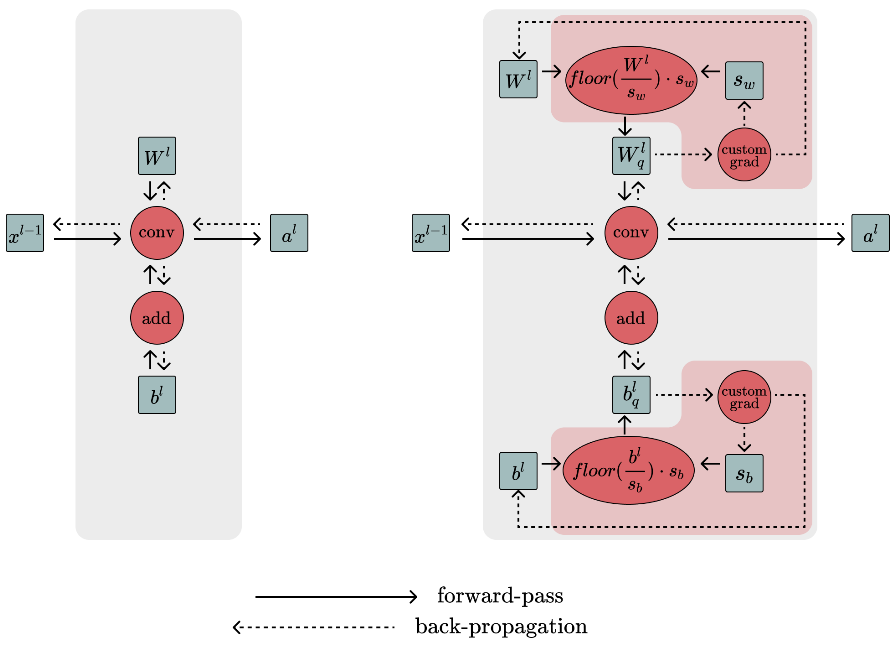


Figure 3.1: A standard convolutional layer (left) and its integration with the nested quantization layer (right) for both weights and bias. Quantization logic is applied to weights and biases during the forward pass, with trainable scaling factors updated using custom gradients in the backward pass. Parameter gradients are passed downward as is.

Since the scaling factor s can have different shapes and be applied at varying levels of granularity, we have enabled scalar, row-wise, and column-wise application granularity for dense layers, as shown in Figure 3.2. For convolutional layers, we additionally support channel-wise granularity, corresponding to the first four scenarios in Figure 2.6.

The nested layer is built upon Tensorflow's `tf.keras.Layer` class, which serves as the base for all Keras layers. The gradient calculation logic is wrapped in Tensorflow's `tf.custom_gradient` decorator, allowing proper functioning of the model's computational graph. Dense and convolutional layers have been implemented as `tf.keras.Layer`

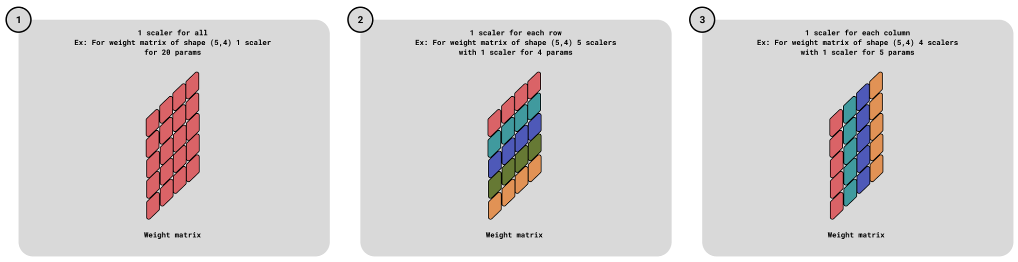

1
1 scaler for all
Ex: For weight matrix of shape (5,4) 1 scaler for 20 params
Weight matrix
2
1 scaler for each row
Ex: For weight matrix of shape (5,4) 5 scalers with 1 scaler for 4 params
Weight matrix
3
1 scaler for each column
Ex: For weight matrix of shape (5,4) 4 scalers with 1 scaler for 5 params
Weight matrix

Figure 3.2: A demonstration of the varying applications of scaling factors, ranging from a single scalar applied to the entire weight matrix (1) to row-wise and column-wise application of vector scalers.

objects with minimal adjustments as well to incorporate the nested layers logic.

3.1.2 Learned Scale Factor

For the trainable scale factor s, we define a custom gradient formula. The gradient of the loss with respect to s, denoted as $\nabla_s L$, is computed as:

$$\nabla_s L = g_s \cdot m,$$

Let's consider both multiplication terms separately. g_s is the main "decision maker" on whether to increase the scale and, therefore, quantize more. It is based on a hyperparameter threshold λ, which is compared against the ratio r.

$$g_s = \begin{cases} 0, & \text{if } r \geq \lambda, \\ -\tanh(\lambda - r), & \text{if } r < \lambda, \end{cases}$$

In turn, r is the ratio between the upstream gradient of the layer's reconstructed parameter with respect to the loss and its absolute value (for simplicity, we done $P_{reconstructed}$ as P_r):

$$r = \frac{|\nabla_{P_r} L|}{\max(\epsilon, |P_r|)}$$

In essence, it conveys the relative impact of the gradient on the parameter's value. A large ratio indicates that the parameter is not ready for aggressive quantization because small perturbations can lead to significant changes in its optimization. Conversely, parameters with a small r are better candidates for quantization since they are less sensitive.

The decision to replace P_r with ϵ when $P_r = 0$ ensures that the corresponding r becomes large, effectively resisting quantization. This behavior is valid regardless of the parameter's sensitivity — if the zero parameter is sensitive, quantization could disrupt future optimization steps, and if it is not sensitive, quantization adds no value since s has a positive non-zero constraint and $\frac{P}{s}$ will remain zero.

The motivation behind using $tanh(\cdot)$ primarily stems from two key reasons. First, it is bounded (in our use case, to $[-1, 0)$), which prevents excessive gradient magnitudes. Second, unlike sigmoid, it does not require any additional rescaling since it is symmetric around 0. An additional point is that $tanh(\cdot)$ saturates comparably faster, potentially allowing for more decisive gradients, but it is uncertain how much influence this has.

Now that g_s is covered, let's take a look at m defined as:

$$m = \max\left(|P_{\text{quantized}}|\right)$$

where the shape of m corresponds to the shape of s. For example, if s is a row-wise scaler, then m will hold the maximum value from each row of $|P_{quantized}|$. Similarly if s is a scalar scaler, then m represents the maximum across the entire layer parameter.

A larger m indicates a wider range of quantized values, implying the parameter can tolerate coarser quantization. In contrast, a smaller m means a narrower range, where aggressive quantization could be rather harmful. As a result, by multiplying g_s with m, the adjustment to the scale becomes proportional to the parameter's range. This encourages more aggressive quantization for parameters with larger ranges while being more conservative for smaller ones.

To sum this part up, the intuition is that the gradient adjustment for the scale factor s adapts dynamically based on both the sensitivity of the parameter (g_s) and its range (m). Sensitive parameters are left with a "zero vote," while the less sensitive ones have a say on how much quantization they can tolerate.

The final touch is that the scale gradients are initially calculated for each parameter value but are then aggregated along the corresponding granularity axes. This reflects the collective behavior of parameters within the same granularity, where only those deemed quantizable and with a meaningful "say" contribute to the overall adjustment, while sensitive parameters express their resistance with a "zero vote."

In Chapter 4, we will present experimental results for different values of λ, offering guidance on the optimal values for both dense and convolutional layers.

3.2 Custom Loss Functions

3.2.1 Penalty for Inverse Scale Factor Magnitude

3.2.2 Constraint on Bin Count for Quantization

3.2.3 Deviation between Quantized and Original Values

4 Experiments

This chapter details the experiments conducted in the context of this thesis. We evaluate the proposed methods on three image classification datasets: MNIST [27], CIFAR-10 [26], and Imagenette [16], which is a subset of 10 easily classified classes from the ImageNet dataset. First, we provide an overview of the experimental setup. We then present the results of the gradient ratio thresholding method, as detailed in Section 3.1, followed by the results of the custom loss terms discussed in Section 3.2.

4.1 Experimental Setup

Software and Hardware Setup. As briefly mentioned in Section 3.1, our custom methods are implemented in TensorFlow. Specifically, we used TensorFlow version 2.11.0, running on Python 3.10.14. All experiments are conducted on a server equipped with two NVIDIA A40 GPUs, each with 48 GB of memory, running on CUDA 11.4 and driver version 470.256.02.

Experiment Networks. For simplicity and tractability, we define custom networks for each dataset. For MNIST, we use a small network consisting of two dense layers, each adjusted to incorporate nested quantization layers for both weights and biases. For CIFAR-10, we use a convolutional network with three blocks of convolutional layers, each adjusted to incorporate nested quantization layers for both kernels and biases. These are followed by two dense layers. The Imagenette model is a ResNet-inspired architecture, featuring an initial quantized convolutional block followed by four stages of residual blocks. Each residual block incorporates the adjusted convolutional layers with nested quantization for both kernels and biases. For the experimentation with custom loss terms, we use equivalent networks without the nested quantization layer logic.

Baseline Hyperparameters and Initialization. Using the hyperparameters described in Table 4.1, each network optimizes its parameters using the Adam optimizer and the sparse categorical cross-entropy loss function. Weights and biases are initialized with random normal values, while scale factors are initialized with very small constant values for all cases. Scale factors are constrained to be positive and non-zero to avoid numerical issues during division. An example implementation for reproducing these networks is available at the following GitHub Gist: INSERT LINK

Table 4.1: Network Settings for Different Datasets

Hyperparameter	**MNIST**	**CIFAR-10**	**Imagenette**
Learning Rate	0.0001	0.0001	0.001
Batch Size	32	128	64
Epochs	100	100	100
Seed for Reproducibility	42	42	42

4.2 Optimal Thresholds for Nested Quantization Layers

For the nested quantization layer method, we evaluate the results across different thresholds λ for both dense and convolutional layers. The first subsection presents results for fully connected layers trained on MNIST, while the second subsection focuses on convolutional layers trained on CIFAR-10 and Imagenette.

4.2.1 Fully Connected Layers

As mentioned earlier, we use a small network with two dense layers trained on the MNIST dataset. These two dense layers consist of weight matrices W_1 and W_2, along with bias vectors b_1 and b_2. We examine three scenarios: applying the scale factor rowwise, columnwise, and as a single scalar for the entire weight matrix. For all scenarios, a single scalar scale factor is used for each bias. The configurations are summarized in Table 4.2.

Table 4.2: Scale Factor Granularity

Granularity	W_1 (784×128)	b_1 $(128,)$	W_2 (128×10)	b_2 $(10,)$
Rowwise scaler	s $(784,)$	s $(,)$	s $(128,)$	s $(,)$
Columnwise scaler	s $(,128)$	s $(,)$	s $(,10)$	s $(,)$
Scalar scaler	s $(,)$	s $(,)$	s $(,)$	s $(,)$

Note: $(,)$ represents a scalar value — scalars technically have no dimensions.

The experiments have shown that the optimal quantization is observed for a penalthy threshold of $\lambda = 1e - 10$ across all three scenarios. This conclusion is based on Pareto front plots in Figure 4.1, which illustrate the trade-off between accuracy loss and quantization. Specifically, we observe that, for all scenarios, the number of unique integer values after quantization, aggregated across all parameters of the two dense layers, reaches its lowest point at $\lambda = 1e - 10$ without significantly degrading the accuracy.

As an example, the actual integer values after quantization for the rowwise scenario is depicted in Figure 4.2 for all four parameters separately. While the total number of unique values is only 10, the range spans from -5 to 5. This means that 4 bits are sufficient to represent the quantized values in the resulting layers, as $\lceil \log_2(11) \rceil = 4$, covering all 10 unique values within the range.

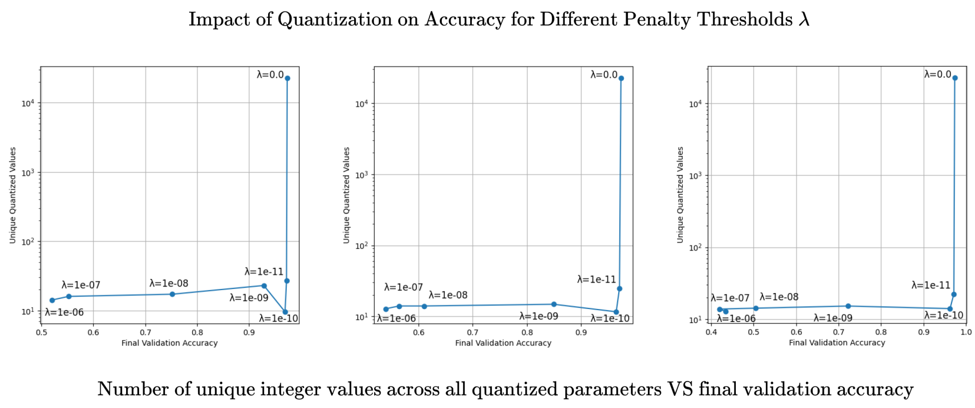


Figure 4.1: Rowwise (left), columnwise (middle) and scalar (right) scaling factor applied to weights of dense layers, with a scalar scaling factor — to biases. We observe optimal quantization for $\lambda = 10$ in all cases. The values are averaged from the results of 5 training runs with different seeds.

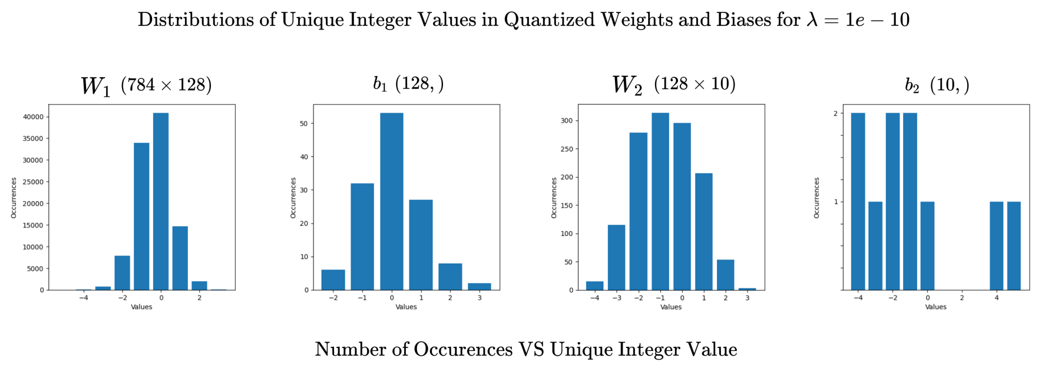


Figure 4.2: Rowwise scaling factors are applied to the weights of dense layers, while a scalar scaling factor is applied to the biases. Results are taken from the run with seed number 42.

There is, however, an interesting phenomenon to highlight. In all scenarios, when the penalty threshold becomes sufficiently large, the last parameter of the network, b^2 with shape (10,), begins to aggressively compensate for the quantization of the other parameters. This observation is supported by the Pareto front plots in Figure 4.3, which, similar to Figure 4.1, illustrate the trade-off between accuracy and quantization. However, instead of the number of unique values, they show the range of the final quantized integer values. The increase in range as the penalty threshold increases is entirely attributable to the last bias vector. This behavior can be explained by the fact that, initially, when the other parameters are aggressively quantized, the last bias vector becomes critically important to the network and is less affected by quantization. However, when it eventually starts to undergo significant discretization, the model is already beyond repair.

Despite the behavior of the last bias vector, we consider this insignificant, as biases are typically not quantized due to their limited contribution to the network's redundancy. By the same token, it does not change the fact that the penalty threshold value of $\lambda = 1e-10$ still remains the optimum for dense layers, as shown in Figure 4.4, where validation accuracy stays nearly unchanged at $\lambda = 1e-10$, while validation loss is within acceptable ranges from that of the baseline with no quantization as depicted in Figure 4.5 — all this despite the parameters being quantized to a range spanning from -5 to 5.

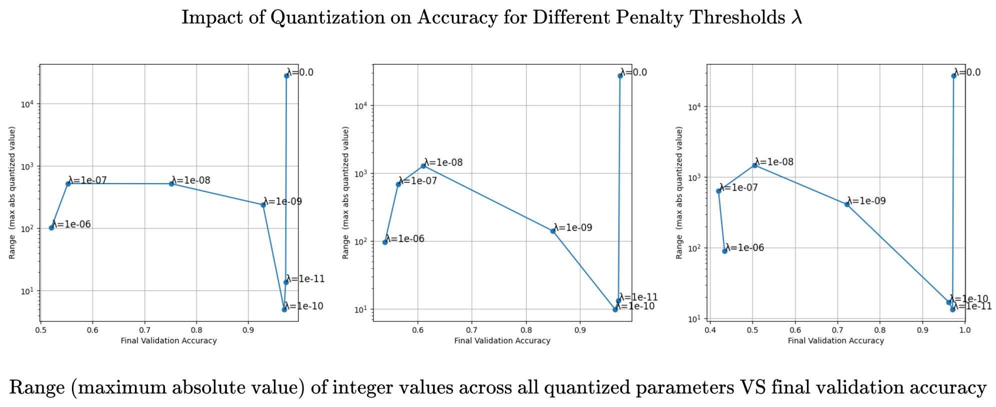


Figure 4.3: Rowwise (left), columnwise (middle) and scalar (right) scaling factor applied to weights of dense layers, with a scalar scaling factor — to biases. The values are averaged from the results of 5 training runs with different seeds.

ADD THE RESULTS WHERE QUANTIZATION STARTS MID TRAINING TO PROVIDE INFO FOR GRADUAL QUANTIZATION.

4.2.2 Quantization Threshold Coefficient

4.3 Systematic Analysis of Custom Loss Terms

4.3.1 Overall Results

4.3.2 Dataset-Specific Insights

4.3.3 Further Implications

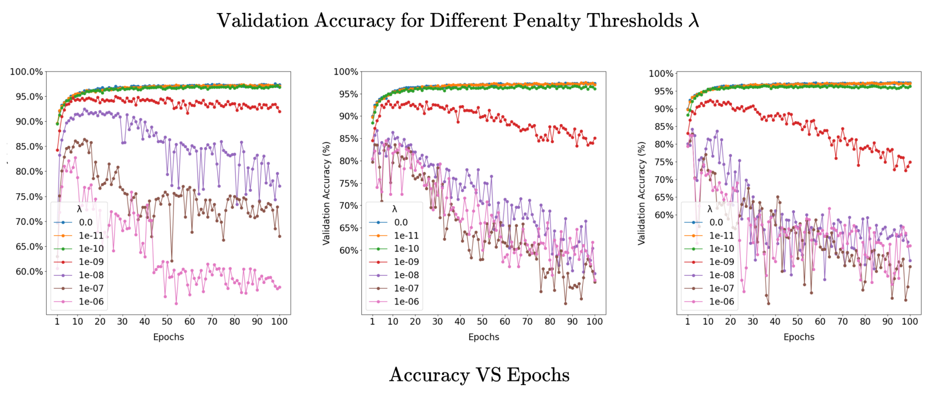


Figure 4.4: Rowwise (left), columnwise (middle) and scalar (right) scaling factor applied to weights of dense layers, with a scalar scaling factor — to biases. Results are taken from the run with seed number 42.

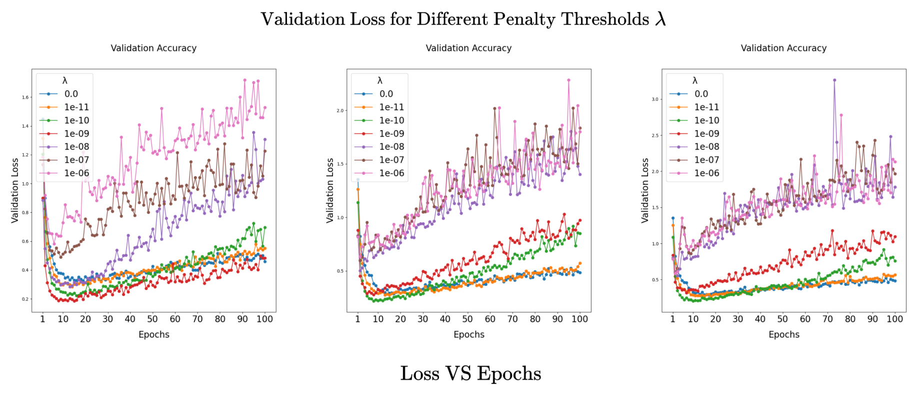


Figure 4.5: Rowwise (left), columnwise (middle) and scalar (right) scaling factor applied to weights of dense layers, with a scalar scaling factor — to biases. Results are taken from the run with seed number 42.

5 Related Work

A significant amount of scientific work has been done on QAT. This research can be categorized based on different characteristics, which are covered separately in the following paragraphs.
Model architecture.
RNN - [33]
CNN - [36]
CNN - [4] DNN - [10] Tranformer bases - [24]
Quantization target parameters.
weights and activations - [25]
weights and activations - [18]
weights - [35]
weigths - [33]
binary weights and input activations - [36]
gradients - [44] layer inputs and weights - [42] weights an activations - [43] weights activations and gradients - [44] **Granularity of quantization.**
Handling of differentiability. STE - [43]
Quantization precision.
binary weights and input activations - [4]
binary weights and activations - [18]
binary weights and input activations - [36]
ternary weights - [33]
higher precision for more important parameters and lowe precision for less important ones - [22]
mixed precision - [39] mixed precision - [5]
Integration with pruning & other techniques.
pruning and Huffman Codign - [13]
distillation - [35]
higher precision for more important parameters and lowe precision for less important ones or pruning - [22] knowledge distillation [40]
Modifications to loss functions. regularization term WaveQ - [6]
proposes a regularization term into the loss function to push the weight values towards +1 and -1 [38]
introduces a novel loss formulation where each quantization has different importance - [15] **Other interesting approaches.** k-means for parameters [10]
I BERT article has a good related work overview

6 Conclusions

Bullet points: - not all layers react similarly to quantization. This gives room for experimenting with different thresholds for different Layers - combined scenarios are not tested - could be tested on standard networks like ALexNet - The scale updates will potentially be improved by some kind of normalization/heuristics

List of Acronyms

ML	Machine Learning

Bibliography

[1] Dorra Ben Khalifa and Matthieu Martel. “Rigorous Floating-Point to Fixed-Point Quantization of Deep Neural Networks on STM32 Micro-controllers”. In: *10th International Conference on Control, Decision and Information Technologies, CoDIT 2024, Vallette, Malta, July 1-4, 2024*. IEEE, 2024, pp. 1201–1206. DOI: 10.1109/CODIT62066.2024.10708400. URL: https://doi.org/10.1109/CoDIT62066.2024.10708400.

[2] Yoshua Bengio, Nicholas Léonard, and Aaron Courville. “Estimating or Propagating Gradients Through Stochastic Neurons for Conditional Computation”. In: *arXiv preprint arXiv:1308.3432* (2013).

[3] Jun Chen et al. “Propagating Asymptotic-Estimated Gradients for Low Bitwidth Quantized Neural Networks”. In: *IEEE J. Sel. Top. Signal Process.* 14.4 (2020), pp. 848–859. DOI: 10.1109/JSTSP.2020.2966327. URL: https://doi.org/10.1109/JSTSP.2020.2966327.

[4] Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. “BinaryConnect: Training Deep Neural Networks with binary weights during propagations”. In: *Advances in Neural Information Processing Systems 28: Annual Conference on Neural Information Processing Systems 2015, December 7-12, 2015, Montreal, Quebec, Canada*. Ed. by Corinna Cortes et al. 2015, pp. 3123–3131. URL: https://proceedings.neurips.cc/paper/2015/hash/3e15cc11f979ed25912dff5b0669f2cd-Abstract.html.

[5] Yinpeng Dong et al. “Stochastic Quantization for Learning Accurate Low-Bit Deep Neural Networks”. In: *Int. J. Comput. Vis.* 127.11-12 (2019), pp. 1629–1642. DOI: 10.1007/S11263-019-01168-2. URL: https://doi.org/10.1007/s11263-019-01168-2.

[6] Ahmed T. Elthakeb et al. “Gradient-Based Deep Quantization of Neural Networks through Sinusoidal Adaptive Regularization”. In: *CoRR* abs/2003.00146 (2020). arXiv: 2003.00146. URL: https://arxiv.org/abs/2003.00146.

[7] Steven K. Esser et al. “Learned Step Size quantization”. In: *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*. OpenReview.net, 2020. URL: https://openreview.net/forum?id=rkgO66VKDS.

[8] Angela Fan et al. “Training with Quantization Noise for Extreme Model Compression”. In: *Proceedings of the International Conference on Learning Representations (ICLR)*. Facebook AI Research, LORIA, Inria. 2021.

[9] Amir Gholami et al. “A Survey of Quantization Methods for Efficient Neural Network Inference”. In: *arXiv preprint arXiv:2103.13630* (2021).

[10] Yunchao Gong et al. “Compressing Deep Convolutional Networks using Vector Quantization”. In: *CoRR* abs/1412.6115 (2014). arXiv: `1412.6115`. URL: `http://arxiv.org/abs/1412.6115`.

[11] Robert M. Gray and David L. Neuhoff. “Quantization”. In: *IEEE Transactions on Information Theory* 44.6 (1998), pp. 2325–2383.

[12] Philipp Gysel et al. “Ristretto: A Framework for Empirical Study of Resource-Efficient Inference in Convolutional Neural Networks”. In: *IEEE Trans. Neural Networks Learn. Syst.* 29.11 (2018), pp. 5784–5789. DOI: `10.1109/TNNLS.2018.2808319`. URL: `https://doi.org/10.1109/TNNLS.2018.2808319`.

[13] Song Han, Huizi Mao, and William J. Dally. “Deep Compression: Compressing Deep Neural Network with Pruning, Trained Quantization and Huffman Coding”. In: *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings*. Ed. by Yoshua Bengio and Yann LeCun. 2016. URL: `http://arxiv.org/abs/1510.00149`.

[14] Geoffrey E. Hinton, Oriol Vinyals, and Jeffrey Dean. “Distilling the Knowledge in a Neural Network”. In: *CoRR* abs/1503.02531 (2015). arXiv: `1503.02531`. URL: `http://arxiv.org/abs/1503.02531`.

[15] Lu Hou, Quanming Yao, and James T. Kwok. “Loss-aware Binarization of Deep Networks”. In: *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net, 2017. URL: `https://openreview.net/forum?id=S1oWlN9ll`.

[16] Jeremy Howard and Sylvain Gugger. “Fastai: A Layered API for Deep Learning”. In: *Inf.* 11.2 (2020), p. 108. DOI: `10.3390/INFO11020108`. URL: `https://doi.org/10.3390/info11020108`.

[17] Gao Huang et al. “Densely Connected Convolutional Networks”. In: *2017 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2017, Honolulu, HI, USA, July 21-26, 2017*. IEEE Computer Society, 2017, pp. 2261–2269. DOI: `10.1109/CVPR.2017.243`. URL: `https://doi.org/10.1109/CVPR.2017.243`.

[18] Itay Hubara et al. “Quantized Neural Networks: Training Neural Networks with Low Precision Weights and Activations”. In: *CoRR* abs/1609.07061 (2016). arXiv: `1609.07061`. URL: `http://arxiv.org/abs/1609.07061`.

[19] Benoit Jacob et al. “Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference”. In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2018, pp. 2704–2713.

[20] Chutian Jiang. “Efficient Quantization Techniques for Deep Neural Networks”. In: *Proceedings of the 2021 International Conference on Signal Processing and Machine Learning*. 2021.

[21] Sangil Jung et al. “Learning to Quantize Deep Networks by Optimizing Quantization Intervals With Task Loss”. In: *IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2019, Long Beach, CA, USA, June 16-20, 2019*. Computer Vision Foundation / IEEE, 2019, pp. 4350–4359. DOI: `10.1109/CVPR.2019.00448`. URL: `http://openaccess.thecvf.com/content%5C_CVPR%5C_2019/html/Jung%5C_Learning%5C_to%5C_Quantize%5C_Deep%5C_Networks%5C_by%5C_Optimizing%5C_Quantization%5C_Intervals%5C_With%5C_CVPR%5C_2019%5C_paper.html`.

[22] Soroosh Khoram and Jing Li. “Adaptive Quantization of Neural Networks”. In: *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*. OpenReview.net, 2018. URL: `https://openreview.net/forum?id=SyOK1Sg0W`.

[23] Daya Shanker Khudia et al. “FBGEMM: Enabling High-Performance Low-Precision Deep Learning Inference”. In: *CoRR* abs/2101.05615 (2021). arXiv: `2101.05615`. URL: `https://arxiv.org/abs/2101.05615`.

[24] Sehoon Kim et al. “I-BERT: Integer-only BERT Quantization”. In: *Proceedings of the 38th International Conference on Machine Learning*. 2021, pp. 5506–5518.

[25] Raghuraman Krishnamoorthi. “Quantizing deep convolutional networks for efficient inference: A whitepaper”. In: *arXiv preprint arXiv:1806.08342* (2018).

[26] Alex Krizhevsky. “Learning Multiple Layers of Features from Tiny Images”. In: (2009), pp. 32–33. URL: `https://www.cs.toronto.edu/~kriz/learning-features-2009-TR.pdf`.

[27] Yann LeCun, Corinna Cortes, and CJ Burges. “MNIST handwritten digit database”. In: *ATT Labs [Online]. Available: http://yann.lecun.com/exdb/mnist* 2 (2010).

[28] Yann LeCun, John S. Denker, and Sara A. Solla. “Optimal Brain Damage”. In: *Advances in Neural Information Processing Systems 2, [NIPS Conference, Denver, Colorado, USA, November 27-30, 1989]*. Ed. by David S. Touretzky. Morgan Kaufmann, 1989, pp. 598–605. URL: `http://papers.nips.cc/paper/250-optimal-brain-damage`.

[29] Qun Li et al. “Investigating the Impact of Quantization on Adversarial Robustness”. In: *CoRR* abs/2404.05639 (2024). DOI: `10.48550/ARXIV.2404.05639`. arXiv: `2404.05639`. URL: `https://doi.org/10.48550/arXiv.2404.05639`.

[30] Pavlo Molchanov et al. “Pruning Convolutional Neural Networks for Resource Efficient Inference”. In: *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net, 2017. URL: `https://openreview.net/forum?id=SJGCiw5gl`.

[31] Pierre-Emmanuel Novac et al. “Quantization and Deployment of Deep Neural Networks on Microcontrollers”. In: *CoRR* abs/2105.13331 (2021). arXiv: `2105.13331`. URL: `https://arxiv.org/abs/2105.13331`.

[32] Kazuki Okado et al. "Channel-wise quantization without accuracy degradation using Δloss analysis". In: *ICMLT 2022: 7th International Conference on Machine Learning Technologies, Rome, Italy, March 11 - 13, 2022*. ACM, 2022, pp. 56–61. DOI: 10.1145/3529399.3529409. URL: https://doi.org/10.1145/3529399.3529409.

[33] Joachim Ott et al. "Recurrent Neural Networks With Limited Numerical Precision". In: *CoRR* abs/1611.07065 (2016). arXiv: 1611.07065. URL: http://arxiv.org/abs/1611.07065.

[34] Eunhyeok Park, Sungjoo Yoo, and Peter Vajda. "Value-Aware Quantization for Training and Inference of Neural Networks". In: *Computer Vision - ECCV 2018 - 15th European Conference, Munich, Germany, September 8-14, 2018, Proceedings, Part IV*. Ed. by Vittorio Ferrari et al. Vol. 11208. Lecture Notes in Computer Science. Springer, 2018, pp. 608–624. DOI: 10.1007/978-3-030-01225-0_36. URL: https://doi.org/10.1007/978-3-030-01225-0%5C_36.

[35] Antonio Polino, Razvan Pascanu, and Dan Alistarh. "Model compression via distillation and quantization". In: *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*. OpenReview.net, 2018. URL: https://openreview.net/forum?id=S1XolQbRW.

[36] Mohammad Rastegari et al. "XNOR-Net: ImageNet Classification Using Binary Convolutional Neural Networks". In: *CoRR* abs/1603.05279 (2016). arXiv: 1603.05279. URL: http://arxiv.org/abs/1603.05279.

[37] Álvaro Domingo Reguero, Silverio Martínez-Fernández, and Roberto Verdecchia. "Energy-efficient neural network training through runtime layer freezing, model quantization, and early stopping". In: *Comput. Stand. Interfaces* 92 (2025), p. 103906. DOI: 10.1016/J.CSI.2024.103906. URL: https://doi.org/10.1016/j.csi.2024.103906.

[38] Wei Tang, Gang Hua, and Liang Wang. "How to Train a Compact Binary Neural Network with High Accuracy?" In: *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence, February 4-9, 2017, San Francisco, California, USA*. Ed. by Satinder Singh and Shaul Markovitch. AAAI Press, 2017, pp. 2625–2631. DOI: 10.1609/AAAI.V31I1.10862. URL: https://doi.org/10.1609/aaai.v31i1.10862.

[39] Zhe Wang et al. "RDO-Q: Extremely Fine-Grained Channel-Wise Quantization via Rate-Distortion Optimization". In: *Computer Vision - ECCV 2022 - 17th European Conference, Tel Aviv, Israel, October 23-27, 2022, Proceedings, Part XII*. Ed. by Shai Avidan et al. Vol. 13672. Lecture Notes in Computer Science. Springer, 2022, pp. 157–172. DOI: 10.1007/978-3-031-19775-8_10. URL: https://doi.org/10.1007/978-3-031-19775-8%5C_10.

[40] Yi Wei et al. "Quantization Mimic: Towards Very Tiny CNN for Object Detection". In: *Computer Vision - ECCV 2018 - 15th European Conference, Munich, Germany, September 8-14, 2018, Proceedings, Part VIII.* Ed. by Vittorio Ferrari et al. Vol. 11212. Lecture Notes in Computer Science. Springer, 2018, pp. 274–290. DOI: `10.1007/978-3-030-01237-3\_17`. URL: `https://doi.org/10.1007/978-3-030-01237-3%5C_17`.

[41] Carole-Jean Wu et al. "Sustainable AI: Environmental Implications, Challenges and Opportunities". In: *CoRR* abs/2111.00364 (2021). arXiv: `2111.00364`. URL: `https://arxiv.org/abs/2111.00364`.

[42] Edouard Yvinec et al. "SPIQ: Data-Free Per-Channel Static Input Quantization". In: *CoRR* abs/2203.14642 (2022). DOI: `10.48550/ARXIV.2203.14642`. arXiv: `2203.14642`. URL: `https://doi.org/10.48550/arXiv.2203.14642`.

[43] Dongqing Zhang et al. "LQ-Nets: Learned Quantization for Highly Accurate and Compact Deep Neural Networks". In: *Computer Vision - ECCV 2018 - 15th European Conference, Munich, Germany, September 8-14, 2018, Proceedings, Part VIII.* Ed. by Vittorio Ferrari et al. Vol. 11212. Lecture Notes in Computer Science. Springer, 2018, pp. 373–390. DOI: `10.1007/978-3-030-01237-3\_23`. URL: `https://doi.org/10.1007/978-3-030-01237-3%5C_23`.

[44] Shuchang Zhou et al. "DoReFa-Net: Training Low Bitwidth Convolutional Neural Networks with Low Bitwidth Gradients". In: *CoRR* abs/1606.06160 (2016). arXiv: `1606.06160`. URL: `http://arxiv.org/abs/1606.06160`.

Appendix

Add additional experimental results that do not need to be directly included in the thesis body.