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 on end-user devices, the need to reduce the memory and computing requirements of such models is becoming ever more pressing. Interestingly, the very redundancy that contributes to the inefficient nature of predictive models, offers the remedy to the malady of resource-intensiveness by creating opportunities for exploitation. One way to exploit the redundancy of Neural Networks (NNs) is to *quantize* them — to approximate their full-precision representations with lower-bit-width counterparts. Among the various quantization approaches, *learned quantization* presents itself as a promising research area with room for contribution, considering the vast landscape of opportunities for ML to autonomously learn their optimal quantization strategies. Hence, this work introduces two techniques for learned quantization, achieving effective model compression without incurring significant accuracy degradation. The first technique involves custom regularization terms that directly take into account quantization goals, the second novel approach incorporates a custom scaling factor gradient calculation that utilizes the gradient of the parameters that are being quantized. As a result, we achieve a memory usage reduction of up to 8× is achieved for the model weights trained on MNIST, CIFAR-10, and Imagenette, a simplified ImageNet dataset.

Zusammenfassung

Machine Learning (ML) Modelle sind dafür bekannt, äußerst ressourcenintensiv zu sein. Da sie immer häufiger auf Endgeräten zum Einsatz kommen, wird die Notwendigkeit, Speicher- und Rechenaufwand zu verringern, zunehmend dringlich. Interessanterweise liefert die Redundanz, die maßgeblich zur Ineffizienz von Neuronalen Netzen beiträgt, zugleich auch den Ansatzpunkt, um genau diese Ineffizienz zu verringern. Eine Möglichkeit, die Redundanz von Neuronalen Netzen (NNs) auszunutzen, ist die Quantisierung – also die Annäherung ihrer Präzisionsdarstellungen an Varianten mit geringerer Bit-Breite. Unter den verschiedenen Ansätzen der Quantisierung erweist sich *Learned Quantization*, oder auch erlernte Quantisierung, aufgrund des breiten Spektrums an Möglichkeiten, bei denen ML Modelle selbstständig optimale Quantisierungsstrategien erlernen können, als ein vielversprechendes Forschungsfeld mit Raum für neue Beiträge. In dieser Arbeit werden daher zwei Techniken der Learned Quantization vorgestellt, die eine effektive Modellkompression ermöglichen, ohne zu wesentlichen Einbußen bei der Genauigkeit zu führen. Die erste Technik verwendet speziell angepasste Regularisierungsterme, die die Ziele der Quantisierung direkt berücksichtigen. Der zweite, neuartige Ansatz integriert eine einzigartige Berechnung des Gradienten für den Skalierungsfaktor, der den Gradienten der zu quantisierenden Parameter nutzt. Mit diesen Methoden lässt sich der Speicherbedarf für die Modellparameter – trainiert auf MNIST, CIFAR-10 und Imagenette (einer vereinfachten Version von ImageNet) – um das bis zu Achtfache reduzieren, ohne dass die Leistung maßgeblich beeinträchtigt wird.

Contents

Abstract vi

Abstract (German) viii

1 Introduction 1

2 Background 3
2.1 Fundamentals of Deep Learning . . . 3
2.1.1 Dense and Convolutional Layers . . . 3
2.1.2 Loss Functions and Regularization . . . 6
2.1.3 Forward-Pass and Back-Propagation . . . 7
2.2 Basics of Quantization . . . 8
2.2.1 Purpose and Definition . . . 8
2.2.2 Core Quantization Approaches . . . 9
2.3 Learned Quantization . . . 11
2.3.1 Trade-offs and Challenges . . . 11
2.3.2 Common Methods . . . 12

3 Learned Quantization Schemes 15
3.1 Nested Quantization Layer . . . 15
3.1.1 Core Logic and Structure . . . 15
3.1.2 Learned Scale Factor . . . 16
3.2 Custom Loss Function Terms . . . 18
3.2.1 Design and Integration . . . 18
3.2.2 Loss Term Definitions . . . 20

4 Experiments 23
4.1 Experimental Setup . . . 23
4.2 Optimal Thresholds for Nested Quantization Layers . . . 23
4.2.1 Fully Connected Layers . . . 24
4.2.2 Convolutional Layers . . . 25
4.3 Systematic Analysis of Custom Loss Terms . . . 27

5 Related Work 31

6 Conclusions 33

Bibliography **35**

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 [3], its 175 billion parameters need a whopping 700 GB of storage in total — 4 bytes for each parameter represented in single-precision floating-point format (FP32). This costliness of modern ML models has increased 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 converting its FP32 parameters to 8-bit integers (INT8), we can reduce its storage requirement from 700 to just 175 GB — a 4× decrease.

Quantization, however, comes with its own trade-offs. While it is claimed that quantization may improve a model's generalization abilities by introducing noise that acts as regularization [7], it is still commonly expected to reduce accuracy. To limit the impact of quantization, a subfield known as *learned quantization* has emerged, focusing on developing schemes that allow models to learn their own quantization parameters during training, while preserving accuracy.

Various approaches of learned quantization have already been proposed [11, 24, 46, 48], each employing unique quantization schemes, while overcoming the inherent issue of non-differentiability of rounding operations. However, most of these methods rely on a predefined bit width, including but not limited to binary quantization [7, 19, 40] or an arbitrary user-defined bit width [48], thereby lacking the ability to dynamically control the "intensity" of quantization during training.

Therefore, in this thesis, we address this gap by contributing the following:

- In Subsection 3.1.2, we introduce a novel method that uses the gradient-to-parameter ratio to determine how much a parameter is adjusted relative to its value. With this ratio and a hyperparameter λ controlling quantization intensity, the model learns its quantizing scaling factors at various granularities.
- We provide a modular framework in Subsection 3.1.1 that can be easily integrated into a wide range of applications and layers with minimal adjustments, ensuring flexibility and usability.
- We investigate a method for the model to learn its scaling factors through a regularization term that can be tuned via a hyperparameter, enabling further control over the "intensity" of quantization.

2 Background

This chapter addresses the theoretical and contextual background necessary to understand the key concepts and methodologies that form the foundation of this thesis. The first Section 2.1 introduces the fundamentals of deep NNs, which serve as a basis for the discussion. The next Section 2.2 broadens the perspective by exploring the concept of quantization, followed by a final Section 2.3 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 in Subsection 2.1.1 and progressing to loss functions with regularization in Subsection 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:

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

- $input \in \mathbb{R}^d$ 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.
- $w \in \mathbb{R}^d$ (weights) is a vector of parameters associated with that specific neuron, defining the importance of each input received by this neuron.
- $b \in \mathbb{R}$ (bias) is an additional scalar parameter specific to the neuron, which shifts the result of the weighted sum, allowing for more flexibility.
- $f(\cdot) : \mathbb{R} \to \mathbb{R}$ is the *activation function*, a nonlinear function applied to the weighted sum of inputs and bias in that specific neuron, allowing for more complexity.

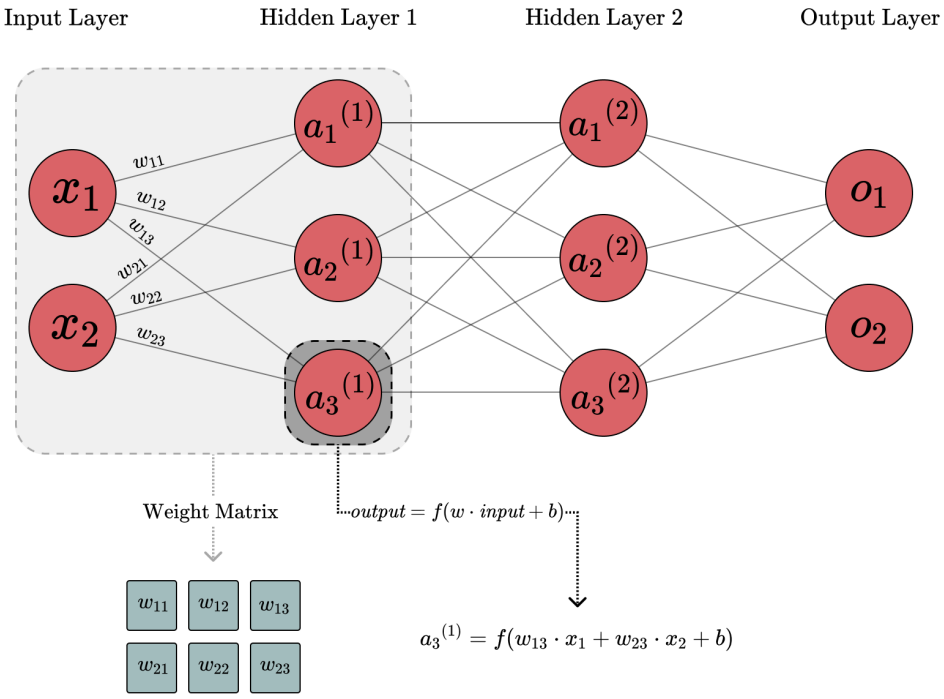

Figure 2.1: An example of a NN with two hidden dense layers, showing the connections between neurons in adjacent layers.

- *output* $\in \mathbb{R}$ 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*. The *weight matrix* (W), which combines the weight vectors of all neurons in a dense layer, grows linearly with each added neuron or input node, embodying the interconnectedness of such layers.

This interconnectedness introduces the inherent redundancy, or the over-parameterizedness of NNs [13]. 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 [20].

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, or *kernel*, over an input to find patterns. Figure 2.2, for example, shows the application of the Sobel kernel that detects edges on the input image.

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 includes

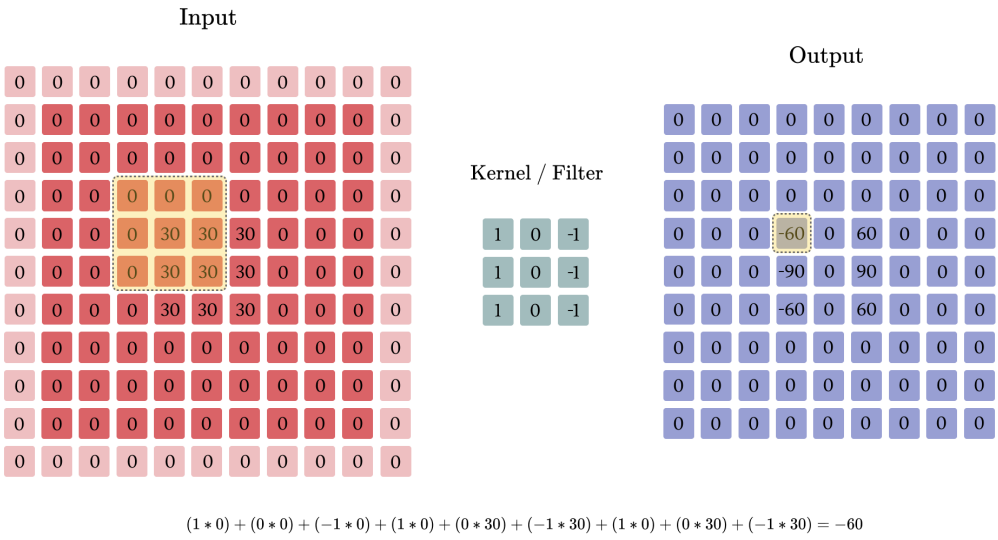

Figure 2.2: A 3×3 kernel sliding over a padded input matrix to compute the output feature map.

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.

Mathematically, a convolutional layer can be represented as:

$$y_{i,j,k} = f\left(\sum_{m=1}^{c}\sum_{p=1}^{h}\sum_{q=1}^{w} x_{i+p-1,j+q-1,m} \cdot w_{p,q,m,k} + b_k\right)$$

- h, w are the height and width of the filter, respectively.
- c 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.
- $f(\cdot) : \mathbb{R} \to \mathbb{R}$ is the activation function.

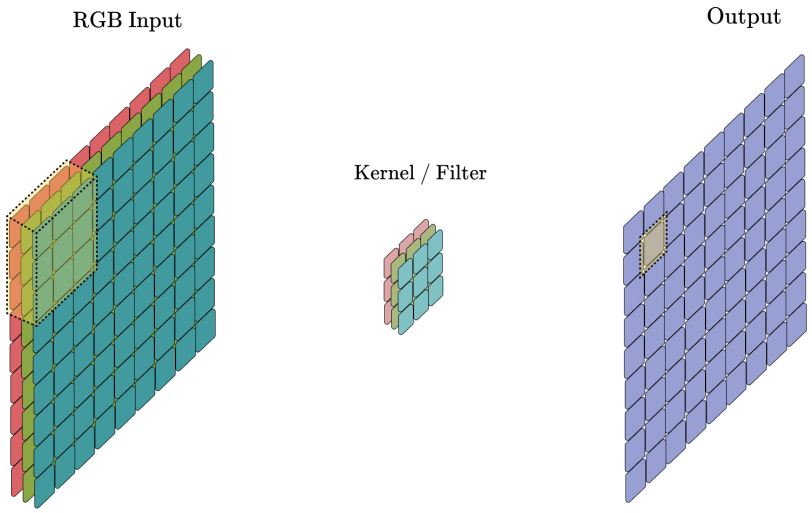

Figure 2.3: A 3×3x3 kernel (filter) sliding over an RGB input matrix to produce a single-channeled output feature map.

In other words, 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 [18], presenting an opportunity for quantization. Together, dense and convolutional layers form the building blocks of convolutional neural networks (CNNs), the core of breakthroughs in computer vision [29] [28] [42]. Thus, both dense and convolutional layers will be the focus of this work.

2.1.2 Loss Functions and Regularization

The weights and biases of NN layers 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.

The loss function alone cannot ensure a neural network generalizes well to unseen data, as it focuses solely on minimizing error on the training set, which can lead to overfitting or a failure to capture the desired generalization properties. To mitigate overfitting and promote generalization, a *regularization term* is added to the loss function to penalize unwanted behaviors.

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:

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

- $\mathcal{L}_{\text{task}}$ is the original task-specific 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.

Regularization terms may have different objectives, be it for better generalization, improved robustness, or encouraging sparsity in the model parameters. The current work employs multiple custom regularization terms that encourage quantization that we will discuss in detail in Section 3.2.

2.1.3 Forward-Pass and Back-Propagation

The repetition of the mathematical operations described earlier in 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. This output is then compared with the ground truth by the loss function that produces an error as explained in Subsection 2.1.2. The error is then 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 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, as shown below:

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

- $\frac{\partial L}{\partial o_1}$ 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.

The chain rule, and consequently backpropagation, is a critical bottleneck for quantization. This issue, along with its solution, will be detailed in Section 2.3.

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 [44]. In this regard, studies have examined quantization within the context of Green AI as a method to reduce the carbon footprint of ML models [41].

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 [40] [46] [1][36].

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 [31]. 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 [14]. 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 [13].

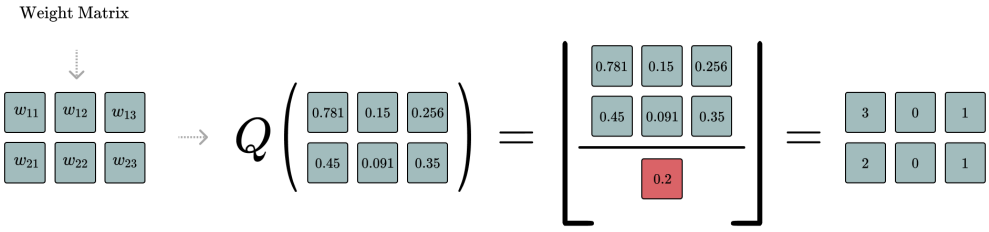

Figure 2.4: Uniform quantization of an arbitrary weight matrix.

Since ML models are generally considered redundant or over-parameterized, there are multiple points where quantization can be applied. Specifically, 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 Chapter 5. For now, we will focus on a few basic approaches from the general categories of both *data-driven* and *data-free* methods [45] to provide a basic understanding of the NN quantization process.

A basic form of data-free quantization, or *post-training quantization* [23], involves converting already trained parameters from FP32 to a lower bit-width format without using the initial training data. One approach is to apply *uniform quantization* that maps real values to a set number of *bins*. The formula can be written as:

$$Q(r) = \left\lfloor \frac{r}{S} \right\rfloor$$

- $Q(\cdot)$ denotes the quantization operation.
- r is the real value of a given model parameter in higher bit-width representation.
- S is a scaling factor.

As a result, we end up with values that can directly be cast to a lower bit-width representation, forming a discrete set of bins instead of an almost continuous range of real numbers, as shown in Figure 2.4.

Unlike data-free quantization — as the name suggests — data-driven quantization involves retraining the model using the initial data. An example of this approach is the Ristretto framework [15], 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

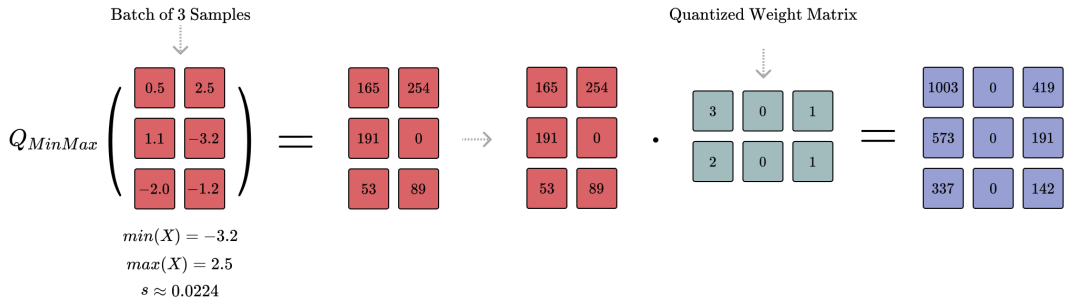

Figure 2.5: Min-max quantization of input data to 8 bits, followed by matrix multiplication with the quantized weight matrix.

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 basic example of data-driven quantization is min-max quantization on input data, as shown in Figure 2.5, using the formula for 8-bit quantization:

$$Q_{MinMax}(x) = \left\lfloor \frac{x - \min(X)}{s} \right\rfloor, \quad \text{where} \quad s = \frac{\max(X) - \min(X)}{2^8 - 1}$$

This method can also be implemented 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.

Previously, we discussed *where* quantization could be applied in a model, mentioning weights, kernels, and biases as the focus of this thesis. Figure 2.4 and Figure 2.5 show examples of quantization using a scalar scaling factor. However, scaling factors can be applied at varying levels of detail, bringing forth the concept of *quantization granularity*.

Granularity refers to the level of detail at which *quantizers* — such as scaling factors, offset adjustments, or other binning strategies — are applied, ranging from a single quantizer for an entire kernel (coarse granularity) to separate ones for individual spatial locations, channels, or filters (fine granularity).

Figure 2.6 illustrates various scaling factor granularities for the kernels of convolutional layers. Despite this wide range of possibilities, channel-wise quantization is currently the standard for convolutional layers [13], 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 [25]. Thus, in the experiments, we focus our quantization methods on channel-wise granularity for

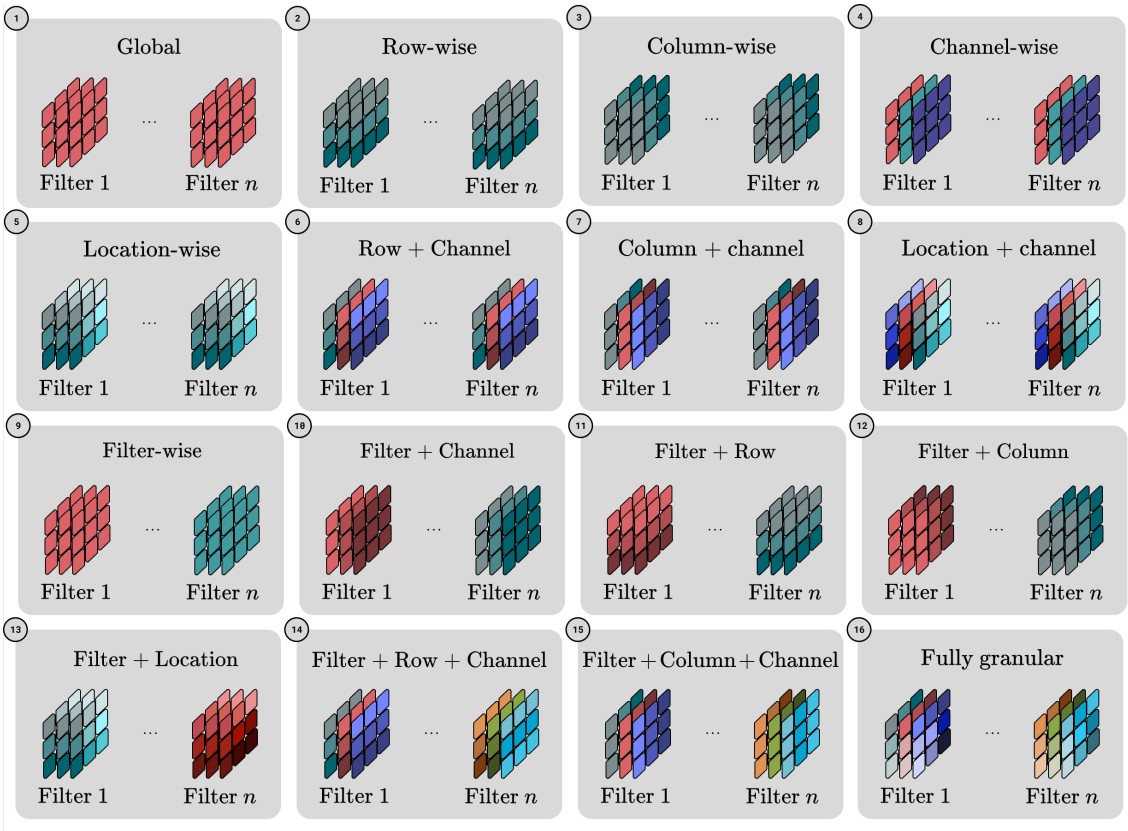

Figure 2.6: Various applications of scaling factors, ranging from a single scalar applied to the entire kernel (1) to separate scalars assigned to spatial dimensions (e.g., 1, 2), channels (4), filters (9), and other granular configurations.

convolutional layers and row-wise granularity for dense layers, while also exploring a few additional applications.

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 widely 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 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 [46] [11], 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 [38]. 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 $\lfloor \cdot \rfloor$ to hidden layer outputs:

$$a_{1,q}{}^{(1)} = \lfloor a_1{}^{(1)} \rfloor$$

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 $a_{1,q}{}^{(1)} = \lfloor a_1{}^{(1)} \rfloor$, 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] [12] [46]. 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 [4] — are the cornerstone of quantization-aware training (QAT) [22], 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) [24], 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 [11] defines a single trainable quantization parameter (step size) with an explicit gradient formula and also uses STE for standard parameter updates. LQ-Nets [46] 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.

Another way to achieve learned quantization is through the use of a regularization term. Such regularization terms are often periodic functions, such as sinusoidal functions [35] [8], which pull model parameters toward the nearest step — or quantization bin. A notable example is WaveQ [9], which combines a sine regularizer with an additional term that encourages lower bit widths without predefined bit constraints. A more straightworfward regularizer is the Mean Squared Quantization Error [6] that directly minimizes the difference between the high-precision parameters and their quantized coutnerparts.

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 with adjustable intensity. The first one, a custom quantization layer featuring a threshold for gradient based scale updates, will be discussed in Section 3.1. The second scheme, which focuses on custom regularization terms with a configurable penalty rate, will be covered second in Section 3.2.

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 parameters 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:

$$P_{\text{quantized}} = \left\lfloor \frac{P}{s} \right\rfloor$$

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 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 of $P_{\text{reconstructed}}$ and applies a custom logic, which will be detailed in Subsection 3.1.2. Essentially, this approach solves two problems with one tool — updating both P and s using the same gradient, though 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.

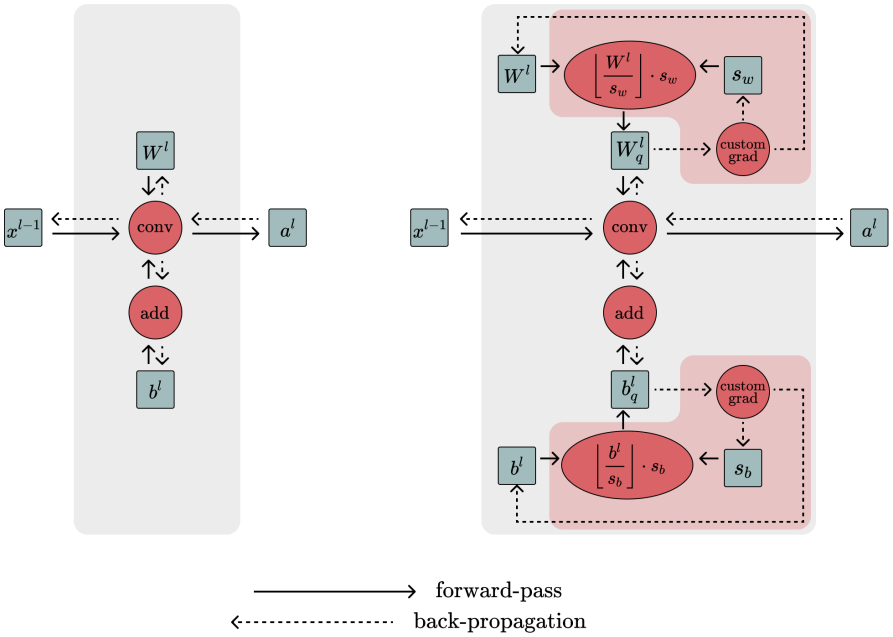

Figure 3.1: A convolutional layer (left) and its integration with the nested quantization layer (right) for both weights and bias. Trainable scaling factors are updated using custom gradients. Parameter gradients are passed downward as is.

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` 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 hyper-parameter 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 denote $P_{reconstructed}$ as P_r):

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

The formula conveys the relative impact of the gradient on the parameter's value. A large r 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:

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

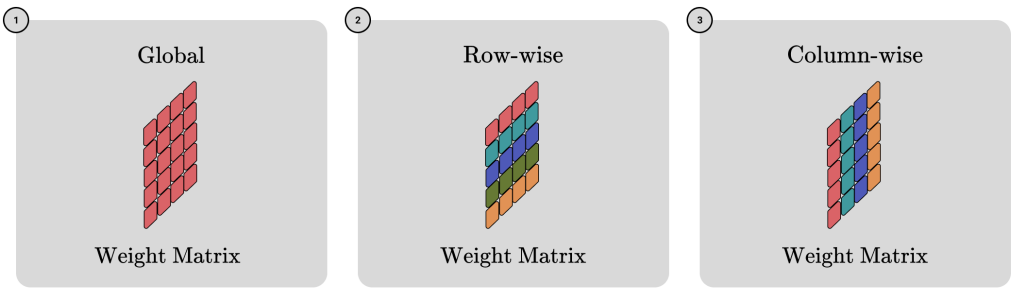

Figure 3.2: Various applications of scaling factors, ranging from a single scalar applied to the entire weight matrix (1) to row-wise and column-wise application of vector scalers.

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.

Now, what if all parameters are deemed sensitive with $r \geq \lambda$ and $g_s = 0$? This would mean that scale factor s does not receive gradient updates. Such a situation could result in an unfavorable outcome where the parameters themselves increase in magnitude while the scale factor remains unchanged, loosening the achieved quantization. To address this, we apply a small constant update tied to λ in cases where all parameters have $r \geq \lambda$. As a result, the initial formula for g_s becomes:

$$g_s = \begin{cases} -\tanh(\lambda), & \text{if } \forall P_r,\ r \geq \lambda \\ -\tanh(\lambda - r) \cdot \mathbf{1}(r < \lambda), & \text{otherwise} \end{cases}$$

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". Even when all parameters are sensitive and each casts a "zero vote", we still force the "elections" by introducing a constant gradient term to avoid stagnation in scale factor updates.

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 Function Terms

Building on the nested quantization layers covered in Section 3.1, we define three custom loss function terms that can be used in combination with the task-specific loss. We will first explain how these terms fit into the model's structure and training process, then provide the details of their individual formulations.

3.2.1 Design and Integration

To incorporate custom loss terms into the model's training process, we utilize the nested layers described in Section 3.1, with a slight modification. The modification lies in the

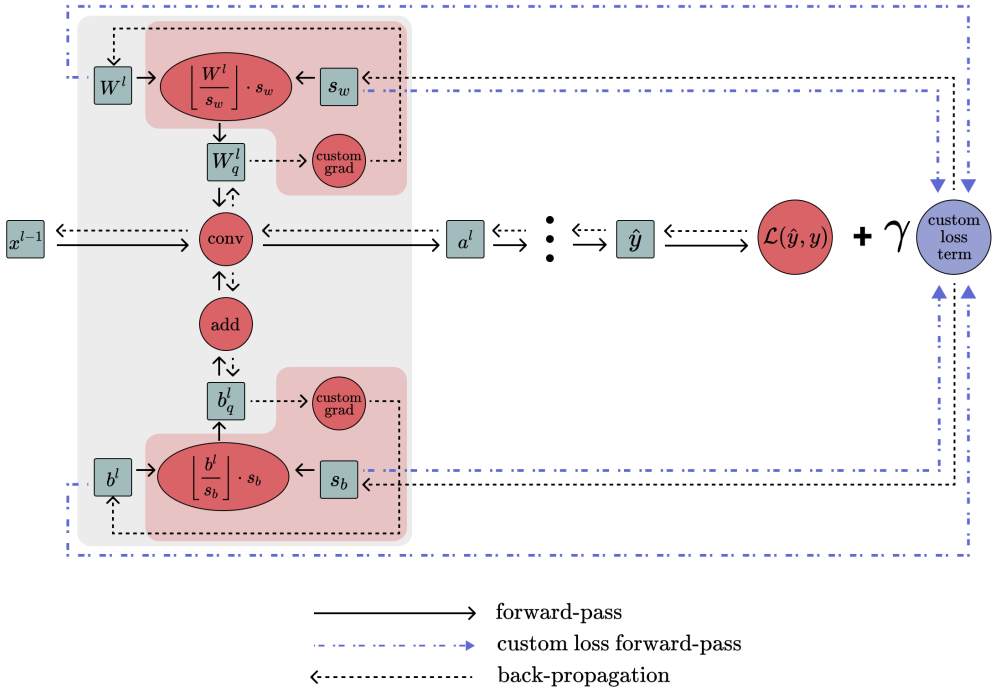

Figure 3.3: Custom loss term using scale factors and model parameters integrated into training.

gradient calculation of the nested layers, where we similarly use the STE for parameter updates, but do not provide any gradient updates for the scale factors altogether. This is equivalent to setting $\lambda = 0$, which disables gradient updates for the scale factors.

Instead, to update scale factors, we define a custom loss term $\mathcal{L}_{custom}$ scaled by a penalty rate γ, an adjustable hyperparameter. This loss term depends on the model parameters and scale factors and is added to the task-specific loss to produce the total loss:

$$\mathcal{L}_{\text{total}} = \mathcal{L}_{\text{task}} + \gamma \cdot \mathcal{L}_{\text{custom}}$$

As visualized in Figure 3.3, scale factors receive gradient updates from the custom loss term, while the gradients for model parameters are influenced by both the task-specific loss and the custom loss term. To illustrate this further, if we set $\gamma = 0$, the scale factors will not receive any updates, and the model will rely on the STE to adapt to the initial values of the scale factors during training.

The custom loss terms are implemented as Python objects to which the model's nested quantization layers are passed. These objects have access to the latest values of parameters and scale factors at any point during training, which provides flexibility in how we encourage quantization.

Unlike the ratio-based approach described in Section 3.1, this approach defines a for-

ward pass to calculate the custom loss terms, rather than directly manipulating gradient calculations. As a result, it may appear more straightforward and will be shown to achieve equally good results in Chapter 4.

3.2.2 Loss Term Definitions

We use the same notation, with P representing the parameters of a nested quantization layer (weights, bias or kernel) and s as the corresponding scale factor. Additionally, we denote the elements in the scale factor s as $\{s_1, \ldots, s_K\}$.

MaxBin Penalty. We name the first custom loss term MaxBin Penalty and define the function $\text{maxbin}(P, s)$ as the collection of per-s_k maxima:

$$\text{maxbin}(P, s) = \left(\max_{(i,j):s_{i,j}=s_k} \frac{|P_{i,j}|}{s_k} \right)_{k=1}^{K}$$

where $P_{i,j}$ is an element of P, and $s_{i,j}$ — the corresponding scaling element. The result is of the same shape as s and conveys the maximum bin value, or range, for each s_k.

For the weights W^l and bias b^l of a nested quantization layer indexed by l, we apply $\text{maxbin}(\cdot)$ to compute the maximum bin values relative to s_W^l and s_b^l:

$$\text{bin\_max}_{W^l} = \text{maxbin}(W^l, s_W^l), \quad \text{bin\_max}_{b^l} = \text{maxbin}(b^l, s_b^l)$$

We then calculate the layer-specific MaxBin penalty $\mathcal{L}_{MaxBin}^l$ as the weighted sum of the mean maximum bin values for W^l and b^l, scaled by their total number of parameters:

$$\mathcal{L}_{MaxBin}^l = \#(W^l) \cdot \text{mean}\Big(\text{bin\_max}_{W^l}\Big) \; + \; \#(b^l) \cdot \text{mean}\Big(\text{bin\_max}_{b^l}\Big)$$

where $\#(W^l)$ and $\#(b^l)$ represent the total number of parameters in W^l and b^l.

As the next step, we sum these layer-wise penalties over all nested quantization layers $l \in L$, then normalize by the total number of parameters in those layers.

$$\mathcal{L}_{MaxBin} = \frac{\sum_{l \in L} \mathcal{L}_{MaxBin}^l}{\sum_{l \in L} \Big(\#(W^l) + \#(b^l) \Big)}$$

In the end, we scale $\mathcal{L}_{MaxBin}$ with an adjustable penalty rate γ and integrate it into the loss objective as follows:

$$\mathcal{L}_{total} = \mathcal{L}_{task} + \gamma \cdot \mathcal{L}_{MaxBin}$$

where $\mathcal{L}_{task}$ represents the task-specific loss function, and γ controls the contribution of the MaxBin penalty to the total loss.

Intuitively, the MaxBin penalty encourages larger scale factor values that result in a small quantization range, while discouraging the ones that result in higher bin values.

Inverse Penalty. We name the second custom loss term Inverse Penalty and define the function $\text{inverse}(s)$ as the mean of the inverse scale factor values:

$$\text{inverse}(s) = \text{mean}\left(\frac{1}{s_{i,j}}\right)$$

where $s_{i,j} > 0$, since s has a positive non-zero constraint.

For the weights W^l and bias b^l of a nested quantization layer l, with scale factors s_W^l and s_b^l, the layer-specific inverse penalty is defined as:

$$\mathcal{L}_{Inv}^l = \#(W^l) \cdot \text{inverse}\big(s_W^l\big) \; + \; \#(b^l) \cdot \text{inverse}\big(s_b^l\big)$$

As the next step, we sum these penalties across all nested quantization layers $l \in L$, and normalize by the total number of parameters in those layers:

$$\mathcal{L}_{Inv} = \frac{\sum_{l \in L} \mathcal{L}_{Inv}^l}{\sum_{l \in L}\Big(\#(W^l) + \#(b^l)\Big)}$$

Finally, we scale $\mathcal{L}_{Inv}$ with an adjustable penalty rate γ and integrate it into the loss objective, just like we did with the MaxBin Penalty.

Unlike the Maxbin Penalty, the Inverse Penalty directly encourages larger scale factors, without taking into account the resulting bin values.

Difference Penalty. We name the third custom loss term Difference Penalty and define the following function:

$$\text{difference}(P, s) \; = \; \text{mean}\left(\left|P - \frac{P}{s}\right|\right)$$

which measures the average absolute difference between the original parameters P and their scaled values, where s is the corresponding scale factor.

For a nested quantization layer l with weights W^l and biases b^l, and their respective scale factors s_W^l and s_b^l, the layer-specific Difference Penalty is given by:

$$\mathcal{L}_{Diff}^l \; = \; \#(W^l) \cdot \text{difference}\Big(W^l,\, s_W^l\Big) \; + \; \#(b^l) \cdot \text{difference}\Big(b^l,\, s_b^l\Big)$$

Next, we sum these Difference Penalties across all nested quantization layers and normalize by the total number of parameters in these layers the same way we did for $\mathcal{L}_{MaxBin}$ and $\mathcal{L}_{Inv}$. In the end we similarly add it to the task-specific loss scaled by γ.

Intuitively, the Difference Penalty encourages larger scale factors, as smaller ones result in a greater difference between the original and scaled parameters, increasing the penalty.

To summarize, while these custom loss terms are the only source of gradient updates for the scale factors, weights and biases are updated both through the main task-specific loss and, where relevant, through the custom loss terms. The Inverse Penalty, for instance, affects only the scale factors and does not directly influence the weights or biases.

We note that during the forward-pass, parameters are quantized by their respective scale factors, with the STE ensuring that the weights and biases can adapt to the learned quantization in the backward-pass.

In Chapter 4, we will present the comparison of the three loss terms and provide guidance on the optimal values of γ.

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 [30], CIFAR-10 [27], and Imagenette [17], 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

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

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
Seeds for Reproducibility	1-3,42,1337	1-3,42,1337	1-3,42,1337

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 have weight matrices W_1 and W_2, along with bias vectors b_1 and b_2. We examine three scenarios: applying the scale factor row-wise, column-wise, and as a single scalar for the entire weight matrix. For all scenarios, a scalar scale factor is used for each bias vector. The configurations are shown in Table 4.2.

The experiments have demonstrated 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. We see 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 24, the range spans from -12 to 11. This means that 5 bits are sufficient to represent the quantized values in the resulting layers, as $\lceil \log_2(24) \rceil = 5$, covering all 24 unique values within the range. This can even be further reduced using compression techniques like Huffman coding [21], considering the concentration of values around 0 in the distributions of weight quantizations.

Interestingly, as depicted in Figure 4.4, the validation loss for $\lambda = 1e - 10$ consistently remains lower than the baseline $\lambda = 0.0$ throughout the training process. At the

Table 4.2: Scale Factor Granularity

Parameter	**Row-wise**	**Column-wise**	**Scalar**
W_1 $(784, 128)$	s $(784, 1)$	s $(1, 128)$	s $(1, 1)$
b_1 $(128, 1)$	s $(1, 1)$	s $(1, 1)$	s $(1, 1)$
W_2 $(128, 10)$	s $(128, 1)$	s $(1, 10)$	s $(1, 1)$
b_2 $(10, 1)$	s $(1, 1)$	s $(1, 1)$	s $(1, 1)$

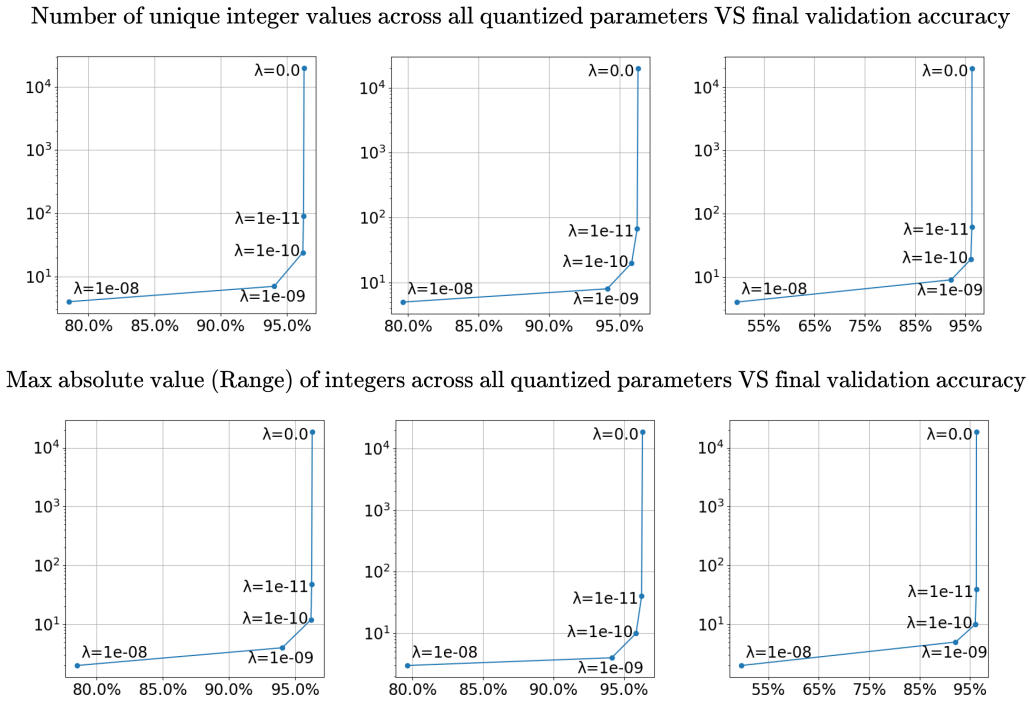

Figure 4.1: Impact of quantization on accuracy for different λ. Rowwise (left), column-wise (middle) and scalar (right) scaling factors applied to weights of dense layers, with a scalar scaling factor — to biases.

same time, the validation accuracy, shown in Figure 4.3 remains almost identical to the baseline. This indicates that the penalty threshold $\lambda = 1e - 10$ not only preserves classification accuracy but also improves generalization in terms of loss, making the model more confident in its predictions.

In terms of quantization granularity, we see that a scalar scaling factor applied to weights produces the worst results compared to row-wise and column-wise granularity. This is evident from the steeper accuracy degradation and greater loss increase for $\lambda = 1e - 8$, as shown in Figure 4.3 and Figure 4.4. The poor performance of the scalar scaling factor suggests that layer-wise quantization, which it represents, is too coarse to account for the variability in parameter distributions within a layer. Therefore, finer granularities, such as row-wise or column-wise scaling, are more suitable candidates for optimal quantization granularity of dense layers.

4.2.2 Convolutional Layers

For the CIFAR-10 dataset, we define a CNN, with six nested convolutional layers. Each convolutional layer has a kernel and a bias. We examine four scenarios: applying the scale factor channel-wise, row-wise, column-wise, and as a single scalar for the entire

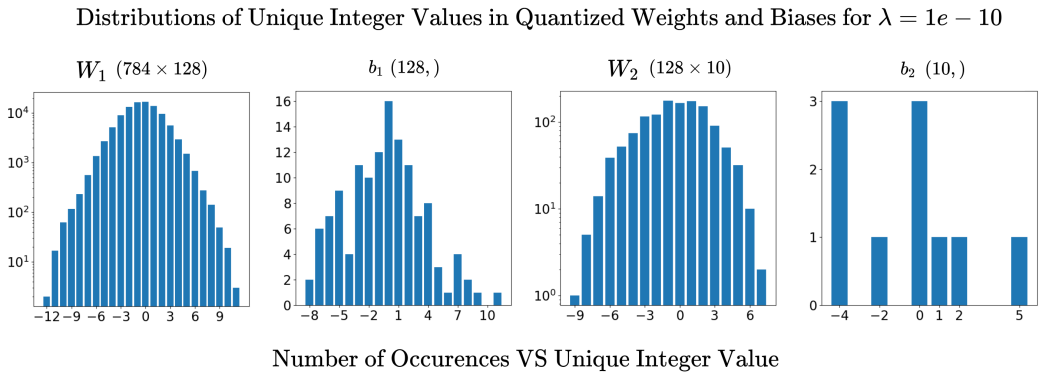

Figure 4.2: Rowwise scaling factors applied to the weights of dense layers, with a scalar scaling factor — to the biases.

kernel. For all scenarios, a scalar factor is used for each bias vector, similar to how we quantized dense layers. The configurations are shown in Table 4.3 for clarity.

The experiments show that the optimal value for the hyperparameter is $\lambda = 1e-11$, across all four scenarios, as depicted in Figure 4.5, with all scenarios producing approximately the same results.

As an example, the integer values after quantization for the row-wise scenario across all quantized parameters range from -22 to 32, resulting in 60 unique values that can be represented with $\lceil \log_2(60) \rceil = 6$ bits.

Compared to the quantization of dense layers, we notice in Figure 4.7 that the validation loss does not decrease with quantization for $\lambda = 1e-11$, which yields the best results. Additionally, as evident from Figure 4.6, the accuracy degrades slightly, indi-

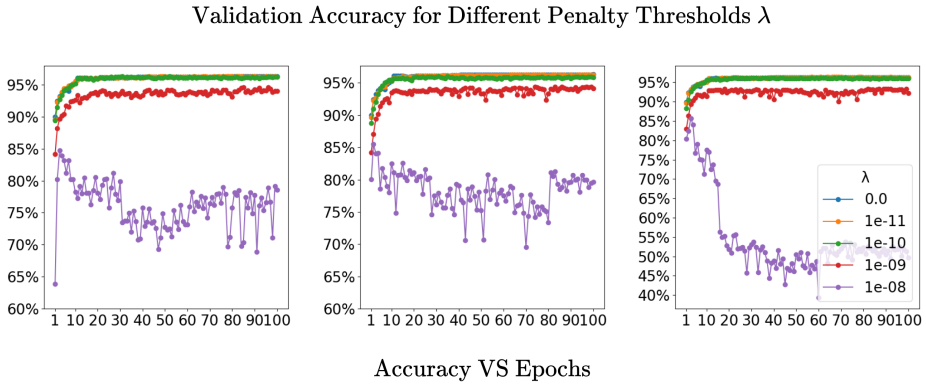

Figure 4.3: Rowwise (left), columnwise (middle) and scalar (right) scaling factor applied to weights of dense layers, with a scalar scaling factor — to biases.

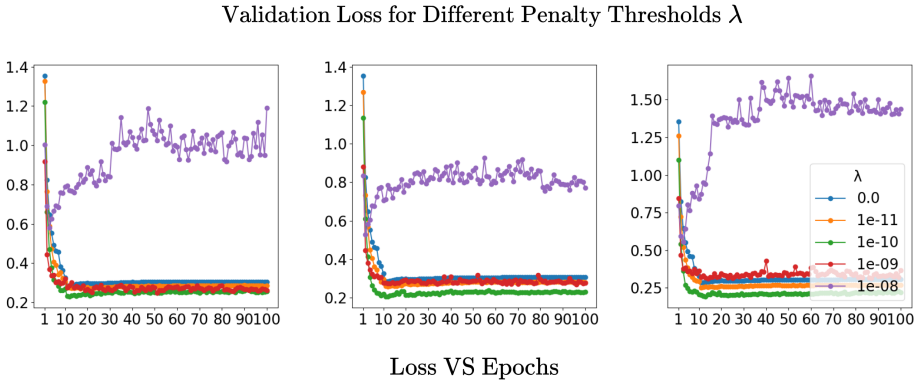

Figure 4.4: Rowwise (left), columnwise (middle) and scalar (right) scaling factor applied to weights of dense layers, with a scalar scaling factor — to biases.

Table 4.3: Scale Factor Granularity for Convolutional Kernels

Kernel	**Channel-wise**	**Row-wise**	**Column-wise**	**Scalar**
K_{1a} $(3, 3, 3, 32)$	$(1, 1, 3, 1)$	$(3, 1, 1, 1)$	$(1, 3, 1, 1)$	$(1, 1, 1, 1)$
K_{1b} $(3, 3, 32, 32)$	$(1, 1, 32, 1)$	$(3, 1, 1, 1)$	$(1, 3, 1, 1)$	$(1, 1, 1, 1)$
K_{2a} $(3, 3, 32, 64)$	$(1, 1, 32, 1)$	$(3, 1, 1, 1)$	$(1, 3, 1, 1)$	$(1, 1, 1, 1)$
K_{2b} $(3, 3, 64, 64)$	$(1, 1, 64, 1)$	$(3, 1, 1, 1)$	$(1, 3, 1, 1)$	$(1, 1, 1, 1)$
K_{3a} $(3, 3, 64, 128)$	$(1, 1, 64, 1)$	$(3, 1, 1, 1)$	$(1, 3, 1, 1)$	$(1, 1, 1, 1)$
K_{3b} $(3, 3, 128, 128)$	$(1, 1, 128, 1)$	$(3, 1, 1, 1)$	$(1, 3, 1, 1)$	$(1, 1, 1, 1)$

Note: $(1, 1, 1, 1)$ denotes a scalar, broadcasted across the kernel for consistency.

cating that the quantization process introduces some impact on the model's predictions. However, the pareto plots in Figure 4.5

4.3 Systematic Analysis of Custom Loss Terms

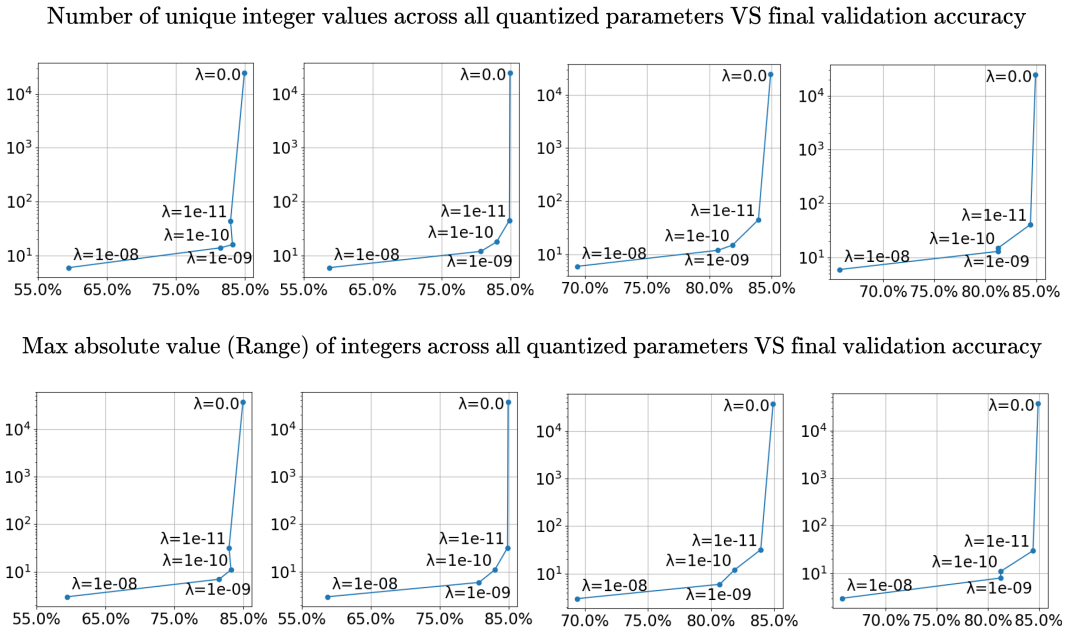

Figure 4.5: Impact of quantization on accuracy for different λ. Channel-wise, row-wise, column-wise and scalar scaling factors applied to kernels of convolutional layers, with a scalar scaling factor — to biases.

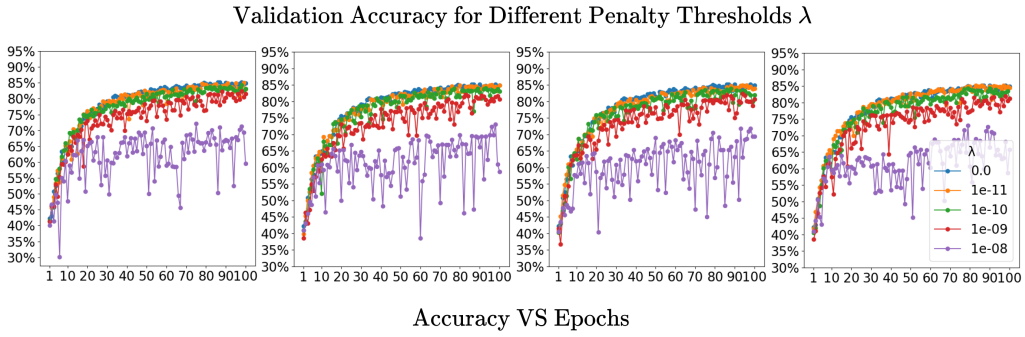

Figure 4.6: Channel-wise, row-wise, column-wise and scalar scaling factor applied to kernels of convolutional layers, with a scalar scaling factor — to biases.

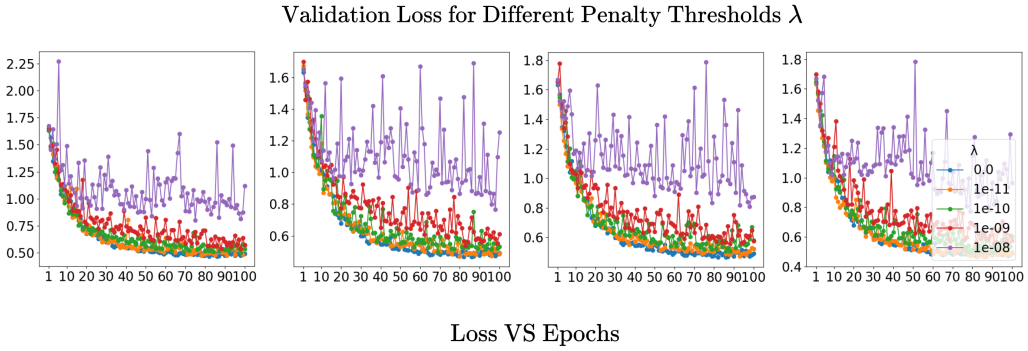

Figure 4.7: Channel-wise, row-wise, column-wise and scalar scaling factor applied to kernels of convolutional layers, with a scalar scaling factor — to biases.

5 Related Work

The hyperparameter space of deep neural networks is so vast that classifying research in the field of learned quantization into rigid groups is inherently challenging. Nevertheless, we aim to outline a rough distinction between different approaches by emphasizing their most apparent aspects, after first establishing the position of learned quantization within the broader field of quantization.

Data-Free vs. Data-Driven Quantization. As discussed in Chapter 2, data-free quantization relies solely on the intrinsic properties of ML models, such as weight distribution statistics, to determine quantization parameters [34]. In contrast, data-driven quantization uses the original training data to fine-tune the model to the quantization strategy [45]. Although learned quantization does not strictly align with the definition of a fine-tuning process, it incorporates the full training pipeline, jointly optimizing both the model and the quantization parameters. This characteristic makes it inherently data-driven. Thus, we regard learned quantization as a distinct subcategory within data-driven quantization approaches.

PTQ vs. QAT. PTQ [49] can fall into either the data-free or data-driven category, depending on the parameters being quantized. If it's just weights, it's data-free since weights can be quantized without relying on input data. However, if activations are included, it's data-driven because activation quantization needs input data for calibration. Since PTQ doesn't involve training on the full dataset, it's distinct from learned quantization. QAT, on the other hand, does involve full training, where the model adapts to the quantization strategy. This makes it part of learned quantization, which may also involve learning the quantization parameters themselves as part of the process.

Static vs. Dynamic Quantization. Static quantization is an offline method that quantizes weights and activations before inference, whereas dynamic quantization is applied on-the-fly during inference. Both static and dynamic quantization methods are

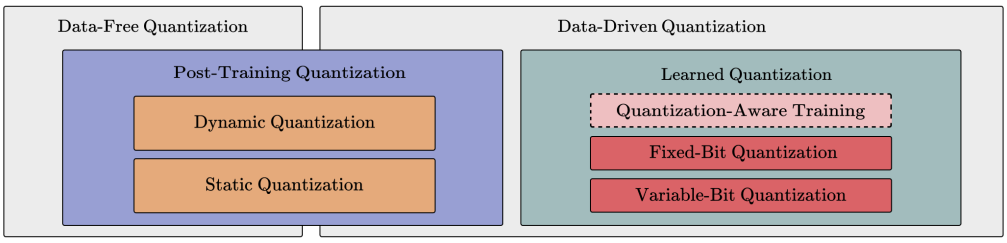

Figure 5.1: Learned quantization in a broader context.

post-training approaches. In the context of learned quantization, static quantization might refer to a fixed quantization scheme, and dynamic quantization — to a scheme that depends on trainable quantizers. However, these terms are not commonly used this way in the literature. Therefore, we consider static and dynamic quantization separately from learned quantization.

Having established the broader context of quantization, we will now attempt to classify the various approaches within learned quantization itself.

Fixed-Bit vs. Variable-Bit. Learned quantization schemes can be roughly divided into two broader groups. The first group involves quantization schemes that use a fixed target bit size, while the second group aims to learn or effectively determine the lowest feasible bit size. This distinction is not widely discussed in the literature, but we view it as a recognizable pattern. For example, LQ-Nets [46] compute binary encodings of model parameters using a predefined bit size, whereas LSQ [11] learns step sizes in a way that can adaptively reduce bit width requirements.

Quantization Target. NNs offer multiple opportunities for applying quantization. While some learned quantization approaches focus exclusively on model weights [39] [37] [7] [10], others extend this to include activations [26] [20] [40] [45] [46] and even gradients [32] [47]. As notable examples, DoReFa-Net [48] quantizes all three with arbitraty bitwidths, whereas Quantized Neural Networks [20] constrain weights and activations to either -1 or $+1$, and quantize gradients to 6 bits.

Quantization Precision. The target lower bit precision, and whether it is adjustable, varies between methods — ranging from rigid constraints to only a few values, to the flexibility of setting the desired bitwidth. For instance, BinaryNet [19] and XNOR-Net [40] represent extreme cases of quantization, where weights and activations are binarized. Ternary Weight Networks [33] take a slightly less aggressive approach by quantizing weights to $-1, 0, 1$. In contrast, DoReFa-Net [48] allows the target bit size to be defined.

Quantization Granularity. Learned quantization methods differ in how granularly the quantization is applied, that is, the level of detail at which model parameters are quantized. To illustrate, Benoit et al. [22] implement layer-wise quantization — similar to QIL [24] and PACT [5] — whereas LQ-Nets [46] employ channel-wise quantization for weights, despite using layer-wise quantization for activations.

Combination with Other Techniques. A number of learned quantization methods work in unison with other model compression approaches. For example, Deep Compression [16] combines quantization with pruning, while Polino et al. [39] and Wei et al. [43] jointly leverage quantization and knowledge distillation.

To summarize, the landscape of learned quantization is as diverse as it gets. While the classifications outlined in this chapter are by no means exhaustive, they represent those we find most apparent and distinguishable. Together with Chapter 2, this chapter aims to provide a broader pitcture of learned quantization. Building on this picture, we will reflect on the contributions of this thesis and outline potential directions for future research in the conclusion.

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 - Most literature argues that having an additional parameter to tune is based - The grad calculation may be considered a bit too much - In real life, if the model is hard to track, it will be hard to finetune

- but this is, if not the first, then at least one of the first approaches that is based on the gradient of the parameter - there's also a theory that the ratio |dy| / |paramter| converges for all parameters, so this is very valid - the nested logic is cool

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] Tom B. Brown et al. "Language Models are Few-Shot Learners". In: *CoRR* abs/2005.14165 (2020). arXiv: 2005.14165. URL: https://arxiv.org/abs/2005.14165.

[4] 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.

[5] Jungwook Choi et al. "PACT: Parameterized Clipping Activation for Quantized Neural Networks". In: *CoRR* abs/1805.06085 (2018). arXiv: 1805.06085. URL: http://arxiv.org/abs/1805.06085.

[6] Yoojin Choi, Mostafa El-Khamy, and Jungwon Lee. "Learning Sparse Low-Precision Neural Networks With Learnable Regularization". In: *IEEE Access* 8 (2020), pp. 96963–96974. DOI: 10.1109/ACCESS.2020.2996936. URL: https://doi.org/10.1109/ACCESS.2020.2996936.

[7] 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.

[8] Ahmed T. Elthakeb, Prannoy Pilligundla, and Hadi Esmaeilzadeh. "SinReQ: Generalized Sinusoidal Regularization for Automatic Low-Bitwidth Deep Quantized Training". In: *CoRR* abs/1905.01416 (2019). arXiv: 1905.01416. URL: http://arxiv.org/abs/1905.01416.

[9] 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.

[10] Steven K. Esser et al. "Convolutional networks for fast, energy-efficient neuromorphic computing". In: *Proc. Natl. Acad. Sci. USA* 113.41 (2016), pp. 11441–11446. DOI: 10.1073/PNAS.1604850113. URL: https://doi.org/10.1073/pnas.1604850113.

[11] 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.

[12] 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.

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

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

[15] 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.

[16] 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.

[17] 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.

[18] 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.

[19] Itay Hubara et al. "Binarized Neural Networks". In: *Advances in Neural Information Processing Systems 29: Annual Conference on Neural Information Processing Systems 2016, December 5-10, 2016, Barcelona, Spain*. Ed. by Daniel D. Lee et al. 2016, pp. 4107–4115. URL: https://proceedings.neurips.cc/paper/2016/hash/d8330f857a17c53d217014ee776bfd50-Abstract.html.

[20] 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.

[21] David A. Huffman. "A Method for the Construction of Minimum-Redundancy Codes". In: *Proceedings of the IRE* 40.9 (1952), pp. 1098–1101. DOI: 10.1109/JRPROC.1952.273898.

[22] 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.

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

[24] 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.

[25] 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.

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

[27] 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.

[28] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. "ImageNet Classification with Deep Convolutional Neural Networks". In: *Advances in Neural Information Processing Systems 25: 26th Annual Conference on Neural Information Processing Systems 2012. Proceedings of a meeting held December 3-6, 2012, Lake Tahoe, Nevada, United States*. Ed. by Peter L. Bartlett et al. 2012, pp. 1106–1114. URL: https://proceedings.neurips.cc/paper/2012/hash/c399862d3b9d6b76c8436e924a68c45b-Abstract.html.

[29] Yann LeCun, Yoshua Bengio, and Geoffrey E. Hinton. "Deep learning". In: *Nat.* 521.7553 (2015), pp. 436–444. DOI: 10.1038/NATURE14539. URL: https://doi.org/10.1038/nature14539.

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

[31] 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.

[32] Zhouhan Lin et al. "Neural Networks with Few Multiplications". 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/1510.03009.

[33] Bin Liu et al. "Ternary Weight Networks". In: *IEEE International Conference on Acoustics, Speech and Signal Processing ICASSP 2023, Rhodes Island, Greece, June 4-10, 2023*. IEEE, 2023, pp. 1–5. DOI: 10.1109/ICASSP49357.2023.10094626. URL: https://doi.org/10.1109/ICASSP49357.2023.10094626.

[34] Markus Nagel et al. "Data-Free Quantization Through Weight Equalization and Bias Correction". In: *2019 IEEE/CVF International Conference on Computer Vision, ICCV 2019, Seoul, Korea (South), October 27 - November 2, 2019*. IEEE, 2019, pp. 1325–1334. DOI: 10.1109/ICCV.2019.00141. URL: https://doi.org/10.1109/ICCV.2019.00141.

[35] Maxim Naumov et al. "On Periodic Functions as Regularizers for Quantization of Neural Networks". In: *CoRR* abs/1811.09862 (2018). arXiv: 1811.09862. URL: http://arxiv.org/abs/1811.09862.

[36] 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.

[37] 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.

[38] 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.

[39] 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.

[40] 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.

[41] 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.

[42] Christian Szegedy et al. "Going deeper with convolutions". In: *IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2015, Boston, MA, USA, June 7-12, 2015*. IEEE Computer Society, 2015, pp. 1–9. DOI: 10.1109/CVPR.2015.7298594. URL: https://doi.org/10.1109/CVPR.2015.7298594.

[43] 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.

[44] 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.

[45] 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.

[46] 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.

[47] Hantian Zhang et al. "ZipML: Training Linear Models with End-to-End Low Precision, and a Little Bit of Deep Learning". In: *Proceedings of the 34th International Conference on Machine Learning, ICML 2017, Sydney, NSW, Australia, 6-11 August 2017*. Ed. by Doina Precup and Yee Whye Teh. Vol. 70. Proceedings of Machine Learning Research. PMLR, 2017, pp. 4035–4043. URL: http://proceedings.mlr.press/v70/zhang17e.html.

[48] 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.

[49] Xiaotian Zhu, Wengang Zhou, and Houqiang Li. "Adaptive Layerwise Quantization for Deep Neural Network Compression". In: *2018 IEEE International Conference on Multimedia and Expo, ICME 2018, San Diego, CA, USA, July 23-27, 2018*. IEEE Computer Society, 2018, pp. 1–6. DOI: 10.1109/ICME.2018.8486500. URL: https://doi.org/10.1109/ICME.2018.8486500.