Signal Processing Systems
Mekelweg 4,
2628 CD Delft
The Netherlands
https://sps.ewi.tudelft.nl/

SPS-2025-5174821

M.Sc. Thesis

Adaptive Compression of Deep Learning Models for Edge Inference via Bayesian Decomposition and Quantization Gates

Joris van de Weg

Abstract

With the growing developments in Artificial Intelligence (AI), deep learning models have become an attractive solution for industrial applications such as machine health monitoring and predictive maintenance. To enable real-time analysis and reduce reliance on cloud infrastructure, it is often more practical to process sensor data directly on edge devices. However, while deep learning models offer improved performance, their high memory and computational demands often exceed the limited resources of edge devices. Moreover, compression requires a lot of hyperparameter tuning, which is unique for each model, layer, and application. To address these limitations, this work utilizes dynamic Bayesian compression, which reduces model size and computational costs. By introducing learnable gate variables that control the quantization precision and the rank of decomposed factors, the model can adaptively determine the most efficient configuration for each layer during training. This results in a more flexible, end-to-end trainable compression scheme that maintains performance while significantly improving deployability on edge devices.

Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology

Adaptive Compression of Deep Learning Models for Edge Inference via Bayesian Decomposition and Quantization Gates

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

ELECTRICAL ENGINEERING

by

Joris van de Weg
born in Nieuwerkerk aan den IJssel, The Netherlands

This work was performed in:

Signal Processing Systems Group
Department of Microelectronics
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology

Delft University of Technology

Copyright © 2025 Signal Processing Systems Group
All rights reserved.

DELFT UNIVERSITY OF TECHNOLOGY
DEPARTMENT OF
MICROELECTRONICS

The undersigned hereby certify that they have read and recommend to the Faculty of Electrical Engineering, Mathematics and Computer Science for acceptance a thesis entitled **“Adaptive Compression of Deep Learning Models for Edge Inference via Bayesian Decomposition and Quantization Gates”** by **Joris van de Weg** in partial fulfillment of the requirements for the degree of **Master of Science**.

Dated: 12/09/2025

Chairman: ______________________________
prof.dr.ir. Justin Dauwels

Advisor: ______________________________
ir. Sinian Li

Committee Members: ______________________________
dr.ir. Jie Yang

____

Abstract

With the growing developments in Artificial Intelligence (AI), deep learning models have become an attractive solution for industrial applications such as machine health monitoring and predictive maintenance. To enable real-time analysis and reduce reliance on cloud infrastructure, it is often more practical to process sensor data directly on edge devices. However, while deep learning models offer improved performance, their high memory and computational demands often exceed the limited resources of edge devices. Moreover, compression requires a lot of hyperparameter tuning, which is unique for each model, layer, and application. To address these limitations, this work utilizes dynamic Bayesian compression, which reduces model size and computational costs. By introducing learnable gate variables that control the quantization precision and the rank of decomposed factors, the model can adaptively determine the most efficient configuration for each layer during training. This results in a more flexible, end-to-end trainable compression scheme that maintains performance while significantly improving deployability on edge devices.

Acknowledgments

I would like to express my gratitude to my supervisor, Justin Dauwels, for his insightful guidance and support throughout this project. His expertise and feedback have been invaluable in shaping the direction of my thesis.

I am also deeply thankful to Sinian Li for her constant willingness to help and for generously sharing her knowledge whenever I had questions. Her encouragement and constructive input made a real difference in the progress of this work.

I would further like to thank Jie Yang, for kindly taking the time to serve on my thesis committee and for their valuable feedback and contributions.

Beyond academia, I would like to express my appreciation to my family. I would like to thank my mom and dad for their continuous support, love, and care. I cannot forget to thank my brother and sister, for whom I always strive to be better.

I am also very grateful to my dear roommate, Daan, for taking care of me during the stressful periods of this thesis with food and support. Furthermore, I want to thank my friends from before university, as well as those I met during my bachelor's and master's, for their companionship, encouragement, and fun moments they brought throughout the years.

Finally, I would like to specially thank Burcu for making my days brighter, keeping me motivated, and for her unconditional support.

Joris van de Weg
Delft, The Netherlands
12/09/2025

Contents

Abstract **v**

Acknowledgments **vii**

1 Introduction **1**
- 1.1 Compression Techniques . . . 2
  - 1.1.1 Knowledge Distillation . . . 3
  - 1.1.2 Quantization . . . 3
  - 1.1.3 Tensor Decomposition . . . 3
  - 1.1.4 Pruning . . . 5
- 1.2 Research Questions . . . 6
- 1.3 Thesis Outline . . . 6
- 1.4 Project Information . . . 7

2 Background on Random Forest **9**

3 Methodology **13**
- 3.1 Decomposition . . . 13
  - 3.1.1 Matrix Multiplication . . . 13
  - 3.1.2 Convolutional Operations . . . 20
- 3.2 Quantization . . . 24
  - 3.2.1 Uniform Quantization . . . 24
  - 3.2.2 Controllable Quantization . . . 25
- 3.3 Gate Variables . . . 27
- 3.4 Loss Function . . . 30
  - 3.4.1 Derivation of the ELBO . . . 31
  - 3.4.2 Definition of ELBO . . . 32
  - 3.4.3 Derivation of the KL Divergence . . . 32
  - 3.4.4 Definition of the Loss Function . . . 35

4 Experimental Framework **37**
- 4.1 Datasets . . . 37
- 4.2 Experimental Setup . . . 39
- 4.3 Metrics . . . 42

5 Experimental Results **45**
- 5.1 Gate Utilization - PU Dataset . . . 45
- 5.2 Compression Sensitivity & Comparison - CWRU Dataset . . . 47
- 5.3 Compression Sensitivity - PU Dataset . . . 47
- 5.4 Comparison - PU Dataset . . . 52

6 Discussion **57**

7 Conclusion **61**

A Additional Dataset Results **67**
A.1 Compression Sensitivity - CWRU Dataset 67
A.2 Comparison - CWRU Dataset 69

List of Figures

1.1 Strategies for model size reduction. 2

2.1 Decision tree visualization. 9

3.1 Illustration of gated low-rank decomposition. 18
3.2 Effective truncation using gated decomposition. 19
3.3 Gated interaction-based decomposition. 19
3.4 1D convolution for a single filter 21
3.5 Structure of the `Conv1D` weight tensor. 21
3.6 Two-step convolution layer. 22
3.7 Illustration of gated two-step convolution. 23
3.8 Uniform quantization. 25
3.9 Residual error in quantization. 26
3.10 Sigmoid mapping from logit to gate probability. 28
3.11 Effect of temperature on the sigmoid function. 29

5.1 Gate activations for a low λ_d and low λ_q 45
5.2 Gate activations for a low λ_d and higher λ_q 46
5.3 Gate activations for a high λ_d and lower λ_q 46
5.4 Gate activations for a high λ_d and high λ_q 47
5.5 Base–SVD accuracy trade-off under varying λ_d 48
5.6 Base–Tucker accuracy trade-off under varying λ_d 49
5.7 Compact–SVD accuracy trade-off under varying λ_d 50
5.8 Compact–Tucker accuracy trade-off under varying λ_d 51
5.9 Accuracy–compression trade-offs across model types 52
5.10 Best λ_d: accuracy–model size 53
5.11 Best λ_d: accuracy–FLOPs 53
5.12 Best λ_d: accuracy–inference time 54

A.1 CWRU Base–SVD accuracy trade-off under varying λ_d 67
A.2 CWRU Base–Tucker accuracy trade-off under varying λ_d 68
A.3 CWRU best λ_d: accuracy–model size 69
A.4 CWRU best λ_d: accuracy–FLOPs 69
A.5 CWRU best λ_d: accuracy–inference time 70

List of Tables

4.1 Class structure comparison for CWRU and PU Datasets. 38
4.2 Model architecture with parameter counts 41
4.3 Inner dimensions and parameter counts 41

5.1 Accuracy and compression ratios 55
5.2 FLOPs and inference time 55

A.1 Accuracy and compression ratios (CWRU) 70
A.2 FLOPs and inference time (CWRU) 70

Introduction

1

Over the last decade, Artificial Intelligence (AI) models have shifted from being an academic pursuit to a very accessible everyday tool for everyone. Systems such as ChatGPT show that with sufficient parameters, data, and computing power, deep learning can effectively generalize across language, vision, and other domains [1]. The breakthroughs are possible due to the transformer architecture [2] and the continued scaling of the models. These very large 'foundation' models need to be supported by central cloud centers for both training and inference. While the 'bigger is better' movement of foundation models dominates the headlines, a complementary trend is quietly unfolding under the radar: Edge AI.

Edge AI is the practice of applying AI algorithms on local devices as close to the data source as possible. This practice makes real-time inference possible, where latency, bandwidth, and privacy risks are minimized [3]. Driven by the increase in usage of and research on AI, computing at the edge level is becoming increasingly important for the scaling and modernization of business operations. According to a recent survey, 83% of industry leaders believe that edge computing will be essential for staying competitive over the next decade [4]. It is not a surprise that the industrial sector will experience this shift as manufacturing alone accounts for about 15% of the global GDP (Gross Domestic Product) [5].

Many complex systems consisting of various machinery and components are used in the industrial sector. In many of these systems, component failure can lead to costly downtime and reduced operational efficiency [6]. Consequently, continuous and accurate condition monitoring of industrial machines is essential. Modern industrial machines are equipped with various sensors that generate time series data, which are sequences of measurements collected over time that reflect the operating conditions [7]. To achieve real-time analysis and reduce the dependency on cloud infrastructure, it is more practical and desirable to put the processing of the machine's sensor data directly on Edge Devices.

However, these Edge Devices embedded within or close to machines are resource-limited, meaning they have strict memory and computational capacity constraints [3, 8]. A common approach is the use of shallow machine learning models combined with expert features. These models are small in terms of their computations and memory; thus, they are well-suited for deployment on Edge Devices. However, because of their simplicity, they may lack the necessary capacity to effectively process data from complex environments. This limitation can be solved by deep learning models, as they have a greater capacity to learn complex and meaningful patterns [9, 10]. Yet, this improved performance comes at the cost of significantly higher memory and computational demands, which can easily exceed the limits of an Edge Device. Therefore, when using deep learning methods in resource-constrained environments, it is necessary to make the model as small as possible in terms of memory and computational needs, while maintaining performance.

A deep learning model is a collection of operations organised in layers, where each layer consists of weights. These weights are learned during training, and their number directly impacts the size of the model and how many computations are required to give an output [9]. Thus, when compressing the model, the goal is to reduce the memory needed to store the model and to simplify the computations it performs. This challenge can be approached by using different compression techniques.

1.1 Compression Techniques

The compression of models is a significant and interesting research area, not only for edge deployment but for the whole AI field. The first and most straightforward way to achieve compression is by designing a model as compact as necessary to fit the constraints by reducing the inner dimensions. However, this compact model may lack the capacity to match the best possible model in terms of accuracy, resulting in undesired results. This approach is not always the most suitable one, depending on the model requirements. The other alternative is to compress the better-performing model in terms of its memory and/or computational needs by changing its architecture or numerical representation [11, 12].

A legitimate first thought one could have is that compression would lead to lower performance. Counter-intuitively, compression can actually lead to better-performing models with higher accuracy and more robust performance [13]. This is because ‘large’ models can capture more diverse and complex patterns, which can be preserved even after compression. Compressed models can therefore outperform compact models that were designed to fit the constraints of an Edge Device. The main compression methods are tensor decomposition, quantization, knowledge distillation (KD), and pruning. Each of these technique reduce the model’s memory and/or computational needs in distinct ways [14]. A diagram showing these methods is displayed in Figure 1.1.

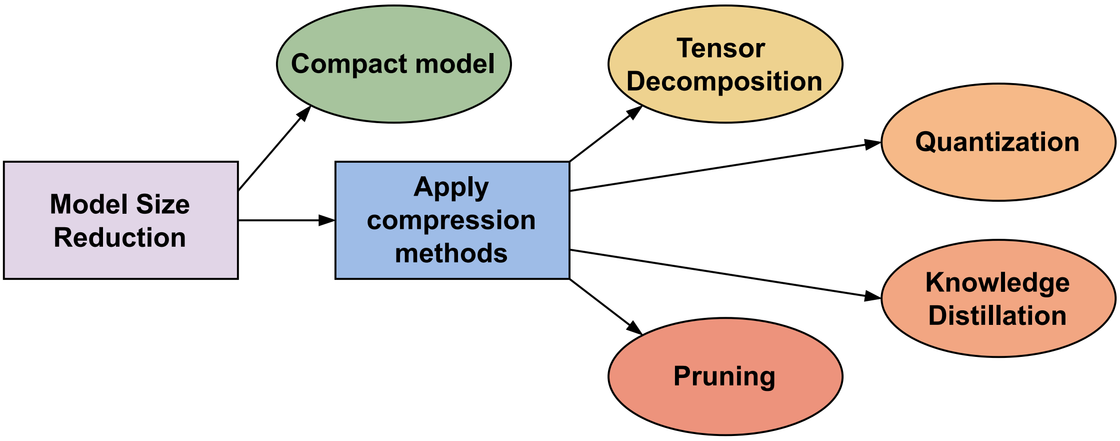


Figure 1.1: Strategies for model size reduction, including compact model design and post-training compression methods.

In the following sections, these four techniques are explained in detail.

1.1.1 Knowledge Distillation

KD is the method of mimicking the behaviour of a ‘best’ performing model, the teacher, with a smaller compact model, the student. This process can be done in multiple ways. First, there is response-based KD, where the soft output probabilities of the student model are matched to the teacher model [15, 16]. The other two families of KD guide the student even more by aiming to align their internal processes with those of the teacher model. This can be feature-based, where the student is trained to have the same output features at specific layer blocks from the teacher model. Another practice is to mimic the teacher indirectly by matching relations between features, instead of matching them directly, which is relation-based KD.

KD has been shown to achieve higher accuracies, and this type of training works better than training a model with hard labels alone [15, 16, 17]. However, this method requires a sufficiently trained teacher model, an additional distillation step, and modifications to the architecture to allow the transfer of knowledge. Moreover, KD is sensitive to a mismatch in capacity between student and teacher, as a very large teacher model can negatively affect the KD process [18]. Therefore, KD is an effective but delicate practice, making it inflexible. As a result, KD is not always a practical strategy to employ for Edge Devices, since their dynamic operating environments often require costly retraining.

1.1.2 Quantization

The second method of compression is quantization, which changes the numerical representation of the model parameters from 32-bit floating-point to lower-bit fixed-point formats. Changing the precision of parameters significantly reduces the model’s memory space, which is very useful for Edge AI applications [13]. Moreover, when the Edge Device has the ability to perform calculations with lower bit representations, it results in a speed up of the inference path and therefore decreased latency[19]. However, as one might expect, reducing precision can lead to degradation in accuracy, particularly in models with many concatenated layers, where quantization errors accumulate through consecutive layers [20]. To mitigate this, post-training quantization and quantization-aware training should be used to retain model performance while benefiting from compression [13].

Furthermore, applying mixed quantization can further optimize the process. One layer could be influential to the performance when changed to a lower numerical precision, in comparison to some layers which are more sensitive and thus need a higher-bit format [21]. Similarly to KD, but to a lower extent, fine-tuning or retraining a quantized model is not as simple, and finding the right quantization levels for every layer will become a computationally expensive problem [20].

1.1.3 Tensor Decomposition

Tensor decomposition is another powerful tool for model compression. It works by factorizing high-dimensional weight tensors (higher order matrices) into lower-dimensional components [22]. In simpler terms, it is the collection of methods that represents a big weight tensor as a multiplication of smaller tensors or matrices. For compression purposes, the goal is to make these components sufficiently small in terms of their number of parameters, resulting in a structured reduction in parameters. These methods not only lead to smaller models but can also decompose and optimize intensive matrix operations, which are very helpful for reducing latency in real-time applications. Tensor decomposition can be categorized into three families: Canonical Polyadic (CP) decomposition, Tucker decomposition, and Tensor-Train (TT) decomposition [23]. CP decomposition represents a tensor as a sum of rank-one components; it is simple, but it can achieve a significant compression rate [24]. Tucker decomposition breaks the tensor down into smaller cores by introducing a small tensor along each of its dimensions, making it a flexible method [25]. The last family, TT decomposition, decomposes a tensor into a series of cores and has been shown to be extremely effective for very large matrices and tensors [26]. Unlike the other two decompositions, the family of TTs can be of a complex structure, making them harder to implement on simple hardware. The main issue shared by all three of the techniques is related to how the inner ranks (dimensions) of the decompositions are chosen. There are multiple ways to determine the inner rank, but these methods are delicate processes that require a significant amount of time or rely on human input. However, when the 'right' inner dimension is found and chosen, tensor decomposition can show high compression ratios.

To reduce the loss of information from the tensor decomposition-based compression, Yin et al. [27] framed the decomposition problem as an optimization problem with constraints on the tensor rank using the Alternating Direction Method of Multipliers (ADMM) method. By first training the model to have a low-rank structure and then converting it into a TT, their experimental results showed compression ratios of 2.3× and 2.4×, while maintaining or even improving classification accuracy. It solves the issue of weight matrices not being of a low-rank nature. A different approach named hierarchical Tucker-2 (HT-2) decomposition was proposed by Gabor et al. [28]. Unlike standard tensor decomposition techniques, HT-2 further factorizes the convolutional kernel weight tensors into multiple low-rank components, offering better compression with minimal accuracy loss. So, by carefully designing factorizations, the performance can exceed that of other state-of-the-art methods, such as pruning and quantization. The method is shown to be especially effective for CNN models trained on CIFAR-10 and ImageNet.

Recent work has shown that combining the compression methods of quantization and tensor decomposition can often show even more promising results. It reduces both the number and the precision of the weights, which define the model's size. For example, by integrating 8-bit integer quantization and low-rank representations, Olutosin et al. [29] achieved a 57× reduction in model size compared with 4× and 17× compression factors that would have been achieved with either using only quantization or TT decomposition methods, respectively. Alnemari et al. [30] proposed a compression framework that integrates tensor decomposition methods, such as CP, Tucker, and TT, with binary neural networks. Their experiments showed that this joint approach achieved compression rates up to 168×. Although it can reach high levels of compression due to the binary layers, it struggles to maintain performance. The main takeaway is that these results demonstrate that quantization and decomposition act as complementary compression strategies, offering the best trade-off between compactness and accuracy.

1.1.4 Pruning

The last technique, pruning, removes weights, neurons, or even entire channels that contribute little to nothing to the final output. This removal of unimportant parts can be performed during or after training. Unstructured pruning is a simple technique that sets unimportant weights to zero, creating sparse matrices, which in turn makes storage and computation more efficient. Then there is structured pruning, on which there has been more focus in recent research, as it removes complete blocks from the layers [31]. It can be used for more aggressive compression of models. Determining which parts of the layers are unnecessary can be based on the magnitude of the weights, sensitivity analysis, or the application of learnable masks that learn what to remove [32]. Pruning based on magnitude is not always as effective; also, sensitivity analysis with pruning is a time-consuming matter. The learnable mask approaches, on the other hand, are very interesting because they reduce the need for user-defined rules and can be automatically applied to different datasets. Learnable mask methods refer to approaches that learn a mask controlling the pruning of parts of the layers through standard gradient-based optimization. The strength of pruning is thus dependent on how the loss function, the regularizer, is defined. Therefore, it does not require any trial-and-error iterations to find the best pruning settings, making it highly attractive for saving time. With human intervention removed and a completely data-driven model, it can even lead to better results, provided that the right regularizer is chosen. Pruning can thus also refer to the process by which the model decides the size of its layers, and it can even be extended to other compression methods mentioned earlier.

Bayesian Bits [33] is an excellent example of this principle, as it employs dynamic, learnable gating mechanisms through which pruning is performed to control the level of quantization. To be more precise, for every layer, the method rewrites each weight as a summation of different quantization levels. Each level is controlled by these gates, where, during training, the precision can be increased or decreased following a loss function. This mechanism enables the model to balance accuracy and compression based on the data without manually finding the optimal layer quantizations. Bayesian LoRA [34] advances this idea by extending the Bayesian Bits pipeline with learnable masking for decomposition in the LoRA framework. The original LoRA [12] was introduced to fine-tune large transformer models through a trainable low-rank decomposition. Through this, the number of trainable parameters, memory footprint, and computational costs are reduced drastically. As mentioned earlier, deciding on an inner rank is a delicate matter, and by allowing the model itself to prune the dimensions during decomposition, the model can find the optimal decomposition. Both papers [33] and [34] had promising results, maintaining accuracy while achieving a significant reduction in the number of basic operations per second and model size.

Although Edge Devices do not run large models such as those mentioned in Bayesian LoRA, the main insights and principles can still be of great value for edge computing purposes. Models designed for Edge Devices, to perform sensor-based tasks, mainly consist of convolutional layers and dense layers [35]. Recent work has shown that attention mechanism-based architectures, as introduced in transformers [2], are emerging as an essential component in the edge domain [36, 37]. Moreover, the approach will be extended to convolutional layers, a critical component for sensor-based tasks. To the best of our knowledge, there is currently no work that provides a comprehensive compression framework for these deep learning methods, unifying Bayesian compression with learnable gates to dynamically control by pruning the quantization and decomposition. By allowing models to control which components to remove and which precision to use, the optimal compression can be discovered. This contributes to making the compression of deep learning models for edge computing both practical and flexible.

1.2 Research Questions

The research carried out in this thesis is guided by key research questions. These questions influenced the design of the methodology and shaped the evaluation of the results. The main and sub-research questions that guide this thesis are defined as:

Main Research Question: How can deep learning models with complex layers be compressed through adaptive methods that reduce memory and computation while preserving inference performance on resource-constrained edge devices?

Sub Research Questions:

- How effectively can adaptive compression methods balance model complexity with resource constraints during inference on edge devices?
- How can hardware constraints, such as maximum model size, be incorporated into the compression objective?
- To what extent do compressed deep learning models outperform shallow machine learning baselines on edge-relevant metrics such as accuracy, computation, and memory?
- How do quantization and decomposition interact within a joint gate-based framework, and what advantages does joint optimization provide over isolated or sequential compression?

1.3 Thesis Outline

The structure of the thesis is as follows:

Chapter 2: Background on Random Forest
This chapter covers materials that are not directly related to the project but are necessary to understand the methodology and results. The shallow machine learning method used as a baseline for comparing the proposed compression method is described.

Chapter 3: Methodology
This chapter explains the details of how the proposed compression method works. It describes the inner workings of decomposition for the mentioned deep learning blocks, quantization, gate variables that control the compression, and the definition of the loss function through a Bayesian perspective.

Chapter 4: Experimental Framework
This chapter provides an elaboration of the dataset, the definition of the model architecture used to test the compression method, and the metrics used for assessment.

Chapter 5: Experimental Results
As the chapter name suggests, the achieved results are presented, providing insight into the performance and characteristics of the compression method.

Chapter 6: Discussion
This chapter provides a summary of all the results and discusses the strengths and shortcomings of the dynamic compression method. This chapter serves as a critical eye to the research.

Chapter 7: Conclusion
This chapter concludes the thesis by summarizing the main insights gained from the research and experiments. Additionally, this chapter discusses potential directions for future research in model generation for edge computing.

1.4 Project Information

This research was conducted as part of R-PODID, which is supported by the Chips Joint Undertaking and its members, including the top-up funding by National Authorities of Italy, Turkey, Portugal, The Netherlands, Czech Republic, Latvia, Greece, and Romania under grant agreement n° 101112338.

Background on Random Forest

2

As mentioned in Chapter 1, classical machine learning methods are often used for the application of Edge Devices. This chapter will define the Random Forest model, which will be used as a reference later in this work.

Random Forest is an ensemble learning technique, meaning it combines multiple models to make one prediction. For Random Forest, this model is the decision tree, which can be applied for classification or regression [38]. A single decision tree splits the input space $\mathcal{X} \subseteq R^d$ into regions that are orthogonal to a dimension axis. Each of these regions refers to a classification label, which makes the decision tree a piecewise constant function. Thus, the output of the model is determined by finding which region the input sample falls under. A simple visualization of this model and its regions is shown in Figure 2.1.

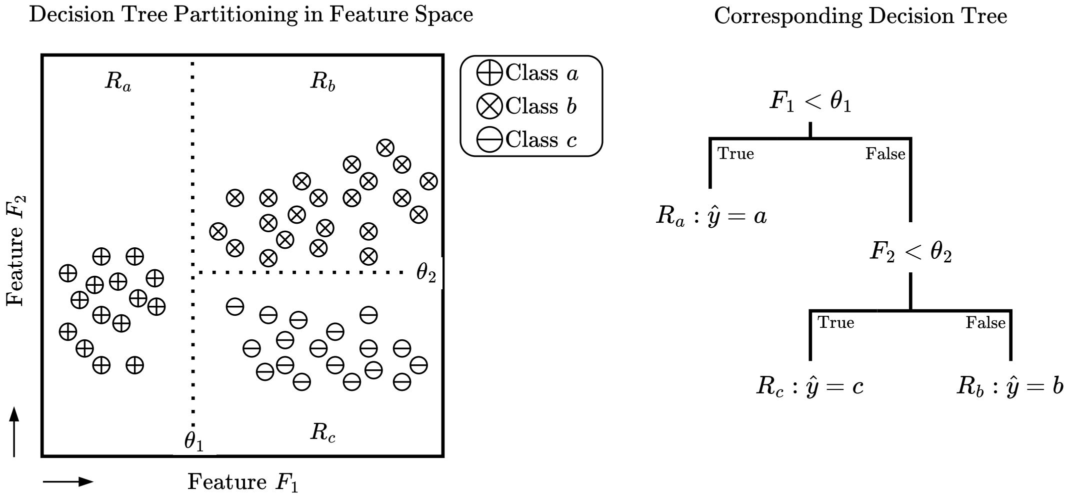

Decision Tree Partitioning in Feature Space
R_a
R_b
R_c
θ_1
θ_2
Feature F_1
Feature F_2
⊕Class a
⊗Class b
⊖Class c
Corresponding Decision Tree
$F_1 < \theta_1$
True
False
$R_a : \hat{y} = a$
$F_2 < \theta_2$
True
False
$R_c : \hat{y} = c$
$R_b : \hat{y} = b$

Figure 2.1: Visualization of the decision tree partitioning in feature space and its corresponding decision tree.

This decision tree is iteratively built by recursively performing binary splits on the dataset, partitioning it into regions until an end condition is met. A split divides the feature space by checking the condition whether a feature lies on one side or the other of a threshold θ. This split creates two subsets, resulting in a left (L) and right (R) child node as shown in Figure 2.1. Every child node contains a part of the total dataset $\mathcal{D}$, which can be defined as:

$$\mathcal{D}_L = \{\mathcal{D} \,|\, x_{i,j} \leq \theta\}, \qquad \mathcal{D}_R = \{\mathcal{D} \,|\, x_{i,j} > \theta\}. \tag{2.1}$$

The goal of splitting is to maximize the separation between the classes, thus creating nodes where there is mainly one class present. The purity of a node can measure the effectiveness of this separation. The contrary of purity, impurity, can be represented by, for example, the Gini index [38]. It quantifies the total variance over all the possible classes within a node and is calculated as:

$$G(n) = \sum_{k=1}^{K} \hat{p}_{n,k}(1 - \hat{p}_{n,k}). \tag{2.2}$$

Here, $\hat{p}_{n,k}$ represents the proportion of samples in node n that belong to the k-th class. So, the Gini index will become zero when a node contains samples solely from one class, making it a pure node. In decision trees, the focus lies on minimizing this impurity by splitting further and further. To generate a tree where its end nodes (leaves) are as pure as possible, every split should identify the dimension and its associated threshold that maximizes the reduction in impurity. This optimal split $s*$ and the impurity reduction $\Delta G(n, s)$ are defined as:

$$s^\star = (j, \theta) = \arg\max_{(j,\theta)} \Delta G_n(j, \theta), \qquad \Delta G(n, s) = G(n) - p_L\, G(n_L) - p_R\, G(n_R). \tag{2.3}$$

Here, j and θ represent the dimension and its associated threshold for the best possible split. The child proportions p_L and p_R are used to calculate the reduction in impurity and are calculated as:

$$p_L = \frac{|\mathcal{D}|}{|\mathcal{D}_L|}, \qquad p_R = \frac{|\mathcal{D}|}{|\mathcal{D}_R|}. \tag{2.4}$$

Here $|\mathcal{D}|$ is the total number of samples in the parent node, and $|\mathcal{D}_{L/R}|$ is the total number of samples in the child node. The splitting will stop when the impurity G becomes zero, the minimum leaf size is reached, or the maximum tree size is met.

Decision trees have advantages in being interpretable and computationally efficient. It requires little data preparation as it does not require feature scaling [39]. They can outperform linear models when a non-linear relationship exists in the data. Decision trees are thus simple to understand and to use, but they have some limitations. Because the splits are made perpendicular to an axis (input dimension), decision trees are sensitive to how the data is rotated, making them unstable. At this point, ensemble methods can be introduced, which can reduce variance by averaging the classification over many models.

As mentioned previously, an ensemble method combines multiple models to generate an output. Making a Random Forest model starts with bagging. It is the process of generating different datasets (samples) by bootstrapping, which is the random resampling of the main dataset. Each of these so-called bootstrap samples is the same size as the original dataset, but can contain duplicate or missing observations. Every bootstrap sample is then used to train a decision tree, and then each decision tree votes together to form a final classification. To further improve the robustness and decorrelate the trees, an additional component of randomness is introduced. Instead of considering all feature dimensions to find the optimal split $s*$, a subset of all the features is randomly chosen. This means that trees will explore different splitting thresholds, making them more diverse from one another. Therefore, Random Forest combines bagging with the random selection of features, resulting in a more robust model. This process can be explained as

$$\hat{f}_{\mathrm{RF}}(\mathbf{x}) = \mathrm{mode}\,\{f_b(\mathbf{x}; D_b^*)\}_{b=1}^{B}\,, \tag{2.5}$$

where each f_b is a decision tree trained on a bootstrap sample $\mathcal{D}_b$ of the original dataset, and at each split f_b uses a random subset of features. The final classification is determined by identifying the most frequent prediction among all trees, which is referred to as the mode.

As this chapter has shown, the inner workings of the Random Forest are quite straightforward. It depends only on integer additions and comparisons, and its computational costs scale with the product of the tree depth and the number of trees $\mathcal{O}(B \cdot \text{depth})$. As only one path is followed for every tree, the inference path is highly efficient. Secondly, despite its relatively simple architecture, it can be state-of-the-art in accuracy for a variety of applications. A study from Fernández Delgado [40] compared the performance of 179 classifiers from 17 families over 121 datasets. The paper showed that the highest accuracy was achieved using a Random Forest-based classifier. Moreover, the top five best-performing classifiers were from the Random Forest family. Their only shortcoming is the model size. When many tree models (high B), which are also large themselves (high depth), are used, the overall RF model can grow large.

In conclusion, Random Forest is an efficient and effective shallow machine learning model, possibly the best for edge-based computing. It is robust to noise and overfitting, and it can outperform linear models when non-linear relationships are found in the data. Although the model can grow larger, its inference process will still be the most efficient out of all other types of models. Therefore, Random Forest is used as a model to beat in terms of accuracy and model size.

3

Methodology

This chapter covers how different layers of a deep learning model are dynamically compressed with decomposition and quantization methods. After the compression methods are established, the gate variables and the loss function, which control the compression, are explained.

3.1 Decomposition

Decomposition is the process of representing a matrix or tensor as a product of smaller and therefore more manageable factors. As mentioned in Chapter 1, it is a commonly used method for compressing neural networks, as it can effectively reduce the number of parameters and computational costs. There are different types of operations that define layers, and therefore, the possible decompositions differ. The first and most fundamental type of operation found in models is matrix multiplication, which performs a linear transformation. It is a standard and essential part of many layers, like dense layers and self-attention layers. The second one is the convolutional operations. The following subsections will explain how these two operations can be decomposed for compression purposes.

3.1.1 Matrix Multiplication

The matrix multiplication is a linear transformation that maps the input space to a different vector space or representation. Dense layers are one of the most commonly seen components in deep learning architectures, which use this linear transformation at their core. Given an input matrix $\mathbf{X} \in \mathbb{R}^{l \times m}$, the same linear mapping is performed for every row, and the dense layer is mathematically represented as:

$$\mathbf{Z} = \mathbf{X}\mathbf{W} + \mathbf{1}_l \mathbf{b}^\top, \qquad \mathbf{Y} = \phi(\mathbf{Z}). \tag{3.1}$$

Here, $\mathbf{W} \in \mathbb{R}^{m \times o}$ is the weight matrix and $\mathbf{b} \in \mathbb{R}^{o}$ is the bias vector, which are learnable parameters of the layer. To apply the bias transformation to every row, the bias vector is multiplied by $\mathbf{1}_\ell \in \mathbb{R}^\ell$, which is a column vector consisting of ones. To create a dense layer capable of learning complex relationships that are beyond the capacity of linear mappings, an element-wise non-linear function $\phi(\cdot)$ is used. The layer provides the model with the ability to learn powerful transformations between input and output features. Every input dimension contributes to every output dimension so that the layer can capture global interactions across the feature space.

Another type of layer, which also uses this simple matrix multiplication, is the multi-head attention (MHA) mechanism found in the Transformer architectures [2]. The attention layer projects the input into three separate spaces, which are *queries* ($\mathbf{Q}$), *keys* ($\mathbf{K}$), and *values* ($\mathbf{V}$), each obtained via a dense linear transformation:

$$\mathbf{Q} = \mathbf{X}\mathbf{W}_Q, \qquad \mathbf{K} = \mathbf{X}\mathbf{W}_K, \qquad \mathbf{V} = \mathbf{X}\mathbf{W}_V. \tag{3.2}$$

Here, $\mathbf{W}_Q, \mathbf{W}_K \in \mathbb{R}^{m\times d_k}$ and $\mathbf{W}_V \in \mathbb{R}^{m\times d_v}$ are the learnable weight matrices that project the input. The output of the attention mechanism is then computed as a weighted combination of these value vectors, where the similarity between queries and keys determines the weights. The core computation of self-attention is represented as:

$$\text{Attention}(\mathbf{X}) = \text{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^\top}{\sqrt{d_k}}\right)\cdot \mathbf{V}, \qquad \mathbf{Y} = \text{Attention}(\mathbf{X})\,\mathbf{W}_O. \tag{3.3}$$

The dot-product attention scores are scaled by $\sqrt{d_k}$ to prevent large values that could push the softmax towards extreme outputs. Finally, $\mathbf{W}_O \in \mathbb{R}^{d_v\times m}$ is the output projection, which maps the attention output back to the model's original dimension. In contrast to dense layers, self-attention does not treat all input features equally; instead, it dynamically focuses on the most relevant parts of the input for each position. By adaptively weighting interactions, attention layers excel at modeling long-range dependencies that would be difficult to capture with fixed transformations alone, such as those in dense layers.

Both layers rely primarily on the same underlying linear transformation between an input matrix $\mathbf{X}$ and a weight matrix $\mathbf{W}$ as seen in Equations (3.1) and (3.2). Although the structural role of both layers differs, their storage and computational needs mainly depend on the dimensionality of their weight matrices. For the dense layer, the parameter count N_{dense} is dependent on the learnable weight matrix $\mathbf{W}$ and bias $\mathbf{b}$:

$$N_{\text{dense}} = \underbrace{m \cdot o}_{\mathbf{W}} + \underbrace{\quad o \quad}_{\mathbf{b}}. \tag{3.4}$$

The dominant computational cost arises from multiplying the input matrix $\mathbf{X}$ with the weight matrix $\mathbf{W}$, which requires $\mathcal{O}(\ell m o)$ operations. The cost scales linearly with the input length ℓ. For the attention mechanism, the situation is similar but involves multiple projection matrices, which results in a parameter count of:

$$N_{\text{MHA}} = \underbrace{m \cdot d_k}_{\mathbf{W}_Q} + \underbrace{m \cdot d_k}_{\mathbf{W}_K} + \underbrace{m \cdot d_v}_{\mathbf{W}_V} + \underbrace{d_v \cdot m}_{\mathbf{W}_O}. \tag{3.5}$$

The computational cost of the MHA layer consists of the input projections to obtain $\mathbf{Q}, \mathbf{K}$ and $\mathbf{V}$, the dot-product attention, and the final linear output projection:

$$\mathcal{O}\Big(\underbrace{\ell(2md_k + md_v)}_{\text{input projections}} + \underbrace{\ell^2(d_k + d_v)}_{\text{dot-product attention}} + \underbrace{\ell\, d_v\, m}_{\text{output projection}}\Big) \tag{3.6}$$

Equation (3.6) shows that the computational load of the attention layer depends primarily on two factors. First, the input sequence length ℓ scales the number of operations linearly for the input/output projections and quadratically for the dot-product operation. This term cannot be reduced through compression, as it is determined by the previous layer or the input data itself. The second factor concerns the terms related to the dimensions of the weight matrices, which can be modified within the layer. For simplicity, it will be assumed that ℓ is much smaller than the weight dimensions, making matrix projections the dominant contributor to the computational cost. Therefore, the matrix multiplication $\mathbf{X} \cdot \mathbf{W}$ becomes the main target for compression to effectively reduce computational cost. This compression will be achieved by applying decomposition on $\mathbf{W}$, separating it into a product of smaller matrices.

There are various methods to decompose the linear transformation step, but not all of them are suitable for deployment on Edge Devices. For example, the TT decomposition, introduced in section 1.1.3, can aggressively reduce the number of parameters in high-dimensional weight tensors by representing them as a product of low-rank 3D tensors, known as TT-cores [20, 13]. For a d-dimensional tensor $\mathcal{A} \in \mathbb{R}^{n_1 \times n_2 \times \cdots \times n_d}$, TT decomposition is formally written as:

$$\mathcal{A}(i_1, \ldots, i_d) = \mathbf{G}_1(i_1)\mathbf{G}_2(i_2) \cdots \mathbf{G}_d(i_d), \tag{3.7}$$

where each $\mathbf{G}_k(i_k)$ is a matrix of size $r_{k-1} \times r_k$, and the sequence $(r_0, r_1, \ldots, r_d)$ defines the TT-ranks with $r_0 = r_d = 1$. To allow the TT to exploit multi-dimensional correlations, it is beneficial to reshape a 2D weight matrix $\mathbf{W} \in \mathbb{R}^{m \times o}$ into a higher-order tensor, such that it would lead to a significant parameter decrease. However, this reshaping introduces additional computational costs during inference, as the TT needs to be contracted back to enable multiplication with the input. This overhead will most likely outweigh the benefits in latency-constrained environments, such as Edge Devices. For this reason, $\mathbf{W}$ will not be reshaped in this work, and a TT will be, in this case, reduced to only two cores, which is equivalent to a standard matrix factorization. Consequently, the focus will shift to these simpler decompositions that are specifically designed for matrices.

These ‘simple’ decompositions for matrices can be distinguished into two broad categories based on their structural characteristics: low-rank factorizations and interaction-based factorizations that allow for component interactions.

Low-Rank Factorization

Low-rank factorization approximates the weight matrix as the product of two smaller matrices as:

$$\mathbf{W} \approx \mathbf{A}\mathbf{B}, \tag{3.8}$$

where $\mathbf{A} \in \mathbb{R}^{m \times r}$ and $\mathbf{B} \in \mathbb{R}^{r \times o}$, and r denotes the inner rank. For compression and edge deployment, r should be much smaller than the dimensions m and o of $\mathbf{W}$, such that the structure reduces both the number of parameters and the number of operations required during inference.

This simple form is efficient and effective, but a practical modification can be suggested to make it more suitable for training. By introducing an intermediate diagonal matrix $\mathbf{E} \in \mathbb{R}^{r \times r}$, the model can directly adjust the scaling of each latent component of $\mathbf{A}$ and $\mathbf{B}$ [34]. The decomposition is then defined as:

$$\mathbf{W} \approx \mathbf{A}\mathbf{E}\mathbf{B}. \tag{3.9}$$

Here, $\mathbf{E}$ is the diagonal matrix containing the scaling elements for every component in $\mathbf{A}$ and $\mathbf{B}$, and it is defined as:

$$\mathbf{E} = \mathrm{diag}(\mathbf{e}), \quad \mathbf{e} = [s_1, s_2, \ldots, s_r]^T. \tag{3.10}$$

The definition, shown in Equation 3.9, can be interpreted as the summation of rank-1 matrices, which closely resembles the singular value decomposition (SVD) and CP decomposition. SVD is a matrix factorization that expresses a matrix as the product of two orthogonal matrices and a diagonal matrix of singular values, whereas CP decomposition was previously explained in section 1.1.3. The formation of $\mathbf{E}$ allows the model to independently control the importance of each rank-1 component without changing the underlying directional vectors in $\mathbf{A}$ and $\mathbf{B}$. This decoupling between magnitude and direction has been shown to lead to better expressiveness, faster convergence, and higher accuracy [41]. However, in cases where the model's task is relatively simple, this difference in performance becomes less prominent.

Interaction-Based Factorization

When $\mathbf{E}$ in Equation 3.9 is changed from a diagonal matrix to a full core matrix, the latent components in $\mathbf{A}$ and $\mathbf{B}$ are no longer independent but can interact with one another. This will be referred to as interaction-based factorization, an extension of the low-rank factorization that retains the same overall structure, but with $\mathbf{E} \in \mathbb{R}^{r \times r}$ being a full core matrix. This formulation resembles the matrix form of the Tucker decomposition and offers a greater representational capacity and compression by modelling dependencies between different components. However, this increase in complexity makes the factorization more challenging to optimize effectively.

Low-rank and interaction-based factorizations do not guarantee a reduction in the number of parameters of W. To achieve a significant decrease, as needed for model compression, truncation needs to be applied.

Truncation

The core idea behind compression by decomposition is to select the inner rank r such that the decomposition of the weight matrix adopts a low-rank structure, thereby reducing the number of parameters. However, an exact decomposition does not reduce this rank itself. To achieve compression, it is typically combined with truncation, which discards components of lower importance, therefore reducing the rank. A common approach is to decompose a pre-trained weight matrix $\mathbf{W}$ and then truncate it, reducing r while attempting to preserve as much of the original representation as possible. For example, when decomposing the matrix with SVD, truncation is achieved by removing the smallest singular values and their associated eigenvectors. However, in the application of edge inference, this two-step static approach is less practical.

First of all, applying truncation on the decomposed weight matrix does not guarantee that it can be represented as a low-rank structure. Weight matrices do not have to be low-rank, meaning that their singular values decay slowly; thus, most components contribute significantly to the representation. As a result, the aggressive removal of components is not possible without a significant reduction in the representational capacity. In these scenarios, the decomposed matrix cannot be compressed meaningfully while still maintaining accuracy. An alternative approach is to impose a low-rank constraint during training on the weight matrix generation, which encourages a clear distinction between essential and negligible components, making the truncation process easier. However, this approach introduces a high dependence on the chosen strength of the low-rank constraint, which can be different for every layer, complicating the training process. While any performance loss resulting from the truncation can be mitigated through fine-tuning, this does not guarantee the recovery of lost performance when the truncation is too aggressive. Therefore, the model should learn both the decomposition and the effective truncation simultaneously during training to effectively compress the model. In this approach, the weight matrices **A**, **E**, and **B** as in Equation 3.9 will become trainable parameters, enabling the model to optimize the rank and scaling of every component in an end-to-end manner.

Before explaining the mechanism that enables the model to control the truncation, the overall architecture must be defined. The weight matrix will be replaced by a factorized series, represented by the effective weight matrix defined as:

$$\mathbf{W}_{\text{eff}} = \mathbf{A} \cdot \mathbf{E} \cdot \mathbf{B}. \tag{3.11}$$

The compression achieved by this approach depends directly on the chosen inner rank r.

For the **low-rank formulation**, where **E** is a diagonal matrix, the number of learnable parameters is:

$$N_{\text{p}} = \underbrace{m \cdot r}_{\mathbf{A}} + \underbrace{r}_{\mathbf{E}} + \underbrace{r \cdot o}_{\mathbf{B}} = r \cdot (m + o + 1). \tag{3.12}$$

To make sure the decomposition does not introduce more parameters than the original weight matrix, an upper bound for the inner rank must be enforced. Using the expression for N_{dense} from Equation 3.4 and N_p from Equation 3.12 , the upper bound is be defined as:

$$r_{\max} = \left\lfloor \frac{m \cdot o}{m + o + 1} \right\rfloor, \tag{3.13}$$

where $\lfloor \cdot \rfloor$ is the floor operator that ensures the bound is an integer.

For the **interaction-based formulation**, where **E** is a full $r \times r$ matrix, the number of parameters becomes:

$$N_{\text{p}} = \underbrace{m \cdot r}_{\mathbf{A}} + \underbrace{r \cdot r}_{\mathbf{E}} + \underbrace{r \cdot o}_{\mathbf{B}} = r \cdot (m + o + r). \tag{3.14}$$

For this scenario, the upper bound on r is derived by solving $r \cdot (m + o + r) \leq m \cdot o$:

$$r_{\max} = \left\lfloor \frac{-(m + o) + \sqrt{(m + o)^2 + 4 \cdot m \cdot o}}{2} \right\rfloor. \tag{3.15}$$

The formulations of N_{p} and $r_{\max}$ for both decomposition types ensure that the number of parameters does not exceed that of the weight matrix. However, while these upper bounds constrain the maximum possible rank, the actual effective rank should ideally be learned dynamically during training, allowing the model to balance expressiveness and compression.

Controllable Decomposition

The previous subsections have explained how a weight matrix $\mathbf{W}$ can be compressed by replacing it with a low-rank decomposition. As discussed, the level of compression is dependent on the inner rank r, which determines the dimensionality of the total structure. To give the model control over this rank, so-called gate variables are introduced, denoted with g. A gate variable is a binary element that can be either active or inactive, effectively switching components on or off. For this section, it is sufficient to understand that the model is capable of changing the state of these gates. The complete explanation and role of gate variables will be discussed in detail later in Section 3.3.

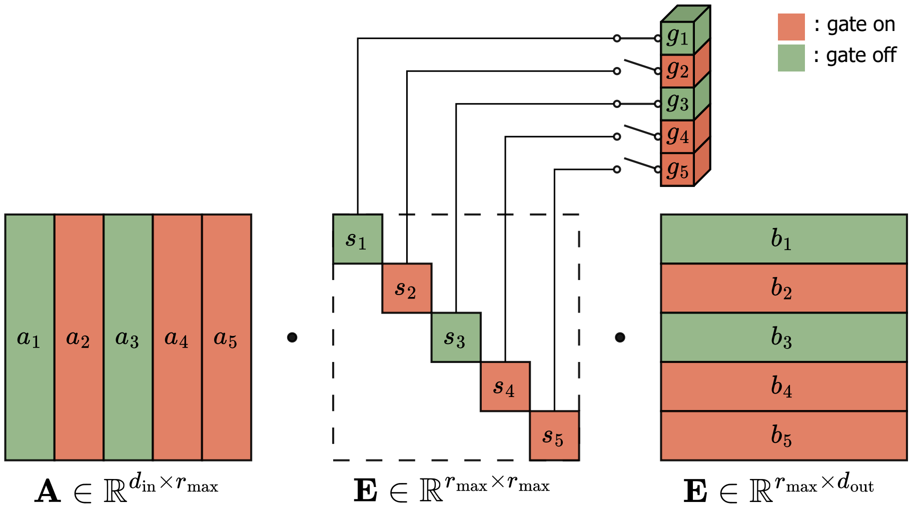
To dynamically control the inner rank, each latent component is coupled to a gate variable, except for one that remains active, which ensures the layer always produces an output. The gates will thus be enforcing selective pruning on the decomposition structure. The vector of gate variables $\mathbf{g}$, containing r_{max} number of gates, is defined as:

$$\mathbf{g}^T = \begin{bmatrix} 1 & g_1 & g_2 & g_3 & \dots & g_{r_{\text{max}}} \end{bmatrix}, \tag{3.16}$$

where $g_i \in \{0, 1\}$. As explained, the leading element is fixed to 1 to guarantee at least one active path. With this gating mechanism, the formulation of the effective weight matrix becomes:

$$\mathbf{W}_{\text{eff}} = \mathbf{A} \cdot \text{diag}(\mathbf{g} \odot \mathbf{e}) \cdot \mathbf{B}, \tag{3.17}$$

where $\odot$ denotes the element-wise multiplication, which mimics the behaviour of truncation of the matrices. In other words, if a gate is inactive ($g_i = 0$), the corresponding singular component e_i is suppressed, and the associated column in $\mathbf{A}$ and row in $\mathbf{B}$ no longer contribute to the reconstruction. Vice versa, active gates ($g_i = 1$) ensure that their corresponding



g_1
g_2
g_3
g_4
g_5
🟥 : gate on
🟩 : gate off
a_1 a_2 a_3 a_4 a_5
s_1
s_2
s_3
s_4
s_5
b_1
b_2
b_3
b_4
b_5
$\mathbf{A} \in \mathbb{R}^{d_{\text{in}} \times r_{\text{max}}}$
$\mathbf{E} \in \mathbb{R}^{r_{\text{max}} \times r_{\text{max}}}$
$\mathbf{E} \in \mathbb{R}^{r_{\text{max}} \times d_{\text{out}}}$

Figure 3.1: Illustration of the gating mechanism applied to a low-rank decomposition. Each column of $\mathbf{A}$ and row of $\mathbf{B}$ corresponds to a latent component, modulated by the diagonal entries of $\mathbf{E}$. The gates (red: inactive, green: active) control whether each component contributes to the effective weight matrix. One gate is always on to ensure stable training.

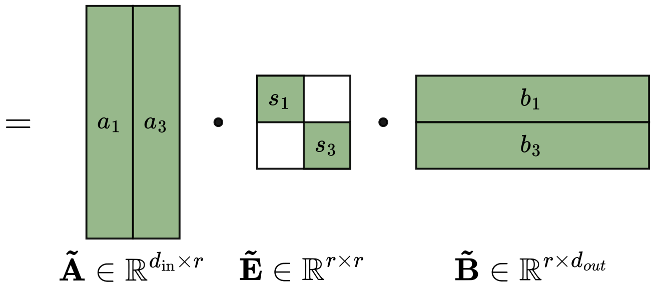

=
a_1
a_3
s_1
s_3
b_1
b_3
$\tilde{\mathbf{A}} \in \mathbb{R}^{d_{\text{in}} \times r}$
$\tilde{\mathbf{E}} \in \mathbb{R}^{r \times r}$
$\tilde{\mathbf{B}} \in \mathbb{R}^{r \times d_{out}}$

Figure 3.2: Resulting truncated decomposition after gates deactivate certain components. Inactive components (red) are pruned away, leaving a reduced decomposition ($\tilde{\mathbf{A}}, \tilde{\mathbf{E}}, \tilde{\mathbf{B}}$) that defines the effective weight matrix. This dynamic pruning provides the model with adaptive control over the rank.

components contribute to the effective weight matrix. In this way, the model gains the ability not only to learn the decomposition parameters but also to determine which latent components are necessary for the task. The process of selectively activating or deactivating latent components is illustrated in Figure 3.1 and Figure 3.2.

For the interaction-based formulation, $\mathbf{E}$ is a full core matrix of size $r_{\text{max}} \times r_{\text{max}}$. Therefore, it requires an extended gating mechanism consisting of two independent gate vectors $\mathbf{g}^1$ and $\mathbf{g}^2$ that control the rows and columns of $\mathbf{E}$, respectively. Therefore, the corresponding effective weight matrix is:

$$\mathbf{W}_{\text{eff}} = \mathbf{A} \cdot \left(\text{diag}(\mathbf{g}^1) \cdot \mathbf{E} \cdot \text{diag}(\mathbf{g}^2)\right) \cdot \mathbf{B}. \tag{3.18}$$

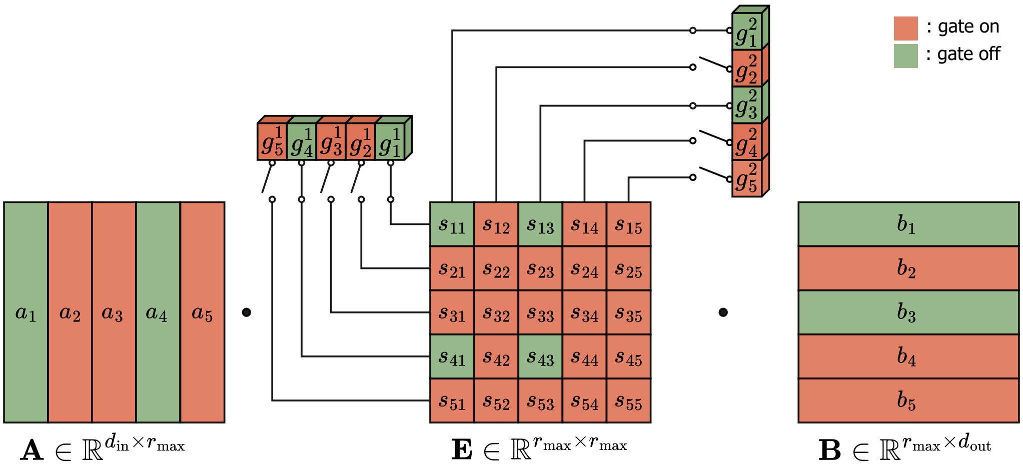

: gate on
: gate off
g^1_5 g^1_4 g^1_3 g^1_2 g^1_1
g^2_1 g^2_2 g^2_3 g^2_4 g^2_5
a_1 a_2 a_3 a_4 a_5
s_{11} s_{12} s_{13} s_{14} s_{15}
s_{21} s_{22} s_{23} s_{24} s_{25}
s_{31} s_{32} s_{33} s_{34} s_{35}
s_{41} s_{42} s_{43} s_{44} s_{45}
s_{51} s_{52} s_{53} s_{54} s_{55}
b_1 b_2 b_3 b_4 b_5
$\mathbf{A} \in \mathbb{R}^{d_{\text{in}} \times r_{\text{max}}}$
$\mathbf{E} \in \mathbb{R}^{r_{\text{max}} \times r_{\text{max}}}$
$\mathbf{B} \in \mathbb{R}^{r_{\text{max}} \times d_{\text{out}}}$

Figure 3.3: Gated interaction-based decomposition. The latent components in $\mathbf{A}$ and $\mathbf{B}$ are connected through the full core matrix $\mathbf{E}$. Two gate vectors control the decomposition: $\mathbf{g}^1$ for the rows and $\mathbf{g}^2$ for the columns. Inactive gates (red) block corresponding rows/columns in the core, thereby pruning away unused interaction paths, while active gates (green) allow information flow.

With this formulation, the model can deactivate entire rows or columns of the core matrix, ensuring that certain latent interaction paths do not contribute to the final weight matrix. This gives the model the ability to not only learn which components are important, but also which cross-component interactions are necessary for representing the data. A graphical illustration of this mechanism is given in Figure 3.3.

3.1.2 Convolutional Operations

Another frequently used and powerful operation block in deep learning architectures is the convolutional layer. Unlike dense layers, convolutional layers are very effective for extracting local patterns in the data due to their ability to focus on small parts of the data by using the same filter over the input. This layer makes it possible for the model to detect recurring structures, and this makes convolutional operations essential for sensor applications. In this thesis, the focus is on one-dimensional convolution layers (`Conv1D`), which are especially suitable for processing sequential input data such as time series. `Conv1D` operates by sliding a learnable filter, also called a kernel, over the input sequence and performing a discrete convolution at each time step.

The layer receives input $\mathbf{X} \in \mathbb{R}^{T \times C_{\text{in}}}$, where T is the number of time steps and C_{in} is the number of input channels (or features). The layer applies C_{out} learnable filters $\mathbf{W}^{(j)} \in \mathbb{R}^{k \times C_{\text{in}}}$, each with a temporal window size of k, where $j \in \{1, \ldots, C_{\text{out}}\}$ indexes the filters. A single filter j produces an output vector $\mathbf{y}^{(j)} \in \mathbb{R}^{T}$ by convolving over the input for each time step. The output at time step $t \in \{1, \ldots, T\}$ is computed as:

$$\mathbf{y}_t^{(j)} = \phi\left(\sum_{\ell=0}^{k-1} \mathbf{W}_\ell^{(j)} \cdot \mathbf{x}_{t+\ell-\lfloor k/2 \rfloor} + b^{(j)} \right), \tag{3.19}$$

where:

- $\mathbf{x}_{t+\ell-\lfloor k/2 \rfloor} \in \mathbb{R}^{C_{\text{in}}}$ is the input vector at offset $t + \ell - \lfloor k/2 \rfloor$,
- $\mathbf{W}_\ell^{(j)} \in \mathbb{R}^{C_{\text{in}}}$ is the weight vector at kernel position ℓ of filter j,
- $b^{(j)} \in \mathbb{R}$ is the bias term for filter j,
- ϕ is a non-linear activation function, such as ReLU.

A visualization of the convolution operation found in Equation 3.19 is further illustrated with Figure 3.4.

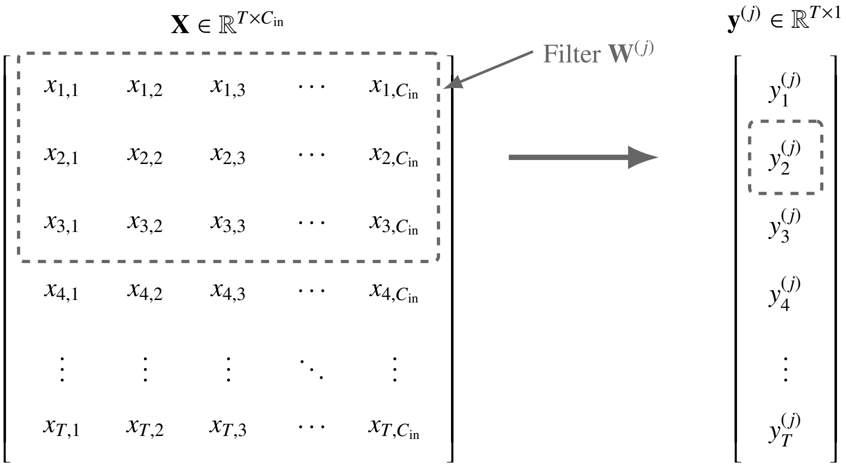

$\mathbf{X} \in \mathbb{R}^{T\times C_{\text{in}}}$
$\mathbf{y}^{(j)} \in \mathbb{R}^{T\times 1}$
Filter $\mathbf{W}^{(j)}$
$x_{1,1}$ $x_{1,2}$ $x_{1,3}$ $\cdots$ $x_{1,C_{\text{in}}}$
$x_{2,1}$ $x_{2,2}$ $x_{2,3}$ $\cdots$ $x_{2,C_{\text{in}}}$
$x_{3,1}$ $x_{3,2}$ $x_{3,3}$ $\cdots$ $x_{3,C_{\text{in}}}$
$x_{4,1}$ $x_{4,2}$ $x_{4,3}$ $\cdots$ $x_{4,C_{\text{in}}}$
$x_{T,1}$ $x_{T,2}$ $x_{T,3}$ $\cdots$ $x_{T,C_{\text{in}}}$
$y_1^{(j)}$
$y_2^{(j)}$
$y_3^{(j)}$
$y_4^{(j)}$
$y_T^{(j)}$

Figure 3.4: Graphical explanation of the 1D convolution in Equation (3.19). The input $\mathbf{X} \in \mathbb{R}^{T\times C_{\text{in}}}$ is convolved with filter $\mathbf{W}^{(j)}$ to produce the single-filter output $\mathbf{y}^{(j)} \in \mathbb{R}^{T\times 1}$. Each element $\mathbf{y}_t^{(j)}$ results from a weighted sum over a local temporal window of the input (size k) across all C_{in} channels, followed by the nonlinearity $\phi(\cdot)$.

Applying all C_{out} filters to the input matrix $\mathbf{X}$ results in the full output of the convolutional layer, which is given by:

$$\mathbf{Y} = [\mathbf{y}^{(1)}, \mathbf{y}^{(2)}, \ldots, \mathbf{y}^{(C_{\text{out}})}] \in \mathbb{R}^{T\times C_{\text{out}}}. \tag{3.20}$$

In $\mathbf{Y}$, each row corresponds to a time step and each column corresponds to the output of a filter. For simplicity, and to express the convolutional operation more compactly, the process can also be expressed as:

$$\mathbf{Y} = \phi\,(\mathbf{X} * \mathbf{W}) + \mathbf{b}, \tag{3.21}$$

where $\mathbf{W} \in \mathbb{R}^{k\times C_{\text{in}}\times C_{\text{out}}}$ is the full weight tensor of the layer, illustrated in Figure 3.5, and $*$ denotes a 1D convolution.

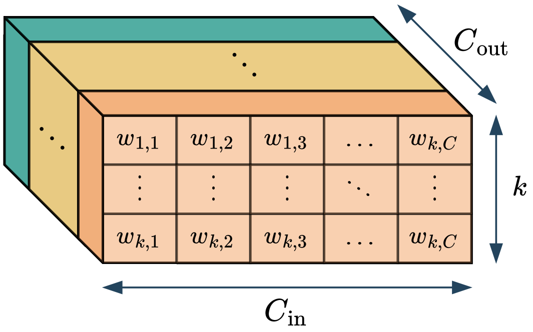

C_{out}
k
C_{in}
$w_{1,1}$ $w_{1,2}$ $w_{1,3}$ $\ldots$ $w_{k,C}$
$w_{k,1}$ $w_{k,2}$ $w_{k,3}$ $\ldots$ $w_{k,C}$

Figure 3.5: Visualization of the weight tensor $\mathbf{W} \in \mathbb{R}^{k\times C_{\text{in}}\times C_{\text{out}}}$ used in a `Conv1D` layer. The kernel has a temporal dimension k, spans all C_{in} input channels, and is repeated for each of the C_{out} output filters.

The total number of trainable parameters in a `Conv1D` layer is:

$$N_{\text{conv1D}} = \underbrace{C_{\text{in}} \cdot C_{\text{out}} \cdot k}_{\text{regular conv}} + \underbrace{C_{\text{out}}}_{\text{bias}} . \tag{3.22}$$

The first term represents the weights of the kernels, and the second term corresponds to one bias per output filter. When both the number of input channels C_{in} and the number of output channels C_{out} are large, the memory and computational footprint can become too significant due to the higher parameter count. To mitigate this problem, just like with the simple linear transformation, the convolution operation can be dynamically truncated. However, directly changing the number of filters also alters the output matrix $\mathbf{Y}$, since its dimensionality depends on C_{out}. To ensure the output dimensionality remains unaffected by the compression mechanism, the layer must first be reformulated as a two-step convolutional structure.

The two-step convolution can be seen as the low-rank factorization of the convolutional kernel. Instead of directly learning a large kernel $\mathbf{W}$ of shape $k \times C_{\text{in}} \times C_{\text{out}}$ for one convolution, the operation is decomposed into two consecutive layers. The first 1D convolution works the same as described in Equation 3.19, but, ideally, with a reduced number of filters r, referred to as the low-rank temporal convolution. The second step is a so-called pointwise convolution, which has a kernel size of 1 and performs a linear projection from the r intermediate channels back to the desired proposed C_{out} outputs without changing the temporal resolution. The aim of this decomposition is to enable the compression layer while making sure it maintains the same capabilities as a standard convolution and ensuring consistency in the model's output. Figure 3.6 illustrates this decomposition.

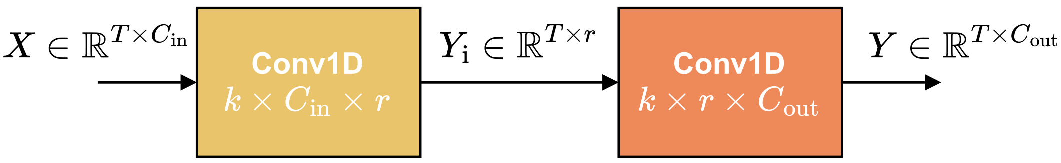

$X \in \mathbb{R}^{T \times C_{\text{in}}}$
Conv1D
$k \times C_{\text{in}} \times r$
$Y_{\text{i}} \in \mathbb{R}^{T \times r}$
Conv1D
$k \times r \times C_{\text{out}}$
$Y \in \mathbb{R}^{T \times C_{\text{out}}}$

Figure 3.6: Two-step convolution layer. The first step applies a temporal convolution with reduced rank r, followed by a pointwise (1×1) convolution that projects back to the output dimension C_{out}, ensuring fixed dimensionality while enabling compression.

This operation can be expressed as:

$$\mathbf{Y} = \left(\phi\left(\mathbf{X} * \mathbf{W}^{(1)}\right)\right) * \mathbf{W}^{(2)} + \mathbf{b}, \tag{3.23}$$

where:

- $\mathbf{W}^{(1)} \in \mathbb{R}^{k \times C_{\text{in}} \times r}$ is the kernel of the temporal convolution,
- $\mathbf{W}^{(2)} \in \mathbb{R}^{1 \times r \times C_{\text{out}}}$ is the pointwise kernel that linearly combines the r intermediate channels,
- $\mathbf{b} \in \mathbb{R}^{C_{\text{out}}}$ is the bias vector,
- ϕ is a non-linear activation function inserted between the two steps.

Truncation

In the same manner of matrix decomposition, the inner rank r of the two-step convolution decides the strength of compression. The total number of parameters in the decomposed convolution layer is:

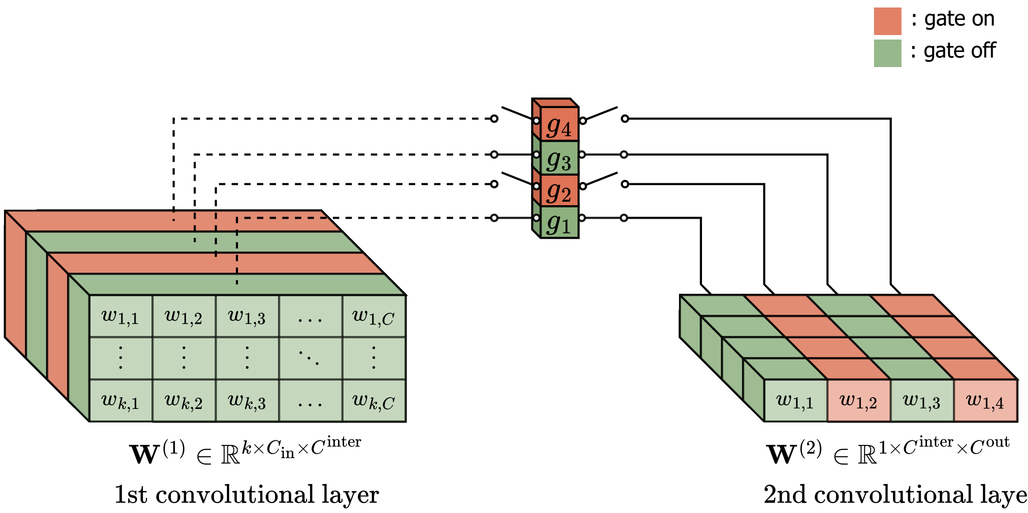
$$N_p = \underbrace{C_{\text{in}} \cdot r \cdot k}_{\text{temporal conv}} + \underbrace{r \cdot C_{\text{out}}}_{\text{pointwise conv}} + \underbrace{C_{\text{out}}}_{\text{bias}} = r \cdot (C_{\text{in}} \cdot k + C_{\text{out}}) + C_{\text{out}}. \tag{3.24}$$

To make sure that the parameter count of the factorized kernels does not exceed the original full-rank kernel, the upper bound on r is defined as:

$$r_{\max} = \left\lfloor \frac{C_{\text{in}} \cdot C_{\text{out}} \cdot k}{C_{\text{in}} \cdot k + C_{\text{out}}} \right\rfloor. \tag{3.25}$$

Controllable Decomposition

Similar to the matrix decomposition discussed in Section 3.1.1, the gate variables can be introduced to control the effective decomposed convolution. Each filter in the low-rank convolution is associated with a gate, which will give the model the ability to keep or deactivate certain filters during training. The vector $\mathbf{g}$ contains all the gate variables, thus for all filters. Since the output of the first low-rank convolution depends on the number of filters active, the kernel size of the pointwise convolution also changes dynamically. This is achieved by applying the same gates inside each kernel of the pointwise convolution, which effectively masks out the inactive filter. This gating can be expressed as a channel-wise mask on the pointwise kernel:



$\mathbf{W}^{(1)} \in \mathbb{R}^{k \times C_{\text{in}} \times C^{\text{inter}}}$

1st convolutional layer

$\mathbf{W}^{(2)} \in \mathbb{R}^{1 \times C^{\text{inter}} \times C^{\text{out}}}$

2nd convolutional layer

Figure 3.7: Illustration of the gating mechanism applied to a two-step convolutional layer. The first convolution $\mathbf{W}^{(1)}$ projects the input into $r_{\max}$ intermediate channels. A gate vector $\mathbf{g}$ determines which of these channels remain active: inactive gates (red) prune the corresponding filters, while active gates (green) allow information flow. The pointwise convolution $\mathbf{W}^{(2)}$ then combines only the active channels to produce the final C_{out} outputs. This enables dynamic compression of the convolutional kernel while preserving the output dimensionality.

$$\widetilde{\mathbf{W}}^{(2)}[:,i,:] = g_i \cdot \mathbf{W}^{(2)}[:,i,:] \quad \text{for } i = 1,\ldots,r_{\max}, \tag{3.26}$$

where $\widetilde{W}^{(2)}$ represents the the adjusted pointwise kernel after gating. This is then combined with the temporal layer, which results in the overall output:

$$\mathbf{Y} = \left(\phi\left(\mathbf{X} * \mathbf{W}^{(1)}\right)\right) * \widetilde{\mathbf{W}}^{(2)} + \mathbf{b}. \tag{3.27}$$

The visual representation of the total gating process, as described in Equation 3.27, can be found in Figure 3.7. This approach allows the model to dynamically adjust the number of active filters during training, guided by the optimization objective, which will be discussed later. As a result, it encourages parameter efficiency by suppressing unnecessary filters while retaining the expressive power of the original full-rank convolution.

In summary, the decomposition of deep learning layers provides a great tool for reducing the number of parameters and computation while retaining expressive capacity. The level of compression of these decompositions is dependent on inner rank r and can differ per layer. Therefore, by introducing gate variables to deactivate or activate components from the decomposition, the model can dynamically find the optimal dimensionality for every layer. The gating mechanism can thus be interpreted as a form of pruning, applied dynamically during training rather than post hoc. This paves the way for hardware-aware model compression, where the parameter budget can be directly enforced or penalized during optimization.

3.2 Quantization

The second method to compress the model is through quantization. It is a fundamental technique that lowers the precision level of parameters as fewer bits are required to represent weights, thereby reducing the storage space required. To enable controllable quantization within the model, the standard formulation needs to be rewritten and extended, such that the quantization process itself can be adjusted during training. This section will define uniform quantization and how it is altered to the controllable formulation.

3.2.1 Uniform Quantization

Every layer contains a weight matrix $\mathbf{W}$ which consists of elements/weights (x) ranging from α to β. Standard uniform quantization divides this range, $[\alpha, \beta]$, into equal intervals separated by boundary values. Each weight is mapped to the nearest representative value from this set of boundary values, also referred to as quantization levels. The step size s defines the distance between every level and is computed as:

$$s = \frac{\beta - \alpha}{2^b - 1}, \tag{3.28}$$

where b is the bitwidth, the number of bits per weight to represent each quantized value, and 2^b is the number of available quantization levels. With the step size, every weight x can be transformed into a quantized weight x_q by rounding it to the nearest level as:

$$x_q = s \left\lfloor \frac{x}{s} \right\rceil, \tag{3.29}$$

with $\lfloor\cdot\rceil$ denoting the round-to-nearest-integer function. The resulting x_q is the closest representable value to x on the quantized grid. This process is visualized in Figure 3.8 shown below.

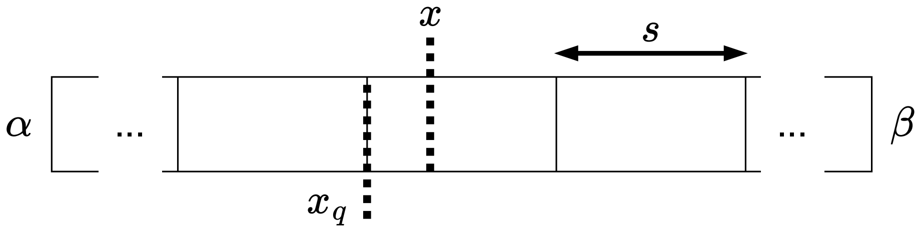


Figure 3.8: Uniform quantization. The weight range $[\alpha,\beta]$ is divided into intervals of equal width s. A weight x is mapped to the nearest quantization level x_q, which lies on the discretized grid defined by the step size. This discretization reduces precision while compressing the representation.

While uniform quantization is simple, there are some disadvantages to this way of compression.

First of all, its effectiveness is heavily dependent on selecting the most fitting values for α, β, and b. Uniform quantization is usually applied on a pre-trained model, from which its corresponding α and β can be deduced and defined. The most effective bitwidth is then found through grid search optimization. However, the chosen configuration may not be optimal, which, in combination with the accuracy degradation introduced by quantization, can result in additional performance loss. To mitigate the performance degradation, fine-tuning following quantization is commonly applied. However, it may only partially recover the lost accuracy.

Secondly, the optimal quantization parameters, α, β, and b, can differ across different layers. Applying the same quantization uniformly across all layers may lead to second-rate results. Deriving layer-specific values can improve the quantized model's performance, but it increases the number of hyperparameters to tune. This process becomes computationally expensive and impractical when working with deeper networks. These limitations motivate the need for a more adaptive quantization mechanism.

3.2.2 Controllable Quantization

Thus, uniform quantization lacks adaptability by fixing the hyperparameters and allows for differences between the training and inference behaviours, making it a static process. To overcome this, the quantization will be reformulated as a dynamic and controllable quantization. The model itself will learn the appropriate values for α, β, and b for each layer during training. To be more specific, these parameters will no longer be fixed but will become learnable variables [33]. Instead of mapping weights as Equation 3.29, this method will decompose the quantization into a sequence of residual corrections, each corresponding to a higher level of precision.

Applying quantization to a weight x at a given bitwidth b_n yields x_{b_n}, which most likely differs from the original value. Logically, choosing a higher bitwidth, b_{n+1}, will result in an improved approximation $x_{b_{n+1}}$. The difference between these two approximations will be referred to as the residual error ϵ:

$$\epsilon_{b_{n+1}} = x_{b_{n+1}} - x_{b_n} \quad \longrightarrow \quad x_{b_{n+1}} = x_{b_n} + \epsilon_{b_{n+1}}. \tag{3.30}$$

This relationship makes it possible to express the higher-precision quantized value $x_{b_{n+1}}$ as the sum of a lower-precision value plus residual corrections. This process is illustrated in Figure 3.9 below.

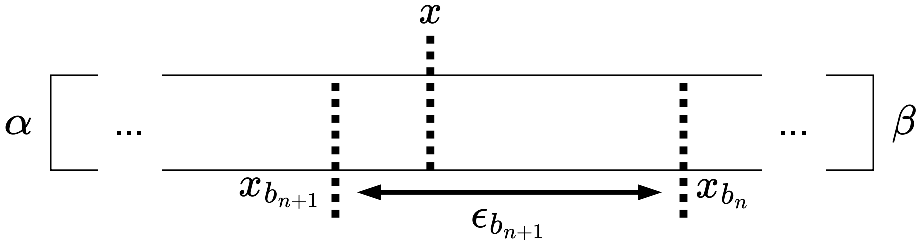


Figure 3.9: Residual error in quantization. Quantizing x with bitwidth b_n yields x_{b_n}, while a higher bitwidth b_{n+1} produces $x_{b_{n+1}}$, which is closer to the original value. The difference $\epsilon_{b_{n+1}} = x_{b_{n+1}} - x_{b_n}$ represents the residual correction, which refines the lower-precision approximation.

To make the quantization controllable for the model, this additive property must hold, which is not guaranteed.

As briefly mentioned earlier, the step size defines the distance between every possible quantization level to which a numeric value can be mapped. The set that contains all these levels for a certain bitwidth b is defined as:

$$Q_b = \left\{\alpha + k \cdot s_b \;\middle|\; k \in \{0, 1, \dots, 2^b - 1\}\right\} \tag{3.31}$$

The additive property defined in Equation 3.30 remains only valid if the quantization levels at bitdwidth b_n, Q_{b_n}, are a subset of those at b_{n+1}, $Q_{b_{n+1}}$:

$$Q_{b_n} \subseteq Q_{b_{n+1}}. \tag{3.32}$$

This property is fulfilled when the step size s_{b_n} is an integer multiple of the step size $s_{b_{n+1}}$:

$$\frac{s_{b_n}}{s_{b_{n+1}}} \in \mathbb{N} \quad \Longleftrightarrow \quad \frac{2^{b_{n+1}} - 1}{2^{b_n} - 1} \in \mathbb{N}. \tag{3.33}$$

In other words, the property is ensured when the quantization levels set for a higher bitwidth partially overlap with the previous quantization level set and insert new ones in between them consistently. The largest set of bitwidths for which this holds is $b \in 2, 4, 8, 16, 32$.

The residual error $\epsilon_{b_{n+1}}$ is computed by quantizing the remaining difference between the original value and its current approximation:

$$\epsilon_{b_{n+1}} = s_{b_{n+1}} \left\lfloor \frac{x - x_{b_n}}{s_{b_{n+1}}} \right\rceil, \tag{3.34}$$

where $s_{b_{n+1}}$ is the step size associated with bitwidth b_{n+1}. The step size is recursively defined as:

$$s_{b_{n+1}} = \frac{s_{b_n}}{2^b + 1}, \tag{3.35}$$

where b is the base number of bits added at each stage, which is $b = 2$ in the case of doubling. Combining everything, the quantization can be represented as a summation of the base quantization and error residuals ϵ_{b_n}.

$$x_q = x_2 + \epsilon_4 + \epsilon_8 + \epsilon_{16} + \epsilon_{32}. \tag{3.36}$$

Now that the quantization is rewritten (Equation 3.36), it can be made controllable for the model by introducing gate variables. For quantization purposes, they are denoted as g. Exactly as for the decomposition described in Section 3.1, these variables are binary and are controlled by the model. By adding a gate variable, g_i for each precision level, $i \in \{4, 8, 16, 32\}$, the model can enable and disable residual components. The adjusted and controllable expression for the quantization is defined as:

$$x_q = x_2 + g_4 \left(\epsilon_4 + g_8 \left(\epsilon_8 + g_{16} \left(\epsilon_{16} + g_{32}\epsilon_{32}\right)\right)\right). \tag{3.37}$$

In this nested structure, a higher precision can only be reached if the preceding gates are all active. This design enables the model to adaptively trade off precision and efficiency, learning where representation is beneficial while suppressing unnecessary bitwidth expansions.

3.3 Gate Variables

The previous sections 3.1 and 3.2 introduced how gate variables can control the dimensionality of layers and the precision of their parameters. This section will go into how these gating variables are defined and learned during training.

As mentioned before, the gate variable g_i acts like a switch that can either be turned on, $g_i = 1$, or off, $g_i = 0$. However, using such a binary mechanism would make the system non-differentiable and incompatible with gradient-based optimization. To make it possible for the model to learn whether a gate should be active or not, the gate should be a continuous variable that is differentiable. To achieve this, the sigmoid function is used as it maps a real-valued ϕ_i to the interval $(0, 1)$:

$$p_i = \sigma(\phi_i) = \frac{1}{1 + e^{-\phi_i}}, \tag{3.38}$$

with $p_i \in (0, 1)$ and ϕ_i is learnable prameter for the model. Intuitively, if a decomposition or precision component contributes positively to the accuracy, then the model should push ϕ_i during the training process to a higher value. A higher value will make the corresponding gate p_i approach 1 as seen in Figure 3.10. Vice versa, for a gate that should be turned off, its ϕ_i should be shifted to a low value, such that sigmoid maps it to close to 0.

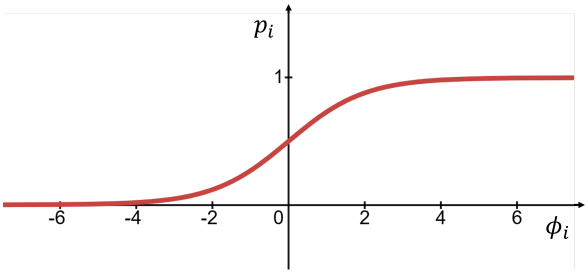


Figure 3.10: Sigmoid mapping from logit to gate probability. The learnable logit ϕ_i is transformed by $\sigma(\cdot)$ into $p_i \in (0, 1)$, which can be interpreted as the probability that gate i is active. Larger ϕ_i produces larger p_i.

Although the relaxation by the sigmoid function gives differentiability, the model still needs discrete gates in the forward pass. This switch effect is required to define the effective model structure, which ensures that the outputs only depend on the active components. A straightforward approach would be to set a threshold, and consider the gates active if their probability are above this value. However, for a better exploration of ϕ_i and reducing its early convergence to suboptimal values, a stochastic Bernoulli sampling scheme is used during the training phase:

$$g_i \sim \text{Bernoulli}(p_i). \tag{3.39}$$

This introduces randomness, discouraging co-adaptation in the same way as structured dropout, and improves the generalization of the model. In conclusion, there is a greater need for the architecture to be stable, so a deterministic hard decision threshold is suited:

$$\hat{g}_i = \begin{cases} 1 & \text{if } p_i > \tau \\ 0 & \text{otherwise} \end{cases} \tag{3.40}$$

Under the assumption that the cost of activating an ineffective gate and the cost of deactivating a useful one are the same, $\tau = 0.5$ is the neutral and unbiased choice.

Nevertheless, this sampling of g_i is non-differentiable. To make back propagation possible, the gradient of the Bernoulli sampled gate can be approximated by a differential version using a straight-through estimator (STE). This approach replaces the backward pass function with a differentiable surrogate, while leaving the forward pass unchanged. The forward computation will thus still use the sampled binary gate g_i. The backward computation will use the differentiable probability p_i through STE. The straight-through estimator is defined as:

$$\tilde{g}_i = g_i + p_i - \text{stop\_gradient}(p_i), \tag{3.41}$$

which means,

$$\frac{\partial \tilde{g}_i}{\partial \phi_i} = \frac{\partial p_i}{\partial \phi_i}. \tag{3.42}$$

In these equations, $\tilde{g}_i$ is the pseudo-binary gate, which acts as a binary gate during inference but has a gradient as if $\tilde{g}_i = p_i$:

$$\tilde{g}_i = \begin{cases} g_i & \text{(used in the forward pass)} \\ p_i & \text{(used for gradient computation in the backward pass)} \end{cases} \tag{3.43}$$

To encourage early-stage exploration and confident decisions later, a temperature variable T is introduced into the sigmoid function. It will control the flattening strength on the sigmoid function as:

$$p_i = \sigma\left(\frac{\phi_i}{T}\right). \tag{3.44}$$

When a relatively higher T is applied, the sigmoid function becomes flatter and thus the probabilities will be pushed closer towards 0.5. In combination with Bernoulli sampling, this will increase the stochasticity during training. In simpler words, the model will explore more gate configurations instead of committing to one direction. A relatively lower T will make the probabilities out of the sigmoid function behave more sharply, approaching 0 or 1 for more values of ϕ. These effects of temperature scaling are illustrated in Figure 3.11.

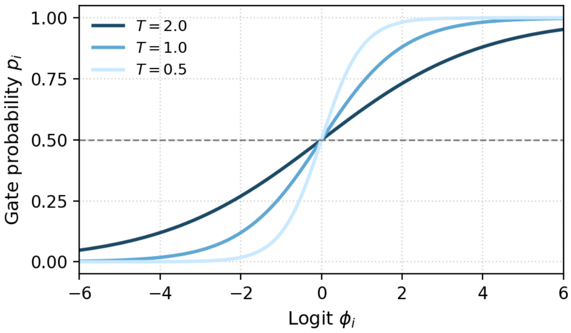


Figure 3.11: Effect of the temperature T on $\sigma(\phi/T)$.

To ensure that a wide range of configurations is explored during the initial training phase and gradually converges to more decisive selections, the temperature is initialized at a high value and steadily decreased. This temperature scheduling is defined as:

$$T = \max(T_{\text{min}}, \alpha^k T_0), \tag{3.45}$$

where T_0 is the initial temperature, $\alpha \in (0, 1)$ is the decay rate and k refers to the update step. The lower bound T_{min} is there to ensure that the temperature does not become close to zero.

To summarize the essence, the gating mechanism is defined by the temperature-controlled sigmoid and Bernoulli sampling

$$p_i = \sigma\left(\frac{\phi_i}{T}\right), \quad g_i \sim \text{Bernoulli}(p_i). \tag{3.46}$$

This stochastic gating ensures that the model does not overcommit to specific gates too early in the training process and encourages regularization. From a compression standpoint, the gates allow the model to choose which components and precision are needed adaptively through the loss function. There is no need for manually set hyperparameters or fixed structure definitions. The network learns which layers require additional dimensionality and a higher precision, and which ones can be compressed further. After training, the learned gating provides insight into which layers are sensitive to compression, helping to understand the model's behaviour and offering additional insights. The level of compression will be defined by the loss function, which will be the topic of the next section.

3.4 Loss Function

To ensure the model compresses itself, it should be optimized not only for the task at hand but also for minimizing its size and computational requirements. Therefore, the loss function for the model must be a balance between performance and compression. This section will define how this balance is derived.

When using gate variables, they are seen as stochastic latent variables whose values are not deterministic but must be inferred during training. As described in Section 3.3, the stochastic nature enables flexible model capacity, allowing the network to dynamically adjust the structure used to process each input. The structure is thus dependent on random variables drawn from a learnable distribution. In supervised problems, the goal is to predict the target variables $\mathbf{y}$ given an input $\mathbf{x}$ and the model parameters. With stochastic gates, however, this relationship also depends on the latent variables $\mathbf{z}$. For cleaner and standard notation, the collection of all gate variables is represented as $\mathbf{z}$. Therefore, the deterministic formulation has to be rewritten as a Bayesian formulation:

$$p(\mathbf{y} \mid \mathbf{x}) = \int p(\mathbf{y} \mid \mathbf{x}, \mathbf{z}; \theta)\, p(\mathbf{z})\, d\mathbf{z}. \tag{3.47}$$

Here, θ represents all deterministic model weights and $\mathbf{z}$ the latent binary gates. The function $p(\mathbf{y} \mid \mathbf{x}, \mathbf{z}; \theta)$ is the likelihood of the model predicting y given gates $\mathbf{z}$ and parameters θ, and $p(\mathbf{z})$ is the prior. Together they define the marginal likelihood $p(\mathbf{y} \mid \mathbf{x})$, also referred to as the model evidence, shown in Equation 3.47. It is the probability that the model assigns to the input data (y, x) after averaging over all possible gating patterns, weighted by the likelihood of the pattern occurring. In simpler terms, it gives an estimate of how likely it is to observe that the model will predict output y for input x, considering both the deterministic and latent variables. Therefore, during training, the goal is to maximize the marginal likelihood, which encourages the model to make, on average, correct predictions.

The prior is chosen by the model designer, and it should reflect any beliefs and properties about the model parameters. In the context of compression, the goal is to minimize the number of gates turned on. However, once the data has been observed, the prior can be updated after seeing both the input and the output. This is quantified by the posterior, which can be defined using Bayes' rule:

$$p(\mathbf{z} \mid \mathbf{x}, \mathbf{y}) = \frac{p(\mathbf{y} \mid \mathbf{z}; \theta)\, p(\mathbf{z})}{p(\mathbf{y} \mid \mathbf{x})}. \tag{3.48}$$

The posterior combines the prior $p(\mathbf{z})$, the assumption about the gating variables, with the likelihood $p(\mathbf{y} \mid \mathbf{z}; \theta)$, which quantifies how well each gating configuration explains the observed data. The denominator, $p(\mathbf{y} \mid \mathbf{x})$, is the earlier defined marginal likelihood, ensuring the posterior is properly normalized. In summary, the posterior $p(\mathbf{z} \mid \mathbf{x}, \mathbf{y})$ encodes the updated beliefs about the gates after observing the data. The marginal likelihood is then denoted as:

$$p(\mathbf{y}|\mathbf{x}) \approx \int p(\mathbf{y}|\mathbf{z}; \theta)\, p(\mathbf{z}|\mathbf{x}, \mathbf{y})\, d\mathbf{z}. \tag{3.49}$$

Finding the posterior distribution is the key in Bayesian Machine learning, but in most cases, it is intractable and computationally infeasible. Looking at Equation 3.48, the denominator requires computing the marginal likelihood, which is the integral over all possible model parameter combinations (see Equation 3.47). When working with deep models consisting of a large number of parameters, there will be an infinite number of combinations to integrate over. Therefore, to evade this, the posterior is approximated by a variational distribution denoted as $q(\mathbf{x}|\mathbf{z};\phi)$. This variational posterior is tractable and replaces the prior in Equation 3.47, leading to:

$$p(\mathbf{y}|\mathbf{x}) \approx \int p(\mathbf{y}|\mathbf{z};\theta)\, q(\mathbf{z}|\mathbf{x};\phi)\, d\mathbf{z}. \tag{3.50}$$

Thus, the goal is to make the variational posterior $q(\mathbf{z}|\mathbf{x})$ as close as possible to the true posterior $p(\mathbf{z}|\mathbf{x})$. The discrepancy, the difference, between two distributions can be measured by the Kullback-Leibler (KL) divergence:

$$\mathrm{KL}\left(q(\mathbf{z}|\mathbf{x}) \,||\, p(\mathbf{z}|\mathbf{x})\right) = \mathbb{E}_{q(\mathbf{z}|\mathbf{x})}\left[\log \frac{q(\mathbf{z}|\mathbf{x})}{p(\mathbf{z}|\mathbf{x})}\right]. \tag{3.51}$$

3.4.1 Derivation of the ELBO

As mentioned, the overall goal is to find the model parameters and latent variable configurations that maximize the marginal likelihood as defined in Equation 3.47, which is intractable. Therefore, the function Equation 3.49 is maximized with the tractable variational posterior. Although it is tractable, it is calculated as a sum of products, which can be numerically unstable and is hard to optimize for. Consequently, it is common to maximize the logarithm of the marginal likelihood instead, which turns products into sums:

$$\log p(\mathbf{y} \mid \mathbf{x}) = \log \int q(\mathbf{z} \mid \mathbf{x};\phi) \frac{p(\mathbf{y},\mathbf{z} \mid \mathbf{x};\theta)}{q(\mathbf{z} \mid \mathbf{x};\phi)} d\mathbf{z}. \tag{3.52}$$

Taking the logarithm of the marginal likelihood makes this function concave, but the log remains outside the integral, which is hard to compute. This difficulty is solved by applying Jensen's inequality, written as:

$$\log(\mathbb{E}[X]) \geq \mathbb{E}[\log(X)]. \tag{3.53}$$

It states that the expectation of a concave function is less than or equal to the function of the expectation, which results in a tractable lower bound:

$$\log p(\mathbf{y} \mid \mathbf{x}) \geq \int q(\mathbf{z}) \log \frac{p(\mathbf{y},\mathbf{z} \mid \mathbf{x};\theta)}{q(\mathbf{z})}, d\mathbf{z} = \mathbb{E}_{q(\mathbf{z}|\mathbf{x})}\left[\log p(\mathbf{y},\mathbf{z} \mid \mathbf{x};\theta) - \log q(\mathbf{z})\right]. \tag{3.54}$$

This right-hand side can then be maximized as a replacement objective, known as the Evidence Lower Bound (ELBO). With the chain rule property shown in Equation 3.55

$$p(\mathbf{y},\mathbf{z} \mid \mathbf{x};\theta) = p(\mathbf{y} \mid \mathbf{z},\mathbf{x};\theta)\, p(\mathbf{z}), \tag{3.55}$$

and by recalling Equation 3.51, the ELBO can be even further decomposed as:

$$\mathbb{E}_{q(\mathbf{z}|\mathbf{x})}\left[\log p(\mathbf{y},\mathbf{z} \mid \mathbf{x};\theta) - \log q(\mathbf{z})\right] = \mathbb{E}_{q(\mathbf{z}|\mathbf{x})}\left[\log p(\mathbf{y} \mid \mathbf{z},\mathbf{x};\theta)\right] - \mathrm{KL}\left(q(\mathbf{z})||p(\mathbf{z})\right) = \mathrm{ELBO}. \tag{3.56}$$

3.4.2 Definition of ELBO

The first term of the ELBO, which is:

$$\mathbb{E}_{q(\mathbf{z}|\mathbf{x})}[\log p(\mathbf{y} \mid \mathbf{z}; \theta)], \tag{3.57}$$

captures how well the model fits the data. In other words, it measures how well the model can predict the observed labels under different gate configurations. Since the expectation over $q(\mathbf{z})$ is non-computable for large models, it is approximated using Monte Carlo sampling. For the loss function Equation 3.85, the gating mask is sampled for each input in the batch, and the log-likelihood is averaged over the batch:

$$\frac{1}{N} \sum_{i=1}^{N} \log p_\theta(\mathbf{y}_i \mid \mathbf{x}_i, \mathbf{z}_i), \tag{3.58}$$

where each $\mathbf{z}_i \sim q(\mathbf{z})$. This is the average log-likelihood across the batch, which is the same as the categorical cross-entropy, the standard loss component for multi-class classification problems. The second term in the ELBO is:

$$\mathrm{KL}\left(q(\mathbf{z}) \,||\, p(\mathbf{z})\right). \tag{3.59}$$

As defined earlier, this is the KL divergence, which acts as a regularizer. This controls the model's compression through the latent gates and penalizes overfitting by discouraging the variational posterior and the prior from deviating.

3.4.3 Derivation of the KL Divergence

As defined in Section 3.3, the gating variables are Bernoulli random variables, and the distribution over the gates can be fully specified by their respective activation probabilities. This is for both the prior $p(z)$ and the variational posterior $q(z)$. The prior over a single gate variable is

$$p(z) = \mathrm{Bern}(z; \pi), \qquad \text{where } \pi = \mathbb{P}_p(z = 1). \tag{3.60}$$

Here, $\pi = p(1)$ is the prior probability of the gate being active. The variational posterior is

$$q(z) = \mathrm{Bern}(z; r), \qquad \text{where } r = \mathbb{P}_q(z = 1). \tag{3.61}$$

Here, $r = q(1)$ is the probability under the variational posterior that the gate is active and is learned and therefore is data-dependent. Then, the KL divergence Equation 3.51 is defined as follows:

$$\begin{aligned} D_{\mathrm{KL}}(q(z) \,||\, p(z)) &= \sum_{z \in \{0,1\}} q(z) \log \frac{q(z)}{p(z)} \quad &(3.62)\\ &= q(1) \log \frac{q(1)}{p(1)} + q(0) \log \frac{q(0)}{p(0)} \quad &(3.63)\\ &= r \log \frac{r}{\pi} + (1 - r) \log \frac{1 - r}{1 - \pi}. \quad &(3.64) \end{aligned}$$

By grouping the entropy terms as:

$$H(q) = -r \log r - (1 - r) \log(1 - r), \tag{3.65}$$

the KL divergence can be simplified to:

$$D_{\mathrm{KL}} = -H(q) + r(-\log \pi) - (1 - r) \log(1 - \pi). \tag{3.66}$$

In the context of Bayesian Bits [33] and many compression methods, the prior is chosen to encourage sparsity (compression), as

$$\pi = p(1) = e^{-\lambda}, \qquad (p(z) = \mathrm{Bern}(z; e^{-\lambda}) \tag{3.67}$$

Here, $\lambda > 0$ is the sparsity parameter selected, and the KL divergence will explicitly depend on it:

$$D_{\mathrm{KL}}(q(z) \,||\, p(z)) = -H(q) + r(-\log e^{-\lambda}) - (1 - r) \log(1 - e^{-\lambda}). \tag{3.68}$$

When high sparsity is enforced by $\lambda \gg 1$, the $\log(1 - e^{-\lambda})$ term and entropy term can be approximated to zero. Therefore, the estimation of the KL divergence becomes:

$$D_{\mathrm{KL}}(q(z) \,||\, p(z)) = \lambda \cdot r = \lambda \cdot q(z). \tag{3.69}$$

As discussed at the beginning of this chapter, there are two different types of gating mechanisms, independent gates (Section 3.1) and dependent gates (Section 3.2). This difference of properties changes when the total KL divergence:

$$D_{\mathrm{KL}}(q(\mathbf{z}) \,||\, p(\mathbf{z})), \qquad \text{where} \quad \mathbf{z} = (z_1, z_2, \ldots, z_L), \tag{3.70}$$

is calculated for L gating variables.

Independent Gates

When all the gates are independent, the prior and variational posterior are constructed as fully factorized Bernoulli distributions.

$$p(\mathbf{z}) = \prod_{\ell=1}^{L} p(z_\ell), \qquad q(\mathbf{z}) = \prod_{\ell=1}^{L} q(z_\ell), \tag{3.71}$$

where each $p(z_\ell)$ and $q(z_\ell)$ are Bernoulli distributions, as defined earlier. The joint KL divergence between the full variational posterior and prior over all gates is defined by

$$D_{\mathrm{KL}}(q(\mathbf{z}) \,\|\, p(\mathbf{z})) = \sum_{\mathbf{z} \in 0,1^L} q(\mathbf{z}) \log \frac{q(\mathbf{z})}{p(\mathbf{z})} \tag{3.72}$$

Given the product structure, the numerator and denominator inside the log also factorize:

$$\frac{q(\mathbf{z})}{p(\mathbf{z})} = \prod_{\ell=1}^{L} \frac{q(z_\ell)}{p(z_\ell)}, \tag{3.73}$$

and results in:

$$\log \frac{q(\mathbf{z})}{p(\mathbf{z})} = \sum_{\ell=1}^{L} \log \frac{q(z_\ell)}{p(z_\ell)}. \tag{3.74}$$

Now, the sum over all possible gate configurations $\mathbf{z}$ can be rewritten as:

$$D_{\mathrm{KL}}(q(\mathbf{z}) \parallel p(\mathbf{z})) = \sum_{\mathbf{z}} q(\mathbf{z}) \sum_{\ell=1}^{L} \log \frac{q(z_\ell)}{p(z_\ell)} \tag{3.75}$$

$$= \sum_{\ell=1}^{L} \sum_{\mathbf{z}} q(\mathbf{z}) \log \frac{q(z_\ell)}{p(z_\ell)}. \tag{3.76}$$

Due to independence, for any fixed ℓ, the sum over all $\mathbf{z}$ of $q(\mathbf{z})$ times a function of z_ℓ reduces to the marginal expectation over z_ℓ:

$$\sum_{\mathbf{z}} q(\mathbf{z}) \log \frac{q(z_\ell)}{p(z_\ell)} = \sum_{z_\ell} q(z_\ell) \log \frac{q(z_\ell)}{p(z_\ell)} = \sum_{\ell=1}^{L} D_{\mathrm{KL}}(q(z_\ell) \parallel p(z_\ell)). \tag{3.77}$$

Therefore, the joint KL divergence for L independent gates decomposes into a sum over the gates:

$$D_{\mathrm{KL}}(q(\mathbf{z}) \parallel p(\mathbf{z})) = \sum_{\ell=1}^{L} D_{\mathrm{KL}}(q(z_\ell) \parallel p(z_\ell)) \approx \sum_{\ell=1}^{L} r_\ell \lambda. \tag{3.78}$$

Hierarchical Gates

However, for more sophisticated setups, such as quantization-aware pruning, the activation of higher-order gates depends on the activation of lower-order gates. To take these dependencies into account, the total probability will be derived using the chain rule of probability, resulting in conditional distributions:

$$q(\mathbf{z}) = q(z_1) \prod_{j=2}^{J} q(z_j \mid z_{j-1}). \tag{3.79}$$

Here, z_1 represents the base gate variable, which corresponds to the lowest precision quantization bit. Higher-indexed z_j gates represent finer quantization bits that are only meaningful if the preceding gate z_{j-1} is active. Starting from the definition of the KL divergence Equation 3.72, and by substituting the product of conditionals, and using properties of the logarithm, the KL divergence is expanded as follows:

$$D_{\mathrm{KL}}(q(\mathbf{z}) \parallel p(\mathbf{z})) = \sum_{\mathbf{z}} q(\mathbf{z}) \left[\log \frac{q(z_1)}{p(z_1)} + \sum_{j=2}^{J} \log \frac{q(z_j \mid z_{j-1})}{p(z_j \mid z_{j-1})} \right] \tag{3.80}$$

$$= \sum_{z_1} q(z_1) \log \frac{q(z_1)}{p(z_1)} + \sum_{j=2}^{J} \sum_{\mathbf{z}} q(\mathbf{z}) \log \frac{q(z_j \mid z_{j-1})}{p(z_j \mid z_{j-1})}. \tag{3.81}$$

Here, J is the total number of gates in this hierarchical structure. This expression can be rewritten as a sum of marginal and conditional KL divergences:

$$D_{\mathrm{KL}}(q(\mathbf{z}) \,\|\, p(\mathbf{z})) = D_{\mathrm{KL}}(q(z_1) \,\|\, p(z_1)) + \sum_{j=2}^{J} \mathbb{E}_{q(z_1,\ldots,z_{j-1})} \left[D_{\mathrm{KL}}(q(z_j \mid z_{j-1}) \,\|\, p(z_j \mid z_{j-1})) \right], \tag{3.82}$$

where $\mathbb{E}_{q(z_1,\ldots,z_{j-1})}[\cdot]$ denotes the expectation with respect to the joint distribution over all parent gates under the variational posterior q. This is the same as scaling the KL divergence with the probability that all parent gates are turned on:

$$\mathbb{E}_{q(z_1,\ldots,z_{j-1})} \left[D_{\mathrm{KL}} \left(q(z_j \mid z_{j-1}) \,\|\, p(z_j \mid z_{j-1}) \right) \right] = \left(\prod_{k=1}^{j-1} q(z_k = 1) \right) D_{\mathrm{KL}} \left(q(z_j \mid z_{j-1} = 1) \,\|\, p(z_j \mid z_{j-1} = 1) \right). \tag{3.83}$$

Resulting in:

$$D_{\mathrm{KL}}(q(\mathbf{z}) \,\|\, p(\mathbf{z})) \approx \sum_{j=1}^{J} \lambda_q \left[\prod_{k=1}^{j} q(z_{l,k} = 1) \right] \tag{3.84}$$

3.4.4 Definition of the Loss Function

By maximizing the $\mathcal{L}_{\mathrm{ELBO}}$ defined in Equation 3.56, the model can be found that gives an accurate prediction and finds a good approximation to the true posterior. In deep learning, however, training is usually a minimization problem. Therefore, the loss function used during optimization is defined as the negative ELBO:

$$\mathcal{L}_{\mathrm{total}} = -\mathrm{ELBO} = -\log p_\theta(\mathbf{y} \mid \mathbf{x}, \mathbf{z}_i), + \mathrm{KL}\left(q(\mathbf{z}) \,\|\, p(\mathbf{z})\right) \tag{3.85}$$

Finally, with this total loss function and the conclusion regarding the KL divergence term, it can be described as three components for clarity:

$$\mathcal{L}_{\mathrm{total}} = \mathcal{L}_{\mathrm{CE}} + \lambda_q \cdot \mathcal{L}_{\mathrm{Q}} + \lambda_d \cdot \mathcal{L}_{\mathrm{D}} \tag{3.86}$$

Here:

- $\mathcal{L}_{\mathrm{CE}}$ is the cross-entropy loss.
- $\mathcal{L}_{\mathrm{Q}}$ is the penalty associated with quantization.
- $\mathcal{L}_{\mathrm{D}}$ is the penalty associated with decomposition.
- λ_q and λ_d are scalar hyperparameters associated with controlling prior. Therefore, in a more practical sense, they control the trade-off between accuracy and compression. The higher λ, the higher the compression.

The decomposition penalty, corresponding to independent gates as discussed in this section, is:

$$\mathcal{L}_{\mathrm{D}} = \frac{1}{|\mathcal{S}_D|} \sum_{l \in \mathcal{S}_D} \frac{1}{|\mathcal{G}_l|} \sum_{g \in \mathcal{G}_l} q(z_{l,g} = 1). \tag{3.87}$$

Here, $\mathcal{S}_D$ represents the set that contains every layer l whose structure has been decomposed, combined with the gating mechanism. Every layer l contains gates $z_{l,g}$ which are defined by their set $\mathcal{G}_l$. This part of the loss function calculates, in combination with λ_d, the total KL divergence loss. The strength of penalizing the independent gates having a high probability of being turned on is dependent on the prior definition.

The quantization penalty, corresponding to hierarchical gates, is:

$$\mathcal{L}_{\mathrm{Q}} = \frac{1}{|\mathcal{S}_Q|} \sum_{l \in \mathcal{S}_Q} \frac{1}{|\mathcal{B}|} \sum_{j \in \mathcal{B}} \left[\prod_{k=1}^{j} q(z_{l,k} = 1) \right]. \tag{3.88}$$

Likewise, the $\mathcal{S}_Q$ refers to the set that contains every layer l where quantization using gates is applied. $\mathcal{B}$ is the set of bitwidths for every layer, which is $\{2, 4, 8, 16, 32\}$. The function iterates over all layers, aggregates the probability of every gate, and divides this total by the total number of gates. Similarly, in combination with the prior's parameter λ_q, $\mathcal{L}_{\mathrm{Q}}$ represents the KL divergence. The strength of compression depends on the chosen prior.

By including both types of KL regularization in the loss, the model is encouraged to be simultaneously sparse while maintaining high task performance. The separation of the parameters also offers flexible control over the relative importance of quantization and decomposition penalties, and can be tuned according to the desired accuracy/compression trade-off. In practice, λ_q and λ_d are found on validation performance. When both hyperparameters are chosen too large, the model will be forced to be very small, but this will come at the cost of accuracy. When chosen too large, vice versa. This trade-off will be discussed more in depth in Chapter 5.

Experimental Framework

4

Putting all the components together discussed in Chapter 3, the proposed framework will be referred to as Bayesian Joint Compression (BJC). This chapter presents the experimental framework used to evaluate the BJC. It begins by describing the datasets used for training and evaluation. Then, the experimental setup, including the input configuration and model definitions, is explained. Finally, the evaluation metrics are given.

4.1 Datasets

To evaluate the proposed compression framework for Edge AI scenarios, two benchmark datasets for bearing fault detection are used: the Paderborn University (PU) Dataset [42] and the Case Western Reserve University (CWRU) Dataset [43]. Both datasets consist of a large collection of time series vibration data measured under various operational conditions and fault types, making them ideal for evaluation and comparison [44]. CWRU is the most frequently used benchmark for machine learning–based fault diagnosis because of its relatively large number of fault settings [45]. However, a limitation of the CWRU Dataset is that many studies achieve almost perfect classification accuracy on their models [46]. This saturation limits the ability to study the (subtle) performance differences, such as the impact of compression methods across different hyperparameter settings. On the other hand, the PU Dataset has been identified as a more challenging classification task due to its more overlapping class distributions [47]. While it is capable of achieving high accuracies, it never reaches perfect accuracy, better reflecting real-world scenarios. Therefore, the CWRU Dataset is important for its comparability and class diversity, while the PU Dataset ensures that conclusions can be drawn under more realistic conditions. Together, they provide a balanced evaluation setting.

A bearing is a crucial component in the industry that reduces friction between two surfaces in contact [45, 42]. Without bearings, the direct contact between surfaces would generate significant heat due to friction. Bearings can be found in every machine with moving parts, such as electric motors, cars, and wind turbines. They handle rotating motion by using balls as their rolling element. In short, they convert a sliding motion into a rolling motion, which is much more durable and energy-efficient. The bearing consists of three different parts:

- Inner Race: The inner ring of the bearing, which rotates along with the shaft it is mounted on.
- Outer Race: The outer ring of the bearing, which is stationary and houses the rolling balls.
- Rolling Elements (Balls): The rolling elements rotate between the inner and outer races, enabling smooth motion while reducing friction.

The goal of these datasets is to perform condition monitoring and detect health status through real-time analysis. The decline in performance can be caused by faults in each of the three elements, with varying severities, and may result from different failure mechanisms.

The CWRU Dataset contains time series vibration data of bearings, with faults introduced in all three bearing components, where each fault can have different fault diameters. These diameters can be of four levels, 0.007″, 0.014″, 0.021″, and 0.028″, thus separating different levels of damage. However, due to the availability in the dataset, only the first three damage levels are considered for the classification process, resulting in a total of 10 classes: nine fault classes and one healthy class [43].

The PU Dataset contains a wide range of fault types and damage modes, even more so than the CWRU Dataset. It includes the main mode and origin of the damage, which are categorized into two groups [42]:

- Pittings: The formulation of small craters, or pits, on the bearing. They are caused by repeated use and thus are a sign of fatigue. They can be on the outer ring and the inner ring.
- Indentations: Localized surface depressions or impressions. They are the result of immediate or localized force, moments of stress.

It further differentiates faults by their origin (presence in the inner race, outer race, or both components) and by the extent of the damage, making the dataset considerably more detailed than CWRU. Certain fault combinations are well-represented, whereas others appear only once, resulting in an imbalanced dataset for some classification tasks. To ensure that the evaluation is fair and comparable, this work considers only the main fault modes. Furthermore, since the inner race contains only one distinct type of fault, the dataset is reduced to six classes, one healthy class and five faulty classes [42]. This selection ensures that the classification task remains challenging, while avoiding a severe class imbalance that would otherwise bias the analysis. Another advantage of the PU Dataset is its great availability of time-series measurements from multiple sensors, including vibration, current, and torque

Table 4.1: Comparison of class structures used in this work for the CWRU and PU Datasets.

Class	CWRU (fault type and diameter)	PU (fault type and origin)
1	Healthy	Healthy
2	Inner race fault (0.007”)	Inner race fault (pitting)
3	Inner race fault (0.014”)	Outer race fault (pitting)
4	Inner race fault (0.021”)	Outer race fault (indentation)
5	Outer race fault (0.007”)	Combined fault (inner + outer, pitting)
6	Outer race fault (0.014”)	Combined fault (inner + outer, indentation)
7	Outer race fault (0.021”)	–
8	Ball fault (0.007”)	–
9	Ball fault (0.014”)	–
10	Ball fault (0.021”)	–

signals. However, this work will only use vibration data, as it is the most widely adopted signal type in the literature and allows for comparability with existing studies [44, 48]. The PU Dataset also includes measurements from artificially damaged bearings; however, our preliminary experiments have shown that these samples do not significantly impact the model, as they do not represent realistic degradation patterns. Therefore, bearing samples with actual damages from accelerated lifetime tests are used for training and evaluating the models. For performance evaluation, each dataset was split into five different train–test sets, and the results of every fold were saved. An overview of the resulting class structure for both datasets is presented in Table 4.1.

4.2 Experimental Setup

In this section, the experimental setup for the bearing fault classification task will be defined. The focus will be on explaining how the datasets are used and which models will be used for the evaluation.

To investigate the influence of the initial model size and the optimal choice for compression, two models with identical frameworks but different dimensionalities are trained for each dataset. Firstly, there is the base model, which is optimized for maximum accuracy using the minimal needed dimensions. Secondly, the compact model is designed to be much smaller, while allowing for the accuracy to be lower in comparison with the baselıine model. Both will be compressed with the proposed framework, and their performance will be assessed for multiple metrics, which will be discussed in Section 4.3. The dimensionality of this general model differs per task and dataset, but the framework remains consistent.

Input

The first element of the model is the input layer. When dealing with raw sensor data measured from a bearing, deep learning methods are usually applied directly. This means that the model itself learns the features during training, thus not relying on predefined features. However, using statistical features can be quite advantageous in the context of Edge Devices for multiple reasons. First of all, dealing with raw data requires deeper architectures, and its first layers require a high amount of parameters and computations to retain the most meaningful characteristics about the time series. Therefore, by first applying feature extraction through statistical features, the input data is compressed, and logically, the model is reduced in dimensionality. Secondly, there is a lot of research on which features are practical and most effective for bearing fault detection [49]. Lastly, it is not uncommon to combine statistical features with deep learning models. The model can be improved to be more robust and build upon the given features, and does not have to discover the indicators from the raw data. [49, 50]. Therefore, given that the research focus is on compression, and to outperform shallow machine learning models in accuracy and compactness, statistical features as the input are the more suitable choice.

To increase the number of training samples, every recording in the datasets is split into non-overlapping segments of two seconds. To enable the model to detect changes in this vibration signal, each of these two-second sequences is divided into T non-overlapping windows, and for each window F statistical features are extracted. The input layer will therefore receive a matrix of $T \times F$ consisting of feature vectors, each associated with a different timestamp. This input should provide a compressed but informative description of the signal.

Each feature vector consists of F elements. The statistical features used for training and evaluating the model include common time-domain indicators and more domain-specific ones often used in vibration analysis. The time-based features are mean, standard deviation, minimum, maximum, skewness, and kurtosis. The features derived from research in vibration classification include the root mean square (RMS), clearance factor, crest factor, and the derived combinations, maximum-mean, and maximum–minimum difference. These features have been proven effective in classifying vibration signals of bearings [49].

Model Definition

The models and their compressed versions follow the same structure, meaning they consist of the same layers but not necessarily the same dimensionality. It is designed to create a balance between high accuracy and compactness.

The input matrix is received by the convolutional layer block, which is used to extract low-level temporal features. This block contains two consecutive 1D convolutional layers. As described in Section 3.1, convolutional layers are well-suited for capturing local correlations across time windows. This makes them an effective first-layer choice for the processing of the input matrix. This block is then followed by the Global Average Pooling (GAP), which reduces the temporal dimension by averaging across time steps. This dimensionally reduced output will be the input for the next element of the model, the MHA layer. It is common for attention mechanisms to operate on inputs containing many time step sequences, as they are powerful in capturing long-range dependencies. However, because the classification task is relatively simple, experiments have shown that the GAP placed between the convolution and MHA blocks improves performance. The GAP reduces the dimensionality, which would otherwise be too high, and promotes better generalization. This is also beneficial from a model compression perspective, because a reduction in T reduces the number of computations (see Section 3.1.1). The second convolutional layer contains the number of filters required to match the same dimensionality as the attention layer. After attention, a feed-forward neural (FFN) block is applied, consisting of two dense layers. The intended task for this block is to take the features from attention and transform them into a more useful representation. Finally, the output layer is a fully connected dense layer that projects the high-dimensional output of the FFN onto the C-class output space. Each corresponds to a different health diagnosis, as listed in Table 4.1. An overview of the model layers, including their input/output dimensions and the number of parameters, is shown in Table 4.2. Here, T refers to the number of windows per input per sample, and F refers to the number of features per window. c represents the number of filters in the first convolutional layer, D is the dimensionality of the attention layer, and M is the hidden dimension of the FNN block.

The model, designed to maximize classification performance, is therefore rather large in terms of the number of parameters and total computations, and is referred to as the base

Layer	**In → Out (channels)**	**Output shape**	**Parameters**
Input	$F \to F$	(T, F)	–
Conv1D	$F \to c$	(T, c)	$(KF + 1)\,c$
Conv1D final	$c \to D$	$(T - (K - 1), D)$	$(Kc + 1)\,D$
GlobalAvgPool1D	–	(D)	–
Reshape	–	$(1, D)$	–
Multi-Head Attention	$D \to D$	$(1, D)$	$4D^2 + 4D$
Feed-Forward Dense1	$D \to M$	$(1, M)$	$DM + M$
Feed-Forward Dense2	$M \to D$	$(1, D)$	$MD + D$
GlobalAvgPool1D	–	(D)	–
Dense (head)	$D \to C$	(C)	$DC + C$

Table 4.2: Layer structure of the CNN-Transformer model, showing input–output dimensionality, output shapes, and parameter counts for the parametrized layers.

model. It will serve as a reference for the best-performing model in terms of accuracy for the classification problem. Then there is the compact model, which is designed to significantly reduce the number of parameters while sacrificing a slight decrease in classification accuracy compared to the base model. It provides a lightweight structure suitable for real-time or resource-constrained applications. The compact model uses the same building blocks as the base model but is optimized for smaller dimensionalities over all layers. The compression on both models should indicate whether it is more advantageous to compress the best model that can be designed or a more compact model. Table 4.3 defines the inner dimensions c, D, and M used in the base and compact model for the PU and CWRU Datasets.

Dataset	**c**	D	M	**Total parameters**
PU (base)	32	64	64	32,902
PU (compact)	32	16	16	4,438
CWRU (base)	16	32	32	8,906
CWRU (compact)	8	16	16	2,538

Table 4.3: Architectural hyperparameters defining the inner dimensions of the CNN-Transformer models (c: convolutional filters, D: attention dimension, M: feed-forward dimension) for the PU and CWRU Datasets, along with the resulting total parameter counts.

The Random Forest model will be used as the benchmark shallow machine learning model to beat. Its hyperparameters were selected through a grid search to ensure fair comparison with the deep learning model. As discussed in Chapter 2, Random Forests are one of the most effective shallow machine learning models for environments that are resource-constrained. Due to its simple nature, the computational load is almost negligible compared to deep learning methods. It has also been demonstrated to be the most effective model for vibration-based fault diagnosis. This makes Random Forest a strong benchmark for evaluating the benefits of the proposed compressed deep learning models.

4.3 Metrics

Four key metrics are used to assess the performance of the proposed dynamic compression method. The metrics, number of parameters, storage cost, computational cost, and inference time are described below:

Number of Parameters

The most straightforward metric for evaluating the model's compression is the number of trainable parameters. The parameter count is given by

$$N_{\text{total}} = \sum_{l \in L} N_l, \tag{4.1}$$

where N_l is the number of parameters in layer l. While informative, this metric will only reveal the effects of the decomposition and the influence of λ_d, since quantization effects cannot be perceived. Furthermore, the number of parameters does not provide the necessary insights for deployment on resource-limited hardware. A model with fewer parameters can still require large amounts of memory if those parameters are stored in high-bit formats. The same applies when a model with a relatively large parameter count can still be deployed if it is quantized to low bit-widths. Therefore, the number of parameters will serve as a baseline metric. In the following chapter, results will be shown with the term 'effective number of parameters'. As the only parameters that are counted are the ones that are associated with active gates, and therefore contribute to the effective output.

Parameter Storage

To build upon the parameter count metric, the most realistic measure of joint compression is the model's memory footprint, which represents its size. Minimizing the model size while maintaining accuracy is the core goal of this compression method. The number of bits needed to store all the parameters of the model can be described as:

$$\text{Storage} = \sum_{l \in L} b_l \cdot N_l, \tag{4.2}$$

where b_l denotes the bit-width used to quantize the layer l. This metric accounts for both the reduction in N_l because of pruning the decomposition structure and the decrease in b_l due to quantization. Therefore, the model's size provides the most insightful information regarding compression and deployment on hardware-constrained devices.

Floating Point Operations

Floating-point operations (FLOPs) quantify the total number of arithmetic operations performed in one forward pass. This metric will serve as a measure of computational efficiency.

The quantization introduces an overhead, as the parameters are converted from their low-precision representation to 32-bit. The decomposition pruning decreases this overhead, as the fewer parameters there are, the lower the overhead computations are. Moreover, due to the upper bound on the decomposition, the model is guaranteed to decrease in operations, aside from the overhead. This metric is therefore chosen to illustrate this trade-off and analyze the reduction in computation.

Inference Time

Lastly, the inference time is also taken into account. While FLOPs can provide a hardware-independent estimate of complexity, the actual runtime depends on other factors. Decomposition reduces the number of parameters but increases the number of steps within the model, which can lead to higher latency. Similarly, quantization introduces an additional step before using any weight matrix, which can also raise the inference time. It will therefore reflect whether the theoretical saving in parameters, storage, and FLOPs will also result in a practical performance gain. The inference time will be measured over many runs to achieve a fair estimate.

Together, along with accuracy, these four metrics capture different aspects of compression:

- **Number of Parameters:** Structural reduction.
- **Parameter Storage:** True memory footprint.
- **FLOPs:** Computational complexity.
- **Inference Time:** Empirical performance during deployment.

By analyzing these metrics together, the compressed models can be evaluated not only for their compactness and accuracy but also to determine if they are a practical improvement in resource-constrained environments.

5

Experimental Results

This chapter presents the experimental results obtained by applying the dynamic Bayesian Joint Compression (BJC) framework on the defined Base and Compact models. To systematically explain the different compression strategies, the results will be divided into four parts.

First, to illustrate how the gates behave and what layers are important, the number of active gates per layer is plotted. This is done for a few different values of λ_d and λ_q. Secondly, for each compression scenario, whether the factorization is of SVD or Tucker structure, and for each initial model size, the performance will be shown for many different regularization combinations. Another goal is to identify and determine the best-performing λ_q for each model. Thirdly, the optimal quantization parameters for each of the four scenarios are presented together in one figure, allowing for a conclusion on the influence of initial model dimensions. Finally, the best-performing compressed models are compared with each other and the random forest model.

5.1 Gate Utilization - PU Dataset

To understand how the decomposition and quantization gates respond to various regularization strengths, the active gate fractions are shown per layer for four settings of λ_d and λ_q. When both regularization terms are weak, the model should have many active gates, and increasing one should affect its corresponding domain. Therefore, a higher λ_d will make sure that the number of parameters is reduced. Moreover, a higher λ_q should force the model size to be more heavily compressed. The results presented in this section aim to demonstrate the difference in outcome when one parameter is changed while the other is kept constant. The results are shown for the compression of the Base model, where the weight matrices are replaced with Tucker structures.

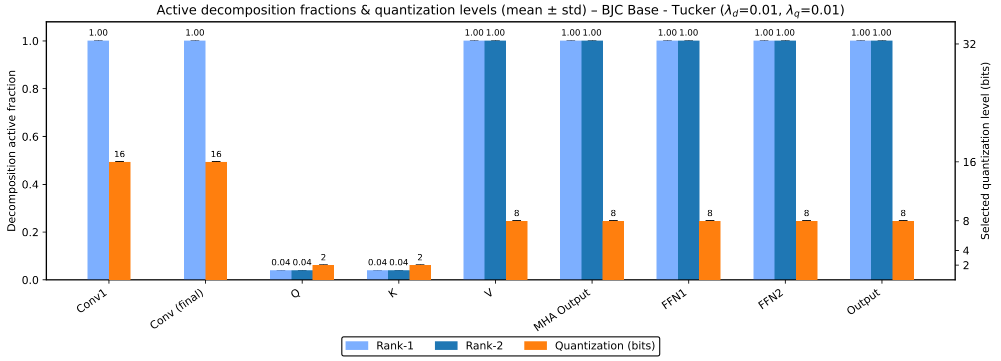


Figure 5.1: Mean fraction of active gates for decomposition and quantization per layer in the case of λ_d and λ_q being low (± standard deviation (std)).

Figure 5.1 shows the Base model after compression using low regularization values. The inner ranks of the full interaction-based decomposition show that almost all layers have almost all their gates turned on. The query and key are the only weight matrices that have been minimized. The gates associated with quantization behave differently. Every layer has been compressed to some degree with quantization. In comparison to decomposition, which has not decreased the parameter count in many layers, the quantization element ensured the model is overall represented in a lower precision.

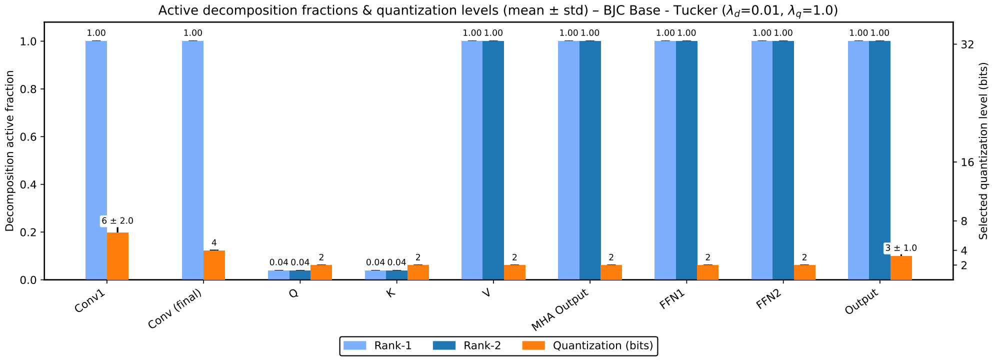


Figure 5.2: Mean fraction of active gates for decomposition and quantization per layer in the case of λ_d being low and λ_q being high (± std).

When the λ_q is increased, Figure 5.2 shows the expected behaviour. The precision levels of the stored weight matrices are significantly decreased. λ_q remains unchanged, and therefore the gates converge to the same highly active configuration as in the previous scenario.

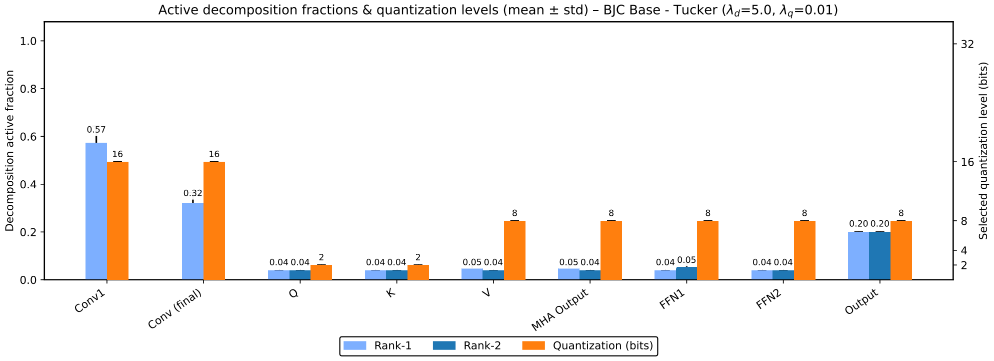


Figure 5.3: Mean fraction of active gates for decomposition and quantization per layer in the case of λ_d being high and λ_q being low (± std).

Then, when the regularization parameter of the factorization is increased, many more gates are deactivated. The layers that contain the most active gates are the convolutional layers and the output layer, as can be seen in Figure 5.3.

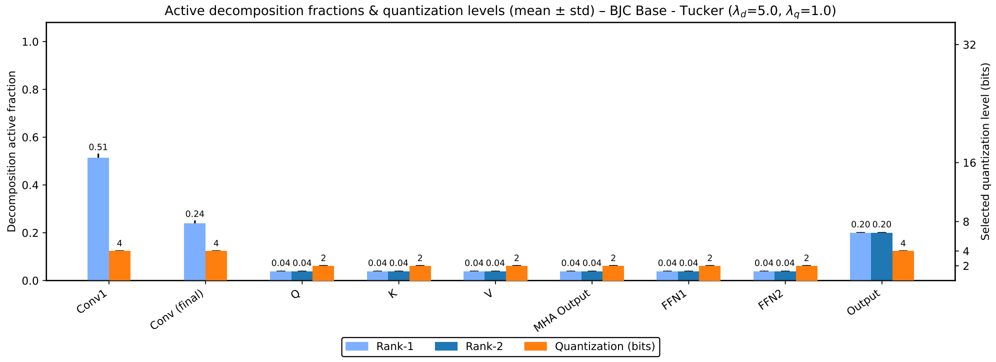


Figure 5.4: Mean fraction of active gates for decomposition and quantization per layer in the case of λ_d and λ_q being high (± std).

Finally, when both regularization parameters are high, the model will be compressed more significantly by sacrificing some accuracy. Figure 5.4 shows this scenario.

These four scenarios have shown the effect of the decomposition and quantization regularization terms. However, Figures 5.1 to 5.4 do not reflect the model's performance. Deactivating almost all gates with high λ_d and λ_q values will not yield the desired accuracy. The following sections will show where the balance lies between accuracy and compression.

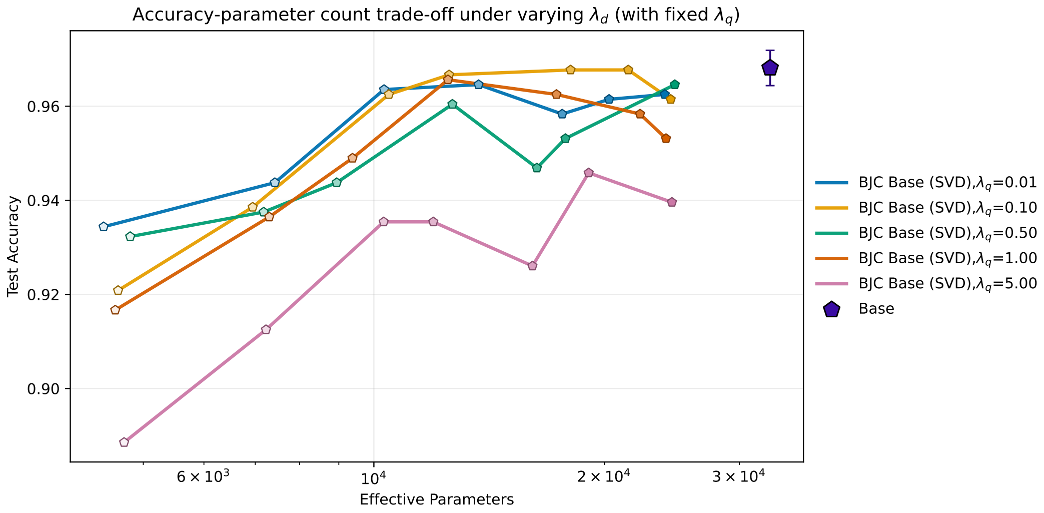
5.2 Compression Sensitivity & Comparison - CWRU Dataset

The experimental results for the CWRU dataset are presented in the Appendix A. Analysing the results will show the same overall characteristics as those obtained on the PU Dataset, and the same conclusions can therefore be drawn. For the sake of clarity and conciseness, the main body of this thesis will only focus on the PU Dataset. This choice is further motivated by the fact that PU poses a more challenging classification problem as mentioned in Section 4.1. As a result, the PU experiments provide a more realistic benchmark and make the impact of compression methods more evident, which is why they are emphasized in the main discussion.

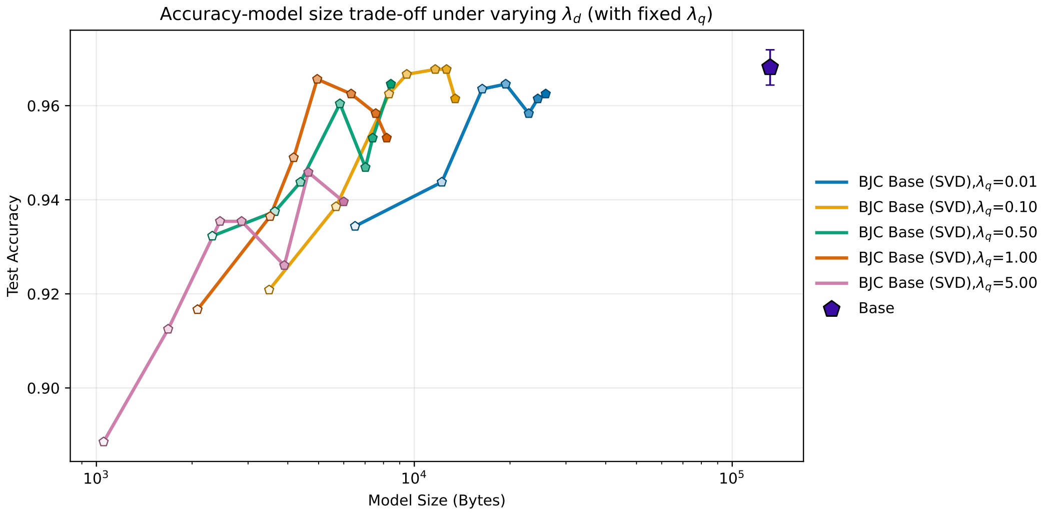
5.3 Compression Sensitivity - PU Dataset

In this section, the effect of the decomposition regularization parameter λ_d on compression is demonstrated, while keeping the quantization regularization parameter λ_q fixed. This is done for multiple λ_q, also to understand the behaviour of dynamic quantization. For each model type, the results show the test accuracy of the compressed models against the number of parameters and model size. Therefore, this section will serve as a sensitivity analysis for every model, showing how changing the λ_d and λ_q influences the accuracy of the model. FLOPs and inference time are not as valuable in this stage and are therefore not discussed in this section.

Figures 5.5a and 5.5b show the results of the compressed Base model for various regularization parameters. For the Figure 5.5a, the first thing that can be noticed is that for the same λ_d, the accuracy is lower for higher λ_q. Secondly, the Base model can be compressed in terms of parameters by almost 3× before a reduction in accuracy. Figure 5.5a shows the model sizes in Bytes, which, as explained in Section 4.3, is the fairest metric for evaluating compression, as it takes into account both the reduction in parameters and the precision level. As can be seen, the higher λ_q, the lower the model size becomes. Another remarkable characteristic is that, to a certain degree, the λ_q can be increased without a significant drop in accuracy. Lastly, the results demonstrate that a 10× reduction in model size is possible without compromising the accuracy of the Base models.

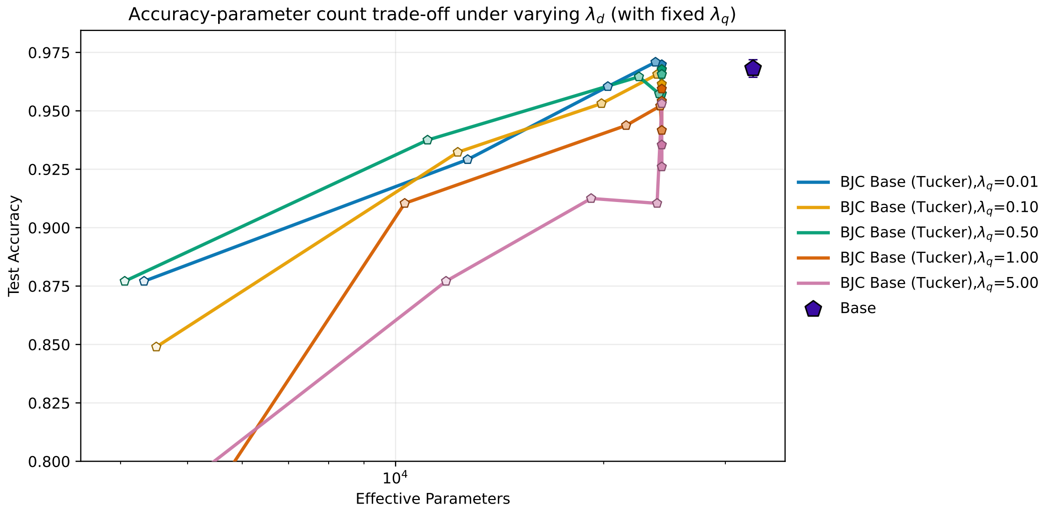


(a) Accuracy vs. Parameters

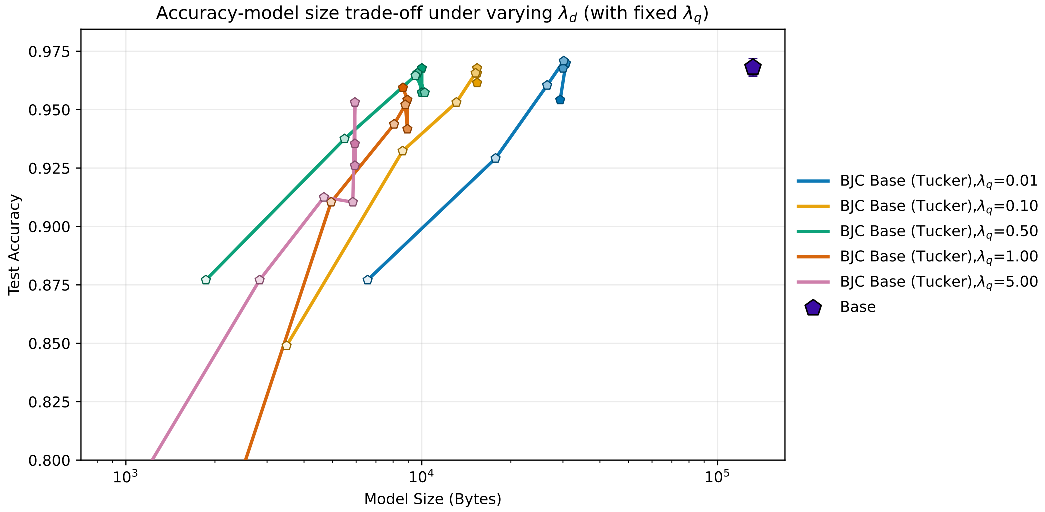


(b) Accuracy vs. Model Size

Figure 5.5: Accuracy trade-offs for the **BJC Base–SVD**. Each curve corresponds to a fixed quantization regularization λ_q (see legend), while the trajectories are obtained by varying the decomposition regularization λ_d over $0.01, 0.05, 0.1, 0.5, 1.0, 2.0, 5.0$.



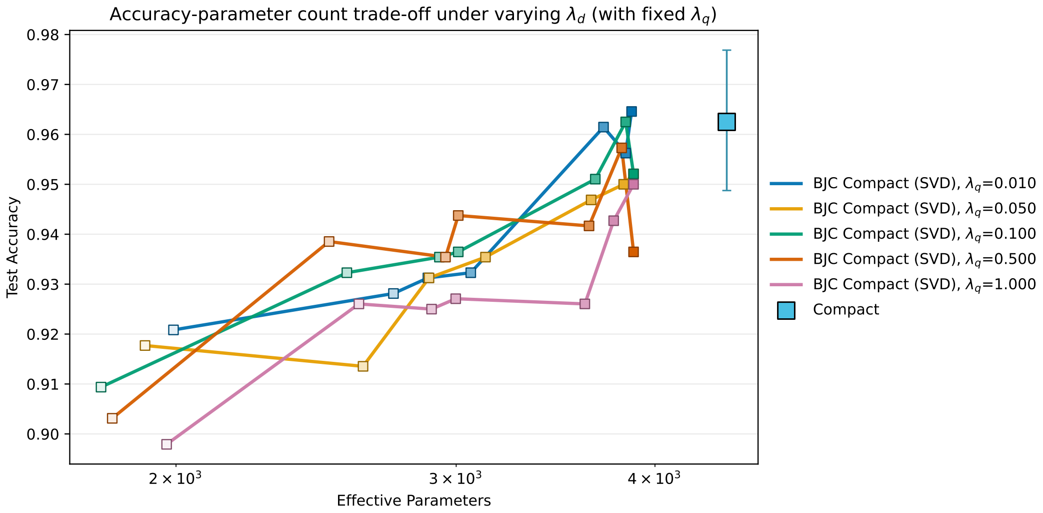
(a) Accuracy vs. Parameters



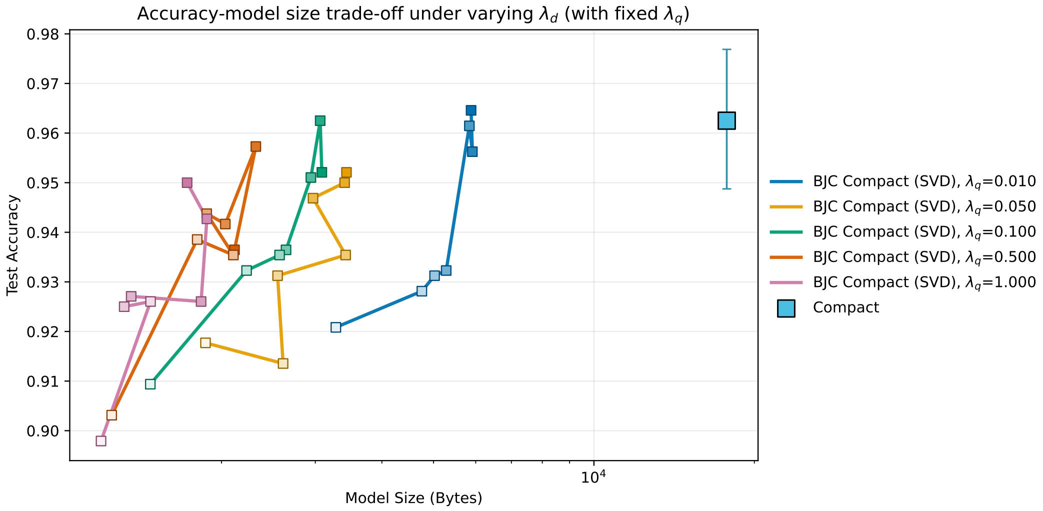
(b) Accuracy vs. Model Size

Figure 5.6: Accuracy trade-offs for the **BJC Base–Tucker**. Each curve corresponds to a fixed quantization regularization λ_q (see legend), while the trajectories are obtained by varying the decomposition regularization λ_d over $0.01, 0.05, 0.1, 0.5, 1.0, 2.0, 5.0$.

When the decomposition is of the Tucker structure, it behaves differently from the SVD structure. Figure 5.6a shows that the compression method does not reduce the number of parameters very effectively. As can be seen from the figure, for many small λ_d, the number of parameters remains unchanged. Then, for a certain higher λ_d, the parameters are reduced, but so is the accuracy. The decomposition proves to be less flexible in reducing, when it should be possible, as seen in the SVD case. Then, Figure 5.6b shows that the quantization strength can be increased without compromising accuracy. Although there is this inflexibility in the reduction of weights, the compression using the Tucker structure can achieve the same accuracy as the Base model.



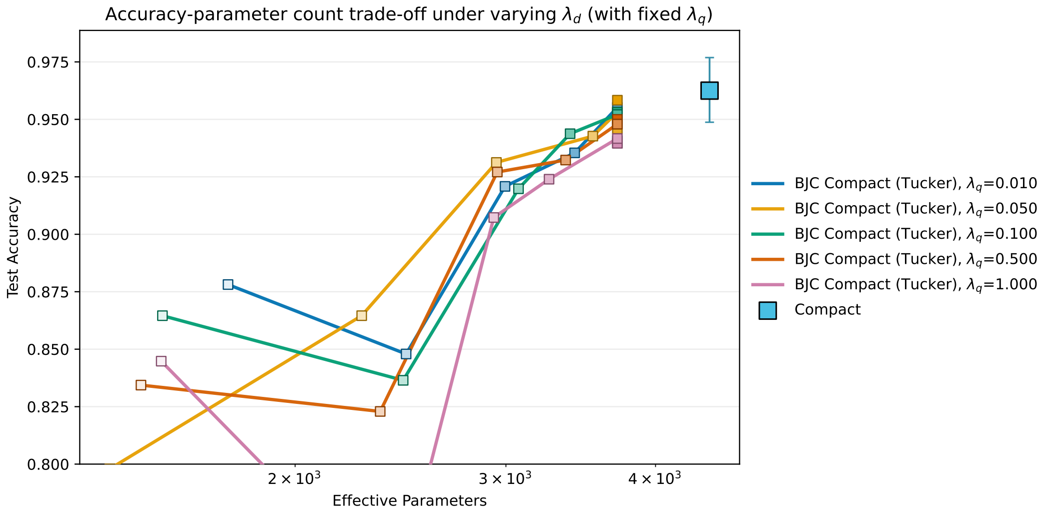
(a) Accuracy vs. Parameters



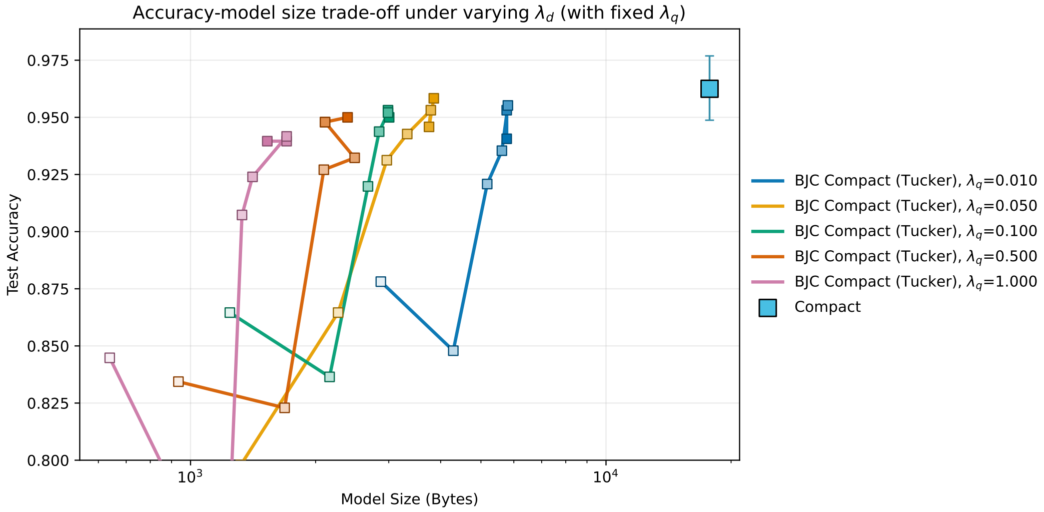
(b) Accuracy vs. Model Size

Figure 5.7: Accuracy trade-offs for the **BJC Compact–SVD**. Each curve corresponds to a fixed quantization regularization λ_q (see legend), while the trajectories are obtained by varying the decomposition regularization λ_d over $0.01, 0.05, 0.1, 0.5, 1.0, 2.0, 5.0$.

The next results are for the compression of the Compact model, shown in Figure 5.7a. The first thing that can be derived is that the number of parameters does not decrease when λ_d is increased. Unlike the BJC Base Tucker case, for the Compact model, it is not unexpected. The model is designed to have performance comparable to the Base model while being as compact as possible. A reduction in parameters would only lead to a degradation in accuracy. The model size, as in previous scenarios Figure 5.7b, shows that for higher λ_q, the model size can further decrease without much accuracy loss. Therefore, the compression does not focus on compression in terms of parameters, but rather on precision.



(a) Accuracy vs. Parameters



(b) Accuracy vs. Model Size

Figure 5.8: Accuracy trade-offs for the **BJC Compact–Tucker**. Each curve corresponds to a fixed quantization regularization λ_q (see legend), while the trajectories are obtained by varying the decomposition regularization λ_d over $0.01, 0.05, 0.1, 0.5, 1.0, 2.0, 5.0$.

Finally, Figure 5.8a and Figure 5.8b present the results for compressing the Compact model. Similar to the results of the compression of the Base model with the Tucker structure, the accuracy changes very drastically when the λ_d becomes very high. As for the number of parameters, an immediate decrease in the accuracy for higher λ_d is not unexpected and is in line with Figure 5.7. The model size again shows that compression under a high quantization regularization parameter yields higher accuracy than under a high decomposition regularization parameter.

5.4 Comparison - PU Dataset

To further analyze the difference between compressing a Base model and a Compact model, the best-performing values of λ_q are chosen and compared. For the SVD structure, the best compression while preserving accuracy is obtained with $\lambda_q = 0.1$. For the Tucker structure, this value is $\lambda_q = 0.5$. These values remain the same for both the Base and Compact models. The corresponding accuracy–compression curves for varying λ_d are shown in Figure 5.9.

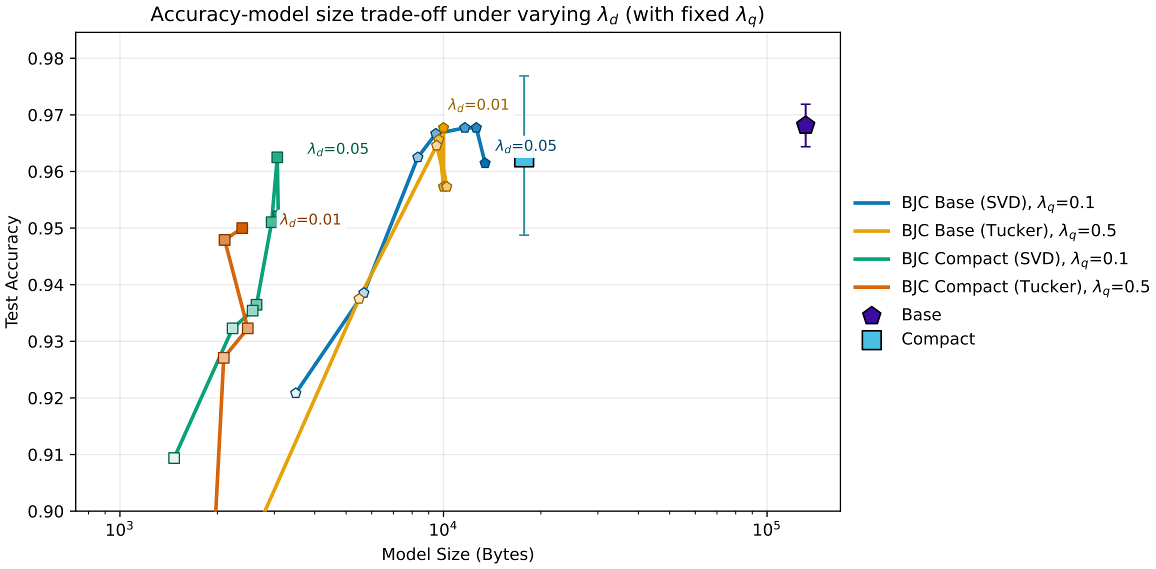


Figure 5.9: Accuracy–model size trade-offs across all configurations (Base–SVD, Base–Tucker, Compact–SVD, Compact–Tucker) under varying λ_d with best found λ_q.

There are several important observations which can be drawn from Figure 5.9. Firstly, the compression of the Base model results in a model that is roughly 2× smaller than the compactly designed model, while maintaining comparable accuracy to the original Base model. This suggests that the compression of the larger Base model can outperform a compact design in terms of both size and accuracy. However, the high dimensionality of the Base model also limits how far it can be compressed without a loss in accuracy. In contrast, when the compression framework is applied to the Compact model, even further compression is possible, extending beyond the compression achieved by the Base model.

In Figure 5.9, the decomposition regularization factor associated with the highest mean accuracy recorded is visualized. To make the results further interpretable, these λ_d associated with the best trade-off between accuracy and model size will only be focused on. With the results more focused displayed, they can be compared to the best-performing shallow machine learning model, the Random Forest model. This is shown in Figure 5.10.

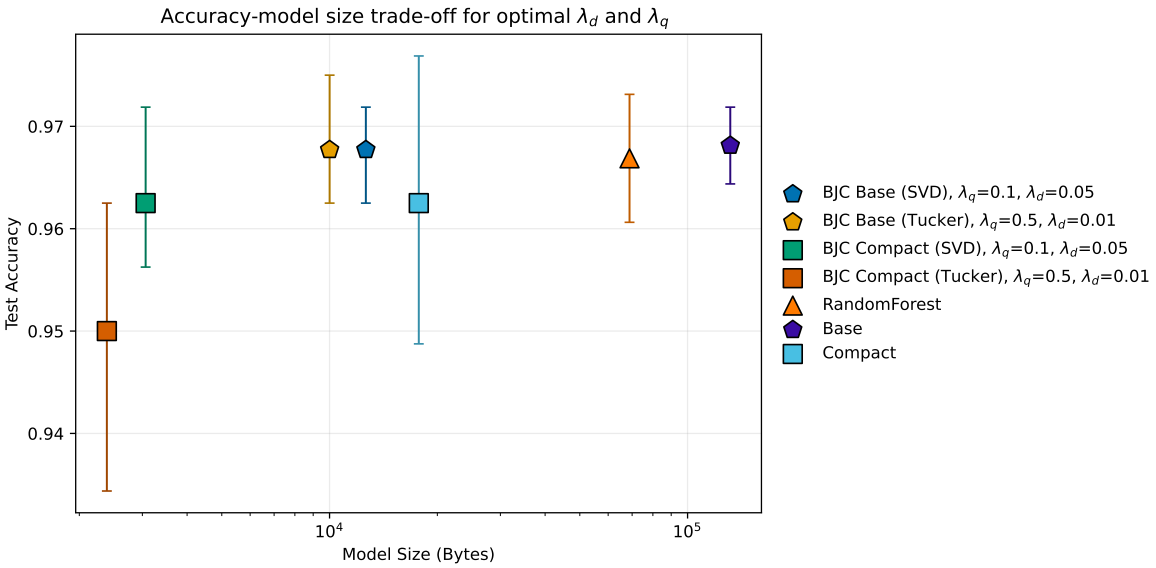


Figure 5.10: Per-configuration results at the selected λ_d: accuracy versus model size. Points are mean ± std across splits.

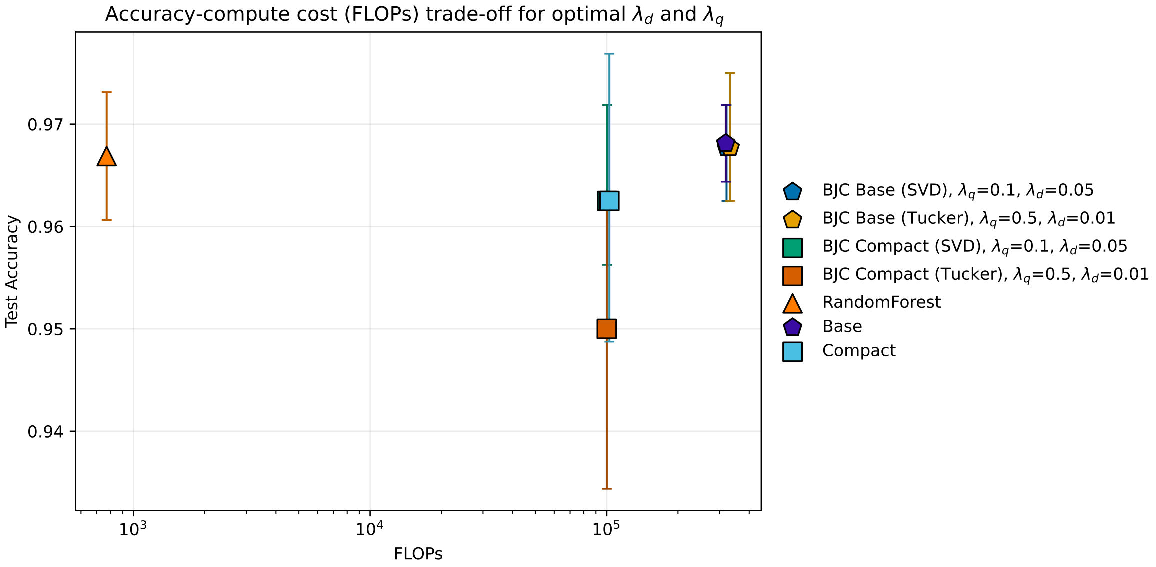


Figure 5.11: Accuracy versus computational cost (FLOPs) at the selected λ_d values for all configurations. Points indicate mean ± std across splits.

Figure 5.11 show the accuracy against floating-point operation counts at the optimal λ_d and λ_q. The proposed framework makes sure that a lower bound is imposed on the number of parameters to ensure that the memory needs do not exceed those of the original model. The number of parameters also mainly decides how many operations the model needs for one forward pass. Therefore, the upper bound also confirms that the number of operations is not higher than that of the Base models. The figure demonstrates that the raw computation count does not grow, but the internal complexity of the model does increase due to the decomposition into additional sublayers.

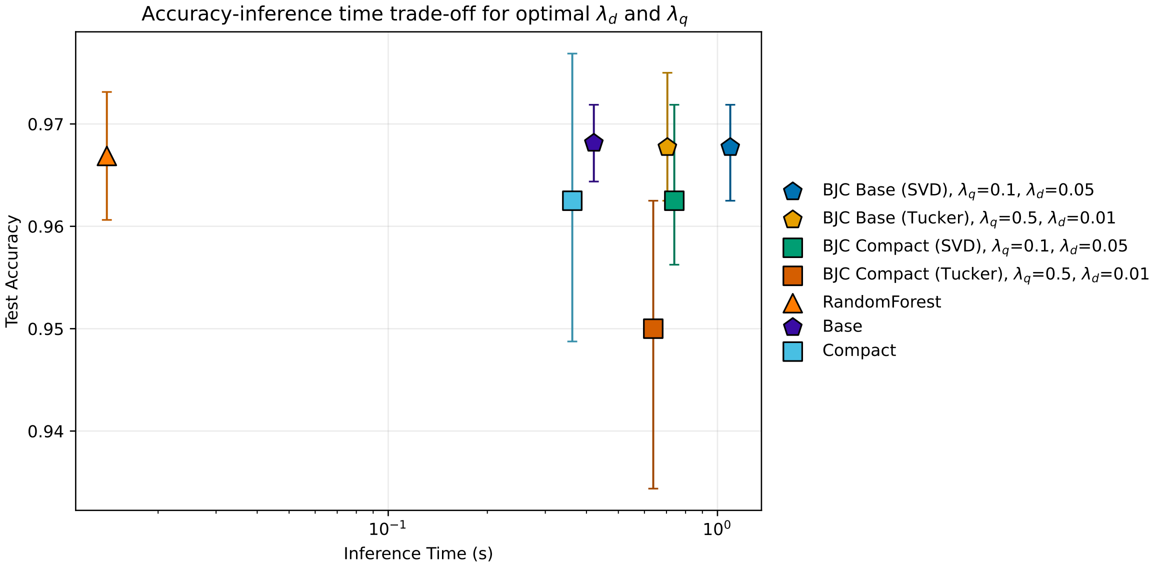


Figure 5.12: Per-configuration results at the selected λ_d: accuracy versus inference time (latency). Points are mean ± std across splits.

The increased depth of the decomposed networks has a direct effect on inference latency. As shown in Figure 5.12, the compressed model generally requires a longer time to perform classification than their uncompressed version. It can be concluded that the compressed model takes longer to produce an output. Interestingly, Tucker-based models seem to have a lower latency compared to the SVD-based ones. However, this increase in latency is, in absolute terms, not very significant and would not affect practical deployment.

The main results of the shown experiments are summarized in Tables 5.1 and 5.2. The table compares baseline models with their compressed counterparts under different decomposition and quantization settings. In this chapter, the best-performing regularization factors that preserved the model accuracy were highlighted. However, to enable a fair comparison between the Base and Compact architectures, an additional compressed Base model is added that matches the accuracy of the Compact model while achieving the highest degree of compression.

Table 5.1: Comparison of models in terms of test accuracy, parameter count, model size, and compression ratios relative to their original model (Base or Compact). CR-Params = compression ratio on parameters; CR-Size = compression ratio on model size. The compression models their regularization parameters are defined as (λ_q,λ_d).

Model Type	**Test Accuracy**	**Parameters**	**Model Size** (Bytes)	**CR-Params**	**CR-Size**
Random Forest	0.967 ± 0.007	17,211 ± 5,164	68,846 ± 20,656	–	–
Base	0.968 ± 0.004	32,902 ± 0	131,608 ± 0	-	-
Compact	0.962 ± 0.017	4,438 ± 0	17,752 ± 0	-	-
BJC Base SVD (0.1,0.05)	0.968 ± 0.004	21,481 ± 304	12,629 ± 55	1.53	10.42
BJC Base SVD (0.5,1.0)	0.966 ± 0.014	12,494 ± 182	4,951 ± 291	2.63	26.58
BJC Base Tucker (0.5,0.01)	0.968 ± 0.005	24,273 ± 0	10,004 ± 472	1.36	13.16
BJC Compact SVD (0.1,0.05)	0.963 ± 0.007	3,834 ± 31	3,064 ± 16	1.16	5.79
BJC Compact Tucker (0.5,0.01)	0.950 ± 0.012	3,717 ± 0	2,388 ± 316	1.19	7.43

Table 5.2: Comparison of models in terms of test accuracy, FLOPs, and inference time. The compression models their regularization parameters are defined as (λ_q,λ_d).

Model Type	**Test Accuracy**	**FLOPs**	**Inference Time (s)**
Random Forest	0.967 ± 0.007	770.400 ± 216.548	0.014 ± 0.004
Base	0.968 ± 0.004	318,420 ± 0	0.421 ± 0.098
Compact	0.962 ± 0.017	102,612 ± 0	0.362 ± 0.017
BJC Base SVD (0.1,0.05)	0.968 ± 0.004	320,818 ± 1,214	1.094 ± 0.156
BJC Base SVD (0.5,1.0)	0.966 ± 0.014	284,940 ± 728	0.973 ± 0.042
BJC Base Tucker (0.5,0.01)	0.968 ± 0.005	332,096 ± 0	0.704 ± 0.004
BJC Compact SVD (0.1,0.05)	0.963 ± 0.007	100,463 ± 124	0.739 ± 0.052
BJC Compact Tucker (0.5,0.01)	0.950 ± 0.012	100,032 ± 0	0.638 ± 0.006

Discussion

6

Chapter 5 has shown the compression of many different model configurations. This chapter presents, interprets, and compares the main findings, while also explaining the trade-offs, limitations, and future work.

Table 5.1 shows the key results for the PU Dataset. The first thing that can be derived is that the compression of the Base and Compact models behaves differently. The Base model can be aggressively compressed in both parameters and precision, where the Compact model is mainly compressed in terms of quantization. When compressing the Base model, there is the possibility for a factor of 2 reduction in the parameters without affecting the accuracy. In combination with quantization, the compression of the Base model, aimed at maintaining accuracy, results in a 10 to 13 times reduction in model size. The results for the compression of the Compact model, however, show that the number of parameters cannot be reduced without sacrificing performance. This outcome is most definitely caused by the model being already optimized in terms of parameters. The Compact models exhibit a parameter reduction of approximately 1.2×, which is most likely due to the upper bound imposed on the inner rank. The main compression of Compact models is therefore primarily dependent on quantization, resulting in a compression ratio of 5.8 to 7.4.

The Base model has a very large architecture, but when compressed, it can become relatively small. Table 5.1 showed that when the Base model is combined with the correct high regularization parameters, the model can match the compression of a Compact model. However, with this accuracy, a higher standard deviation is coupled with it, making the compression effectiveness more uncertain. This limitation is not unexpected, as the layers of their inner ranks must be lower than those of a Compact compressed model. Therefore, when the goal is aggressive compression, it would be more effective to compress a model that is already small. One more important observation from Table 5.1 is that the compression of Compact models returns a more robust model with the possibility of a slightly higher accuracy. Due to the decomposition, the model can capture the patterns of the data better, yielding a higher accuracy.

Another important aspect of this thesis concerns the decomposition method. When the BJC utilizes the SVD structure to replace the weight matrices, the compression effectively reduces the parameters without compromising accuracy, as shown in Figure 5.5a. The results generated for the Tucker structure decomposition, as shown in Figure 5.6a, do not appear to offer a benefit in reducing the inner ranks. However, the models compressed using the Tucker structure are about 20% smaller in model size, while achieving the same accuracy. Thus, the Tucker structure enables the model to utilize lower precision levels for its layers. Therefore, keeping the parameter count high in combination with a complex structure, such as Tucker, the layers can be quantized even further. This phenomenon is not expected, due to the initial belief that the high level of interactions would make it possible for the model to be compressed further. Consequently, another strong possibility is that the two-dimensional gate control is too complex for the model to make use of effectively, as there are more gates to learn. This makes the model more focused through quantization.

FLOPs and inference time are metrics that one would expect to behave similarly, decreasing when the number of parameters is reduced. However, this is not seen in the results shown in Figures 5.11 and 5.12, and Table 5.1. Although the FLOPs indeed decrease when the parameters are reduced and are not increased with the compression framework, the inference time is doubled. The fact that the forward pass takes longer is caused by the increased number of steps within a layer and the extra computational overhead required for recomputing the quantized matrices back to their full precision. As explained in Section 3.2, quantization acts as a reducer in the footprint of parameter storage; however, when the weight matrices are needed, they are computed to full precision.

Then, the results have shown that the used deep learning structure slightly outperforms the Random Forest model in terms of accuracy (see Figure 5.10). Without any compression, the Base model is about twice as large as the RF model. When compressing the Base model, the BJC maintains accuracy while making the model significantly smaller, outperforming the shallow model in both accuracy and model size. Although joint compression can lead to a model with fewer parameters and a smaller size, its computational efficiency cannot match that of the Random Forest model. This, in combination with the matching accuracy, makes the reasoning for the deep learning approach for this use case more questionable.

Having outlined the key findings, the BJC method used to generate the results has some remarks of its own that are not directly visible. First of all, the choice of the regularization factors is an essential step in finding an effective compressed model. The figures shown in Chapter 5 demonstrate that selecting the ‘right’ values requires an extensive grid search. On the other hand, this process does not require a lot of human intervention, as the only input choice is the model itself. Second, the decomposition method is defined by gates that are independently connected to each other. In the original work (Bayesian LoRa), these gates were dependent on each other, meaning that if the earlier component were deactivated, the subsequent gates would also be deactivated. The same principle is used to control the quantization level. This change would put the focus more on model compression, which is desirable for the sake of aggressive reduction in model size. However, this nature is logical for quantization, but for decomposition, exploration is essential. Through hierarchical gates, the decomposition is limited during training, which does not guarantee that the compressed model will maintain the desired accuracy. However, as was seen in the results, the compression of the model was mainly caused by the quantization. Therefore, it might have been the better choice for maintaining accuracy and simplicity, but it did most likely not benefit compression. This could also be extended to the CNN layers.

Lastly, some choices were made for the model initialization, which could have influenced the results. The BJC first defines the model architecture by placing upper bounds on the inner rank of the factorizations. However, it ensures that the compressed model is always smaller than the original model, even when λ_d and λ_q are close to zero. However, this procedure assumes that the initial layer dimensions are already well chosen. This assumption holds when the Base model is relatively large, since most layers then provide sufficient capacity. But, for cases where the optimal architecture would require expanding certain layers while aggressively pruning others, this fixed upper-bound initialization may no longer be ideal, as it restricts the model's flexibility to adapt its structure in a fully data-driven manner.

Secondly, the model structure also depends on the initialization of the gates. In the chosen setup, the gates are initialized to be activated. This allows the model to explore many possible configurations during training, and then to gradually compress itself through the loss function, which encourages sparsity. Although this strategy encourages preserving higher accuracy, it places the entire responsibility for compression on the loss function and the training parameters. If these are not set to optimal values, the model will stay either too large or be compressed too heavily. Choosing a different strategy could provide a better balance.

During this work, some doubts arose regarding the optimal data splitting strategy. Since the datasets are created by measuring different bearings, a split based on bearing IDs would more realistically reflect real-world deployment. As in real-world situations, models are expected to generalize to unseen machines. However, most studies split the dataset through random sampling. This can cause data leakage, as train and test sets most likely contain samples from the same bearing, which present the same behaviour. With self-conducted experiments, the Base model achieved an accuracy of around 60%. Further exploration of this is left for future work. On the other hand, the main goal of this thesis was to evaluate the proposed compression method and demonstrate that accuracy can be largely preserved under compression. Therefore, this optimistic accuracy does not undermine the contribution of this work.

7

Conclusion

Edge AI refers to the practice of deploying AI algorithms directly on local devices that are close to the data source. While the application may differ, the devices themselves are typically resource-constrained in terms of memory and computational power. To still be able to use deep learning on such devices, models must therefore be compressed and optimized to fit within the hardware limitations, while still preserving as much predictive performance as possible. Standard solutions for Edge AI, however, are typically static and depend heavily on manual intervention.

This limitation has highlighted the need for an end-to-end adaptive framework that can integrate multiple compression methods and automatically adjust the model to meet hardware and application requirements, which this thesis addresses by introducing the Bayesian Joint Compression (BJC) framework. BJC adaptively combines decomposition and quantization into a single model. By using Bayesian gate mechanisms, BJC automatically determines during training which components to prune and which precision levels to apply, thus effectively balancing accuracy and efficiency without requiring extensive manual intervention.

The results in Chapter 5 demonstrate that BJC can effectively compress the model size while maintaining accuracy, even improving its robustness, and without increasing the computational load. BJC can adeptively control the effective rank via λ_d and the bit-precision via λ_q. With higher values for the regularization parameters λ_d and λ_q, which control the pruning strengths of decomposition and quantization, the model can be aggressively compressed, with a small, almost negligible reduction in accuracy. Moreover, the experiments confirm that while quantization is the dominant driver of compression, decomposition provides the flexibility to compress the model further when needed. Together, these findings directly address the main research question of this thesis introduced in Section 1.2. By combining quantization and decomposition within an adaptive Bayesian framework, BJC can achieve significant reductions in parameter count, memory size, and FLOPs, while preserving classification performance. These results indicate that end-to-end, joint compression strategies represent a promising direction for enabling the practical deployment of deep learning on edge devices.

Additionally, the sub-research questions stated in Section 1.2 can also be addressed. First, BJC enables compression by adaptively balancing and tuning the regularization parameters. Low values preserve model capacity, while higher values lead to aggressive compression with only a slight reduction in accuracy. Secondly, although this work did not integrate any direct hardware constraints, the definition of the loss function allows such an extension. Constraints such as maximum model size can be relatively easily integrated as an additional term in the loss function. As stated in the Chapter 6, this would allow compression to be directly guided by device-specific requirements. The third sub-question covers whether the compression method is a good alternative to a simple machine learning model, standard for constrained devices. It does outperform shallow machine learning methods, such as Random Forest, in terms of both accuracy and compact model size. Although the Random Forest is superior in the number of computations of the forward pass, the BJC is a much better choice in all other metrics. Lastly, the results highlighted that quantization was the most important for thorough compression. Furthermore, decomposition is required for a more aggressive compression or when the model itself is not compact. Therefore, joint optimization of both methods results in a better compression-accuracy trade-off. To further extend on the research question, this work has shown that the initial dimensions of the model influence the accuracy and model size trade-off. Although a compact model may not reach the test accuracy of a larger one, it can achieve a significantly smaller size while maintaining higher accuracy under the same size constraint as the larger model.

This chapter is concluded by outlining several possible directions for future research:

- Firstly, an important extension would be to explore the effects of using a hierarchical gate structure for the decomposition component, rather than relying solely on independent gating mechanisms. As discussed in Chapter 6, hierarchical gates could enforce more aggressive compression, but they may also risk limiting the model's ability to preserve accuracy. Investigating this balance would provide insights into whether hierarchical decompositions offer a practical advantage in BJC-based compression.
- Second, it would be valuable to evaluate the method on a broader range of benchmark datasets commonly used in the compression literature. Most studies on neural network compression rely on large-scale datasets, whereas the PU and CWRU Datasets are domain-specific to bearing fault detection. Extending the evaluation to more widely adopted benchmarks would make it easier to compare BJC directly with other state-of-the-art compression methods.
- Thirdly, in the proposed BJC framework, the weight matrices have to be recomputed at every forward pass, as they are stored in a lower precision level. However, it does not exploit the fact that many edge devices support low-bit computation. This would not only improve efficiency but also reduce latency.
- Lastly, the BJC framework is designed as an end-to-end approach without the need for pre-training. However, to further improve the accuracy and possibly the compression, it could be advantageous to use a pre-trained model as the input of the framework to have a better starting point. Moreover, for the application of Edge AI, it would be interesting and valuable to develop a tool that automatically converts an existing model into its compressed version. Such a tool would simplify deployment in real-world scenarios by compressing and fine-tuning models for edge devices without requiring extensive retraining or manual adjustments.

In conclusion, this work offered a solid groundwork for future research. The findings of adaptive BJC demonstrated that a deep learning model can be compressed effectively in terms of memory footprint, without increasing its computational costs, while preserving accuracy more robustness. This framework, therefore, opens promising directions for practical compression and deployment of models on resource-constrained edge devices across a wide range of tasks.

Bibliography

[1] S. Minaee, T. Mikolov, N. Nikzad, M. A. Chenaghlu, R. Socher, X. Amatriain, and J. Gao, “Large language models: A survey,” *ArXiv*, vol. abs/2402.06196, 2024.

[2] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, “Attention is all you need,” 2023.

[3] R. Singh and S. S. Gill, “Edge ai: A survey,” *Internet of Things and Cyber-Physical Systems*, vol. 3, p. 71–92, 2023.

[4] S. Jaber, J. Soldatos, and R. Rao, “2024 state of edge ai report: Exploring the dynamic world of edge ai applications across industries,” tech. rep., DATEurope in collaboration with tinyML Foundation, May 2024. Accessed: 2025-07-09.

[5] World Bank, “Manufacturing, value added (Accessed: 2025-07-09.

[6] D. S. Thomas and B. A. Weiss, “Economics of manufacturing machinery maintenance: A survey and analysis of u.s. costs and benefits,” Tech. Rep. 100-34, National Institute of Standards and Technology, Gaithersburg, MD, June 2020. Accessed August 21, 2025.

[7] T. P. Carvalho, F. A. A. M. N. Soares, R. Vita, R. da P. Francisco, J. P. Basto, and S. G. S. Alcalá, “A systematic literature review of machine learning methods applied to predictive maintenance,” *Computers & Industrial Engineering*, vol. 137, p. 106024, 2019.

[8] W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu, “Edge computing: Vision and challenges,” *IEEE Internet of Things Journal*, vol. 3, no. 5, pp. 637–646, 2016.

[9] Z. Zhu, Y. Lei, G. Qi, Y. Chai, N. Mazur, Y. An, and X. Huang, “A review of the application of deep learning in intelligent fault diagnosis of rotating machinery,” *Measurement*, vol. 206, p. 112346, Jan 2023.

[10] O. Surucu, S. A. Gadsden, and J. Yawney, “Condition monitoring using machine learning: A review of theory, applications, and recent advances,” *Expert Systems with Applications*, vol. 221, p. 119738, Jul 2023.

[11] E. Memmel, C. Menzen, J. Schuurmans, F. Wesel, and K. Batselier, “Position: Tensor networks are a valuable asset for green ai,” 2024.

[12] E. J. Hu, Y. Shen, P. Wallis, Z. Allen-Zhu, Y. Li, S. Wang, L. Wang, and W. Chen, “Lora: Low-rank adaptation of large language models,” 2021.

[13] L. Deng, G. Li, S. Han, L. Shi, and Y. Xie, “Model compression and hardware acceleration for neural networks: A comprehensive survey,” *Proceedings of the IEEE*, vol. 108, no. 4, pp. 485–532, 2020.

[14] Y. Tang, Y. Wang, J. Guo, Z. Tu, K. Han, H. Hu, and D. Tao, "A survey on transformer compression," 2024.

[15] A. Moslemi, A. Briskina, Z. Dang, and J. Li, "A survey on knowledge distillation: Recent advancements," *Machine Learning with Applications*, vol. 18, p. 100605, 2024.

[16] J. Gou, B. Yu, S. J. Maybank, and D. Tao, "Knowledge distillation: A survey," *International Journal of Computer Vision*, vol. 129, p. 1789–1819, Mar 2021.

[17] G. Hinton, O. Vinyals, and J. Dean, "Distilling the knowledge in a neural network," 2015.

[18] J. H. Cho and B. Hariharan, "On the efficacy of knowledge distillation," *CoRR*, vol. abs/1910.01348, 2019.

[19] A. Gholami, S. Kim, Z. Dong, Z. Yao, M. W. Mahoney, and K. Keutzer, "A survey of quantization methods for efficient neural network inference," 2021.

[20] L. Deng, G. Li, S. Han, L. Shi, and Y. Xie, "Model compression and hardware acceleration for neural networks: A comprehensive survey," *Proceedings of the IEEE*, vol. 108, no. 4, pp. 485–532, 2020.

[21] K. Wang, Z. Liu, Y. Lin, J. Lin, and S. Han, "Haq: Hardware-aware automated quantization with mixed precision," 2019.

[22] T. G. Kolda and B. W. Bader, "Tensor decompositions and applications," *SIAM Review*, vol. 51, no. 3, pp. 455–500, 2009.

[23] A. Cichocki, N. Lee, I. Oseledets, A.-H. Phan, Q. Zhao, and D. P. Mandic, "Tensor networks for dimensionality reduction and large-scale optimization: Part 1 low-rank tensor decompositions," *Foundations and Trends® in Machine Learning*, vol. 9, no. 4–5, p. 249–429, 2016.

[24] C. Yang and H. Liu, "Stable low-rank cp decomposition for compression of convolutional neural networks based on sensitivity," *Applied Sciences*, vol. 14, no. 4, 2024.

[25] M. Gabor and R. Zdunek, "Compressing convolutional neural networks with hierarchical tucker-2 decomposition," *Applied Soft Computing*, vol. 132, p. 109856, 2023.

[26] M. Gabor and R. Zdunek, "Convolutional neural network compression via tensor-train decomposition on permuted weight tensor with automatic rank determination," in *Computational Science – ICCS 2022* (D. Groen, C. de Mulatier, M. Paszynski, V. V. Krzhizhanovskaya, J. J. Dongarra, and P. M. A. Sloot, eds.), (Cham), pp. 654–667, Springer International Publishing, 2022.

[27] M. Yin, Y. Sui, S. Liao, and B. Yuan, "Towards efficient tensor decomposition-based dnn model compression with optimization framework," *2021 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, p. 10669–10678, Jun 2021.

[28] M. Gabor and R. Zdunek, "Compressing convolutional neural networks with hierarchical tucker-2 decomposition," *Applied Soft Computing*, vol. 132, p. 109856, Jan 2023.

[29] O. A. Ademola, P. Eduard, and L. Mairo, “Ensemble of tensor train decomposition and quantization methods for deep learning model compression,” *2022 International Joint Conference on Neural Networks (IJCNN)*, p. 1–6, Jul 2022.

[30] M. Alnemari and N. Bagherzadeh, “Ultimate compression: Joint method of quantization and tensor decomposition for compact models on the edge,” *Applied Sciences*, vol. 14, no. 20, 2024.

[31] D. Blalock, J. J. G. Ortiz, J. Frankle, and J. Guttag, “What is the state of neural network pruning?,” 2020.

[32] Y. He and L. Xiao, “Structured pruning for deep convolutional neural networks: A survey,” *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 46, p. 2900–2919, May 2024.

[33] M. van Baalen, C. Louizos, M. Nagel, R. A. Amjad, Y. Wang, T. Blankevoort, and M. Welling, “Bayesian bits: Unifying quantization and pruning,” 2020.

[34] C. Meo, K. Sycheva, A. Goyal, and J. Dauwels, “Bayesian-loRA: LoRA based parameter efficient fine-tuning using optimal quantization levels and rank values trough differentiable bayesian gates,” in *2nd Workshop on Advancing Neural Network Training: Computational Efficiency, Scalability, and Resource Optimization (WANT@ICML 2024)*, 2024.

[35] S. Han, H. Mao, and W. J. Dally, “Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding,” 2016.

[36] N. Setyawan, C.-C. Sun, M.-H. Hsu, W.-K. Kuo, and J.-W. Hsieh, “Microvit: A vision transformer with low complexity self attention for edge device,” in *2025 IEEE International Symposium on Circuits and Systems (ISCAS)*, p. 1–5, IEEE, May 2025.

[37] S.-K. Yeom and T.-H. Kim, “Uniform: A reuse attention mechanism optimized for efficient vision transformers on edge devices,” 2024.

[38] G. James, D. Witten, T. Hastie, and R. Tibshirani, *An Introduction to Statistical Learning: with Applications in R*. Springer Texts in Statistics, Springer, 2013.

[39] A. Géron, *Hands-On Machine Learning with Scikit-Learn, Keras & TensorFlow: Concepts, Tools, and Techniques to Build Intelligent Systems*. Sebastopol, CA: O’Reilly Media, 2nd ed., 2019.

[40] M. Fernandez-Delgado, E. Cernadas, S. Barro, and D. Amorim, “Do we need hundreds of classifiers to solve real world classification problems?,” *Journal of Machine Learning Research*, vol. 15, pp. 3133–3181, 10 2014.

[41] Q. Zhang, M. Chen, A. Bukharin, N. Karampatziakis, P. He, Y. Cheng, W. Chen, and T. Zhao, “Adalora: Adaptive budget allocation for parameter-efficient fine-tuning,” 2023.

[42] C. Lessmeier, J. K. Kimotho, D. Zimmer, and W. Sextro, “Condition monitoring of bearing damage in electromechanical drive systems by using motor current signals of electric motors: A benchmark data set for data-driven classification,” *PHM Society European Conference*, vol. 3, Jul 2016.

[43] C. W. R. University, “Case western reserve university bearing data center.” `https://engineering.case.edu/bearingdatacenter`, 1997.

[44] D. Neupane and J. Seok, “Bearing fault detection and diagnosis using case western reserve university dataset with deep learning approaches: A review,” *IEEE Access*, vol. 8, pp. 93155–93178, 2020.

[45] Z. Zhu, Y. Lei, G. Qi, Y. Chai, N. Mazur, Y. An, and X. Huang, “A review of the application of deep learning in intelligent fault diagnosis of rotating machinery,” *Measurement*, vol. 206, p. 112346, 2023.

[46] T. Saghi, D. Bustan, and S. S. Aphale, “Bearing fault diagnosis based on multi-scale cnn and bidirectional gru,” *Vibration*, vol. 6, no. 1, pp. 11–28, 2023.

[47] B. Zhang, F. Li, N. Ma, W. Ji, and S.-K. Ng, “Open set bearing fault diagnosis with domain adaptive adversarial network under varying conditions,” *Actuators*, vol. 13, no. 4, 2024.

[48] Y. Chen, M. Rao, K. Feng, and M. J. Zuo, “Physics-informed lstm hyperparameters selection for gearbox fault detection,” *Mechanical Systems and Signal Processing*, vol. 171, p. 108907, 2022.

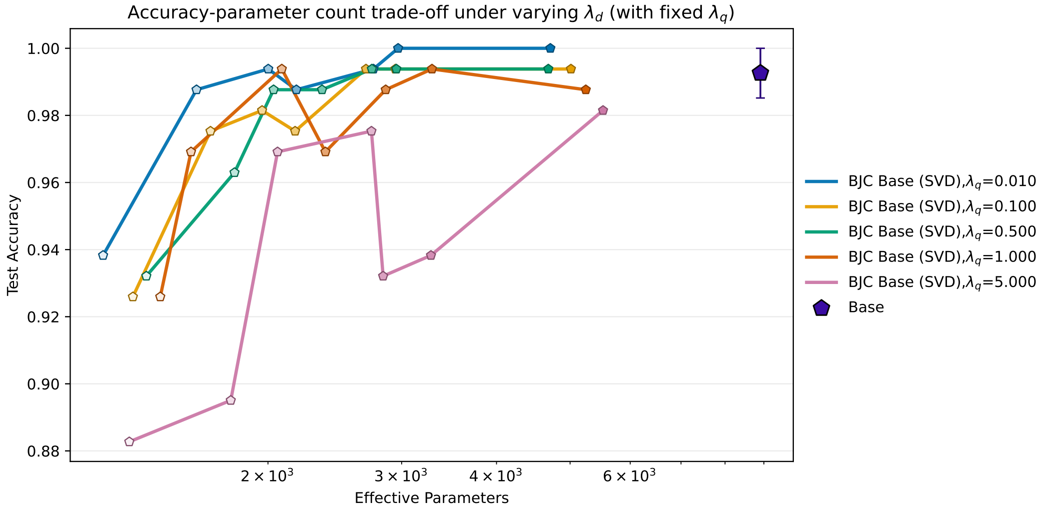
[49] C. Grover and N. Turk, “Optimal statistical feature subset selection for bearing fault detection and severity estimation,” *Shock and Vibration*, vol. 2020, no. 1, p. 5742053, 2020.

[50] D.-T. Hoang and H.-J. Kang, “A survey on deep learning based bearing fault diagnosis,” *Neurocomputing*, vol. 335, pp. 327–335, 2019.

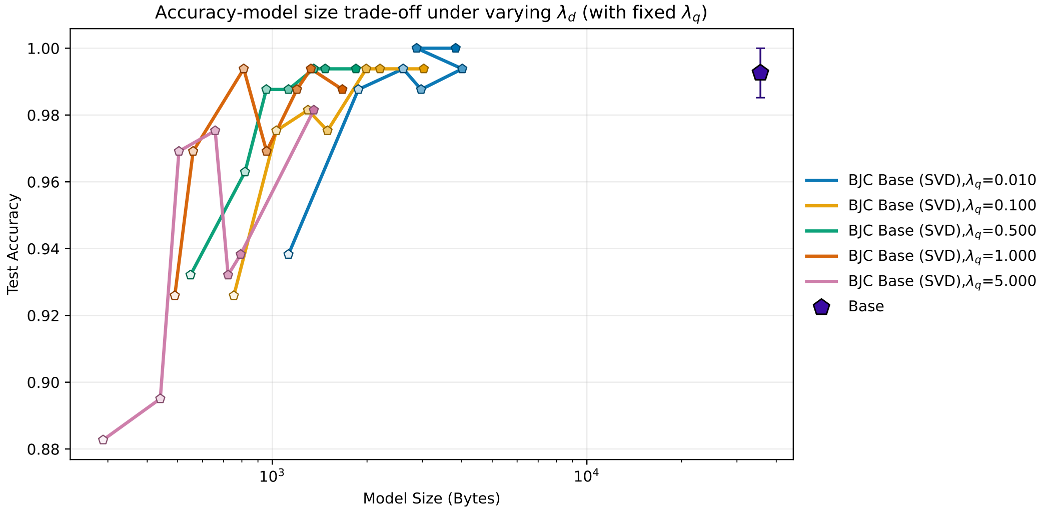
A

Additional Dataset Results

A.1 Compression Sensitivity - CWRU Dataset

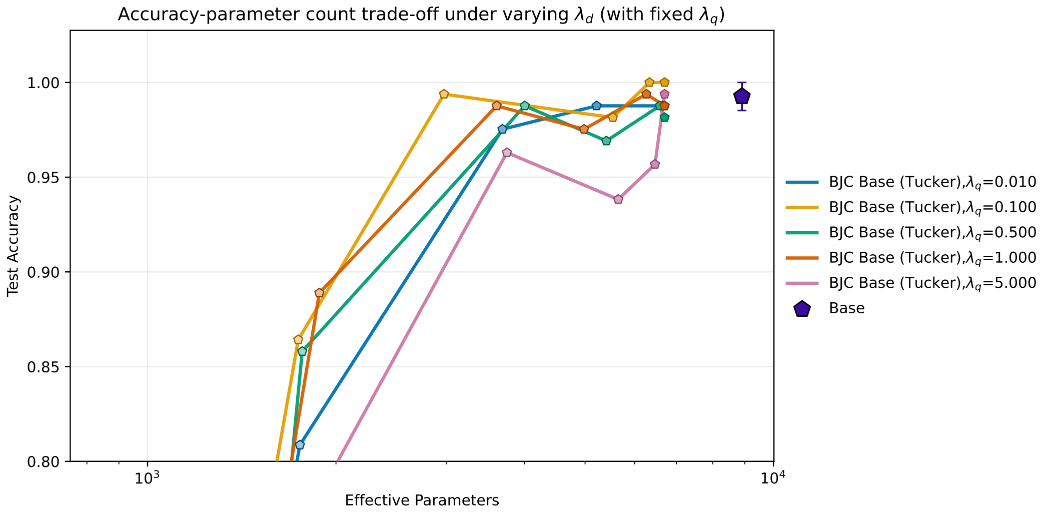


(a) Accuracy vs. Parameters

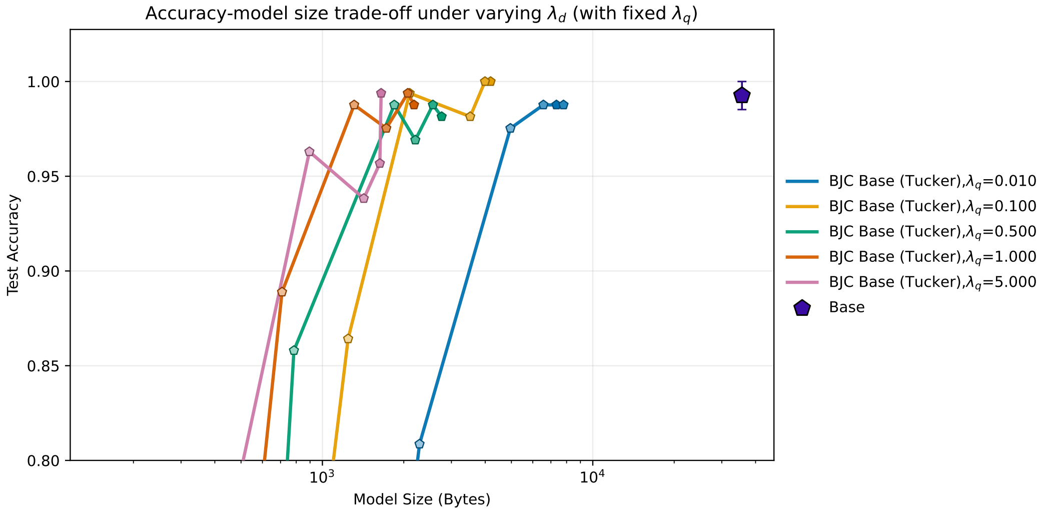


(b) Accuracy vs. Model Size

Figure A.1: Accuracy trade-offs for the **CWRU Base–SVD**. Each curve corresponds to a fixed quantization regularization λ_q, while trajectories are obtained by varying the decomposition regularization λ_d.



(a) Accuracy vs. Parameters



(b) Accuracy vs. Model Size

Figure A.2: Accuracy trade-offs for the **CWRU Base–Tucker**. Each curve corresponds to a fixed quantization regularization λ_q, while trajectories are obtained by varying the decomposition regularization λ_d.

A.2 Comparison - CWRU Dataset

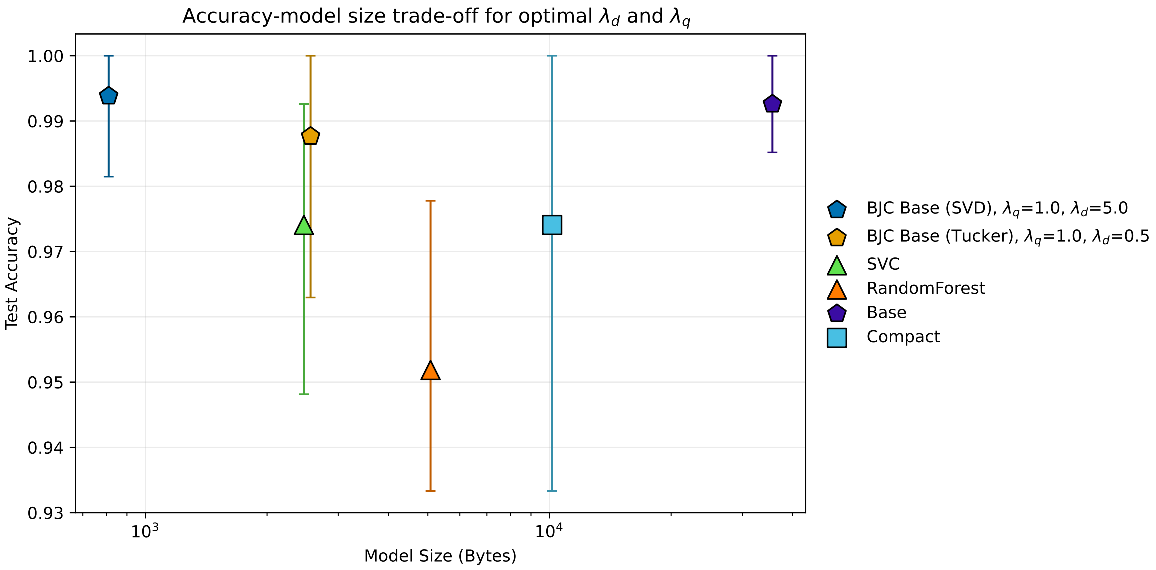


Figure A.3: Per-configuration results at the selected λ_d: accuracy versus model size. Points are mean ± std across splits.

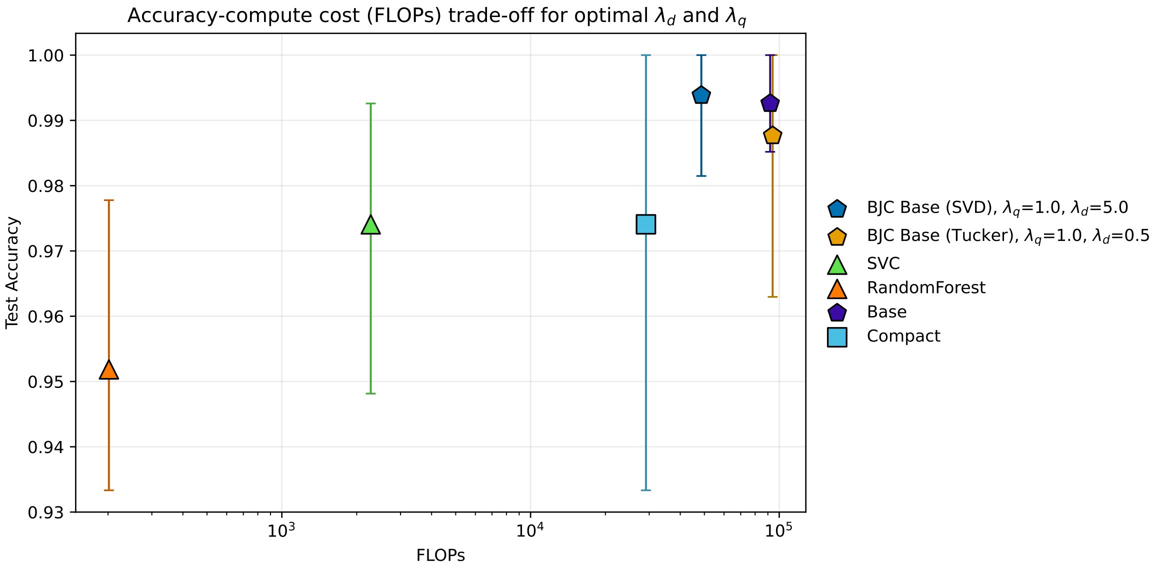


Figure A.4: Accuracy versus computational cost (FLOPs) at the selected λ_d values for all configurations. Points indicate mean ± std across splits.

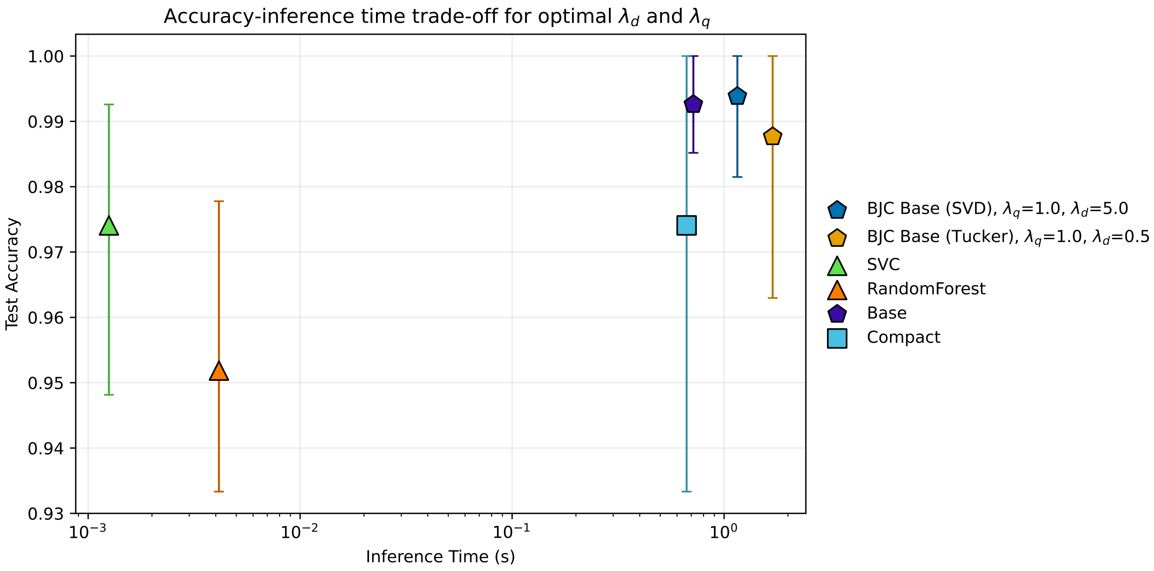


Figure A.5: Per-configuration results at the selected λ_d: accuracy versus inference time (latency). Points are mean ± std across splits.

Table A.1: Comparison of models on the CWRU dataset in terms of test accuracy, parameter count, model size, and compression ratios relative to the uncompressed Base model. CR-Params = compression ratio on parameters; CR-Size = compression ratio on model size. The compression models are defined by regularization parameters (λ_q, λ_d).

Model Type	**Test Accuracy**	**Parameters**	**Model Size (Bytes)**	**CR-Params**	**CR-Size**
SVC	0.974 ± 0.025	617 ± 154	2,467 ± 614	–	–
Random Forest	0.952 ± 0.025	1,269 ± 1,140	5,078 ± 4,559	–	–
Base	0.993 ± 0.009	8,906 ± 0	35,624 ± 0	1.00	1.00
Compact	0.974 ± 0.043	2,538 ± 0	10,152 ± 0	3.51	3.51
BJC Base SVD (λ_q=1.0, λ_d=5.0)	0.994 ± 0.009	2,086 ± 140	811 ± 70	4.27	43.93
BJC Base Tucker (λ_q=1.0, λ_d=0.5)	0.994 ± 0.009	6,265 ± 276	2,069 ± 72	1.42	17.22

Table A.2: Comparison of models on the CWRU dataset in terms of test accuracy, FLOPs, and inference time. The compression models are defined by regularization parameters (λ_q, λ_d).

Model Type	**Test Accuracy**	**FLOPs**	**Inference Time (s)**
SVC	0.974 ± 0.025	2,278 ± 315	0.001 ± 0.001
Random Forest	0.952 ± 0.025	202 ± 188	0.004 ± 0.003
Base	0.993 ± 0.009	91,928 ± 0	0.716 ± 0.071
Compact	0.974 ± 0.043	29,112 ± 0	0.666 ± 0.031
BJC Base SVD (λ_q=1.0, λ_d=5.0)	0.994 ± 0.009	48,616 ± 4,285	1.155 ± 0.053
BJC Base Tucker (λ_q=1.0, λ_d=0.5)	0.994 ± 0.009	92,237 ± 1,656	1.885 ± 0.233