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 [5] and ending with knowledge distillation, quantization presents itself as an especially promising area especically in the sense of learned quantization - the process of making ML models learn their optimal quantization parameters on their own. Hence, the current work employs two techniques that bypass the main issue of learned quantization, that is, the non-differentiability of rounding operations. While the first technique involves custom loss functions that directly take into account quantization goals, the second approach incorporates a custom gradient calculation that is easily integratable to various training scenarios. As a result of these two techniques, a memory usage reduction of up to ..x is obtained on MNIST, CIFAR10, 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 discretization effects, which act like a form of regularization, forcing the model to generalize better. No matter whether quantization results in better or a slightly worse performance, the conclusion 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 non-standard 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 encourage the model to quantize only where it's necessary and feasible.

This paragraph is gonna be about how previous reseacrh has tried to tackle the wbove two questions.

Finally, factually, how this thesis implemented the actions to the goal.

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*.

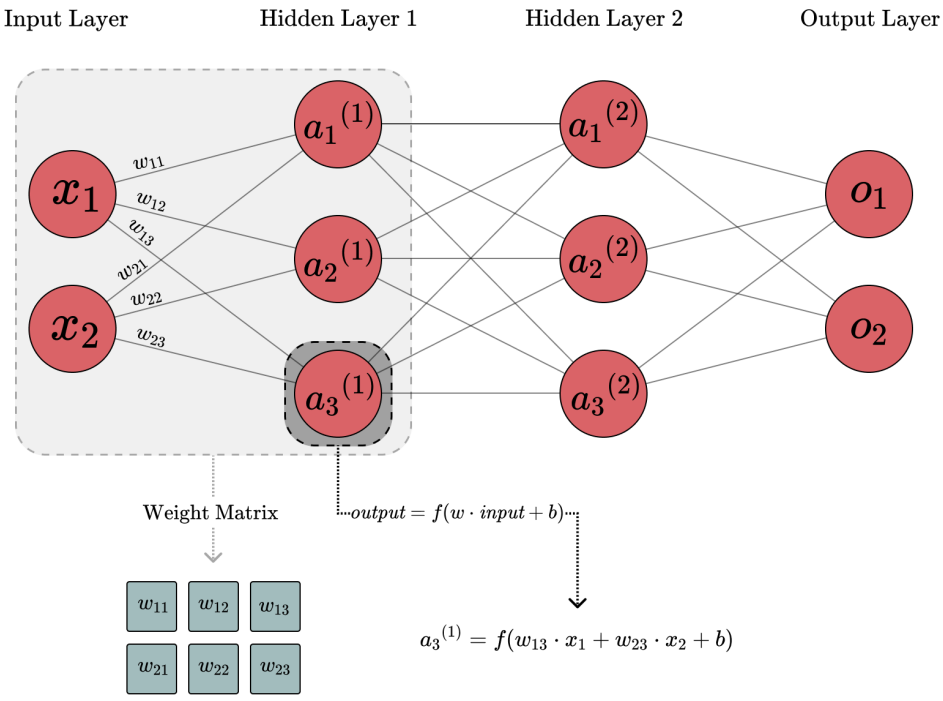

Figure 2.1: An example of a NN with two hidden dense layers, showing the connections between neurons in adjacent layers.

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 [2]. 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 [7].

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.

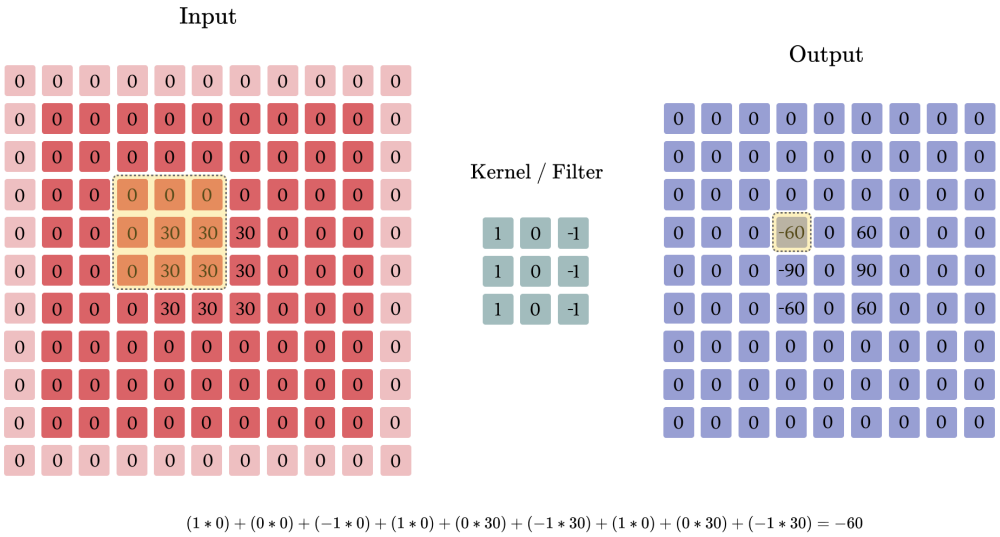

Figure 2.2: A 3×3 kernel (filter) sliding over a padded input matrix to compute the output feature map, demonstrating the interaction between the kernel weights and input values at a specific position.

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.

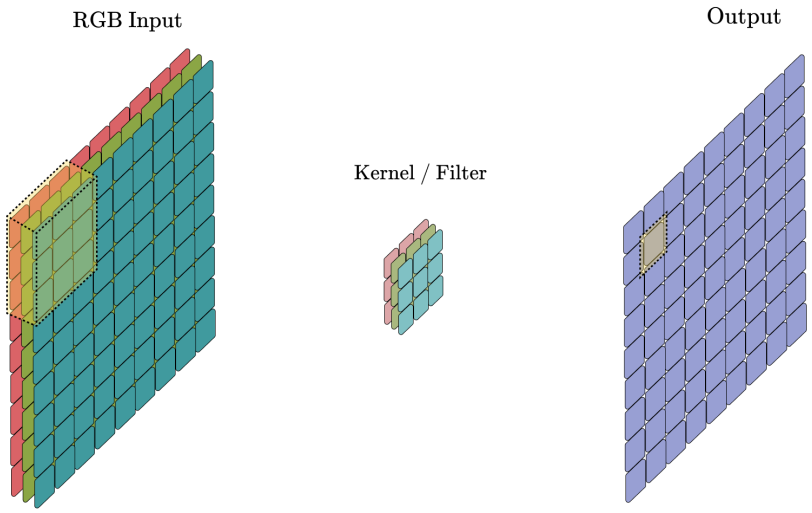

Figure 2.3: A 3×3x3 kernel (filter) sliding over an RGB input matrix to produce a single-channeled output feature map.

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 [6], 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.
INSTEAD OF THIS WE NEED TO EXPLAIN HOW A CLASSICAL WEIGHT UPDATE LOOKS LIKE IN TENSORFLO ALSO ADD how the theory looks like in tensorflow so that it's easier to follow
For example, consider the weight $W_{I1,H1_1}$ represented as the line between *Input 1* and the hidden layer node $H1_1$ in Figure **??**. The gradient of this weight with respect to the loss is calculated using the chain rule:

$$\frac{\partial L}{\partial W_{I1,H1_1}} = \frac{\partial L}{\partial \text{Output 1}} \cdot \frac{\partial \text{Output 1}}{\partial H1_1} \cdot \frac{\partial H1_1}{\partial W_{I1,H1_1}}$$

Where:

- $\frac{\partial L}{\partial \text{Output 1}}$ is the gradient of the loss with respect to *Output 1*.
- $\frac{\partial \text{Output 1}}{\partial H1_1}$ is the gradient of *Output 1* with respect to the output of $H1_1$.
- $\frac{\partial H1_1}{\partial W_{I1,H11}}$ is the value of *Input 1*, since $H1_1$ 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

With humans becoming increasingly dependent on deep learning models disguised as everyday tools on edge devices with limited capacitites — the need for these models to function in a resource- and time-efficient manner is more imperative than ever. Among the many ways to meet this need, quantization has become one of the most common techniques used to reduce the computational and memory costs of deep NNs.

While the term quantization has its roots in the first half of the 20th century [4], in the context of NNs, it has gained renewed importance — the over-parameterization of NNs introduces a certain type of flexibility that allows quantization to be implemented in many different forms [2]. Despite the abundance of these forms and approaches, quantization of NNs generally refers to the process of reducing the bit precision of their parameters, all while maintaining accuracy within an acceptable range.

2.2.2 Quantization Types

Although there is a multitude of ways to classify NN quantization methods, the most general classification is the division between *static, dynamic*, and *learned quantization*.

- In **static quantization**, quantization parameters are fixed before model inference, based on the data observed during training or calibration. This corresponds to Post-Training Quantization (PTQ), which directly quantizes the trained floating-point model [9], using various discretization approaches based on the range and distribution of the parameters being quantized, as well as the level of granularity at which the quantization values are applied.

- In **dynamic quantization**, the quantization parameters are calculated dynamically during inference. This typically applies to activations, as their range changes depending on the input data and, unlike that of the weights, cannot be precomputed with static quantization [11].

- In **learned quantization**, quantization parameters are learned as part of the model training process. This corresponds to Quantization Aware Training (QAT) [8], where quantization is integrated directly into training rather than applied afterwards. Since learned quantization is central to this work, a detailed review of QAT and its applications is provided in the Related Work chapter.

2.3 Learned Quantization

Now that the fundamentals of deep learning have been covered, this section introduces concepts commonly encountered in learned quantization, including the modified forward-pass technique and the challenges it creates for back-propagation.

2.3.1 Strategies and Applications

This

2.3.2 Trade-offs and Challenges

THIS PART WILL GO TO THE NEXT SECTION:

An example approach is to define a regularization term that directly penalizes large differences between full-precision values and their quantized counterparts [17]. This can encourage the model to learn parameters that are easily quantizable without significant performance loss. If the values that are being quantized are W, then this regularization term could look like:

$$\mathcal{L}_{\text{quant}} = \lambda \sum_i \|W_i - q(W_i)\|^2$$

Where:

- λ is a scalar that controls the importance of the quantization penalty.
- W_i represents the full-precision weight value before quantization at index i.
- $q(W_i)$ represents the quantized version of W_i.

The current work uses multiple custom regularization terms that trigger the quantization process during training. These will be discussed in detail in the Experimental Setup section.

3 Learned Quantization

This chapter presents the custom methods of learned quantization developed to address the challenges posed by non-differentiability, while enabling the model to make informed decisions on whether to proceed with quantization or revert to preserve performance. The first section will discuss the *custom quantization layers* written in Tensorflow that can easily be integrated to existing workflows and directly deal with gradient calculation. The next section provides details on the *custom loss functions* as an alternative approach to learned quantization.

This chapter elaborates on the problem that this thesis tries to solve and explains the individual methods used for solving the problem.

3.1 Custom Quantization Layers

In this section, we introduce custom layers built upon Tensorflow's `tf.keras.Layer` class, which serves as the base for all Keras layers. Each custom layer also leverages Tensorflow's `tf.custom_gradient` decorator to define its own gradient computation. For clarity, the upcoming subsections start by showing how to define the corresponding standard, non-quantized layer using `tf.keras.Layer` and `tf.customg_gradient`, then move on to the specific quantized implementations.

3.1.1 Data Quantization Layer

3.1.2 Quantized Dense Layer

3.1.3 Quantized Convolutional Layer

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 provides details about the experiments conducted within the context of this thesis.

You'll need these: there's a formula for precision requirement in [10]

4.1 Experimental Setup

All experiments are caried out on machine XYZ.

4.1.1 Software Setup

4.1.2 Hardware Setup

4.2 Hyperparamter Tuning

4.2.1 Learning Rate and Regularization Coefficients

4.2.2 Quantization Penalty Coefficient

4.3 Dataset-Specific Results

4.3.1 MNIST

4.3.2 CIFAR10

4.3.3 Imagenette

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 - [13]
CNN - [15]
CNN - [1] DNN - [3]

Quantization target parameters.
weights and activations - [12]
weights and activations - [7]
weights - [14]
weigths - [13]
binary weights and input activations - [15]
gradients - [16]

Granularity of quantization.

Handling of differentiability.

Quantization precision.
binary weights and input activations - [1]
binary weights and activations - [7]
binary weights and input activations - [15]
ternary weights - [13]
higher precision for more important parameters and lowe precision for less important ones - [10]

Integration with pruning & other techniques.
pruning and Huffman Codign - [5]
distillation - [14]
higher precision for more important parameters and lowe precision for less important ones or pruning - [10]

Modifications to loss functions.

Other interesting approaches. k-means for parameters [3]

6 Conclusions

This chapter summarizes the contriubtions of the thesis and provides an outlook into future work.

List of Acronyms

ML	Machine Learning

Bibliography

[1] 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.

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

[3] 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.

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

[5] 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.

[6] 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.

[7] 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.

[8] 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.

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

[10] 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.

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

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

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] Bohan Zhuang et al. "Towards Effective Low-bitwidth Convolutional Neural Networks". In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2018, pp. 7920–7928.

Appendix

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