MASTER'S THESIS 2026

Quantization Techniques for Transformer Models on Ethos-U Hardware Accelerators

Märta Holmquist, Emma Kujala

ISSN 1650-2884
LU-CS-EX: 2026-47

DEPARTMENT OF COMPUTER SCIENCE
LTH | LUND UNIVERSITY

EXAMENSARBETE
Datavetenskap

LU-CS-EX: 2026-47

Quantization Techniques for Transformer Models on Ethos-U Hardware Accelerators

Kvantiseringstekniker för transformermodeller på Ethos-U-acceleratorer

Märta Holmquist, Emma Kujala

Quantization Techniques for Transformer Models on Ethos-U Hardware Accelerators

Märta Holmquist
mae@dektis.se

Emma Kujala
emmakujala@hotmail.com

June 26, 2026

Master's thesis work carried out at Arm.

Supervisors: Flavius Gruian, flavius.gruian@cs.lth.se
Oscar Andersson, Oscar.Andersson@arm.com
Adrian Lundell, Andrian.Lundell@arm.com

Examiner: Sven Robertz, sven.robertz@cs.lth.se

Abstract

Efficient transformer inference on edge hardware requires optimization of both model precision and memory utilization. Transformer models are often memory-bound, which makes quantization especially important for reducing memory footprint and data movement. Lower-precision integer formats can improve deployability on embedded devices, but they may also introduce accuracy loss when applied to sensitive operations or tensors with large outliers.

The study focuses on mixed-precision quantization and rotation-based methods, evaluated using vision transformer models deployed on Arm Ethos-U hardware accelerators. Mixed precision is used to preserve accuracy in sensitive parts of the model, while rotations are used to improve weight and activation quantization. The methods are compared using accuracy, memory usage, and hardware-oriented performance metrics. The results confirm that reducing precision has a positive effect on memory and performance, but that mixed-precision is essential in some cases in order to avoid complete model degradation. Rotations can help recover part of the accuracy-efficiency trade-off introduced by lower precision, especially for weights, but effectiveness depends on model architecture.

Keywords: quantization, edge deployment, mixed precision, hardware acceleration, rotation, Hadamard

Acknowledgements

We want to thank Arm and the ExecuTorch team for giving us this opportunity, for sharing their knowledge and resources, and being very generous and welcoming to us as thesis students. Further, we want to thank our supervisors Flavius, Oscar, and Adrian for putting up with us through all of the highs and lows of this project, and the help they have provided.

Contents

1 Introduction **9**
- 1.1 Research Questions . . . 10
- 1.2 Report disposition . . . 11
- 1.3 Division of Labour . . . 11
- 1.4 Statement of AI Use . . . 11

2 Background **13**
- 2.1 Fundamental Concepts . . . 13
- 2.2 Transformer Models . . . 15
  - 2.2.1 Vision Transformers . . . 15
- 2.3 PyTorch . . . 17
  - 2.3.1 Torchvision . . . 18
- 2.4 Microcontroller-Based Edge Inference . . . 18
  - 2.4.1 Memory . . . 18
  - 2.4.2 Computation . . . 18
  - 2.4.3 Integer-Only Inference . . . 19
  - 2.4.4 Ethos-U85 Neural Processing Unit . . . 19
  - 2.4.5 ExecuTorch . . . 19
- 2.5 Quantization . . . 21
  - 2.5.1 Quantization Scheme . . . 21
  - 2.5.2 Symmetric Quantization . . . 22
  - 2.5.3 Calibration . . . 23
  - 2.5.4 Quantization Granularity . . . 23
- 2.6 Quantization Challenges . . . 23
  - 2.6.1 Dynamic Activation Ranges . . . 24
  - 2.6.2 Uneven Quantization Sensitivity . . . 24
  - 2.6.3 Mitigation Strategy: Mixed-Precision Quantization . . . 25
  - 2.6.4 Mitigation Strategy: Rotation-Based Outlier Reduction . . . 26
- 2.7 Benchmarking . . . 29
  - 2.7.1 Model-Level Metrics . . . 30

2.7.2 Hardware-Level Metrics . . . 30

3 Method **33**
3.1 Overview of Method . . . 33
3.2 Initiation Phase . . . 34
3.3 Implementation Phase . . . 35
3.3.1 Rotation Optimization Strategy . . . 35
3.4 Evaluation Phase . . . 36

4 Initiation **37**
4.1 Literature Study . . . 37
4.1.1 Related work . . . 38
4.2 Software Versions and Reproducibility . . . 38
4.3 Analysis Tools and Metrics . . . 38
4.4 Quantization Method Selection . . . 39
4.5 Model Selection . . . 40

5 Implementation **43**
5.1 Ethos-U Quantizer Configuration . . . 43
5.1.1 Mixed-Precision Quantization . . . 44
5.2 Rotation Methods . . . 45
5.2.1 Inserting Rotations . . . 46
5.3 Optimizing Rotations . . . 48
5.3.1 Quantization Simulation . . . 48
5.3.2 Simulated Weight Quantization . . . 49
5.3.3 Simulated Activation Quantization . . . 49
5.3.4 Training Setup Implementation . . . 51
5.3.5 Training Loop . . . 51
5.3.6 Training Evaluation . . . 52
5.3.7 Training Attempts . . . 52

6 Evaluation **55**
6.1 Accuracy Testing . . . 55
6.1.1 Dataset Generation and Sample Size . . . 55
6.1.2 Testing Strategy . . . 56
6.1.3 Random rotations . . . 56
6.2 Hardware Performance Test Strategy . . . 57
6.2.1 Hardware Simulation . . . 57
6.3 Results . . . 60
6.3.1 Rotation impact on accuracy . . . 60
6.3.2 Mixed-Precision Quantization without rotation . . . 65
6.3.3 Performance results . . . 66

7 Discussion **75**
7.1 Method . . . 75
7.1.1 Initiation . . . 75
7.1.2 Implementation . . . 76

7.1.3 Evaluation 77
7.2 Threats to Validity 78
7.2.1 Accuracy Evaluation and Generalizability 78
7.2.2 Implementation and Toolchain Limitations 78
7.2.3 Hardware Performance Estimates 79
7.3 Key Findings and Trade-offs 79
7.3.1 What Were the Best Performers? 80
7.3.2 Recommendations 81
7.4 Quantization to Ethos-U - Lessons Learned 82

8 Conclusions **85**
8.1 Research Questions 85
8.2 Relation to Prior Work 86
8.3 Future Work 86

References **87**

Chapter 1
Introduction

Machine learning (ML) has become a central technology in modern data-driven systems. An ML model is a mathematical model that makes predictions on some given data, a process known as inference. Such models are widely utilized in many different industries and services, including digital services, healthcare, finance, and autonomous systems [1]. In particular, the transformer architecture [2] has become prevalent in the field of deep learning, powering not only large language models, but many other types of models with applications in a wide array of tasks, such as image classification or speech recognition [3, 4].

Many of these models are traditionally deployed in cloud environments, where large amounts of memory, computational resources, and energy are available. However, cloud-based inference introduces several disadvantages. Sending data to a remote server can increase latency, require continuous connectivity, consume communication energy, and raise privacy or security concerns [5]. These limitations have motivated growing interest in edge inference (EI) [6], where trained models are deployed directly on local devices, such as phones or laptops.

Jiang et al. [6] categorize the edge to three tiers according to computational power, memory, and energy constraints as shown in Table 1.1. Our work focuses on inference at the lowest tier, where models are deployed on small microcontroller units (MCUs) that have extremely limited memory capacity and computational power. They are designed for extremely-embedded low-power use cases, such as wearable health monitors or smart sensors used in agriculture [7].

To make neural network inference feasible at this tier, modern MCU platforms often combine general-purpose processors with specialized neural processing units (NPUs) that are designed to accelerate neural network inference [8]. Our work focuses on Arm Ethos-U series of microNPUs [9] that leverage highly specialized architecture and integer-only inference to speed up and enable neural network inference on MCUs based on the Arm Cortex series of processors. Models are deployed to hardware through the Ethos-U backend provided by Arm as part of the ExecuTorch framework [10].

Specialized hardware is however not enough. In the cloud, neural network inference is

Tier	Hardware	Constraints	Parameter Count
High-performance	Laptops	16–64 GB RAM, > 50 TOPS, 20–100 W	7–30B
Mobile/consumer	Phones, tablets	8–16 GB RAM, 10–50 TOPS, 5–15 W	3–7B
Resource-constrained	MCUs	< 2 GB RAM, < 1 TOPS, < 5 W	< 3B

Table 1.1: Edge hardware tiers and feasible model scales, adapted from [6].

executed in high-precision floating-point (FP) numbers that offer the best accuracy, but also require large amounts of memory and are more expensive to compute. To move inference from the cloud to the edge, the floating-point numbers need to be converted to a lower-precision format through an approximation process called quantization [11], one of the most widely used techniques for model compression and acceleration [12]. For example, replacing 32-bit floating-point weights with 8-bit integer (INT) weights reduces the memory required to store the weights by a factor of four. In addition to reducing model size, quantization lowers memory traffic and enables efficient integer arithmetic during inference [13].

While quantization has a positive effect on memory usage and computation speed, the reduction in precision can have extremely adverse effects on model accuracy. Transformer models are especially sensitive to quantization, and preserving good accuracy depends on handling outliers, sensitive operations, and large variations in activation ranges [14]. Recent research has therefore focused on specialized quantization schemes or integer approximations for sensitive operations [15, 16, 17, 18] to address these challenges. Many have achieved accuracy results very close to floating-point performance, demonstrating that low-bit inference with good accuracy is possible for transformer models, but adding them to the existing Ethos-U deployment flow may not be feasible due to the need for tailor-made solutions.

To bridge the gap between theoretical performance and deployment feasibility, we have focused on techniques that mitigate the challenges associated with transformer quantization, but don't require highly specialized operations or schemes to be implemented in the existing deployment flow. These techniques include mixed-precision quantization (MPQ) [19] that keeps sensitive operations at higher precision, and rotation-based methods [20, 21] that aim to reduce outliers by transforming weights and activations into representations that are then easier to quantize. By studying how these methods affect both accuracy and hardware-oriented performance in an existing Ethos-U deployment flow, we aim to provide insight into the challenges of designing quantization strategies that are not tailor-made for one specific model, but remain compatible with the operator, datatype, and compiler constraints of Arm Ethos-U hardware accelerators.

1.1 Research Questions

The main goal of this work is to understand the effects, limitations, and trade-offs that arise when applying the chosen quantization techniques to some transformer models deployed on Ethos-U hardware. In particular, we aim to identify which factors influence whether a quantization method is effective, such as model architecture, sensitivity of individual operations, activation and weight distributions, memory behaviour, and hardware support in the deployment flow. The results are intended to provide guidance for future development of

quantization methods and deployment flows for transformer models on Arm-based microcontroller platforms that use Ethos-U hardware accelerators. To help guide this process, we have defined the following research questions:

RQ1 How do the different quantization approaches affect the performance, accuracy, and memory footprint of encoder transformer models on Embedded Platforms such as Ethos-U?

RQ2 What are the challenges of applying the chosen quantization approaches to transformer models for inference on Ethos-U?

1.2 Report disposition

The remainder of the thesis is structured as follows. Chapter 2 introduces the background concepts needed to understand the work, including transformer models, edge inference, Ethos-U hardware, ExecuTorch, and quantization theory. Chapter 3 presents the overall research method. Chapter 4 describes the initiation phase, including the literature study, model selection, and pilot experiments. Chapter 5 explains the implementation of the quantization and rotation methods. Chapter 6 presents the accuracy and hardware evaluation results. Chapter 7 discusses the implications and limitations of the work, and Chapter 8 concludes the thesis by answering the research questions and outlining possible future work.

1.3 Division of Labour

The work has been shared across all areas and stages of this thesis project, including initiation, implementation, analysis, and report writing. Märta had somewhat more responsibility for the ViT model, and Emma for the Swin. For the report, both authors have contributed to most sections, refined each other's texts, and discussed both content and structure. However, Emma wrote most of the Background and worked more on the overall structure of the report, while Märta constructed tests, collected data, and created most of the Evaluation section.

1.4 Statement of AI Use

AI tools have been used to help find sources, rephrase sentences, extend text segments, check grammar, and to help translate or find suitable words. AI has not been used to generate whole new segments of text, only on rephrasing text we have written ourselves. It has also been used during development to generate simpler pieces of code or scripts, and to help search documentation. Given excel data, AI has helped create tables and figures, as well as improving structure of existing ones. All such output included in the report has been checked for errors.

Chapter 2
Background

This chapter introduces the technical background needed to understand the implementation and evaluation of the quantization techniques we have used. We first define the neural network terminology used throughout the report, then describe the transformer and vision transformer architectures studied in the experiments. The chapter also introduces PyTorch and Torchvision, since the selected models were modified in PyTorch before being exported through the ExecuTorch deployment flow. Later sections describe microcontroller-based edge inference, Ethos-U acceleration, quantization theory, transformer-specific quantization challenges, and the benchmarking metrics used in the evaluation.

2.1 Fundamental Concepts

This section defines the neural network concepts necessary for understanding our work. The emphasis is on tensors, weights, activations, and linear operations, since these are the objects that are quantized, rotated, and inspected during the implementation.

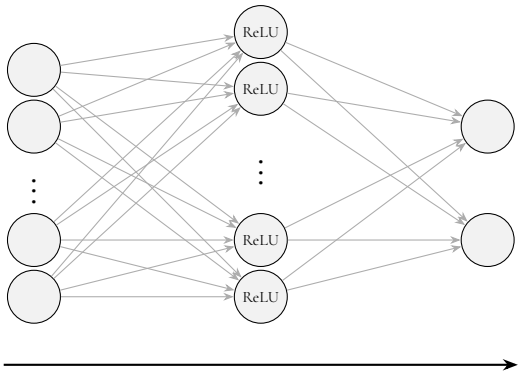


Figure 2.1: Simple fully-connected neural network with one hidden layer.

Neural network: A computational model consisting of layers that transform input data into output predictions. Each layer applies mathematical operations to tensors, often using learned parameters.

Tensor: A multi-dimensional array of numbers. In neural networks, tensors represent inputs, weights, activations, intermediate outputs, and final predictions. Quantization changes how tensor values are represented numerically, for example from floating point to low-precision integers.

Weights: Learned parameters that determine how values from one layer influence values in the next layer. In our work, weights are important both because they occupy memory and because their distributions affect quantization accuracy. Some of the rotation methods modify weight matrices before quantization in order to reduce outlier effects.

Activations: Intermediate tensors produced during the forward pass of the model. Activations depend on the input sample and are therefore more difficult to quantize statically than weights. Many of the challenges studied in this thesis are caused by activation ranges that vary across layers, tokens, channels, or input images.

Attention: Attention is the mechanism introduced in [2] that allows each token to combine information from other tokens in the sequence, by assigning weights to them based on their relevance. Attention projections map activations to the query, key, value, and output representations used by this mechanism.

Multilayer perceptron (MLP): An MLP block is a feed-forward sub-network, typically consisting of two or more linear layers separated by a non-linear activation function. It is applied independently to each token representation to further transform the features after attention.

Linear layer: A linear layer applies a learned linear transformation to an input activation:

$$y = Wx + b. \tag{2.1}$$

Here, x is the input activation, W is the weight matrix, b is the bias, and y is the output activation. Linear layers are central to transformer models because they are used in both attention projections and MLP blocks. Linear layers are also important for rotation-based methods, since orthogonal rotations can often be folded into adjacent linear weights.

Activation function: A nonlinear operation applied to an activation tensor. Nonlinearities allow neural networks to model relationships that cannot be represented by a stack of only linear layers. ReLU and GELU are common activation functions:

$$\mathrm{ReLU}(x) = \max(0, x). \tag{2.2}$$

GELU can be approximated as

$$\mathrm{GELU}(x) \approx x \cdot \Phi(x), \tag{2.3}$$

where $\Phi(x)$ is the cumulative distribution function of the standard normal distribution. Nonlinear operations are relevant for quantization because they can produce asymmetric or difficult activation distributions, and because rotations cannot always be moved freely across nonlinear functions.

Hidden dimension / channel: The hidden dimension is the size of the vector representation used inside the model. In transformer models, each token is represented by a vector of this size. The individual dimensions of this vector are often referred to as channels. Channel-wise variation is important for quantization because a single per-tensor activation scale may need to represent channels with very different ranges.

2.2 Transformer Models

This work focuses on two transformer-based image classification models: the Vision Transformer (ViT) [22] and the Swin Transformer [23]. Transformer models were first introduced by Vaswani et al. [2] and are based on attention mechanisms that allow tokens in a sequence to exchange information. Transformers are widely used in natural language processing, computer vision, speech processing, and multimodal learning [3].

In a transformer, the input is represented as a sequence of tokens. Each token is converted into a vector representation, and the model repeatedly updates these representations using attention and feed-forward layers. The attention mechanism computes how strongly each token should attend to other tokens, meaning that it allows tokens to exchange information with each other.

The original transformer architecture contains both encoder and decoder blocks. The models we have studied are solely encoder-based vision transformers, which means that they process an input image into a representation used for classification, but do not generate an output sequence token by token.

2.2.1 Vision Transformers

Vision Transformers adapt the transformer architecture to image data. Instead of processing words or subword tokens, the image is divided into patches, and each patch is embedded into a token vector. The resulting sequence of patch embeddings is processed by transformer encoder blocks. For image classification, the final representation is used to predict the class of the input image.

The activations in a vision transformer can be viewed as tensors with a token dimension and a channel dimension. The token dimension corresponds to image patches or windows, while the channel dimension corresponds to the hidden representation of each token. This distinction is important for quantization: activation ranges can vary both between tokens and between channels, while quantization parameters are often shared across an entire activation tensor.

The main difference between ViT and Swin is how self-attention is applied. ViT uses global self-attention, where every patch token can attend to every other patch token. This gives a simple architecture, but the attention cost grows with the number of tokens. Swin instead uses shifted-window attention, where attention is computed within local windows and the windows are shifted between blocks. This reduces the computational cost and makes Swin more scalable to larger images, but also introduces additional operations such as attention masking. These architectural differences are relevant for our work because they lead to different quantization behaviour for ViT and Swin.

Encoder Block Architecture

A transformer encoder block consists of two main sublayers: multi-head self-attention and a feed-forward network, often implemented as an MLP. Each sublayer is combined with LayerNorm and a residual connection, as shown in Figure 2.2. These components are central to our work because they are affected differently by quantization. Linear projections are often easier to quantize, while LayerNorm, Softmax, and some attention-related operations are more sensitive.

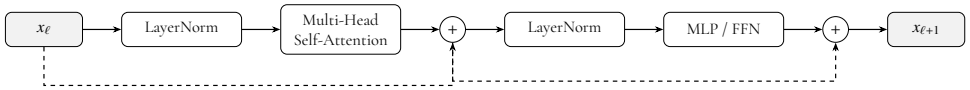


Figure 2.2: Simplified transformer encoder block architecture. The block consists of self-attention and MLP sublayers, each with layer normalization and a residual connection.

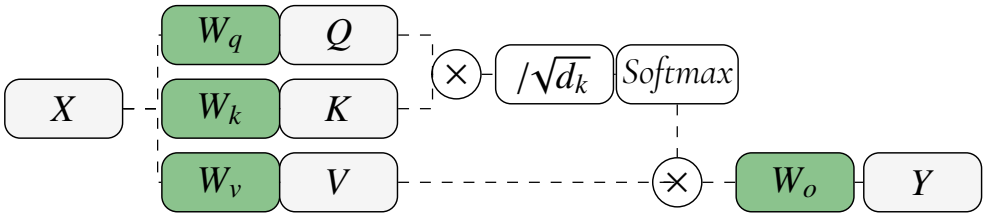
In a transformer encoder, the input to an attention block is a sequence of token embeddings $\boldsymbol{X}$. First, $\boldsymbol{X}$ is projected into query, key, and value tensors:

$$Q = XW_q, \qquad K = XW_k, \qquad V = XW_v. \tag{2.4}$$

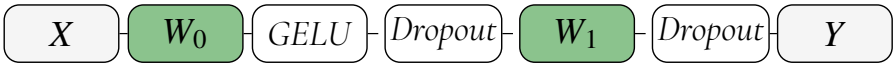
The attention weights are then computed using scaled dot-product attention:

$$\text{Attention}(Q, K, V) = \text{Softmax}\left(\frac{QK^\top}{\sqrt{d_k}}\right)V. \tag{2.5}$$

The Softmax operation converts attention scores into attention probabilities. This operation is important for quantization because the resulting probabilities can be highly non-uniform, with many values close to zero and a few values carrying most of the attention mass. This makes Softmax one of the sensitive operations discussed later in Section 2.6.



(a) Simplified self-attention block.



(b) Simplified MLP block.

Figure 2.3: Simplified attention and MLP block.

Each encoder block also contains an MLP, illustrated in Figure 2.3b. The MLP first expands the hidden dimension, applies a nonlinear activation such as GELU, and then projects

the representation back to the original hidden dimension. The MLP contains large linear weight matrices, making it important for both memory usage and weight quantization.

The residual stream is the main path by which activations flow through the transformer. In Figure 2.2, it is illustrated by the dashed connections around the attention and MLP sublayers. The output of each sublayer is added back to its input. This allows information from earlier layers to be preserved while each block adds new information. The residual stream is also important for rotation-based methods, because rotations must be inserted in ways that preserve the function of the model.

LayerNorm normalizes activations across the feature dimension and applies learned scale and shift parameters. For an input vector x, it can be written as

$$\text{LayerNorm}(x) = \gamma \odot \frac{x - \mu}{\sqrt{\sigma^2 + \epsilon}} + \beta, \tag{2.6}$$

where μ and σ^2 are the mean and variance of the input features, and γ and β are learned parameters. LayerNorm is important for transformer stability, but it is also one of the operations that can be difficult to quantize because it depends on accurate statistics and can receive inputs with large inter-channel variation.

2.3 PyTorch

PyTorch was used to load the selected models, modify their architecture, insert rotations, and run accuracy experiments before export to ExecuTorch. PyTorch is a Python-based deep learning framework where models are represented as subclasses of `torch.nn.Module`. A module contains parameters, submodules, and a `forward` method that defines how input tensors are transformed during inference.

Listing 2.1 shows a minimal PyTorch model. The model contains two linear layers and a ReLU activation. During inference, the input activation x is passed through the first linear layer, the activation function, and the second linear layer. This corresponds to the simple network in Figure 2.1.

Listing 2.1: Simple PyTorch model definition.

```
class SimpleModel(torch.nn.Module):
    def __init__(self):
        super().__init__()

        self.linear1 = nn.Linear(10, 32)
        self.relu = nn.ReLU()
        self.linear2 = nn.Linear(32, 2)

    def forward(self, x):
        x = self.linear1(x)
        x = self.relu(x)
        x = self.linear2(x)
        return x
```

This module structure is important for understanding details of our implementation. The rotation methods were implemented by adding new PyTorch modules to selected parts of the ViT and Swin architectures and modifying their forward passes. During rotation optimization, the original model parameters were frozen while the inserted rotation modules remained trainable. After the PyTorch model had been modified, it was exported to ExecuTorch for quantization and backend lowering.

By default, PyTorch models execute in floating point. Quantized deployment therefore requires an additional export and quantization flow, described later in the ExecuTorch section.

2.3.1 Torchvision

Torchvision is the PyTorch library for computer vision models, datasets, and pretrained weights and we have taken the ViT-B/16 and Swin V2-T model definitions and pretrained weights from there. Using Torchvision models made it possible to start from standard pretrained image-classification models and then apply the same modification, quantization, and evaluation flow to both architectures.

2.4 Microcontroller-Based Edge Inference

Two main factors limit neural network inference on MCUs: memory and computation. Memory determines whether a model can be stored and executed on-device, while computation determines whether inference can be performed within the required latency and energy budget. These constraints are closely connected, since data movement often dominates the cost of inference.

2.4.1 Memory

Memory affects inference in two ways. First, memory capacity limits the size of deployable models, including weights, activations, intermediate buffers, and runtime metadata [6]. Second, memory bandwidth often limits throughput, since inference requires repeatedly moving weights, activations, and partial sums between different levels of the memory hierarchy [24]. Although neural network inference is usually described in terms of multiply-accumulate operations, the cost of moving data can be comparable to, or greater than, the cost of performing the arithmetic itself [25].

The memory hierarchy describes the different storage levels available in a system. Memory close to the compute units is fast and energy-efficient but small, while larger memories are farther away and more expensive to access. Efficient inference therefore depends on reusing data locally and minimizing transfers to slower memory.

2.4.2 Computation

Neural network inference is dominated by linear algebra operations, especially matrix multiplications and convolutions. These operations require large numbers of multiply-accumulate

operations, which are expensive on small processors if executed in software. MCUs are also designed for systems with strict power budgets, so inference must minimize both arithmetic cost and data movement. Specialized accelerators improve efficiency by executing common neural network operations in parallel and by using lower-precision arithmetic, which reduces the area, energy, and memory bandwidth required per operation [25, 6].

2.4.3 Integer-Only Inference

Integer-only inference means that the deployed neural network is executed using integer arithmetic rather than floating-point arithmetic. In a quantized model, floating-point weights and activations are represented by low-precision integers together with scale factors that map between integer values and real-valued ranges. The main arithmetic operations during inference can then be performed using integer multiply-accumulate operations, while rescaling operations are inserted where necessary [11].

This is beneficial for constrained edge devices for two main reasons. First, low-precision integer tensors reduce the memory footprint of weights and activations, which makes it easier to fit models within the limited memory capacity of MCUs. Second, integer arithmetic and reduced tensor sizes lower the cost of computation and data movement. Since data movement can be a major contributor to inference energy, reducing memory traffic is important for energy-efficient deployment [25].

2.4.4 Ethos-U85 Neural Processing Unit

The Arm Ethos-U85 NPU is designed to accelerate neural network inference in Cortex-M based systems. It targets quantized neural networks and executes supported operators using dedicated hardware rather than the general-purpose CPU. The Ethos-U85 supports INT8 weights and INT8 or INT16 activations, with the common deployment path using 8-bit integer quantization for both weights and activations [9]. In the ExecuTorch deployment flow, NPU-compatible parts of the quantized graph are lowered to *TOSA*, which is standardized ML operator language between the model framework and the hardware backend, and compiled with the Ethos-U compiler *Vela*, into an Ethos-U command stream. Operations that are not supported by the Ethos-U backend remain on the CPU. This makes operator support and quantization format important: only subgraphs that satisfy the backend requirements can be delegated to the NPU. In practice, the current ExecuTorch Ethos-U quantization flow mainly targets affine activation and symmetric weight INT8 quantization, as well as symmetric activation and weight INT16 quantization, through the `EthosUQuantizer` [26].

2.4.5 ExecuTorch

Our work uses ExecuTorch to quantize and deploy the PyTorch models onto Ethos-U85 NPU. ExecuTorch uses an ahead-of-time compilation flow: a PyTorch model is first exported to a graph representation, then transformed, quantized, partitioned for available hardware backends, and finally serialized for execution by the ExecuTorch runtime. The Arm Ethos-U backend extends this flow by lowering supported parts of the model to TOSA and compiling them with Vela into command streams that can execute on an Ethos-U NPU.

Figure 2.4 shows the quantization part of the flow. The boxes represent model states, while the arrows represent transformations. Starting from a floating-point PyTorch module, export produces an ATen graph, which is a functional graph representation of the model. Quantization is then applied to this graph. For uniform quantization, the same quantization configuration is applied broadly across the model. For mixed-precision quantization (MPQ), different layers or operator groups may be annotated with different quantization settings, such as different bit-widths or observer configurations. After calibration and conversion, the result is a quantized graph module that can be checked for accuracy before backend lowering.

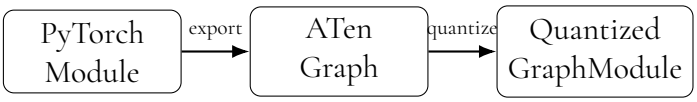


Figure 2.4: Model quantization flow with ExecuTorch.

Figure 2.5 shows the backend lowering and execution flow. The quantized graph is first lowered to the ExecuTorch edge dialect, where operations are represented using a smaller deployment-oriented operator set. The Arm backend then partitions the graph by identifying subgraphs that are compatible with the Ethos-U backend. Supported and quantized subgraphs are delegated to the NPU path, while unsupported operations, unsupported data types, or operations outside the backend's operator coverage remain on the CPU path. This partitioning step is important for MPQ, because only the parts of the graph whose quantization format and operators are supported by the Ethos-U backend can be offloaded to the NPU. After partitioning, the NPU-compatible subgraphs are passed to Vela, Arm's offline compiler for Ethos-U NPUs. Vela lowers these subgraphs into an Ethos-U command stream and applies hardware-specific optimizations based on the selected NPU and memory configuration. In addition, Vela provides different performance estimations for the lowered modules, see section 2.7.2.

The generated command stream is then embedded into the ExecuTorch program and executed by the Ethos-U runtime, while graph regions that were not delegated to the NPU are executed on the CPU. Arm provides a Fixed Virtual Platform (FVP) that simulates an MCU with a Cortex-M CPU and an Ethos-U85 NPU that can be used to execute the runtime.

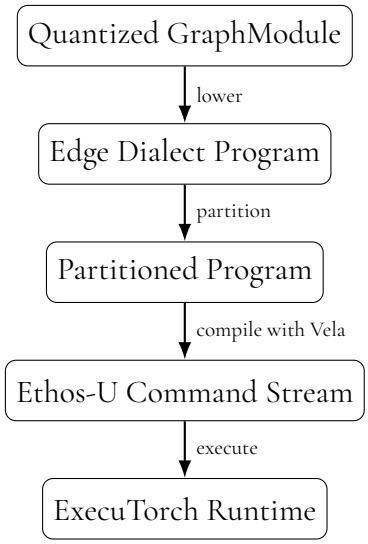


Figure 2.5: Backend lowering and execution flow for Ethos-U.

2.5 Quantization

Quantization is a model compression and acceleration technique where high-precision floating-point values are represented using lower-precision integer values. In neural network inference, quantization is commonly applied to weights and activations. This reduces the memory footprint of the model, lowers memory bandwidth requirements, and enables the use of efficient integer arithmetic on embedded hardware.

Two common approaches are post-training quantization (PTQ) and quantization-aware training (QAT). In PTQ, a pre-trained floating-point model is quantized after training, usually using a small calibration dataset to estimate activation ranges. PTQ is attractive because it does not require retraining the model. In QAT, quantization effects are simulated during training so that the model can adapt to quantization noise. QAT can often achieve higher accuracy, especially at very low bit-widths, but requires access to the training pipeline and is more expensive to apply. This work focuses on PTQ in order to keep the deployment flow lightweight and compatible with existing pre-trained models.

The quantized inference flow in this work is implemented using ExecuTorch and the Ethos-U Quantizer from the Arm backend. The quantizer annotates the exported graph with quantization and dequantization operations, often referred to as Q/DQ nodes. These nodes describe where tensors should be quantized or dequantized and store the quantization parameters used for integer inference. The parameters are estimated by observers during calibration, where representative input data is passed through the model and the observed activation ranges are recorded.

2.5.1 Quantization Scheme

The quantization scheme used in this work follows the affine quantization formulation commonly used for integer-only neural network inference [11]. A real-valued tensor element x is represented by an integer value x_q using a scale factor s and a zero-point z:

$$x_q = \text{clamp}\left(\text{round}\left(\frac{x}{s}\right) + z, q_{\min}, q_{\max}\right), \tag{2.7}$$

where $q_{\min}$ and $q_{\max}$ define the integer range for the selected bit-width. The corresponding dequantized approximation is

$$\hat{x} = s(x_q - z). \tag{2.8}$$

The scale s determines the step size between adjacent quantized values, while the zero-point z is the integer value that represents real zero. Together, the scale, zero-point, and integer range are referred to as the *quantization parameters*.

For affine quantization over a floating-point range $[\alpha, \beta]$, the scale can be computed as

$$s = \frac{\beta - \alpha}{q_{\max} - q_{\min}}. \tag{2.9}$$

The zero-point is then chosen so that the real value zero is exactly representable:

$$z = \text{clamp}\left(\text{round}\left(q_{\min} - \frac{\alpha}{s}\right), q_{\min}, q_{\max}\right). \tag{2.10}$$

This affine scheme is commonly used for activations, whose distributions may be non-negative or asymmetric after nonlinear functions such as ReLU. In these cases, the zero-point allows the quantized range to be shifted so that the available integer values better cover the observed activation range [11, 27].

2.5.2 Symmetric Quantization

Symmetric quantization is a special case of affine quantization where the zero-point is fixed to zero. The quantized value is then computed as

$$x_q = \text{clamp}\left(\text{round}\left(\frac{x}{s}\right), q_{\min}, q_{\max}\right). \tag{2.11}$$

For signed b-bit quantization, the integer range is typically

$$q_{\min} = -2^{b-1}, \qquad q_{\max} = 2^{b-1} - 1. \tag{2.12}$$

A common choice of scale for symmetric quantization is

$$s = \frac{\max(|\alpha|, |\beta|)}{q_{\max}}. \tag{2.13}$$

Symmetric quantization is commonly used for weights because weight distributions are often approximately centered around zero. It also simplifies integer arithmetic, since no zero-point correction is needed for the quantized tensor.

2.5.3 Calibration

Calibration is the process of estimating suitable quantization ranges for activations. The scale factor depends on the floating-point range assigned to the tensor, but using the absolute minimum and maximum values is not always optimal. Rare outliers can increase the quantization range and reduce the effective precision available for the majority of values. Therefore, many PTQ methods use calibration techniques that estimate a representative range rather than simply preserving every observed value.

In the ExecuTorch PTQ flow, observers are attached to the graph during quantization preparation. The model is then evaluated on a calibration dataset, and the observers collect statistics about the tensors that will be quantized. After calibration, these statistics are used to calculate the quantization parameters, and the model is converted to a quantized representation.

In this work, activation ranges are estimated using histogram-based observers. A histogram observer records an approximate distribution of activation values during calibration and then selects quantization ranges based on the collected histogram. This can reduce the effect of outliers compared with simple min/max calibration. For weights, the quantization parameters can be calculated directly from the stored weight tensors, and a min/max observer is usually sufficient.

2.5.4 Quantization Granularity

Quantization granularity describes the scope over which quantization parameters are shared. The choice of granularity can significantly affect accuracy, especially when different channels or tensors have very different value ranges [28].

The simplest option is per-tensor quantization, where one scale and zero-point are shared by all values in a tensor. This is commonly used for activations because activation tensors are produced dynamically at runtime, and per-tensor quantization keeps the implementation simple and efficient.

For weights, per-channel quantization is often preferred. In this scheme, each output channel has its own quantization parameters. This is useful because different channels may have very different weight distributions. If a single shared range is used, channels with small values may lose precision due to outliers in other channels. Per-channel quantization reduces this effect and often improves accuracy with little additional runtime cost.

2.6 Quantization Challenges

Quantization error is not distributed evenly across a transformer model. Some tensors are easy to represent with a low-precision integer range, while others contain outliers, large inter-channel variation, or input-dependent activation ranges that make static quantization difficult. This section introduces the problem of quantizing activations as a central challenge, then describes transformer operations that are especially sensitive to it, and finally presents the two mitigation strategies used in this work: mixed-precision quantization and rotation-based outlier reduction.

2.6.1 Dynamic Activation Ranges

Static activation quantization requires fixed quantization parameters to be chosen during calibration. These parameters must then represent activation values during inference, even though the observed range may vary across inputs, tokens, layers, and channels. If the chosen range is too narrow, large values are clipped. If the range is too wide, the quantization step size increases, reducing precision for the majority of values.

This issue is often caused by outliers or by large variation between channels. For example, in INT8 quantization only 256 discrete values are available. If most activation values lie in the interval $[-10, 10]$, but a few outliers extend the observed range to $[-100, 100]$, then the scale must cover the wider interval. As a result, the dense region around zero is represented with fewer effective quantization levels. The outliers therefore dominate the quantization range, even though they may represent only a small fraction of the activation values.

This problem is prominent in transformer models, where activations can contain large outlier channels, token-dependent variation, and highly non-uniform attention distributions. Several mitigation strategies have been proposed, including clipping-based calibration, per-channel quantization, mixed-precision quantization, and rotation-based transformations.

2.6.2 Uneven Quantization Sensitivity

Quantization does not affect all parts of a transformer equally. Linear layers are often comparatively easier to quantize, especially when weights are quantized per-channel. In contrast, normalization layers, activation functions, and attention-related operations can be more fragile because they either depend on accurate activation statistics or produce distributions that are difficult to represent with a fixed low-precision range.

This uneven sensitivity motivates mixed-precision quantization, where robust parts of the model are quantized aggressively while sensitive operations are kept at higher precision or excluded from quantization. In this work, the most problematic operations were LayerNorm and Softmax. These operations are also repeatedly identified as challenging in prior work on fully quantized Vision Transformers [16, 18, 17].

LayerNorm

LayerNorm was introduced in Section 2.2 as part of the transformer encoder block. In the quantization context, it is one of the most sensitive operations in ViT architectures. Prior work on fully quantized Vision Transformers identifies LayerNorm as a major source of quantization error, especially because its inputs can exhibit large inter-channel variation [16, 18, 17].

The difficulty comes from both the normalization computation and the distribution of the input activations. LayerNorm computes statistics over the feature dimension and normalizes the input using the mean and variance, as shown in Equation 2.6. Estimating and applying this normalization in low precision can introduce error. In addition, because activations are commonly quantized per tensor, a single scale must represent all channels in the activation tensor. If a few channels have much larger values than the rest, they dominate the quantization range and reduce the effective precision available for smaller channels.

LayerNorm is therefore a concrete example of the dynamic range problem described in Section 2.6.1. Errors introduced before or inside LayerNorm can distort the normalized activations and propagate into the following attention or MLP sublayer.

Softmax

Softmax is another operation that is difficult to quantize in transformer models [16, 18, 17]. In self-attention, Softmax converts attention logits into a probability distribution:

$$\text{Softmax}(x)_i = \frac{e^{x_i}}{\sum_j e^{x_j}}. \tag{2.14}$$

Quantizing Softmax is challenging because the exponential function amplifies small perturbations in the input logits. Softmax also depends on relative differences between logits, so quantization noise that changes these differences can significantly alter the resulting attention distribution. In addition, attention probabilities are often highly non-uniform: a small number of tokens may receive most of the probability mass, while many others have values close to zero. Low-bit quantization can therefore cause small probabilities to vanish or large probabilities to saturate, changing the attention pattern used by the model.

In Swin, this issue is further amplified by the shifted-window attention mask. The mask uses large negative values before Softmax to suppress attention outside the current window. These values increase the activation range and make the Softmax-related path especially difficult to quantize.

2.6.3 Mitigation Strategy: Mixed-Precision Quantization

Mixed-precision quantization mitigates uneven quantization sensitivity by assigning different bit-widths or quantization configurations to different parts of the model [19]. Sensitive operations, such as LayerNorm or Softmax-related regions, can be kept at higher precision, while more robust operations are quantized more aggressively. This allows the model to reduce memory and computation cost without applying the same quantization error uniformly to all layers.

Many MPQ methods formulate bit-width assignment as an optimization problem over layers or operator groups [19]. However, the search space can be large, especially for transformer models with many repeated blocks. In addition, some full-integer transformer quantization methods rely on custom approximations for operations such as LayerNorm, Softmax, or GELU [16, 18, 17]. These approximations can achieve high accuracy, but may be difficult to use directly in a deployment flow where the available operators are constrained by the target backend.

In the ExecuTorch quantization flow, MPQ can be viewed as a change in the quantization annotation stage. In uniform quantization, the same quantization configuration is assigned to all supported operators, producing a graph with one consistent quantization scheme. In MPQ, different operators, modules, or subgraphs are annotated with different quantization configurations, or excluded from quantization entirely. The rest of the PTQ flow remains conceptually similar: calibration estimates the quantization parameters for each annotated region, and conversion produces a quantized graph.

This has an important consequence for deployment. A uniformly quantized model can often be represented as one large quantized graph region, whereas MPQ may split the model into several quantized subgraphs with different precision or quantization formats. Each subgraph must independently satisfy the constraints of the deployment backend. In the Ethos-U flow, only subgraphs whose operators and quantization formats are supported by the Ethos-U backend can be delegated to the NPU. Unsupported operations, unsupported precision choices, or transitions between incompatible quantization configurations may break the graph into smaller delegated regions, while the remaining parts are executed on the CPU.

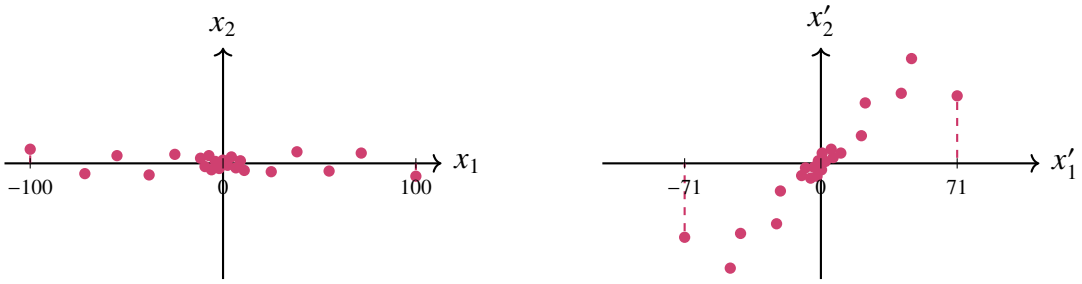
In this work, MPQ is implemented through the quantization configuration interface provided by the Ethos-U quantizer. Different layers or operator types can be assigned different quantization configurations, or excluded from quantization. This makes it possible to focus MPQ on operations known to be sensitive, such as LayerNorm and Softmax, while keeping the rest of the model compatible with the ExecuTorch and Ethos-U deployment flow.

2.6.4 Mitigation Strategy: Rotation-Based Outlier Reduction

Rotation-based methods target the dynamic range problem by changing the basis in which weights and activations are represented. The goal is not to remove information, but to redistribute values across channels so that fewer extreme axis-aligned outliers dominate the quantization range. If the rotated weights or activations have a smaller or more balanced range, they can be easier to quantize at low precision.

This work is based on QuaRot [20] and SpinQuant [21]. Both methods use orthogonal rotations to reduce outlier effects before quantization. The idea is illustrated in Figure 2.6: after rotation, the same data is represented in a different coordinate basis, which can reduce the dynamic range and differences in magnitude across channels.

Rotations can be inserted at different locations in the model architecture. The most useful locations are those where outliers are pronounced and where the rotation can be merged into adjacent weights, since this reduces the number of online rotations required during inference. In transformer models, such locations are commonly found inside the attention block and the MLP. The insertion of rotations is based on the concept of computational invariance used in SliceGPT [29], where orthogonal transformations can be inserted without changing the model function if neighbouring weights are transformed accordingly.



(a) Before rotation. **(b)** After rotation.

Figure 2.6: Conceptual 2D illustration of how rotation changes quantization ranges. Rotating the coordinate basis creates a tighter range and x'_1 and x'_2 components of more similar magnitudes.

Where Can Rotation Be Inserted

Weights and activations between encoder blocks and inside the blocks can be transformed using an orthogonal matrix without changing the model output. Equations 2.15-2.17 show the mathematical reasoning behind this.

Let $\boldsymbol{R} \in \mathbb{R}^{n \times n}$ be an orthogonal matrix:

$$R^\top R = I_n. \tag{2.15}$$

For a linear transformation with weight matrix $W \in \mathbb{R}^{m \times n}$ and input $x \in \mathbb{R}^n$, inserting $R^\top R = I_n$ gives

$$Wx = WI_n x = WR^\top Rx. \tag{2.16}$$

This can be written as

$$Wx = (WR^\top)(Rx). \tag{2.17}$$

Equation 2.17 shows that as long as both the activation and the weights are transformed by the same rotation, the computation remains invariant.

Offline Rotation

This can be extended to the following important insight that makes it possible to add rotations to the model without adding more computations. If an orthogonal transformation $\boldsymbol{R}$ has been applied to an input, it can be compensated by transforming the weight matrix as

$$W \mapsto WR^\top. \tag{2.18}$$

This principle shown in2.18 makes it possible to replace the weight matrix W with $WR^\top$ ahead of time, eliminating the need for calculating it during inference. These transformations are called **offline rotations**, and they can also be used to apply a rotation to an activation.

Online Rotation

Operations within the model are not always linear. **Online rotations** are needed when the activations need to be transformed at a place there is a non-linear operation between linear

operations. Equation 2.17 does not hold if a nonlinear operation such as LayerNorm (LN) is placed between the rotation and the weight matrix as shown by the following.
Consider the equation 2.19, where LN is applied to the activation tensor x before the weights are applied to it.

$$y = W\,\mathrm{LN}(x). \tag{2.19}$$

If the activation is rotated before LN and the inverse rotation is merged into the weight matrix, the transformed computation becomes

$$\tilde{y} = (WR^\top)\,\mathrm{LN}(Rx). \tag{2.20}$$

For this to be equivalent to Equation 2.19, the following equation would need to hold

$$(WR^\top)\,\mathrm{LN}(Rx) = W\,\mathrm{LN}(x). \tag{2.21}$$

This in turn requires the following equivariance for arbitrary orthogonal rotations to hold

$$\mathrm{LN}(Rx) = R\,\mathrm{LN}(x). \tag{2.22}$$

However, this is is not the case for a general orthogonal matrix R, so

$$\mathrm{LN}(Rx) \neq R\,\mathrm{LN}(x) \tag{2.23}$$

Then, consequently we also have

$$(WR^\top)\,\mathrm{LN}(Rx) \neq W\,\mathrm{LN}(x), \tag{2.24}$$

Online rotations need to be calculated during inference, and may therefore have a negative effect on model performance, introducing a tradeoff between that and possible gain in accuracy. To mitigate the effect of adding the extra rotations, a special type of rotation matrix is used.

Efficient Hadamard Rotations

A general dense rotation matrix would be expensive to apply during inference. Rotation-based quantization methods therefore often use structured orthogonal matrices that can be applied efficiently. A common choice is the normalized Walsh–Hadamard matrix, which can be multiplied by a vector using the Fast Walsh–Hadamard Transform.

The normalized Walsh–Hadamard matrices $\tilde{H}_n$ are defined recursively for $n = 2^k$ by

$$\tilde{H}_1 = \begin{bmatrix} 1 \end{bmatrix}, \qquad \tilde{H}_{2n} = \frac{1}{\sqrt{2}} \begin{bmatrix} \tilde{H}_n & \tilde{H}_n \\ \tilde{H}_n & -\tilde{H}_n \end{bmatrix}. \tag{2.25}$$

This normalization makes $\tilde{H}_n$ orthogonal:

$$\tilde{H}_n^\top \tilde{H}_n = I_n. \tag{2.26}$$

For $x \in \mathbb{R}^n$, multiplication by $\tilde{H}_n$ can be computed in $O(n \log n)$ time using the Fast Walsh–Hadamard Transform. This requires only additions, subtractions, and scaling, making it much cheaper than multiplication by a dense rotation matrix.

A randomized Hadamard rotation can be constructed as

$$R = \tilde{H}_n D, \tag{2.27}$$

where

$$D = \text{diag}(d_1, \ldots, d_n), \qquad d_i \in \{\pm 1\}. \tag{2.28}$$

The diagonal matrix D randomly flips signs, while $\tilde{H}_n$ spreads the activation energy across dimensions. Since both matrices are orthogonal, their product is also orthogonal.

Optimizing Rotations

The rotations described above can either be sampled randomly or optimized for a specific model and quantization configuration. Random Hadamard rotations are computationally efficient and may reduce outlier effects without requiring additional training. However, a random rotation is not necessarily the best rotation for a given model. Learned rotations aim to find a transformation that preserves the model function while making the rotated weights and activations easier to quantize.

In rotation-based quantization methods such as SpinQuant [21], the model weights are kept fixed and only the rotation matrices are optimized. This makes the optimization problem smaller than full model fine-tuning. The objective is not to improve the floating-point model, but to reduce the loss introduced by quantization. The rotations are therefore trained under simulated quantization, so that the optimization process sees an approximation of the errors that will occur after the model is quantized.

A challenge is that quantization contains rounding operations, which are not directly differentiable. If the derivative of the rounding function is treated exactly, it is zero almost everywhere, preventing gradients from flowing through the quantizer. A common solution is the straight-through estimator (STE), where the quantizer is used in the forward pass but is approximated as the identity function in the backward pass [30]. This allows gradient-based optimization to update the rotation matrices despite the presence of simulated quantization.

The rotation matrices must also remain orthogonal during optimization. Orthogonality is important because the inverse of an orthogonal matrix is its transpose, which is what allows rotations to be cancelled or folded into neighbouring weight matrices. If optimization breaks orthogonality, the computational invariance described above no longer holds. For this reason, rotation-based methods can use optimization algorithms on the Stiefel manifold, such as Cayley SGD, which update the matrices while preserving orthogonality [31].

In this work, optimized rotations are treated as a post-training quantization method. The pre-trained model weights remain frozen, and only the inserted rotation matrices are trained. After optimization, the learned rotations can be inserted into the deployment model and quantized using the same ExecuTorch and Ethos-U flow as the randomly rotated models.

2.7 Benchmarking

Evaluating quantization methods for deeply embedded inference requires both model-level and hardware-level metrics. A quantization method may improve memory usage and execution time while reducing accuracy, or it may preserve accuracy at the cost of additional

operators, larger intermediate tensors, or less favourable graph partitioning. Benchmarking therefore needs to capture the trade-off between accuracy, memory footprint, memory traffic, and execution cost.

Benchmarking suites such as MLPerf Inference [32] have been developed to compare machine learning systems across different models, software frameworks, and hardware platforms. In this work, the goal is narrower. The hardware target and deployment flow are fixed, and the benchmark is used to compare different quantization configurations for the same model architecture. The most important question is therefore not which platform is fastest, but how a change in quantization strategy affects accuracy, memory behaviour, and execution on the Ethos-U deployment path.

2.7.1 Model-Level Metrics

Accuracy

Accuracy measures how well a model performs the target task. For image classification, accuracy is commonly reported as the percentage of correctly classified images. Top-1 accuracy measures whether the highest-scoring class is correct, while Top-5 accuracy measures whether the correct class appears among the five highest-scoring predictions. Accuracy depends on both the dataset and the model architecture, and should therefore be interpreted in context [33]. In this work, Top-1 and Top-5 accuracy are used to measure how much accuracy is lost or recovered when applying uniform quantization, mixed-precision quantization, and rotations.

Model Size and Operation Count

Model statistics describe the computational and storage requirements of a network. Important metrics include the number of parameters, the number of multiply-accumulate operations (MACs), and the memory required to store weights, activations, and intermediate buffers [25]. These metrics are especially important on microcontroller-class systems, where memory capacity and computational throughput are limited.

For quantized models, the weight memory footprint depends on both the number of parameters and the bit-width used to store them. For example, changing weights from 8-bit to 4-bit ideally halves the storage required for those weights. Activation precision affects the size of intermediate tensors, and therefore has a strong impact on SRAM requirements. In mixed-precision quantization, different parts of the model may use different precisions, so the total memory footprint depends on the bit-width assigned to each region of the graph.

2.7.2 Hardware-Level Metrics

Latency, Cycles, and Throughput

Latency describes the time required to produce the output for a single input, while throughput describes the number of inferences completed per second. Low latency is important for real-time edge applications, while high throughput is important when many inputs must be

processed. On MCU-class systems, batching is often limited by memory capacity, so single-inference latency is usually the most relevant performance metric.

Throughput can be related to hardware performance and model complexity as

$$\text{Throughput} = \frac{\text{operations per second}}{\text{operations per inference}}. \tag{2.29}$$

Equation 2.29 shows that throughput depends both on the computational capability of the hardware and on the number of operations required by the model [33]. However, this expression is only an approximation. In practice, performance also depends on memory bandwidth, operator support, graph partitioning, compiler scheduling, and whether operations execute on the NPU or fall back to the CPU.

In this work, execution performance is mainly evaluated using cycle estimates and derived inference time. The estimated inference time is obtained from the reported cycle count and the assumed clock frequency. These values are useful for comparing configurations compiled under the same assumptions, but should not be interpreted as exact measurements of physical hardware execution.

Memory Usage and Bandwidth

Memory behaviour is a central part of benchmarking on constrained edge devices. Memory capacity determines whether a model can be deployed, while memory bandwidth strongly influences inference time and energy consumption. Important capacity-related metrics include peak SRAM usage, peak DRAM usage, and the memory required for weights, activations, intermediate tensors, and compiler-generated command streams. See section 6.2.1 for information on Ethos-U memory modes.

Memory bandwidth describes how much data is moved during inference. Inference requires weights, activations, and intermediate outputs to be moved through the memory hierarchy. Metrics such as total SRAM bandwidth, total DRAM bandwidth, input bandwidth, output bandwidth, and weight bandwidth therefore help identify whether a configuration is limited mainly by computation or by data movement. Since moving data can be significantly more expensive than arithmetic operations, reducing memory traffic is important for both performance and energy efficiency [34].

Memory Modes

The Ethos-U memory configuration affects both memory usage and performance. In this work, the relevant memory modes are Shared SRAM and Dedicated SRAM. In Shared SRAM mode, intermediate tensors are stored in SRAM while weights are stored in external memory. This can reduce activation traffic to external memory, but requires enough SRAM to hold the largest required intermediate tensors. In Dedicated SRAM mode, both tensors and weights may reside in external memory, while SRAM is used as a dedicated fast memory region or cache. This mode can support larger models, but performance becomes more dependent on external memory traffic.

The same quantized model can therefore have different performance characteristics under different memory modes. For this reason, hardware metrics should be interpreted together with the memory configuration used during compilation.

Delegation and Partitioning

For accelerator-based deployment, it is not sufficient to measure only the total number of operations. It is also important to know which parts of the model are executed on the NPU. In the ExecuTorch Ethos-U flow, supported quantized subgraphs are delegated to the NPU, while unsupported operations or unsupported quantization formats remain on the CPU.

This is especially relevant for mixed-precision quantization. A uniformly quantized model may be represented as one large delegated graph region, while MPQ can split the model into several smaller subgraphs with different quantization configurations. Each delegated subgraph is compiled separately for the NPU, and transitions between CPU and NPU execution can introduce additional memory traffic or runtime overhead. Therefore, the number of delegated subgraphs, the fraction of operations executed on the NPU, and the amount of data moved between partitions are important metrics when evaluating MPQ.

Compiler Estimates and Runtime Profiling

The Ethos-U deployment flow provides two main sources of hardware-oriented metrics. The first is the Vela compiler report, which estimates memory usage, bandwidth, operation counts, and cycle counts for NPU-compatible graph regions. These estimates are useful for comparing many configurations quickly and for identifying trends in memory and compute behaviour.

The second source is runtime profiling on a Fixed Virtual Platform (FVP). FVP profiling can provide runtime-level information such as delegated cycles, end-to-end runtime, CPU wait time, and Performance Monitoring Unit (PMU) counters. PMU counters can indicate whether memory traffic mainly targets SRAM or external memory, and how NPU activity is distributed between MAC execution, weight decoding, and idle time.

Compiler estimates and FVP profiling serve different purposes. Compiler estimates are well suited for broad comparisons between quantization configurations, while FVP profiling provides a more detailed view of execution behaviour for selected configurations. Since compiler estimates depend on graph lowering, scheduling, tensor lifetimes, and memory planning, relative changes between similar configurations are generally more meaningful than absolute values.

Chapter 3
Method

This chapter presents the research method used in the project. The work was organized into three phases: initiation, implementation, and evaluation. The initiation phase established the technical scope of the project by selecting relevant models, quantization methods, tools, and evaluation criteria. The implementation phase focused on applying and adapting the selected methods to the chosen models and deployment flow. The evaluation phase measured the effect of the implemented methods on accuracy, memory behaviour, and hardware-oriented performance metrics.

The purpose of the method was not to find a single optimal configuration for one model, but to use selected vision transformer models as case studies for understanding how mixed-precision quantization and rotation-based methods behave in an Ethos-U deployment flow. The work therefore combines exploratory implementation with controlled comparisons between quantization configurations.

3.1 Overview of Method

Figure 3.1 shows the overall workflow. The project began with a literature study and definition of research goals. This was followed by environment setup, tool development, and pilot experiments. These early steps were used to narrow the scope to post-training quantization, select the target models, and identify which quantization methods were feasible to implement within the existing PyTorch, ExecuTorch, and Ethos-U toolchain.

After the initiation phase, the project moved into an iterative implementation phase. Model modifications, quantization configurations, and rotation insertions were tested continuously using accuracy evaluation and graph inspection tools. This made it possible to detect problematic configurations early, for example cases where a sensitive module did not tolerate low activation precision or where a quantization configuration did not appear in the exported graph as expected.

The final evaluation phase was divided into two parts. First, accuracy testing was used

to compare the implemented configurations and select the most relevant ones for further study. Second, hardware-oriented evaluation was performed on selected configurations using compiler estimates and, where possible, simulation on the Arm Fixed Virtual Platform (FVP). This allowed the final comparison to include not only model accuracy, but also memory usage, memory bandwidth, cycle estimates, and NPU delegation behaviour.

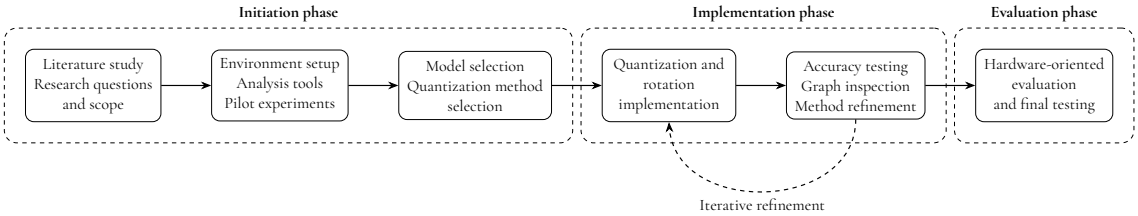


Figure 3.1: Overview of the workflow used in the project.

3.2 Initiation Phase

The initiation phase defined the scope and experimental basis of the project. It consisted of a literature study, development environment setup, tool development, pilot experiments, quantization method selection, and model selection. The purpose was to identify a feasible set of quantization methods and models that could be implemented and evaluated within the Ethos-U deployment flow.

The literature study focused on three areas: quantization of transformer models, deployment of transformers on constrained edge devices, and methods for evaluating accuracy and hardware performance. Based on this study, the project was limited to post-training quantization. This choice was made because PTQ is more compatible with existing pre-trained models and does not require a full model retraining pipeline.

The initiation phase also included the development of analysis tools. These tools were used to compare intermediate outputs between floating-point and quantized models, and collect activation statistics. The goal was to make the implementation phase more systematic by providing evidence for where quantization errors appeared and which parts of the model were most sensitive.

Pilot experiments were used to validate the main assumptions behind the selected methods. Small models were used to study how outliers affect quantization error and whether orthogonal rotations could reduce that effect. These experiments were not intended as final benchmarks, but as a way to test the tooling and confirm that rotation-based methods were relevant to investigate further.

The final model selection was based on model type, task, availability of pre-trained weights, and compatibility with the deployment flow. Image classification was chosen because it is a common edge inference task and provides clear accuracy metrics. ViT-B/16 was selected as a representative vision transformer with a relatively simple encoder architecture, while Swin V2-T was selected as a more complex architecture with shifted-window attention. Together, the two models provided case studies with different architectural properties and different quantization challenges.

3.3 Implementation Phase

The implementation phase applied the selected quantization and rotation methods to the chosen models. This phase was iterative: implementation changes were followed by accuracy tests, graph inspection, and activation or node-output analysis. The results of these checks then guided the next implementation step.

The first part of the implementation focused on configuring the Ethos-U quantizer. Baseline uniform quantization configurations were created for different activation and weight precisions. Mixed-precision quantization was then implemented by assigning different quantization configurations to selected modules or graph patterns. For ViT, the main focus was LayerNorm, which was identified as highly sensitive to low activation precision. For Swin, the main focus was the attention mask and Softmax-related computation, since the shifted-window attention mechanism introduced values that were difficult to quantize.

The second part of the implementation focused on rotation-based methods. Rotation modules were inserted into the ViT and Swin architectures in locations where the transformation could either be merged into weights or applied online while preserving the model function. The floating-point model outputs were compared before and after inserting rotations to verify that the transformations did not change the model behaviour before quantization.

The third part of the implementation explored optimized rotations. Since the final Ethos-U quantization flow is not directly differentiable, a separate fake-quantized training setup was implemented. This setup simulated weight and activation quantization near the inserted rotations, allowing the rotation matrices to be optimized while keeping the original model weights frozen. Several versions of the fake quantization setup and training configuration were tested during the project. The goal was not to exhaustively tune the training procedure, but to investigate whether optimized rotations could improve quantized accuracy compared with random rotations.

3.3.1 Rotation Optimization Strategy

After implementing random rotations, we also investigated whether the rotation matrices could be optimized for the chosen quantization setting. The purpose of this step was not to train the full model, but to adapt the inserted rotations so that the weights and activations became easier to quantize.

Each rotation optimization attempt followed the same general procedure. First, a model was prepared by inserting randomly initialized rotation matrices. The original model parameters were frozen, and only the rotation matrices were kept trainable. Second, fake quantization was enabled in selected parts of the model so that the rotations were optimized under simulated quantization error. Third, the model was trained on a subset of ImageNet-1K and evaluated on a held-out validation subset.

This setup was used because the final Ethos-U quantization flow is designed for deployment rather than gradient-based training. The exported and quantized ExecuTorch graph is not a convenient target for backpropagating gradients to update inserted rotation matrices. Instead, fake quantization was used during training to approximate the quantization effects that would later occur in the Ethos-U quantized model.

The optimization process was exploratory. Since the rotation optimization approach was inspired by SpinQuant, which was developed for language models, the same hyperparameters could not be assumed to work directly for vision transformers. Several training attempts were therefore used to test different fake-quantization schemes, learning rates, and dataset sizes. The resulting trained rotations were evaluated both in the fake-quantized training setup and after quantization with the Ethos-U quantizer.

3.4 Evaluation Phase

The evaluation phase measured how the implemented methods affected accuracy, memory usage, and hardware-oriented performance. It was divided into accuracy evaluation and hardware-oriented evaluation.

Accuracy evaluation was used throughout the project, but served two different purposes. During implementation, smaller accuracy tests were used as a development tool to detect large regressions and guide design choices. In the final evaluation, larger tests were used to compare selected configurations under controlled conditions. The same ImageNet-1K validation split was used throughout the project, with fixed calibration, training, and test sets where possible. This reduced variation caused by differences in calibration data or test data.

The accuracy evaluation compared uniform quantization, mixed-precision quantization, random rotations, trained rotations, and combinations of MPQ and rotations. For random rotations, repeated runs were used for selected configurations to estimate how much accuracy varied between sampled rotation matrices. Fake-quantized models were also evaluated to compare the behaviour of the rotation training setup with the final Ethos-U quantized models.

The hardware-oriented evaluation was performed after the main accuracy experiments had identified the most relevant configurations. The goal was to measure whether the accuracy benefits of MPQ and rotations were accompanied by changes in memory footprint, memory bandwidth, operation count, cycle count, and NPU delegation behaviour. Vela compiler reports were used to obtain memory and performance estimates for several configurations and memory modes. These estimates made it possible to compare the effects of activation precision, weight precision, MPQ, and rotations while keeping other factors fixed.

FVP profiling was used for a smaller number of configurations due to longer runtime. The FVP results were used to complement the compiler estimates with runtime-oriented metrics, such as delegated cycles, CPU wait time, PMU memory counters, and NPU activity counters. Since most hardware results are compiler estimates rather than physical hardware measurements, the evaluation focuses mainly on relative comparisons between similar configurations rather than absolute runtime claims.

Chapter 4
Initiation

In this chapter we explain how the initiation phase was carried out in detail.

4.1 Literature Study

To find relevant literature, we mainly used FINN (Lund University Library search tool) and Google Labs, an AI search tool that allows searching via prompts. Prompt searches on Google Scholar were used to find surveys or reviews on relevant topics, which were then used as sources for additional material. These were complemented by keyword searches on FINN to make sure that we did not miss recently published research and did not only rely on the Google Labs.

We used these types of search prompts on Google Labs:

- "Find peer-reviewed surveys or reviews about transformer/LLM inference on the edge"
- "Find peer-reviewed surveys or reviews about transformer/LLM quantization"
- "What methods have been developed for deploying transformer models on MCUs?"

On FINN the following types of search terms were used and the results were filtered to only include peer-reviewed sources.

- "Transformer" OR "LLM" AND "quantization"
- "ViT" OR "Swin" AND "quantization"
- "mixed-precision quantization" AND "transformer"
- MCU AND "transformer" OR "ViT"

4.1.1 Related work

Our literature review found works in four different areas that are relevant for our efforts.

- **Post-training quantization for ViTs.** PTQ4ViT addresses the quantization difficulty of post-Softmax and post-GELU activations using twin uniform quantization and Hessian-guided calibration [15]. FQ-ViT extends ViT PTQ toward fully quantized inference by introducing Power-of-Two Factor quantization for LayerNorm inputs and Log-Int-Softmax for attention [16]. TSPTQ-ViT similarly targets fully quantized A8W8 ViTs, using two-scale quantization for post-Softmax, post-GELU, and LayerNorm activations [17].
- **Integer-only ViT inference.** I-ViT focuses on integer-only ViT deployment by replacing floating-point Softmax, GELU, and LayerNorm with lightweight integer approximations, enabling end-to-end integer arithmetic during inference [18].
- **Mixed-precision quantization.** LRP-QViT is a mixed-precision quantization method that assigns layer-wise bit-widths according to relevance scores computed using layer-wise relevance propagation, improving the accuracy–compression trade-off compared with uniform low-bit quantization [35].
- **MCU-oriented transformer deployment.** MCUFormer jointly considers ViT architecture design, MCU-specific operator support, and memory scheduling for deployment under tight microcontroller constraints [36]. TinyFormer also targets transformer deployment on microcontrollers, combining neural architecture search, sparsity, and a sparse inference engine to reduce memory and compute requirements [37].

4.2 Software Versions and Reproducibility

The experiments were carried out using ExecuTorch version 1.0 with the Arm Ethos-U backend. Since the Arm backend was under active development during the project, all experiments were fixed to a specific ExecuTorch revision to avoid changes in backend lowering, quantization, or operator support from affecting the results. The revision used in this work corresponds to the merge commit `a06697e` from pull request #17004, "Arm backend: Add matmul decomposition pass" [38]. This pull request was merged into the ExecuTorch main branch on February 2, 2026 [38].

The Ethos-U deployment flow also depends on Vela, Arm's compiler for Ethos-U NPUs. The ExecuTorch Arm backend lowers supported subgraphs to TOSA and uses the `ethos-u-vela` compiler to generate an Ethos-U command stream [39]. For the selected ExecuTorch revision, the Arm Ethos-U backend requirements specify `ethos-u-vela` version `4.5.0` [40]. This Vela version was therefore used for compiling the delegated Ethos-U subgraphs in all hardware performance experiments.

4.3 Analysis Tools and Metrics

We used several different tools to help us analyze what was happening during inference. An existing model explorer tool [41] allowed us to see the exported model and to confirm how

q/dq nodes were placed, as well as their datatypes. We also developed the following tools for analyzing quantization error.

- **Node output inspector**: This tool allowed us to see where there were significant differences in output values between the quantized model and the floating point model during inference
- **Activation collector**: analyze the activations at different points during inference, used to find potential bottlenecks and confirm results

We used several metrics to guide the implementation work and interpret the final results. The benchmarking metrics introduced in Section 2.7 were used to evaluate accuracy, memory usage, bandwidth, and hardware-oriented performance on the Ethos-U85 deployment flow. These metrics made it possible to compare quantization configurations and estimate how well each method would perform on the target NPU.

In addition to the final evaluation metrics, we collected intermediate diagnostic metrics during development. These were used together with the activation collection tools to identify where quantization error occurred and which tensors were difficult to represent at lower precision. The diagnostic metrics included activation ranges, per-channel variation, minimum and maximum tensor values, output differences between floating-point and quantized models, and the placement of quantize/dequantize nodes in the exported graph. These measurements helped guide the selection of mixed-precision exceptions, rotation insertion points, and configurations for further accuracy and hardware evaluation.

4.4 Quantization Method Selection

Many quantization methods and frameworks have been proposed in recent years, especially for transformer models. Several state-of-the-art methods for ViT quantization achieve high accuracy by introducing specialized quantization schemes or custom integer approximations for operations such as Softmax, GELU, and LayerNorm [15, 16, 17, 18]. These methods demonstrate that accurate low-bit transformer inference is possible, but they may be difficult to apply directly in a general deployment flow. In practice, a method must be supported not only by the model implementation, but also by the quantizer, graph lowering, compiler, and target hardware.

For this reason, we selected methods that could be implemented around the existing ExecuTorch and Ethos-U deployment flow rather than by modifying the Arm Ethos-U quantizer itself. The goal was to investigate how far the existing quantizer could be pushed, and whether model-level changes could improve quantized accuracy while remaining compatible with the backend. This constraint made post-training quantization the most realistic scope for the project, since PTQ can be applied to existing pre-trained models without a full retraining pipeline.

Two main directions were selected. The first was mixed-precision quantization (MPQ), where sensitive modules or graph regions are assigned a different quantization configuration or excluded from quantization. This was chosen because prior work shows that transformer layers vary significantly in quantization sensitivity, and because the Ethos-U quantizer provides mechanisms for assigning different quantization configurations to selected modules or

operator patterns. MPQ therefore provided a practical way to study how sensitive operations such as LayerNorm, Softmax, and attention-mask computations affect deployment accuracy.

The second direction was rotation-based outlier reduction, inspired by QuaRot and SpinQuant [20, 21]. Rotation-based methods were selected because they can be implemented as model transformations in PyTorch, before export to ExecuTorch. This makes them attractive for a deployment-oriented study: the model can be modified to make weights or activations easier to quantize, while the final quantization and lowering are still handled by the existing Ethos-U flow. QuaRot motivated the use of random Hadamard rotations, while SpinQuant motivated the later attempt to optimize rotations under simulated quantization.

Other methods considered during the literature study included approaches based on specialized integer approximations, layer-wise bit-width search, pruning, and sparsity. However, these were not selected as the main focus because they would either require additional backend support, a larger optimization framework, or changes to the hardware execution model. The final method selection therefore reflects the main goal of this thesis: to study quantization techniques that are relevant for practical deployment on Ethos-U, while still being feasible to implement within the scope of the project.

4.5 Model Selection

The model selection was based on four criteria: task relevance for edge inference, transformer architecture, availability of pre-trained weights, and compatibility with the Ethos-U deployment flow. We chose to focus on image classification because it is a common edge inference task and because the evaluation metric is well established. Using ImageNet-1K classification also made it possible to compare measured accuracy against published or reported reference accuracy for the selected weights.

Two encoder-based vision transformer architectures were selected: ViT-B/16 and Swin V2-T. ViT-B/16 was chosen because it is a widely used Vision Transformer architecture with a relatively simple encoder structure [22]. Its quantization challenges, especially around LayerNorm and attention-related operations, are also well documented in prior work. This made it a suitable first case study for testing both mixed-precision quantization and rotation-based methods.

Swin V2-T was selected as a second case study because it is still an encoder-based vision transformer, but has a more complex architecture than ViT due to shifted-window attention. The Tiny variant was chosen because it is the smallest Swin V2 model available through Torchvision, making it more realistic for constrained deployment than larger variants. Swin therefore provided a useful contrast to ViT: it shared the same task and deployment flow, but introduced additional quantization challenges related to the shifted-window attention mask and Softmax computation.

Smaller language models and other transformer-based tasks were also considered. However, adding a third task would have required a separate dataset, evaluation pipeline, and possibly different deployment assumptions. Since the goal was to study quantization effects in depth rather than cover many model families superficially, the final scope was limited to two image classification models. This also made it possible to reuse the same dataset, calibration strategy, accuracy metrics, and deployment flow across the experiments.

Table 4.1 summarizes the selected models and weights.

Table 4.1: Chosen models and weights.

Model	Weights	Parameters	Top-1 (%)	Top-5 (%)
ViT-B/16 [42]	IMAGENET1K_SWAG_E2E_V1	86.9M	85.304	97.650
Swin V2-T [43]	IMAGENET1K_V1	28.4M	82.072	96.132

Chapter 5
Implementation

In this chapter, we describe how the quantization and rotation methods were implemented. The implementation consists of three main parts. First, the Ethos-U quantizer was configured for different activation and weight precisions. Second, mixed-precision quantization was implemented by assigning alternative quantization configurations to selected modules or operator patterns. Third, rotation modules were inserted into the ViT and Swin architectures and later optimized under simulated quantization.

5.1 Ethos-U Quantizer Configuration

Model quantization was performed using the Ethos-U quantizer in the ExecuTorch Arm backend. For each experiment, we created an Ethos-U quantizer and configured it with the target Ethos-U85 compile specification, the chosen activation and weight precision, and the calibration setup. The model was then prepared for post-training quantization by inserting observers into the exported graph. Calibration was performed by running representative ImageNet samples through the prepared model, after which the observers were used to calculate the quantization parameters. Finally, the graph was converted to a quantized GraphModule.

The quantization scheme followed the configuration supported by the Ethos-U quantizer. Weights were quantized using symmetric per-channel quantization, with a MinMax observer used to determine the quantization range. Activations were quantized using affine per-tensor quantization, with a HistogramObserver used during calibration. This matches the intended use of per-channel quantization for static weights and per-tensor quantization for dynamically produced activations.

The evaluated precision configurations were written as AxWy, where x is the activation bit-width and y is the weight bit-width. For example, A8W4 denotes 8-bit activations and 4-bit weights. Table 5.1 shows the integer ranges used for the supported precisions. The signed weight ranges omit one negative value in order to keep the range symmetric around zero.

Bit-width	Full-range weights	Full-range activations
INT4	[−7, 7]	–
INT8	[−127, 127]	[−128, 127]
INT16	–	[−32767, 32767]

Table 5.1: Integer quantization ranges for weights and activations. Weight ranges are reduced by one value to ensure true symmetry around zero, enabling further optimizations for symmetric quantization.

After quantization, the exported graphs were inspected using Model Explorer. This was used to verify that quantize/dequantize nodes were inserted in the expected locations, that the intended data types were used, and that modules selected for mixed precision or exclusion were handled correctly.

5.1.1 Mixed-Precision Quantization

The mixed-precision quantization process was guided by both previous work and our own analysis tools. Prior work on ViT quantization identifies operations such as LayerNorm and Softmax as sensitive to quantization. We used this as the starting point, and then used the node output inspector and Model Explorer to confirm which parts of the exported models caused large differences between the floating-point and quantized outputs.

In the implementation, MPQ was handled through the composable Ethos-U quantizer. Instead of applying one global quantization configuration to the whole graph, selected modules or operator patterns were assigned a different quantization configuration or excluded from quantization. The resulting model was then calibrated and converted in the same way as the uniformly quantized models. After conversion, the graph was inspected to confirm that the intended parts of the model had received the correct quantization treatment.

Several MPQ configurations were tested during development. The purpose was to identify which sensitive parts needed higher precision or floating-point execution in order to recover accuracy, while still keeping as much of the model as possible compatible with the Ethos-U deployment flow.

MPQ Process for ViT

For ViT-B/16, the main focus was LayerNorm. Initial uniform A8W8 and A8W4 quantization caused severe accuracy degradation, which indicated that some part of the model was highly sensitive to low-precision activation quantization. This matched prior work identifying LayerNorm as difficult to quantize in ViT architectures.

We therefore configured the Ethos-U quantizer to treat all LayerNorm modules differently from the rest of the model. In the most successful MPQ configurations, the linear layers and attention-related operations were quantized, while LayerNorm was kept in floating point. This allowed the rest of the model to benefit from integer quantization while avoiding the large error introduced by quantizing LayerNorm. Model Explorer was used to verify

that the LayerNorm modules were excluded or assigned the intended configuration, and accuracy testing was used to compare the resulting MPQ models against uniformly quantized baselines.

MPQ Process for Swin

For Swin V2-T, the most problematic part was not LayerNorm but the shifted-window attention mask and the Softmax-related computation. In shifted-window attention, the mask prevents tokens from attending across invalid window boundaries. This is done by adding large negative values, such as -100.0, to masked attention logits before Softmax. These values are intended to become approximately zero after Softmax, but they also create a wide activation range that is difficult to represent with low-precision quantization.

To handle this, we implemented a custom pattern matcher for the attention-mask computation. The pattern matcher identified the graph nodes involved in constructing and applying the attention mask, so that these nodes could be assigned a separate quantization configuration. We then used the composable Ethos-U quantizer to apply the selected configuration to the matched pattern while keeping the rest of the model quantized normally.

Several configurations were tested, including keeping the attention-mask and Softmax-related region in higher precision. Model Explorer was used to confirm that the matched nodes were correctly identified and that the intended quantization configuration was applied. The final choice was based on the resulting accuracy and whether the graph remained compatible with the Ethos-U deployment flow.

5.2 Rotation Methods

The rotation methods were implemented by inserting explicit rotation modules into the PyTorch model before export to ExecuTorch. Each rotation module represents a matrix multiplication with an orthogonal matrix $\boldsymbol{R}$. The same module can also apply the inverse rotation by multiplying with $\boldsymbol{R}^\top$, since $\boldsymbol{R}^{-1} = \boldsymbol{R}^\top$ for orthogonal matrices. This made it possible to use the same implementation for both forward rotations and compensating inverse rotations.

The implementation is shown in Listing 5.1. The rotation matrix is stored as a `torch.nn.Parameter`, which allows the same module to be used both for fixed random rotations and for later optimization. For random rotations, the matrix was initialized using the Hadamard-based rotation utilities from the QuaRot repository [44]. The module design was inspired by the SpinQuant implementation [45].

Listing 5.1: RotateModule class

```
class RotateModule(nn.Module):
    def __init__(self, R_init):
        super().__init__()
        self.weight = nn.Parameter(R_init.float())

    def forward(self, x, transpose=False):
        return x @ (self.weight.T if transpose else self.
            weight)
```

The `transpose` argument determines whether the module applies $\boldsymbol{R}$ or $\boldsymbol{R}^\top$. This was useful when inserting pairs of rotations that should cancel each other, or when merging one side of the transformation into a neighbouring weight matrix.

5.2.1 Inserting Rotations

Rotations were inserted by modifying the forward passes of the attention and MLP blocks. The insertion points were chosen so that the floating-point function of the model was preserved according to the computational invariance described in Section 2.6.4. In practice, this meant that when an activation was multiplied by $\boldsymbol{R}$, the corresponding linear weight was compensated by multiplying it with $\boldsymbol{R}^\top$, either explicitly during the forward pass or by merging the transformation into the weight matrix.

For activation rotations, the model code was modified by applying the corresponding `RotateModule` at selected points in the forward pass. For weight rotations, the linear weights were transformed so that the rotated activation basis was cancelled by the following linear operation. This allowed rotations to change the distributions seen by the quantizer without changing the output of the floating-point model.

After inserting the rotations, we verified the implementation before applying quantization. The original floating-point model and the rotated floating-point model were both evaluated on samples from the ImageNet-1K validation set, and their outputs were compared. This check was used to confirm that the inserted rotations preserved model behaviour up to numerical precision. Only after this equivalence check was the rotated model passed to the quantization flow.

Rotations in ViT

We added three different types of rotation to ViT, shown in Figure 5.1. The first rotation, $\boldsymbol{R}_1$, was used to rotate the main activation tensor before the attention operation and before the MLP block. In the original SpinQuant method, $\boldsymbol{R}_1$ can be applied as a fully offline rotation by keeping the residual stream in a rotated basis between encoder blocks. In ViT, this was not possible in the same way because LayerNorm is applied inside the encoder block. For this reason, $\boldsymbol{R}_1$ had to be inserted after LayerNorm, inside the attention and MLP paths. This preserved the model function but made part of the rotation an online operation.

The second rotation, $\boldsymbol{R}_2$, was inserted in the attention value path. It rotates the value tensor before the attention-value multiplication, while the inverse rotation $\boldsymbol{R}_2^\top$ is applied before the output projection. This follows the same role as in SpinQuant and is intended to reduce quantization error in the value/output-projection path. Since the head dimension can vary between blocks, $\boldsymbol{R}_2$ had to be created separately for each encoder block, while $\boldsymbol{R}_1$ could be shared across all ViT blocks.

The third rotation, $\boldsymbol{R}_3$, was added later and differs from $\boldsymbol{R}_1$ and $\boldsymbol{R}_2$ in both purpose and implementation. While $\boldsymbol{R}_1$ and $\boldsymbol{R}_2$ affect larger activation and weight paths in the attention and MLP blocks, $\boldsymbol{R}_3$ was added specifically to target the quantization of the query and key tensors inside the attention mechanism. It was inspired by the use of rotations for key-value cache quantization in LLM-oriented methods, where the key tensor can be particularly important because it is stored and reused during autoregressive decoding. However, ViT is an encoder model and does not use an autoregressive key-value cache in the same way as a

decoder-only language model. For this reason, R_3 was treated as a more experimental rotation in this work.

Unlike R_1 and R_2, R_3 was not merged into neighbouring weights. Instead, it was always applied online to the query and key tensors after the corresponding projections. This was necessary because R_3 is applied after the Q and K projections, directly before the attention-score computation. At this point there is no following linear weight matrix where the inverse rotation can be folded without also changing the dot-product attention calculation. R_3 therefore introduces an additional runtime operation and only affects activation quantization, not weight quantization. This also means that its expected benefit is more limited than for R_1 and R_2, which affect larger parts of the model and can improve the quantization behaviour of both rotated activations and adjacent weights.

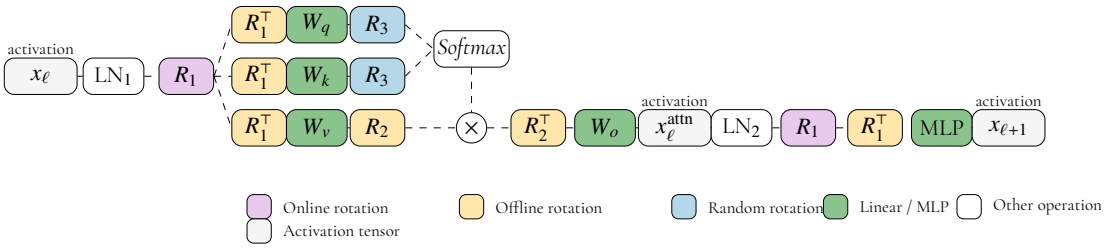


Figure 5.1: Rotation insertion points in the ViT encoder block and the MLP.

Rotations in Swin

We added both R_1 and R_2 to Swin, as shown in Figure 5.2. The role of R_1 was to rotate the activation tensor before the attention block and before the MLP block, similar to the ViT implementation. However, unlike ViT, R_1 could not be shared across all blocks. Swin uses a hierarchical architecture where the hidden dimension changes between stages as the spatial resolution is reduced and the channel dimension is increased. Because the rotation matrix must match the hidden dimension of the activation tensor it is applied to, a separate R_1 had to be created for each attention layer or stage where the hidden dimension differed.

For the attention block, R_1 was inserted before the query, key, and value projections. The corresponding inverse rotation, $R_1^\top$, was applied to the projection weights so that the floating-point function of the model was preserved. In the MLP path, R_1 was inserted before the first MLP projection, with the matching inverse rotation applied to the weight matrix of the first linear layer in the MLP. This follows the same computational-invariance principle as in ViT: the activation is represented in a rotated basis, while the neighbouring weights are transformed to compensate for the rotation.

The second rotation, R_2, was used in the value and output-projection path of the attention block. It rotates the value representation before the attention-value multiplication and applies the inverse rotation before the output projection. In Swin, this rotation could be implemented completely offline by folding the rotation and inverse rotation into the surrounding weight matrices.

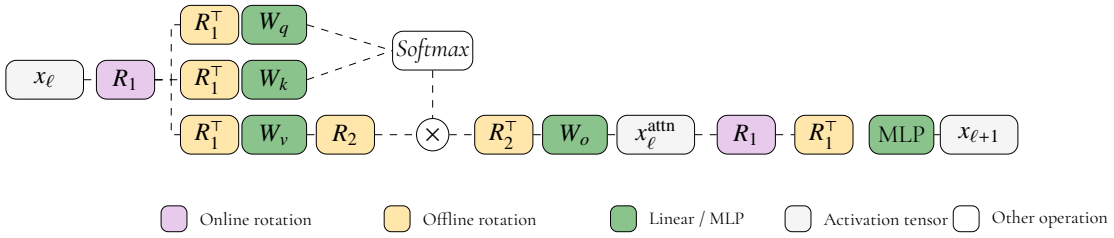


Figure 5.2: Rotation insertion points in the Swin attention and MLP blocks.

5.3 Optimizing Rotations

After implementing random Hadamard rotations, we created training versions of the models to optimize the inserted rotation matrices. The optimization followed the strategy described in Section 2.6.4: the original model weights were frozen, only the rotation matrices were trainable, and fake quantization was used to expose the rotations to simulated quantization error during training.

5.3.1 Quantization Simulation

To optimize rotations for a quantized model, simulated quantization was added to the training versions of the models. The Ethos-U quantizer provides support for quantization-aware training, but we chose to implement fake quantization directly in the PyTorch model code. This gave us more control over where quantization was applied. In particular, we wanted to simulate quantization only around the inserted rotations and the surrounding attention and MLP operations, rather than quantizing the entire model uniformly during training.

In fake quantization, values are quantized and dequantized during the forward pass, but the stored tensor datatype remains floating point. This means that the model experiences the rounding and clipping effects of quantization, while still allowing gradients to be propagated during training. However, the rounding operation is not directly useful for gradient-based optimization. If the quantizer is written as $q(x)$, then the chain rule gives

$$\frac{\partial L}{\partial x} = \frac{\partial L}{\partial q(x)} \cdot \frac{\partial q(x)}{\partial x}. \tag{5.1}$$

For a quantizer that includes rounding, $\frac{\partial q(x)}{\partial x}$ is zero almost everywhere. This would prevent gradients from flowing to the rotation matrices. As described in Section 2.6.4, the straight-through estimator (STE) addresses this by using the quantized value in the forward pass, but approximating the quantizer as the identity function in the backward pass [30]. In practice, this means that the backward pass treats

$$\frac{\partial q(x)}{\partial x} \approx 1. \tag{5.2}$$

Our first implementation used custom fake quantization functions with an STE-based backward pass. This worked for simple tests, but was difficult to tune for transformer activations. A single static activation range was often too harsh and caused excessive accuracy loss,

while a dynamic range computed from each input was too forgiving and did not resemble the static quantization used in the Ethos-U deployment flow.

We therefore implemented a second version using the `IntxFakeQuantizer` from the `torchao` package [46, 47]. This made it possible to define a fake quantization setup closer to the one used by ExecuTorch and the Ethos-U quantizer. Activations were simulated using static asymmetric quantization, while weights were simulated using static symmetric quantization. The bit-widths for activations and weights could be configured independently, which allowed us to train rotations under different simulated AxWy configurations.

Because the fake quantization was static, the training setup required two modes. First, the fake quantizers were placed in calibration mode and representative samples were passed through the model to collect ranges. After calibration, the quantizers were switched to fake quantization mode, where the collected ranges were used during training. This mirrored the post-training quantization flow used later for the Ethos-U quantized models, while still allowing gradients to update the rotation matrices.

5.3.2 Simulated Weight Quantization

During rotation optimization, simulated weight quantization was applied to the linear layers inside the attention block and the MLP. The fake quantization used a static, symmetric per-channel scheme, matching the weight quantization strategy used by the Ethos-U quantizer. The stored model weights remained in floating point, but during the forward pass the weights were replaced by fake-quantized versions. This allowed the training loop to measure the effect of quantized weights while still updating the rotation matrices with gradient-based optimization.

For the training models, rotations were not merged permanently into the weights before each optimization step. Instead, the relevant weight tensors were rotated during the forward pass and then fake-quantized. This ordering was important: the purpose of the rotations was to change the distribution of the tensor before quantization. If fake quantization had been applied before the rotation, the training would not capture whether the rotation improved the quantizability of the weights.

5.3.3 Simulated Activation Quantization

Simulated activation quantization was added to the attention block and the MLP. The goal was to expose the rotation matrices to activation quantization error during training, without having to quantize the entire model. This made the training setup more focused on the parts of the model affected by the inserted rotations.

The activation fake quantizers used static asymmetric quantization, following the same general form as the activation quantization used in the ExecuTorch and Ethos-U flow. Because the quantization was static, the fake quantizers first had to be calibrated on representative data. After calibration, the collected ranges were fixed and used during training.

The fake-quantization placement inside the MLP was the same for ViT and Swin, as shown in Figure 5.3. In this part of the model, fake quantization was inserted after the main operations in the MLP path. This made the MLP setup relatively straightforward compared with the attention blocks, where the models differ more.

The final activation quantization setup was chosen empirically. Quantization points were added until the simulated quantization caused a noticeable accuracy drop, but not so much that the model collapsed to near-zero accuracy. This was necessary because the fake quantization was only an approximation of the final Ethos-U quantization flow. If the simulation was too weak, the rotations would not learn to compensate for meaningful quantization error. If it was too harsh, the training signal became uninformative. The chosen setup should therefore be understood as a practical compromise rather than an exhaustive optimization of fake-quantization placement.

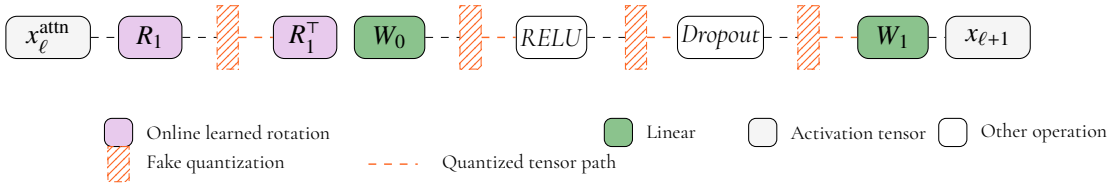


Figure 5.3: Fake-quantization points inside the MLP block.

ViT: Activation Quantization in the Attention Block

For ViT, fake activation quantization was added to the main tensor paths inside the attention block, as shown in Figure 5.4. The query, key, and value tensors were fake-quantized after their linear projections. The attention matrix after Softmax was also fake-quantized, as was the value tensor before the attention-value multiplication.

Not every operation in the attention block was quantized. In particular, the dot-product attention score computation was left in floating point in the fake-quantized training model. Quantizing this operation caused too large an accuracy drop during the simulation, making the training setup too unstable. The final placement was therefore chosen to introduce activation quantization error around the attention computation while still keeping the training problem feasible.

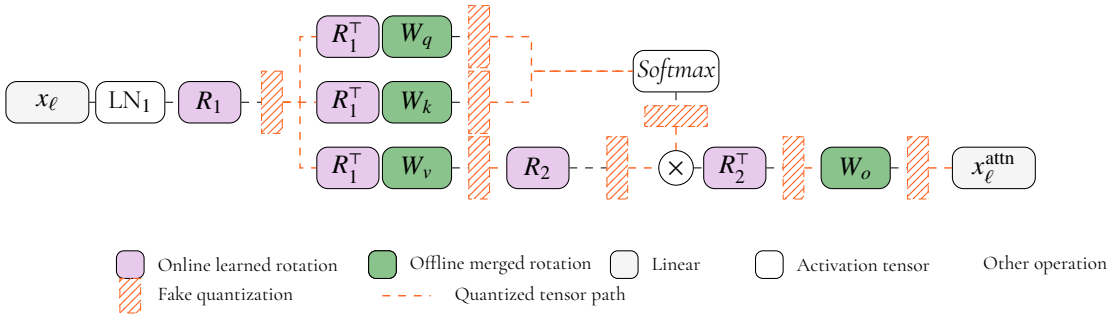


Figure 5.4: Rotation and fake-quantization insertion points inside the ViT attention block.

Swin: Activation Quantization in the Attention Block

For Swin, fake activation quantization was inserted into the shifted-window attention block, as shown in Figure 5.5. The same general principle was used as for ViT: quantization was added around the projected query, key, and value tensors and around the attention output

path. However, the Swin attention block was more sensitive to fake activation quantization because of the shifted-window attention mask.

The attention mask introduces large negative values before Softmax in order to suppress attention across invalid window boundaries. This creates a wide activation range and makes the Softmax-related path difficult to quantize. As a result, the fake-quantization placement for Swin had to be chosen more conservatively than for ViT. The goal was to expose the rotations to the quantization error caused by the attention block, while avoiding a simulation that was so severe that training could no longer recover useful accuracy.

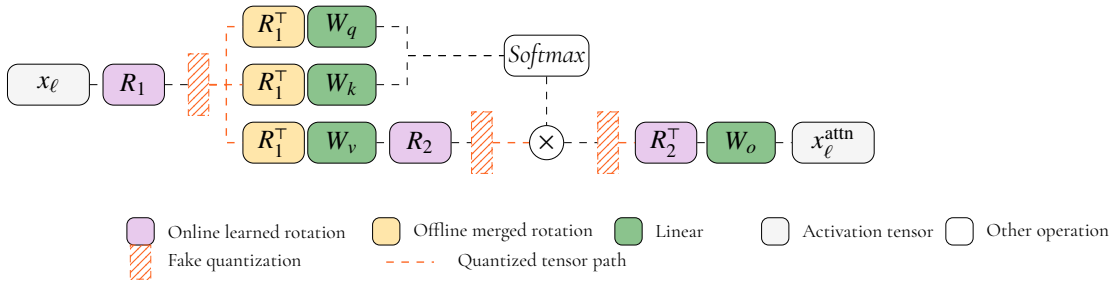


Figure 5.5: Rotation and fake-quantization insertion points inside the Swin attention block.

5.3.4 Training Setup Implementation

A model was prepared for training by inserting new randomly initialized rotation matrices at the same locations used for the random-rotation experiments. These matrices were stored as trainable parameters, while all original model parameters were frozen. This ensured that training could only change the rotations and not the underlying pre-trained model.

Before training, the fake quantizers were placed in calibration mode. A calibration subset was passed through the model to collect activation ranges for the static fake-quantization setup. After calibration, the fake quantizers were switched to quantization mode, and the collected ranges were kept fixed during training.

The training and evaluation subsets were generated from the ImageNet-1K validation set. In most attempts, an 80/20 split was used between training and validation samples. The validation subset was used to monitor whether the optimized rotations improved classification accuracy under fake quantization.

5.3.5 Training Loop

The training loop was implemented by subclassing the `Trainer` class from the HuggingFace `transformers` package. We used this to create a custom `SpinQuantTrainer` that trained only the inserted rotation parameters.

The rotations were optimized using Cayley SGD, following the SpinQuant implementation. Cayley SGD was used because the rotation matrices must remain orthogonal during training. If this constraint is not preserved, the inverse rotation can no longer be represented by the transpose of the matrix, which breaks the computational invariance used when inserting the rotations. The optimizer implementation was taken from the SpinQuant repository [45], and the method is based on efficient optimization on the Stiefel manifold [31].

Cross-entropy loss was used, since the task is image classification. During each training step, the model performed a forward pass with fake quantization enabled. The loss was computed from the model logits and the ImageNet labels, and the backward pass updated only the rotation matrices.

Listing 5.2: Training Loop

```
class SpinQuantTrainer(Trainer):
    def __init__(self, *args,
                 rotation_params=None, calibration=None, **
                    kwargs):
        super().__init__(*args, **kwargs)
        self.rotation_params = tuple(rotation_params or [])
        self.calibration = calibration

    def compute_loss(self, model, inputs, **_):
        x = inputs["x"].to(next(model.parameters()).device)
        y = inputs["labels"].to(x.device)
        return F.cross_entropy(model(x), y)

    def prediction_step(self, model, inputs,
       prediction_loss_only, **_):
        x = inputs["x"].to(next(model.parameters()).device)
        y = inputs["labels"].to(x.device)
        return (loss, logits.detach(), y.detach())
```

5.3.6 Training Evaluation

The training attempts were evaluated in two ways. First, accuracy was measured during training on the validation subset with fake quantization enabled. This showed whether the rotation matrices were improving the simulated quantized model. Second, after training, the optimized rotations were inserted into the deployment model and quantized using the Ethos-U quantizer. This second evaluation was necessary because the fake-quantized training setup only approximated the final Ethos-U quantization flow.

In the first training attempts, only the loss was monitored. This was not sufficient for judging training progress, because Cayley SGD produced larger loss fluctuations than standard unconstrained SGD. Accuracy on the validation subset was therefore added as the main indicator of successful training. A training attempt was considered promising if validation accuracy improved under fake quantization and if the resulting rotations also improved, or at least did not degrade, the accuracy after Ethos-U quantization.

5.3.7 Training Attempts

Several training attempts were carried out to find a usable optimization setup for the rotation matrices. All attempts shown in Table 5.2 used static fake quantization with 4-bit simulated weights and 8-bit simulated activations. This setting was chosen because rotations were expected to be most useful when weight quantization was aggressive, and because the

static fake-quantization setup most closely matched the later post-training quantization flow used with the Ethos-U quantizer.

The initial training attempts used the SpinQuant configuration as a reference point, including a learning rate of 1.5 and a cosine learning-rate scheduler [45]. We first tested this setup on a small training subset of 100 samples to verify that the training loop, fake quantization, and rotation updates worked as expected. The same configuration was then tested with a larger training subset of 1000 samples. Since the validation accuracy did not improve reliably with this setup, we also tested a lower learning rate of 1.0 while keeping the same number of epochs and the cosine scheduler. The Swin experiment used the same lower learning rate, since the ViT experiments suggested that the original learning rate was not necessarily suitable for our models and fake-quantization setup.

The fake-quantization setup was kept static across the attempts in Table 5.2. Before training, the fake quantizers were calibrated on representative data and then switched to quantization mode for optimization. This ensured that the rotations were trained under fixed quantization ranges, similar to the static activation quantization used in the final Ethos-U deployment flow.

Table 5.2 summarizes the main training attempts. The table should be interpreted as an overview of the explored configurations rather than an exhaustive hyperparameter search.

Table 5.2: Training configurations used for rotation optimization.

Model	W bits	A bits	Epochs	LR	Train size	Fake quant
ViT	4	8	4	1.5	100	Static
ViT	4	8	4	1.5	1k	Static
ViT	4	8	4	1.0	1k	Static
Swin-T	4	8	4	1.0	1k	Static

The resulting trained rotations were used in the final accuracy evaluation and compared against both non-rotated models and models using random Hadamard rotations.

Chapter 6
Evaluation

The evaluation phase consisted of accuracy testing and hardware testing.

6.1 Accuracy Testing

The accuracy testing strategy was designed to enable the reliable comparison of model accuracy scores between the different configurations we created, as well as results from other related works. Top-1 and top-5 accuracy tests from the Torchmetrics package were used as the primary performance metrics. Results of the accuracy tests allowed us to quickly detect significant problems in a specific setup, such as unexpected low accuracy due to some module not tolerating a low activation precision, or integer conversion penalties occurring during mixed-precision. We used both custom made and existing tools (such as the model explorer, model inspector, and heat mapping, as described in 4.3) to further investigate these issues.

6.1.1 Dataset Generation and Sample Size

The validation split of the ImageNet-1K [48] dataset available through Huggingface was used for all accuracy testing. To make sure that all calibration, training and accuracy testing was performed on the same set of samples we created a function that split the first 112 images to calibration, the next 1224 to training, and the rest to the testing. The first two sets remained constant throughout the evaluation phase, and only the size of the testing set was changed on occasion. Initial accuracy tests and baselines were conducted on a subset of 1000 or 1224 samples to narrow down the most promising setups for each model. Later tests used a larger number of samples (5224) for improved reliability. The sample size was chosen because an accuracy test ran in floating point was 0.8613, which was very close to the accuracy that was reported on Torchvision for the weights that we used.

6.1.2 Testing Strategy

For each relevant quantization and rotation configuration, a benchmark was run on an unrotated version of that model. Rotations were tested both individually and in combination, as well as random and trained. To determine the variance in performance between random Hadamard rotations, tests were performed multiple times with otherwise static configurations. Other parameters that could affect results that would be directly compared were frozen, such as model mode, calibration, test set size and content. Tests were also conducted on models with fake quantization enabled, to serve as indicator of similarity to the corresponding Ethos-U85 quantized model.

6.1.3 Random rotations

When running accuracy tests with random rotations, there is a variation in performance between different rotations. To determine if an optimized rotation outperforms most of random rotations with a specified confidence interval, enough runs and samples are needed. For a 95% confidence interval with margin of error $m = 0.01$:

- n the number of samples per run,
- x the number of independent runs,
- k the number of correctly classified samples,
- $\hat{p} = \frac{k}{nx}$ the observed top-1 accuracy

Assuming independent Bernoulli trials, the standard error is

$$\mathrm{SE}(\hat{p}) = \sqrt{\frac{\hat{p}(1-\hat{p})}{nx}}$$

Using normal approximation, the two-sided $100(1-\alpha)\%$ confidence interval is

$$\hat{p} \pm z_{\alpha/2}\sqrt{\frac{\hat{p}(1-\hat{p})}{nx}}$$

With a confidence interval of 95%, $z_{\alpha/2} = 1.96$, and the variance maximized at $\hat{p} = 0.5$:

$$\mathrm{MoE}_{\max} = \frac{0.98}{\sqrt{nx}}$$

With $x = 1000$ samples, and $m = 0.01$, the number of runs x may be derived from the upper bound of the margin of error:

$$m = \frac{0.98}{\sqrt{nx}} \Rightarrow x = \frac{1}{n}\left(\frac{0.98}{m}\right)^2 = \frac{1}{1000}\left(\frac{0.98}{0.01}\right)^2 = 9.604$$

Therefore, at least **10** runs are required for this setup.

These tests were run on the most promising quantization setups (A16W4, A8W8, and A8W4, the latter two with LayerNorm in fp32), with random R1+R2, as well as trained (ie

static) R1+R2 together with random R3. The motivation for the first combination was that R1 and R2 were trained together, and results from testing R1 and R2 individually would not be directly comparable to any results using both R1 and R2. Having two random parameters in one test instead of isolating one may affect the variance in the results and does not allow for conclusions about the individual contribution of R1 versus R2, but it provides a fairer comparison to the jointly trained R1+R2 configuration. Though individual contributions of R1 and R2 is also of interest, they were ruled a lower priority. As a single accuracy test of a quantized module on 1000 samples takes at least one hour with the available hardware, a minimal of 10h additional computing time was needed for each configuration. The motivation behind the second configuration of trained R1+R2 combined with random R3 is also that it allows for direct comparison to results using the same trained R1 and R2, while keeping R3 as the isolated varying factor.

6.2 Hardware Performance Test Strategy

The hardware performance tests were designed to generate metrics about memory usage and computation performance in order to gain an understanding how our quantization efforts affected model inference on the Ethos-U85 NPU.

6.2.1 Hardware Simulation

Memory Metrics

The Vela compiler provides a report of memory usage that we have used in our study. This report includes information about the memory mode and various statistics about bandwidth usage and size of used memory. We used this report to compare how the different configurations affected memory usage.

The following configurations were evaluated for both Dedicated and Shared SRAM memory configurations:

Precision	Variants
A16W8	No rot, Rot
A8W8	No rot, MPQ No rot, MPQ Rot
A8W4	No rot, MPQ No rot, MPQ Rot

Table 6.1: Evaluated configurations. "Rot" denotes use of rotations.

Our focus was to compare configurations where all impacting factors were frozen except one, more specifically the ones presented in 6.2.

Comparison axis	Values
Activation precision	A16 vs. A8
Weight precision	W8 vs. W4
Quantization scheme	All modules quantized vs. MPQ
Rotation	No rot vs. Rot

Table 6.2: Design dimensions compared across the evaluated configurations.

The configuration A16W4 that used in accuracy tests were replaced by A16W8 due to compiler constraints, but by comparing A16W8 with A8W8 we can still isolate the impact of lowering the activation precision, and similarly the weight precision impact can be isolated by comparing A8W8 with A8W4.

For MPQ configurations, the graph is split into subgraphs to be executed on either the CPU or the NPU. In this case, the metrics have been collected and processed for all NPU-delegated graphs, for example by being summarized or by having a maximum value extracted.

Metric	Description
Peak SRAM/DRAM used (KiB)	Peak on-chip/off-chip memory required to run the graph.
W SRAM/DRAM BW (MB/inf.)	Weight-related memory traffic per inference.
Total SRAM/DRAM BW (MB/inf.)	Total memory traffic per inference.
NPU ops	Operations executed on the NPU.
MACs	Multiply–accumulate operations.

Table 6.3: Metrics used to compare the evaluated configurations.

To compare a specific metric between two configurations that differ in only one factor, we mainly consider the relative change. However, relative changes across different metrics should not be compared without also considering their absolute magnitudes. For example, a 10% increase in DRAM usage is not necessarily comparable to a 10% increase in SRAM usage, since the corresponding absolute changes in MB may be very different. When comparing different metrics, or configurations that differ in multiple factors, we therefore consider the absolute magnitudes.

Ethos-U Memory Modes

The chosen Ethos-U memory mode directly affects inference latency, SRAM and DRAM requirements, and whether a model can be deployed at all, so performance results must be interpreted in terms of the memory placement assumptions used during compilation and execution.

The Ethos-U NPU provides two memory interfaces: low-latency on-chip memory, typically SRAM, and higher-latency external memory, typically Flash or DRAM. During inference, the compiler uses a **scratch buffer** for intermediate tensors, a **neural network region** for weights and constants, and an optional **fast scratch buffer**. The placements of these regions determine three possible memory modes: SRAM-Only, Shared-SRAM, and Dedicated-SRAM, which provide different trade-offs between performance, SRAM usage, and model size support:

- **SRAM-Only** provides the highest performance since both intermediate tensors and neural network weights reside entirely in on-chip SRAM, but requires the model to be sufficiently small.
- **Shared-SRAM** stores intermediate tensors in SRAM while weights remain in external memory, balancing performance and SRAM usage. This mode requires sufficient SRAM to hold the largest intermediate tensor. The external memory is Read-Only, the on-chip memory interface is Read/Write.
- **Dedicated-SRAM** is commonly used for running very large models. Both tensors and weights reside in external memory, while dedicated SRAM is utilized as a cache. This mode enables support for models that exceed SRAM capacity but depends on high-bandwidth external memory for good performance.

Using SRAM-Only is not realistic in our case, and performance with this memory mode is therefore not evaluated.

Computation Metrics

The FVP provides a summary of collected hardware usage metric estimates, presented in 6.4. The FVP NPU performance timings are not perfectly cycle accurate but are within 10% of the expected execution timings on a physical board. As FVP runs are time costly, we derived a minimal list of 5 required runs to be able to evaluate the design dimensions in 6.2 with regard to the computational metrics.

Metric category	Metrics
Delegation behaviour	NPU delegations
Runtime cost	Delegated cycles, end-to-end inference runtime, CPU wait for NPU
PMU memory counters	SRAM read/write data beats, external read/write data beats
PMU activity counters	NPU idle, MAC active, weight decoder active

Table 6.4: Overview of FVP profiling metrics used to support the performance analysis.

The delegation metrics indicate how many graph regions are offloaded to the Ethos-U NPU in the case of MPQ, while the runtime-cost metrics describe the execution cost observed by the runtime and host processor. The PMU counters are low-level hardware event counters provided by the Ethos-U Performance Monitoring Unit. The memory-related PMU counters indicate whether memory traffic is mainly served by on-chip SRAM or external memory (DRAM), while the activity-related PMU counters describe how NPU execution is distributed between idle time, MAC computation, and weight decoding. Together, these metrics are used to identify what performance is primarily affected by.

6.3 Results

Here we present the results of our tests with focus on rotation and MPQ impact on accuracy, and performance in terms of cycles, estimated inference time, NPU utilization, and memory use. The ViT model dominates the reported results as it was the first model to be investigated and was deemed a higher priority, both due to time constraints and the Swin model lacking rotation-introduced improvement. Swin is hence absent from the performance results subsection.

6.3.1 Rotation impact on accuracy

There were large differences in how useful the rotations were between the models as well as the different quantization schemes and configurations. First we present the accuracy results from introducing randomized hadamard rotations, then trained rotations, and lastly how the fake quantization implementations performed compared to the real quantization. The last mentioned was done in order to evaluate the reliability of the quantization simulation and hence the quality and applicability of the trained rotations when it comes to real quantization.

MPQ was essential for some quantization configurations to produce any useful results, as presented in 6.3.2, and combinations of MPQ and rotations were therefore included in the evaluated rotation setups. In the tables below, setups labeled 'MPQ' marks Mixed-Precision Quantization as follows:

Model	Config	MPQ
ViT	A8W8	LayerNorm FP32
	A8W4	LayerNorm FP32
Swin	A8W8	Attn mask + softmax in A16W8
	A8W4	Attn mask + softmax in A16W4

Table 6.5: Mixed-precision quantization exceptions used for each model and configuration.

Randomized rotations in ViT

Here we present the results from running multiple accuracy tests on 1000 samples with random rotations in ViT, as described in 6.1.3. The $A8W8$ configuration is omitted from the $R3$ evaluation presented in 6.8, since the $R3$ rotation only affects activations and not weights.

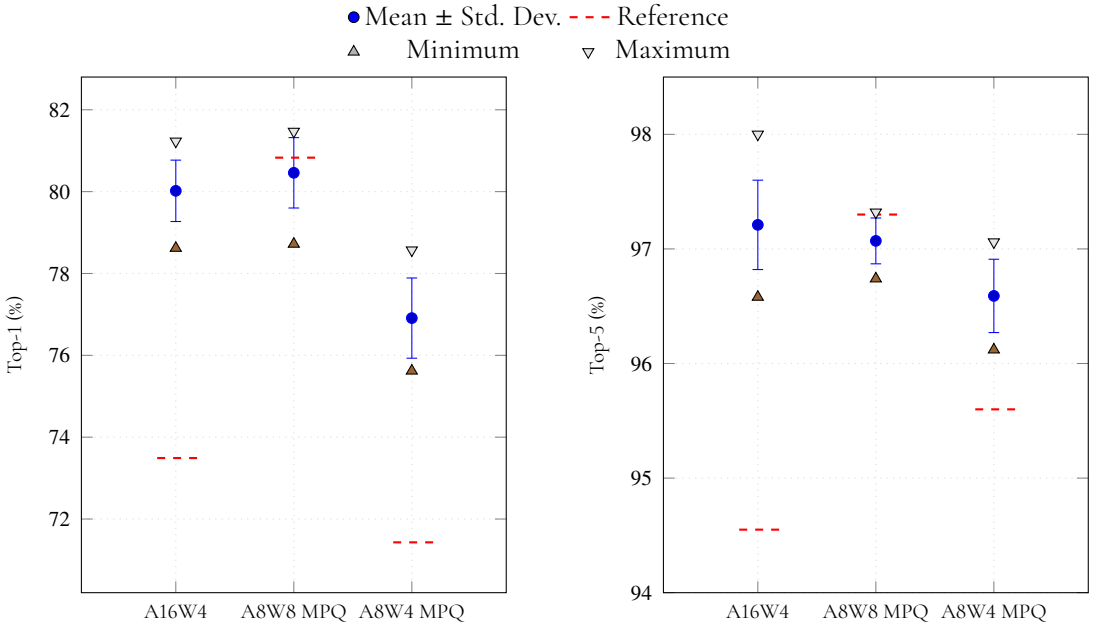
Config	Ref	Min	Δ Ref	Max	Δ Ref
A16W4	73.49	78.62	+5.13	81.23	+7.74
A8W8 MPQ	80.83	78.72	-2.11	81.47	+0.64
A8W4 MPQ	71.43	75.62	+4.19	78.57	+7.14

Table 6.7: Minimum and maximum Top-1 accuracy over 11 runs with randomized R1 and R2 rotations, compared with the reference accuracy without randomized rotations. Δ Ref denotes the difference relative to the reference accuracy.

Config	Metric	Ref	Mean	Δ Ref	Std. Dev.	95 CI
A16W4	Top-1	73.49	80.02	+6.53	0.75	[79.58, 80.46]
	Top-5	94.55	97.21	+2.66	0.39	[96.98, 97.44]
A8W8 MPQ	Top-1	80.83	80.46	-0.37	0.86	[79.95, 80.97]
	Top-5	97.30	97.07	-0.23	0.20	[96.95, 97.18]
A8W4 MPQ	Top-1	71.43	76.91	+5.48	0.98	[76.33, 77.49]
	Top-5	95.60	96.59	+0.99	0.32	[96.41, 96.78]

Table 6.6: Accuracy statistics over 11 runs on 1000 samples with randomized R1 and R2 rotations. Δ Ref denotes the difference between the mean accuracy and the reference accuracy. Std. Dev. denotes standard deviation for the mean.

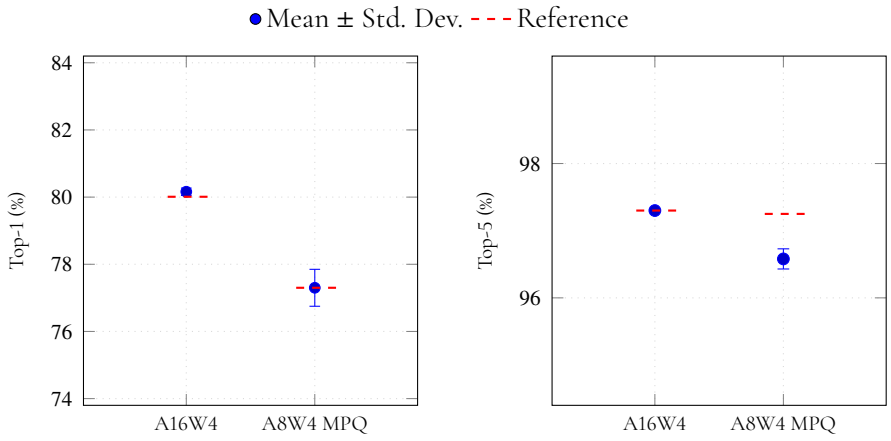
The results in 6.1 show that randomized rotations can substantially improve accuracy for configurations where the reference accuracy is lower, as can be seen for A16W4 and A8W4 MPQ - in these cases, even the worst rotations improves accuracy with several percent points, as can be seen in 6.7. A8W8 MPQ already has a strong reference accuracy and random rotations actually has a negative effect on average, although the best randomized run slightly exceeds the reference. Seemingly, the effect of R1 and R2 rotations becomes more pronounced at lower weight precision, which is not completely unexpected. The benefit of rotations is often more stable for weight quantization since weights are fixed, while activations depend on the input. A rotation can consistently reduce outliers in a weight matrix, but activation outliers may change from sample to sample and may not be equally well handled by the same rotation. Further, the standard deviation for Top-1 accuracy increases with harsher quantization, indicating that lower-precision configurations may be more sensitive to the particular rotation used. The same pattern is not observed for Top-5, which might be explained by Top-5 being less sensitive to small changes in the ranking.



(a) Top-1 accuracy

(b) Top-5 accuracy

Figure 6.1: Accuracy statistics over 11 runs on 1000 samples with randomized R1 and R2 rotations. Points indicate mean accuracy and bars show the standard deviation. Dashed markers indicate the reference accuracy.



(a) Top-1 accuracy

(b) Top-5 accuracy

Figure 6.2: Accuracy statistics over 10 runs on 1000 samples with randomized R3 rotational embeddings. Points indicate mean accuracy and bars show the standard deviation. Dashed markers indicate the reference accuracy.

As visualized in 6.2, the impact of R3 rotations were near none-existing, and even negative in the case of A8W4 MPQ Top-5. As R3 rotations are meant to combat outliers in KV-cache quantization inside the attention mechanism, and the ViT does not rely on an autoregressive

Config	Metric	Ref	Mean	Δ Ref	Std. Dev.	95 CI
A16W4	Top-1	80.01	80.16	+0.15	0.11	[80.09, 80.23]
	Top-5	97.30	97.30	0.00	0.00	[97.30, 97.30]
A8W4 MPQ	Top-1	77.30	77.30	0.00	0.55	[76.95, 77.64]
	Top-5	97.25	96.58	-0.67	0.24	[96.43, 96.73]

Table 6.8: Accuracy statistics over 10 runs on 1000 samples with randomized R3 rotations. Std. Dev. denotes standard deviation for the mean.

KV cache in the same way as a language model, the benefit of improving key quantization should also be smaller. In contrast, R1 and R2 affect larger parts of the model and therefore address more of the quantization error.

Trained rotations

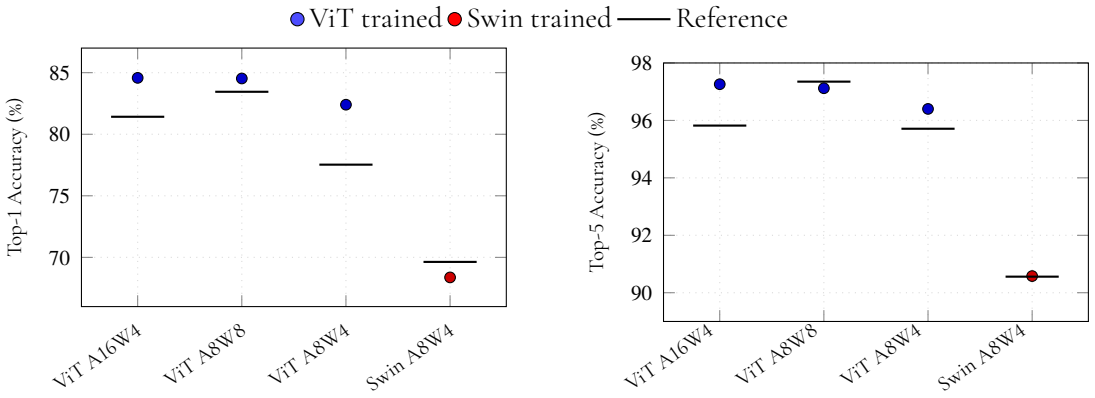
Several implementations of the training were made, as described in 5.3. The last version of the training was performed with A8W4 quantization, and resulting rotations were inserted into models which were then quantized to the below presented configurations. In ViT, the results for A16W4 and A8W8 are therefore not for rotations trained on that specific configuration.

Model	Config	Top-1 Ref	Top-1	Δ Ref	Top-5 Ref	Top-5	Δ Ref
ViT	A16W4	81.42	84.58	+3.16	95.82	97.26	+1.44
	A8W8 MPQ	83.45	84.53	+1.08	97.35	97.12	-0.23
	A8W4 MPQ	77.53	82.40	+4.87	95.71	96.40	+0.69
Swin	A8W4 MPQ	69.63	68.37	-1.26	90.56	90.58	+0.02

Table 6.9: Accuracy statistics on 5224 samples with R1 and R2 trained on A8W4.

6.3 and 6.9 show that the trained R1 and R2 rotations improve Top-1 accuracy close to the ViT FP baseline of 86.13% on the same sample set for all configurations, suggesting that the rotations are somewhat generalizable beyond the A8W4 MPQ setting they were trained for. The results for Top-5 accuracy follows the same reasoning of lower sensitivity as in 6.1.3, where rotation may help the highest-ranked prediction while not uniformly improving the broader top-five ranking.

The trained rotations did not improve the Swin model accuracy, but rather decreased it by 1.26%. Being a relatively small deduction, this could be due to failed training.6.10 shows that the trained rotations introduced a larger negative impact compared to a set of random R1 and R2 rotations, confirming that training was not optimal. However, as both worsened the accuracy, the model might simply not be benefiting from rotations. The training and fake quantization for Swin will be further evaluated in the next section.



(a) Top-1 accuracy

(b) Top-5 accuracy

Figure 6.3: Accuracy comparison between trained R1 and R2 rotations and reference values across quantization configurations run on 5224 samples.

As previously described, several runs with randomized rotations were done, often with fewer samples in order to quickly gain an understanding of how a configuration would behave. However, two singular runs of random rotations on 5224 samples were done and are presented in 6.10. Interestingly enough, these random rotations produced better results than the trained versions (though the ViT rotations were trained for A8W4 quantiztion, not A16W4), confirming that the training was either not successful in finding the best rotations, or that optimal rotations for the A8W4 configuration are not also optimal for the A16W4 configuration.

Model	Rotation	Top-1 Ref	Top-1	Δ Ref	Top-5 Ref	Top-5
ViT A16W4	Random	81.42	84.81	+3.39	95.82	97.32
	Trained*	81.42	84.58	+3.16	95.82	97.26
Swin A8W4 MPQ	Random	69.62	69.25	-0.37	90.56	90.78
	Trained	69.62	68.37	-1.25	90.56	90.58

Table 6.10: Accuracy on 5224 samples with randomized and trained R1 and R2 rotations. Δ Ref denotes the Top-1 accuracy difference relative to the baseline without rotations. Trained* marks use of rotations trained for A8W4 instead of A16W4.

Simulated Quantization

Running accuracy tests on models with fake quantization enabled is important for understanding how effective the training itself was. The baseline result for ViT fake quantization matched the Ethos-U85 quantization results closely, while the difference for the Swin model was a substantial one of about 43%, showing that the Swin fake quantization did not mimic Ethos-U85 quantization accurately. The results in 6.11 show that randomized rotations helped improve the accuracy also for the fake quantization, and that training brought

the gain further. This was especially true for the Swin model, though the low baseline for fake quantization left room for these large improvements.

Worth noting is that only one test was performed on each randomized rotation, meaning it does not represent the average effect introduced by a random rotation.

Model	Quantizer	Rotation	Top-1	Top-5
ViT	Ethos-U85	None	77.53	95.71
	Fake	None	77.10	93.60
		Random	80.10	95.00
		Trained	81.30	95.70
Swin	Ethos-U85	None	69.63	90.88
	Fake	None	26.03	48.64
		Random	30.13	52.89
		Trained	49.96	75.05

Table 6.11: A8W4 Accuracy on 5224 samples under Ethos-U85 and Simulated ("fake") Quantization schemes. Ethos-U85 rows use no R1/R2 rotations, fake quantization rows compare different R1/R2 rotation setups.

6.3.2 Mixed-Precision Quantization without rotation

This subsection presents the MPQ results for models without rotations.

We observed large accuracy differences between the tested MPQ exceptions. For ViT, the baseline A8 configurations without exceptions collapsed to near-zero accuracy, which shows that uniform low-precision quantization was not viable for this model in our deployment flow. Excluding LayerNorm from quantization recovered most of the lost accuracy, reaching 83.45 Top-1 for A8W8 and 77.53 for A8W4. Excluding everything but the linear layers recovered even more accuracy, but was less attractive as a practical deployment strategy because it leaves a much larger part of the model outside the intended low-precision path.

These results identify LayerNorm as the dominant source of quantization error in ViT and explain why the later ViT experiments use LayerNorm FP32 as the MPQ configuration. It provided a large accuracy recovery while remaining closer to the intended deployment setting than keeping the linear layers in FP32. For Swin, the effect of MPQ was much smaller: keeping the attention mask and Softmax in higher precision improved A8W8 accuracy only marginally and did not improve A8W4. This suggests that Swin's quantization bottlenecks were not isolated to the same operations as in ViT, and that a more model-specific MPQ strategy would have been required.

Model	Config	Exception	Top-1	Top-5
ViT	A8W8	None	< 0.03	< 0.06
		LayerNorm FP32	83.45	97.35
		All except Linear FP32	85.81	97.54
	A8W4	None	< 0.03	< 0.06
		LayerNorm FP32	77.53	95.71
		All except Linear FP32	85.62	97.64
Swin	A8W8	None	66.24	93.25
		Attention mask + softmax FP32	67.53	93.63
	A8W4	None	69.63	90.88
		Attention mask + softmax A16W4	69.63	90.60

Table 6.12: Accuracy with mixed-precision exceptions and no rotations.

6.3.3 Performance results

In this section we present effects on system-level metrics, using data from the process of lowering an exported model to the Ethos-U85 edge dialect. The affecting factors we look at are activation precision, weight precision, MPQ, and online rotations. The main purpose of this is to investigate for example if lowering weight or activation precision actually decreases resource use and improves performance, and if MPQ and online rotations introduce overhead and memory-related penalties. Since the rotation impact evaluations showed that R3 rotations did not improve accuracy, only R1 and R2 are included when we denote that a configuration has rotations.

How to read the tables: The horizontal bars in the tables below show the increase or decrease of a metric in percent when changing a configuration. Example: If Peak SRAM used goes from 23500 KiB to 11300 KiB when changing A16 to A8 (W8, with no rotations and using Dedicated SRAM), that means the Peak SRAM used was approximately halved, ie a change of about -50%. This would be represented by a negative bar of the magnitude 50.

The inference times are derived from total cycles, assuming a clock frequency of 0.1 GHz, i.e., 1 ms = 100,000 cycles, and are only estimates.

Activation precision

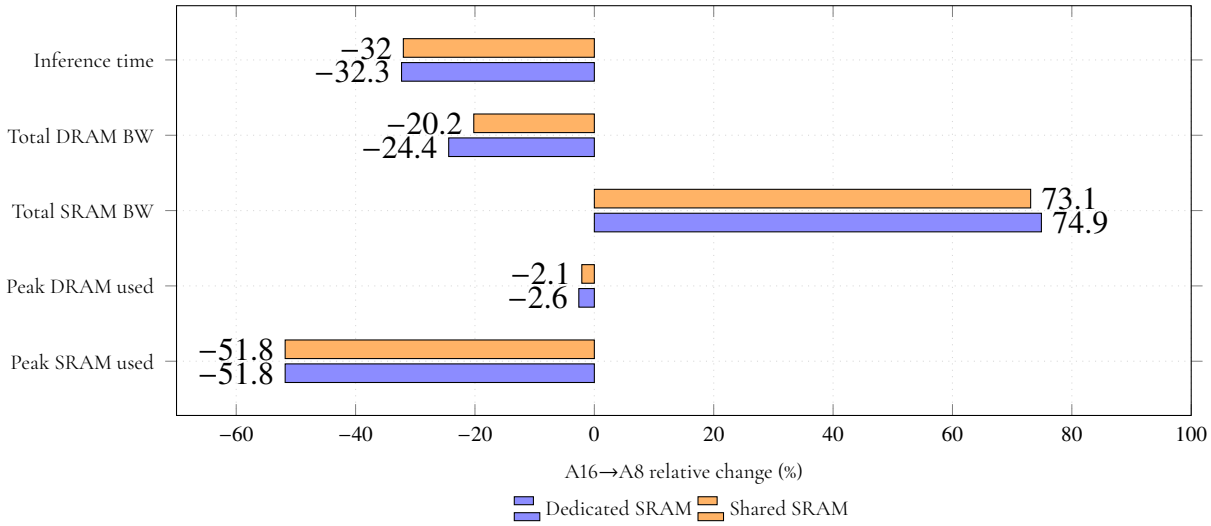
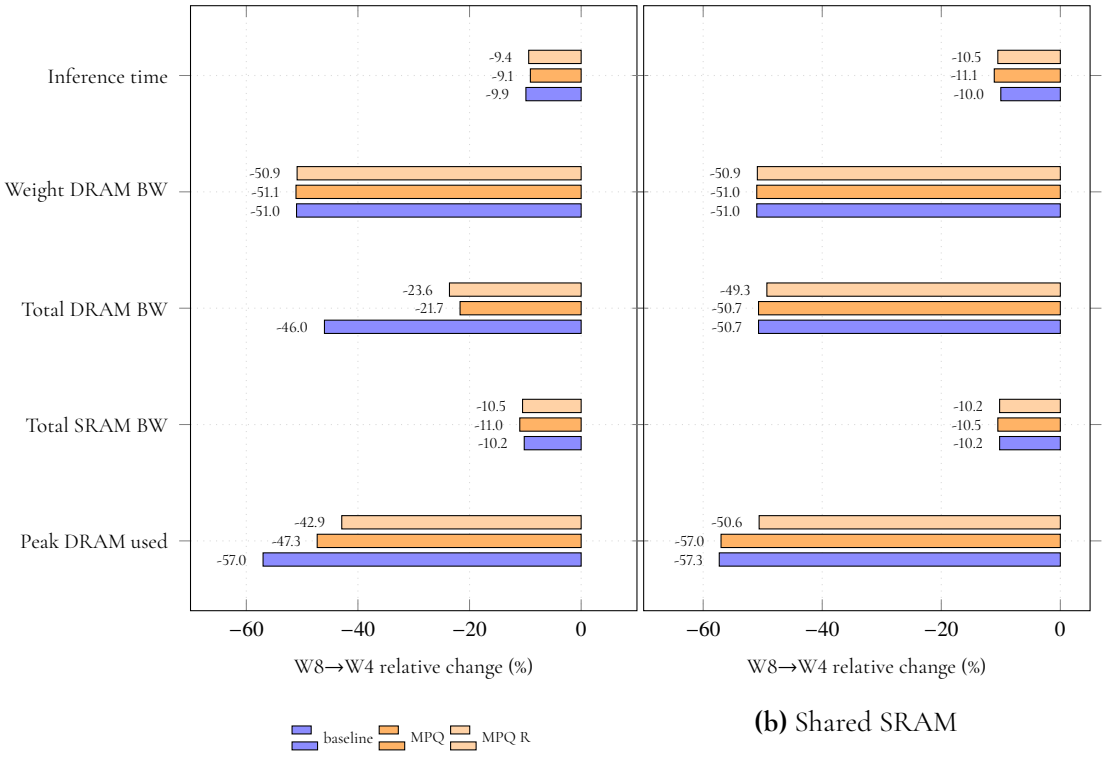


Figure 6.4: A16→A8 relative hardware metric changes under Dedicated SRAM and Shared SRAM. Negative values indicate reductions relative to the corresponding A16W8 configuration. Bandwidth is reported in MB/inference, peak memory usage in KiB, and inference time as relative change.

Reducing activation precision from 16 to 8 bits means that peak SRAM size could be halved for both memory modes. This result makes sense as the intermediate tensors that are stored in SRAM are halved. In Shared SRAM mode the DRAM bandwidth decreased by 20 percent, meaning that the amount of memory accesses to fetch weights was smaller. In dedicated mode the decrease was almost one-fourth, which is reasonable since the bandwidth also includes writes to scratch buffer that require less bytes. SRAM bandwidth increased significantly, and this change seems to be caused by a higher rate of inputs being read to SRAM. In both memory modes there was a 30% improvement in inference time which is to be expected as less data needed to be moved from DRAM, and SRAM could be utilized more.

Weight precision



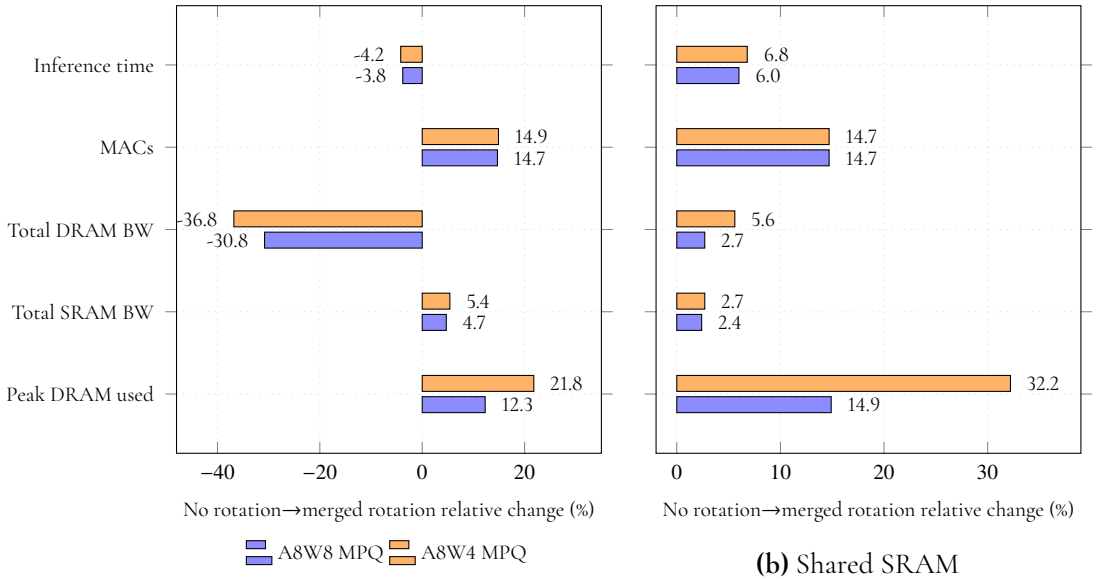
(a) Dedicated SRAM

(b) Shared SRAM

Figure 6.5: W8→W4 relative hardware metric changes under Dedicated SRAM and Shared SRAM. Negative values indicate reductions relative to the corresponding W8 configuration. Bandwidth is reported in MB/inference, peak memory usage in KiB, and inference time as relative change.

The W DRAM BW metric highlights the reduction in weight-related DRAM traffic, which was to be expected as weights are stored in DRAM in both memory modes. These results show how reducing the weight precision not only affects how much memory is needed to store them, but how it also has a significant positive impact on performance during inference that is the result of halving the DRAM bandwidth for weights. It's also noteworthy that in Dedicated SRAM mode when MPQ is used the weights account for a smaller part of the used bandwidth, leading to a lower reduction in total DRAM bandwidth. However, the positive impact on inference time was still very similar to the non-MPQ model. SRAM use remained consistent between weight configurations and the bandwidth reduction is likely caused by the pre-fetched weight tensors that are stored there.

Online rotations



(a) Dedicated SRAM

(b) Shared SRAM

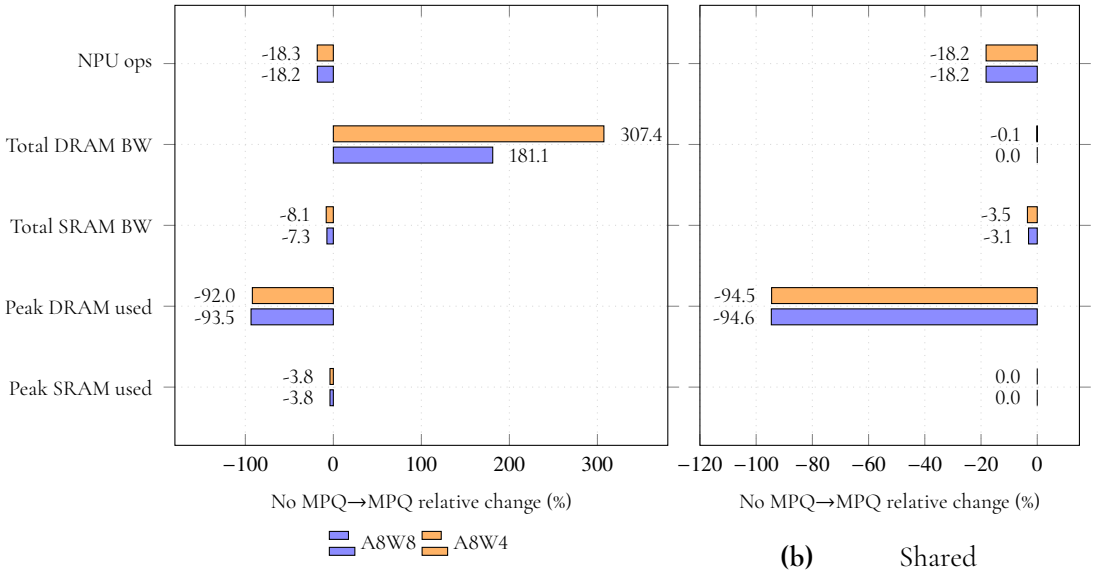
Figure 6.6: No rotation→merged rotation relative hardware metric changes under Dedicated SRAM and Shared SRAM. Negative values indicate reductions relative to the corresponding MPQ no-rotation configuration. Bandwidth is reported in MB/inference, peak memory usage in KiB, MACs as operation count, and inference time as relative change.

The results of 6.6 show that the online rotations introduce a 15% increase in the amount of MACs that are performed during inference. In both modes there is also an increase in the amount of DRAM that is required, and for the Shared SRAM mode the bandwidth required is increased. There is however a large decrease in DRAM bandwidth for the Dedicated SRAM mode, as well as an increase of SRAM usage and inference time. This is counter-intuitive because rotations add computation and extra operators, the expected outcome was increased memory traffic and number of cycles (therefore also the estimated time). An example of estimates are presented in 6.13, where it is evident that adding rotations increases NPU cycles and SRAM access cycles, but reduces DRAM access cycles so much that total cycles decrease. A likely explanation is that the graph changes introduced by the merged rotations affected compiler memory planning, allowing more intermediate data to be retained or reused in SRAM instead of being fetched from DRAM. The result should therefore not be interpreted as a general property of rotations, but as an example of how small graph transformations can have non-obvious effects on memory scheduling and total performance.

Metric	No rotations	Merged rotations	Change
NPU operators	3	5	+66.7%
MACs per inference	2.72B	3.07B	+12.5%
NPU cycles	23.10M	26.35M	+14.1%
SRAM access cycles	4.93M	5.99M	+21.6%
DRAM access cycles	34.21M	10.61M	-69.0%
Total cycles	46.11M	32.10M	-30.4%
Total SRAM traffic (MB/inf.)	138.87	185.17	+33.3%
Total DRAM traffic (MB/inf.)	90.39	26.96	-70.2%
Input DRAM traffic (MB/inf.)	86.35	22.78	-73.6%
Inference time (ms)	46.11	32.10	-30.4%
Throughput (inf./s)	21.69	31.15	+43.6%

Table 6.13: Subgraph example of an unexpected effect of adding merged rotations to A8W8 MPQ in Dedicated SRAM mode.

Mixed-precision quantization



(a) Dedicated SRAM

(b) Shared SRAM

Figure 6.7: No MPQ→MPQ relative hardware metric changes under Dedicated SRAM and Shared SRAM. Negative values indicate reductions relative to the corresponding configuration without MPQ. Bandwidth is reported in MB/inference, peak memory usage in KiB, and inference time as relative change.

The effects of MPQ on performance were most noticeable on Dedicated SRAM mode, as shown by 6.7. The DRAM bandwidth increased with over 300% for the configuration with weights in 4 bit, and over 181% for weights in 8 bit. This is caused by the partitioning between

the NPU and the CPU. In Dedicated mode, the inputs are read from DRAM, which means that once the CPU has produced the output for the LayerNorm operation, the NPU starts executing the next part of the model which is it's own command stream. The new inputs have to then be read from the DRAM, resulting in the large increase in bandwidth. The DRAM use decrease was very large since the NPU executed much smaller programs instead of the entire model. However, roughly 80% of the model was still executed on the NPU. On Shared SRAM mode MPQ has much less penalties to performance, as it does not introduce the same increase in DRAM bandwidth.

Cycle count

Figure 6.8 shows the total estimated NPU cycle count for each configuration, compared between memory modes. It is evident that lower activation precision has a clear impact on the cycle count, and that differences between DRAM and SRAM use are more prominent when MPQ is used.

As MPQ delegates some cycles to the CPU, comparisons between configurations should only be made with this in mind.

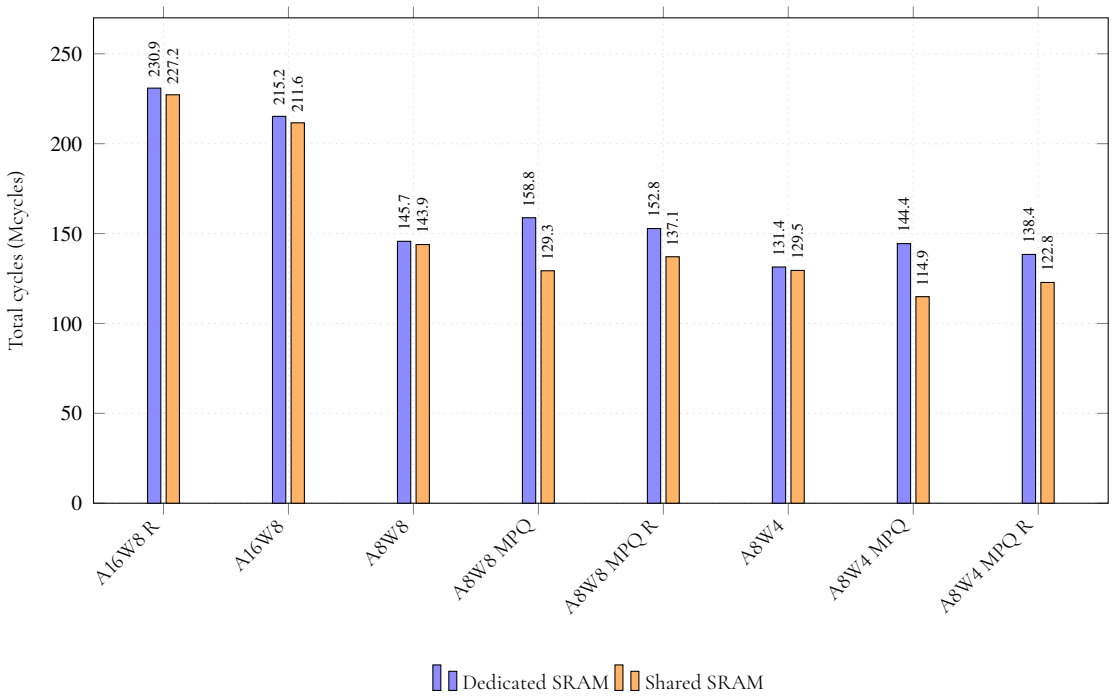


Figure 6.8: Total NPU cycles across activation/weight quantization, MPQ, and rotated/no-rotation variants under dedicated and shared SRAM configurations. Values are shown in million cycles. R marks use of rotation.

Only one of the planned FVP runs were completed before our set time-out, being the A8W4 MPQ configuration where no rotations were added. The table 6.14 shows some data collected by the Performance Monitoring Unit (PMU). Although it is not possible to make comparisons between configurations, the profiling data is useful in gaining a better idea of how MPQ affects performance, and allows some reflection of which observations that we've

PMU event	Counter value
SRAM read data beats received	0
SRAM write data beats written	0
External read data beats received	230,941,521
External write data beats written	28,255,839
NPU idle	13,871
MAC active	489,024,578
Weight decoder active	67,147,204

Table 6.14: Ethos-U PMU counters for A8W4 MPQ and using Dedicated SRAM on FVP.

made in the static performance estimations may be most important for improving real model inference.

Table 6.14 shows that all memory traffic has gone through the DRAM, and that data reads accounted for the majority of the memory traffic. This is not unexpected, as the compiler estimated SRAM usage indicated that the available SRAM would not be sufficiently large for inference and it has therefore not been utilized at all. The static performance numbers assume that there is sufficiently large SRAM available during inference, which means that comparison between the numbers are is not appropriate.

A useful way to visualize the trade-offs is to plot accuracy against resource metrics. In this case, we use a scatter plot to show accuracy as a function of total cycles, and with the size of the dots representing memory usage. Cycles count for MPQ configurations are total cycles for the inference, not only NPU cycles, to keep comparisons fair to non-MPQ configurations. Points that are smaller and closer to the upper-left corner represent more favourable configurations, since they achieve higher accuracy quicker and with lower resource costs.

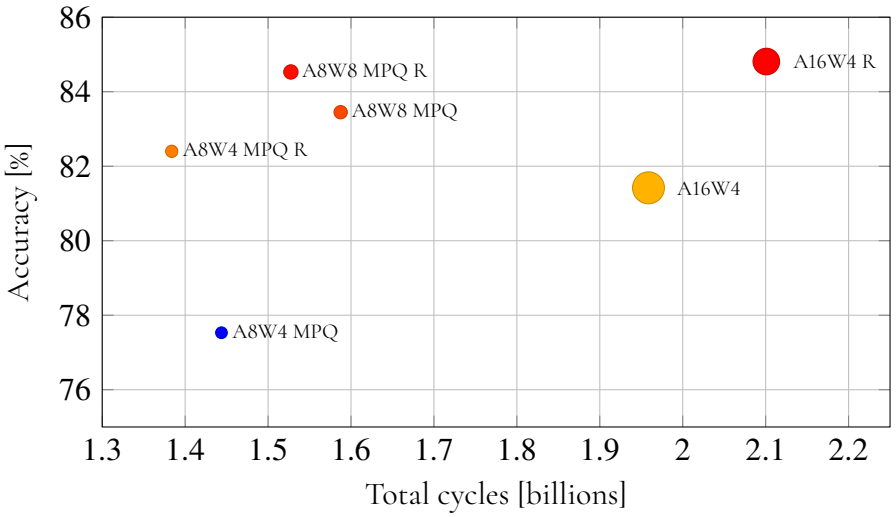


Figure 6.9: Trade-off between accuracy, NPU executed cycles and memory, using Dedicated SRAM as memory mode. Higher accuracy, fewer cycles and smaller memory usage are preferable; therefore, good configurations appear toward the upper-left with smaller markers. Marker size represents total memory usage. Memory and cycles for A16W4 are approximated. R marks use of rotation.

A16W4 is of greater deployment interest than A16W8, as the weights tolerated quantization to INT4 well. Since A16W8 had to be used to extract performance metrics, memory and cycles for A16W4 are approximated using the observed effects of changing weight precision from INT8 to INT4: peak DRAM usage was reduced by approximately 57%, peak SRAM usage was unchanged, and the total number of cycles was reduced by approximately 9%. The estimates are therefore computed as

$$C_{\text{A16W4}} = 0.91 C_{\text{A16W8}}, \qquad D_{\text{A16W4}} = 0.43 D_{\text{A16W8}}, \qquad S_{\text{A16W4}} = S_{\text{A16W8}}.$$

The calculated values should be interpreted as rough estimates rather than direct measurements, as complex compiler optimizations may make these numbers difficult to predict.

The results in 6.9 show a clear trade-off between model accuracy and resource usage. The rotated A16W4 is the most accurate, but also expensive in terms of both cycles and memory. The MPQ configurations shift the points toward the upper-left region of the plot, indicating a better balance between accuracy and efficiency. In particular, A8W8 MPQ R achieves nearly the same accuracy as A16W4 R, while requiring fewer cycles and less memory. Reducing the precision further to A8W4 lowers the resource usage, but also reduces accuracy. The difference between the A8W4 MPQ configurations further indicates that rotation is especially beneficial at lower precision, where quantization error is more severe. An important disclaimer is that the cycles in the figure are only NPU cycles, ie. CPU delegated cycles for MPQ configurations are not included. However, as can be seen in 6.8, there is a large difference between A16W8 and A8W8, showing that activation precision accounts for most of the cycle difference, and not MPQ.

Chapter 7
Discussion

Our study aimed to explore different quantization techniques on transformer-based models and to evaluate their effectiveness on Ethos-U85 NPU. We begin this chapter by discussing the method and how it has helped us to reach the goals of this project. We also reflect on which choices in methodology have been most impactful, and what could have been improved. Next, we discuss the validity of our results, followed by a discussion on the overarching trends and interesting observations we have made that have potential for practical use. We will also attempt to provide an analysis on how future implementation work could be improved.

7.1 Method

Our method consisted of three distinct phases: initiation, implementation, and evaluation. It is important to evaluate and discuss each part of the method, and reflect on the method creation process in order to identify both what went well and possible areas of improvement.

7.1.1 Initiation

The initiation phase focused on defining the goals and research questions of our work. This was a natural place to start, as the rest of the method was designed to meet the goals and provide results that could be used to answer the questions. As our goal was to explore new quantization techniques and gain an understanding of how they affected model performance, the first research question was purposefully designed as an open-ended question. The second question was also open-ended and aimed at providing an opportunity to extrapolate our findings into guidelines for future development of the Ethos-U Quantizer.

In order to gain an understanding of current state-of-the-art methods that could be used in our work, a literature study was conducted. The chosen quantization techniques were studied in practice with small experiments, but here it could have been possible to choose a larger set of potential techniques to pilot test as it is now difficult to say if the chosen

techniques are the best possible. However, as the goal of the work was to ultimately try out some state-of-the-art methods, this is not considered a big issue but a recommendation for future work on how it might be of interest to develop a more rigorous evaluation strategy for identifying potential techniques before committing to a larger implementation project.

The models were also chosen based on the small pilot experiments, and was constrained by Ethos-U support. Here it could also have been a good option to choose some other types of models too for initial testing in order to determine if there were any other transformer models that could have been improved with the chosen techniques. We considered testing models with other use cases, such speech-to-text or object detection, but this would also have required setting up different types of test sets. We planned to instead try the techniques on an LLM after implementation with Vision transformers, but the timeline did not permit this.

All in all, the strategy for choosing quantization techniques and test models could have been improved by identifying a larger pool of candidates and running more pilot experiments to systematically identify the most promising ones. However, this would have required more time and effort that was feasible.

7.1.2 Implementation

We went through several iterations of the implementation phase, intermingled with the accuracy evaluation phase where smaller accuracy tests were run frequently during the implementation work to guide the next iteration. This strategy allowed us to quickly see if the changes had the desired effect and to reconsider design choices. The implementation work can further be divided into two segments, where the first part focused on implementing the randomized rotations, and the second part focused on optimizing them with the strategy described in the method-section. The optimization strategy was repeated several times, and included a larger re-implementation of the fake quantization setup.

Optimization Strategy

As there was only one previous example of rotation optimization available on a different type of transformer model, the optimization strategy was based on a process of trial and error with the goal of quickly testing different configurations and identifying suitable hyperparameters. As each training attempt took roughly 2 hours to complete, the goal of the strategy was to simply find a configuration that gave some result, instead of following a more rigorous plan aimed at identifying the best possible training configuration, both in terms of hyperparameters and in terms of fake quantzation setup. We determined that it was more important to establish a proof-of-concept that optimization was possible, but that it was equally as important to gain an understanding of the practical feasibility of the method before embarking on a more rigorous exploration of different training strategies and hyperparameter tuning.

This approach is evident in the results, as we could on one hand show that the training made a difference when tested against our simulated quantization, but showed no definite improvement with the Ethos-U quantizer. Barring time constraints we would have perhaps attempted a third optimization iteration that utilized the QAT-option available in the Ethos-U quantizer, instead of our own simulation. In conclusion, the optimization strategy was sufficient in showing that it is possible to optimize rotations, but our optimization approach was not ideal for deployment on the Ethos-U.

7.1.3 Evaluation

The evaluation phase was divided into two stages where the accuracy testing was tightly coupled with the implementation phase. The division was made to enable us to first focus on improving the accuracy for quantized models. The hardware metrics stage was only completed once all implementation and accuracy testing had been done. A consequence of this strategy is that the accuracy performance was prioritized during implementation over memory or computational performance. This choice was motivated by the fact that we knew that both rotation and MPQ would incur a penalty in performance, and while we did attempt to introduce as little overhead as possible in our implementation choices, the priority was to maximize accuracy and then measure if the gain in accuracy would be worth it. This choice is reflected in our results as the improvements in accuracy have generally all come with some degradation in hardware performance.

The evaluation phase did provide us with results that were reliable enough for us to answer the first research question, and while it may not be possible to make direct comparisons to results found in related work, it does lay a ground for understanding how quantization affects performance on the Ethos-U and how this performance may be measured, as well as how evaluation can guide future development work.

Accuracy Testing Strategy

The goal of the accuracy testing was to provide performance benchmarks that we could compare to each other, and to scores reported in related work and the floating point performance of the models. We ensured this by using ImageNet-1k as the test set, and by creating a static split for calibration, training, and testing. This split worked well and ensured that factors such as variances in calibration data could not affect the results. One possibility for improvement could have been a more careful choice of the size of calibration set, as there was no clear methodology for choosing the amount of calibration samples. Since this does not have any effect on the comparability of our results with each other, we have also concluded that it was not feasible to explore how the used calibration data and the sample size affected the performance within the available time frame.

Hardware Testing Strategy

Hardware testing constituted the final phase of our project, and included collecting performance data from various configurations that had been built during the previous phases. The goal was to look at both compiler-provided estimates, and data from the FVP. The compiler estimates could be generated without bigger issues and only one configuration, A16W4, could not be tested as that specific setup has not been implemented in the Ethos-U backend yet. We could still gain an understanding of how activation precision affected performance by running tests with A16W8 configuration.

Generating compiler estimated performance numbers was relatively fast, as it took around 10 minutes at most per model configuration. No prioritization due to time constraints was needed and we collected metrics even from configurations that we knew had poor accuracy in order to gain an understanding of how large performance gains could be made by further quantization efforts. The same did not apply to the simulations on FVP that took several hours and were unsuccessful for all but one run. This means that we were not able to gain

the best possible picture of performance numbers. The problems were likely caused by the models being too large, which could have been solved by tweaking the configurations and by adding other model optimization methods. Time was again the limiting factor that forced us to make our performance analyses mostly reliant on the compiler estimates. We have still been able to compare the different configurations and find details that are helpful for analysing the characteristics that different quantization techniques bring to model deployment on Ethos-U.

7.2 Threats to Validity

Several factors may affect the validity of the results presented in this work. These threats are mainly related to the accuracy evaluation, model selection, implementation complexity, and the interpretation of compiler-estimated hardware metrics.

7.2.1 Accuracy Evaluation and Generalizability

One limitation is related to the datasets used during accuracy evaluation. Although the same ImageNet-1K validation split was used throughout the project, not all experiments were run with the same number of samples, and these results are therefore not directly comparable. Smaller test sets were mainly used during the implementation phase to identify promising configurations as well as for the repeated random rotation experiments, while larger test sets were used for the final evaluations.

The calibration set is another potential source of variation. Since transformer activations vary between inputs, a different calibration set or a larger number of calibration samples could lead to different quantization parameters and different accuracy results. Furthermore, the size of the calibration set was not chosen based on a separate methodological study, but was selected early during the experimental phase and then kept constant for the rest of the project.

The choice of models also limits generalizability. The evaluation focuses on two encoder-based models, and one of them was not evaluated as extensively as the other. These models provide useful case studies, but they do not cover the full range of transformer architectures or the different quantization issues that may affect the chosen methods. Even between the two evaluated models, clear differences in response to mixed-precision quantization and rotations were observed, suggesting that results may vary substantially for other transformer architectures.

7.2.2 Implementation and Toolchain Limitations

As previously discussed, another limitation is the mismatch between the fake quantization used during rotation optimization and the quantization applied by the Ethos-U deployment flow. The fake quantization does not fully reproduce all details of the Ethos-U quantizer, graph lowering, or compiler behaviour. As a result, rotations that improve accuracy under fake quantization may not provide the same benefit after export and compilation. Consequently, no definite conclusions can be made about how much accuracy could have been recovered with a training setup that more closely matched the final deployment flow.

The implementation and deployment flow also introduce threats to validity. The complete system includes model modifications in PyTorch, fake quantization during training, export through ExecuTorch, lowering to the Ethos-U backend, compilation with Vela, and evaluation through accuracy and hardware metrics. Each stage can introduce subtle changes to the model graph, quantization parameters, or execution behaviour. We mitigated this by comparing floating-point outputs before and after model transformations, inspecting exported graphs, checking quantization placement, and running accuracy tests throughout the implementation process. Nevertheless, small inconsistencies may remain.

The results are also dependent on the specific versions of the software stack used in the experiments. ExecuTorch, the Arm backend, and Vela are actively developed, and changes to graph lowering, operator support, quantization annotation, or compiler memory planning could affect both accuracy and hardware metrics. The results should therefore be interpreted as valid for the tested toolchain version, rather than as permanent properties of the evaluated models or methods.

7.2.3 Hardware Performance Estimates

The hardware performance results are based on compiler estimates rather than measurements on FPGA or physical hardware. These estimates are useful for comparing configurations and understanding expected trends, but they cannot be interpreted as exact runtime measurements. Compared to absolute values, the relative changes between similar configurations are more likely to remain consistent between estimates and real deployment measurements, but even these ratios should be considered with caution.

The performance comparisons in Section 6.3.3 are also likely affected by compiler memory planning, as illustrated by the unexpected result shown in Table 6.13. In those comparisons, configurations are often compared by changing one intended factor at a time, such as activation precision, weight precision, mixed precision, or rotations. However, changing one of these factors can also change the lowered graph structure and therefore the compiler memory schedule.

The SRAM–DRAM partitioning is determined by the compiler based on graph structure, tensor sizes, tensor lifetimes, and the available SRAM budget. Adding rotations, changing precision, or introducing mixed-precision partitions may alter operator placement, intermediate tensor sizes, and tensor lifetimes. As a result, some activations that were previously placed in DRAM may instead be kept in SRAM, or the other way around. This means that observed differences in memory traffic or cycle count may not be caused only by the intended factor, but also by secondary effects from compiler scheduling.

7.3 Key Findings and Trade-offs

This section discusses the main patterns that emerged from the evaluation and relates them to the research questions. The focus is on which design choices had the largest effect on accuracy, memory behaviour, and estimated inference performance, and on what trade-offs these choices imply for deployment on Ethos-U.

7.3.1 What Were the Best Performers?

The three evaluation areas are discussed first on their own and then in relation to each other, in order to identify combinations that offer a useful balance between accuracy and hardware efficiency.

Achieving Best Accuracy

It was no surprise that the best accuracy scores were tied to configurations with more precision. In addition, the rotations were inserted to help with the dynamic activation range problem. The interesting part here is comparing which of the two had a bigger impact - precision or the rotations.

In the case of ViT it is clear that MPQ had the absolute largest impact on accuracy for A8 configurations, as without it the accuracy score was basically zero. With MPQ, ViT could be quantized to A8W8 and the accuracy would still be close to the reported floating point reference, with a drop of only 3% from the reference. When weights were quantized to four bits, the accuracy score dropped by additional 3%. On the other hand, if activations were quantized to 16 bits and weights to eight bits the accuracy loss was almost non-existent. Decreasing the weight precision to four introduced again a drop in accuracy that was similar to what we observed before.

The interesting thing is that R1 and R2 rotations seemed to be most effective in mitigating the degradation in accuracy that followed from dropping weight precision from eight to four bits. This result indicates that while rotations may not have shown noticeable improvements on activation quantization, they may be helpful in maintaining accuracy when weights are quantized to a lower precision.

However, the above observations apply only to ViT. We did not observe any accuracy gain on the Swin-model with rotations. Reductions in channel variation were observed on Swin when activations were recorded on models with rotation, so the quantization should have been easier in theory. It is likely that other sensitive parts had a stronger effect on quantization that overshadowed the potential help from the rotations. MPQ efforts were likewise not very fruitful, as the process was not as straightforward as compared to ViT. Better results would have required a more concentrated effort on finding a quantization configuration that avoided the effect that the attention mask application had.

Achieving Best Memory Performance

In order to discuss which configurations were best for memory efficiency, the characteristics that imply best memory performance should be defined. In this case, we were interested in analysing which factors were most important for increasing or decreasing SRAM and DRAM bandwidth, and how precision affected peak memory usage.

Here we again observed a trend where lower precision was tied to better SRAM utilization and decreased DRAM bandwidth. The activation precision had the biggest impact on peak SRAM utilization as peak SRAM size is proportional to activation precision, motivating the use of low bit-widths for activations. Our work did not explore activation lower than INT8 as they are not yet supported by the Arm backend, but this may be motivation for future work on enabling it.

As shown by both the compiler estimates on memory usage, as the profiling data gained from the successful FVP run, most memory accesses to DRAM are read operations. This means that quantization of weights to INT4 will therefore always have a positive impact on DRAM bandwidth and peak DRAM usage. Big differences in DRAM usage could also be seen when MPQ was used in Dedicated SRAM mode. In this case the model being broken into multiple graphs results in a very large increase in DRAM bandwidth, as input tensors had to be read from DRAM every time a new partition was executed on the NPU. This indicates that the possible penalties related to MPQ are mostly relevant when Dedicated SRAM more is used, and grow with the number of partitions. The fewer partitions, the lesser impact on DRAM bandwidth.

Achieving Fastest Inference

While estimated cycle counts have been mostly what we expected, some of the results were not. Some trends can be extrapolated, such that a higher SRAM utilization translates to fewer total cycles and a shorter inference time. As we are dealing with estimates, the most meaningful observations are inferred by looking at how the estimates relate to each other. The biggest gaps in inference times are by far caused by the activation precision, where quantization to 16 bits loses clearly to 8 bits, as can be seen when comparing A16W8 to A8W8 in 6.8. The differences between MPQ and non-MPQ results for a configuration were not significant, though the true number of CPU delegated cycles are unknown without running tests on the FVP.

Although online rotations did not increase the cycle count in our experiments, this result is unlikely to hold in the general case. Nevertheless, the additional operations introduced by this number of online rotations appear to have a limited impact on overall inference time.

7.3.2 Recommendations

As shown by the results, there is no configuration that received the best results in all three performance areas. The question is then, are there any details that will always result in a negative or a positive effect, without degradation in other areas? We have attempted to identify which aspects in the different areas have synergy in order to create a more general understanding of how quantization techniques may affect the overall performance.

The first such observation is the relationship between activation precision and performance. While the A16W4 configuration achieved an accuracy score close to the floating point-performance, with full-integer inference, the benefits of integer-only execution do not outweigh the costs, as the A16 configurations were clearly inferior to the A8 configurations in terms of inference speed. The results show that MPQ can be a good option for achieving a good balance between accuracy and inference time, especially when Shared SRAM mode can be used, although this may not be realistic with most models. When Dedicated SRAM is used, MPQ will likely work best when the partitioning balances the cost of increasing DRAM bandwidth with the benefits of the speed-up in computations that the NPU offers. Maximizing the number of NPU operations may not be the best strategy when partitioning the model for inference in terms of performance.

Rotations - Are They Worth It?

The rotations had the biggest positive impact when weights were quantized to four bits, which means that they could be used to regain the accuracy loss caused by the lower bit-width while bringing a large decrease in DRAM bandwidth. In this case the penalties from possible online rotations are likely worth the benefits of reducing DRAM usage seeing that the computations are inexpensive operations compared to memory reads. However, it should also be noted that rotations were not as useful for reducing activation quantization error, and although some slight improvement was seen for the A8W8 configuration, this improvement was not shown to be statistically significant. The same applies to the R3 rotation that was applied in ViT on the key and query tensors. The results indicate that the rotation had virtually no impact on accuracy scores. It is possible that we did not apply the rotations in places that were most optimal, and that further investigations on how best rotation placement for individual models can be carried out and implemented is motivated.

This model specificity is also highlighted by the fact that two online rotations in ViT were required due to the placement of LayerNorm in the model architecture. Some model architectures may be much more suitable for such rotation methods, and offer the possibility of using only offline rotations.

7.4 Quantization to Ethos-U - Lessons Learned

Although our results have showed that it is possible to achieve good accuracy and performance for transformer-models on Ethos-U NPUs, our work has also offered the possibility to reflect on the specific pain points associated with transformer quantization, especially for a framework that is intended for general use cases and has more limited capabilities in implementing model-specific methods.

Our experiences with implementing the rotations shows how different model architectures present unique challenges to quantization that may be very difficult to address with a general quantization scheme. These challenges are clearly seen in the vastly different results we achieved with the two models that were used in the experiments. While we can't say for certain why the rotations were seemingly not helpful for quantization of Swin, the ineffectiveness is likely connected to how the attention mask is constructed and how the Ethos-U quantizer handles its quantization. We suspect that the possible positive effect of the rotations was masked by the large quantization error that the attention mask caused, and that if those specific operations were handled differently the effect of rotations could have been more visible. This is not unreasonable, as we saw with ViT that the quantization error introduced by LayerNorm overshadowed any possible benefits that the rotations could bring when activations were quantized to 8 bits.

These results show how bottlenecks that cause large quantization error can be very difficult to handle for a general deployment framework such as ExecuTorch, as they can be extremely model specific, and one cannot assume that two models would suffer from the same quantization bottlenecks. This highlights how important it may be to provide a general deployment flow that allows a level of customization on the quantization scheme that makes it possible for the end-user to account for the specific challenges that concern the model they are looking to deploy. Furthermore, this raises the important question of how the challenges can be identified in the general case. Through our analysis tools and previous

literature we could quite quickly surmise that LayerNorm was the big bottleneck for ViT, but this was again not the case for Swin. Our tooling was insufficient in providing a good level of understanding where accuracy was lost and what each operator contributed to quantization error in the time frame available to us. Even though we knew that the attention mask was a bottleneck for quantization, we were not able to construct an MPQ scheme that had any significant effect on it. Our conclusion is that while the analysis tools we used could *identify* a bottleneck, knowing where the bottleneck is located is only the first step in resolving the issue. The appropriate mitigation strategy may not always be as easy as excluding a problematic layer, and can instead require the application of a highly specific quantization scheme for a part of the model. This scheme needs to then be balanced with hardware performance as for example in the case of MPQ it should take into account how fragmented the program becomes between the NPU and CPU to avoid incurring memory penalties with a cost that exceeds the benefits of using the NPU for acceleration in the first place.

We believe that further development of tools that help understand the root causes for quantization error may be more useful than implementing specific solutions for some sensitive operations. Such tools could for example visualize the bottlenecks and their surrounding operations. Depending on the operations involved, further analytics could involve calculating the variance in activations or weights distributions to identify suitable places that could benefit from adding rotations that mitigate outliers.

Chapter 8
Conclusions

We conclude our work by answering our research questions, relating our finds to prior work, and reflecting on possible future work.

8.1 Research Questions

RQ1

> How do the different quantization approaches affect the performance, accuracy, and memory footprint of encoder transformer models on embedded platforms such as Ethos-U?

Rotations can improve accuracy but increase the model size and computational cost when applied online during inference. Whether this trade-off is worthwhile depends on the model architecture, as both the amount of accuracy recovered through rotations and the number of online rotations required vary between models. Rotations become increasingly beneficial as quantization precision decreases. Reducing activation precision substantially decreases the memory footprint and improves overall performance, but at the cost of a significant reduction in accuracy for lower activation precisions, unless combined quantization approaches are used. Reducing weight precision exhibits a similar trade-off, although the effects are less pronounced. MPQ offers the best path for balancing accuracy with overall performance, but how the model is partitioned has a big impact on memory usage.

RQ2

> What are the challenges of applying the chosen quantization approaches to transformer models for inference on Ethos-U?

The main challenge is that transformer quantization is highly model- and toolchain-dependent. Although ViT and Swin are both encoder-based vision transformers, they responded differently to MPQ and rotations: ViT was mainly sensitive to LayerNorm quantization, while Swin was more affected by attention mask and Softmax-related computation. Improvements under fake quantization also did not always transfer to the final Ethos-U deployment flow, showing that rotation training must closely match the final quantization and lowering behaviour. In addition, hardware performance depended not only on bit-widths and operation count, but also on graph partitioning, compiler memory planning, and SRAM–DRAM placement. Overall, transformer quantization for Ethos-U must be treated as a combined model, toolchain, and hardware problem.

8.2 Relation to Prior Work

Compared with prior work, our work focuses less on proposing a new quantization operator or model architecture, and more on understanding which transformer quantization techniques are practical in an existing Ethos-U deployment flow. Methods such as PTQ4ViT [15], FQ-ViT [16], I-ViT [18], and TSPTQ-ViT [17] show that accurate ViT quantization is possible when sensitive operations such as LayerNorm, Softmax, and GELU are handled with specialized quantization schemes or integer approximations. Our work instead evaluates how far backend-supported post-training quantization can be pushed using mixed-precision exceptions and rotation-based outlier reduction. Compared with LRP-QViT [35], our MPQ approach is manual and deployment-driven rather than based on automated bit-width search. Compared with QuaRot [20] and SpinQuant [21], our rotation experiments investigate whether similar outlier-reduction ideas transfer from LLMs to encoder-based vision transformers. Compared with MCUFormer [36] and TinyFormer [37], we do not design a new MCU-specific transformer, but evaluate standard Torchvision ViT and Swin models in an ExecuTorch and Ethos-U85 deployment flow.

8.3 Future Work

If deemed worthwhile, future work could focus on improving the match between rotation optimization and the final Ethos-U quantization flow, for example by using quantization-aware training support closer to the deployed backend. The methods could also be evaluated on a broader range of transformer architectures to better understand when MPQ and rotations generalize. Finally, more complete FVP profiling or measurements on FPGA or physical Ethos-U hardware would be needed to validate the compiler-estimated performance trends.

Overall, this thesis shows that efficient transformer deployment on Ethos-U requires more than lowering numerical precision. The effectiveness of quantization methods depends on model architecture, sensitive operations, activation and weight distributions, compiler behaviour, and the target memory configuration.

References

[1] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *Nature*, vol. 521, no. 7553, pp. 436–444, 2015. [Online]. Available: https://doi.org/10.1038/nature14539

[2] A. Vaswani *et al.*, "Attention is all you need," in *Advances in Neural Information Processing Systems*, I. Guyon *et al.*, Eds., vol. 30. Curran Associates, Inc., 2017. [Online]. Available: https://papers.nips.cc/paper/7181-attention-is-all-you-need

[3] T. Lin *et al.*, "A survey of transformers," *AI Open*, vol. 3, pp. 111–132, 2022. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2666651022000146

[4] T. Mowbray, "A survey of deep learning architectures in modern machine learning systems: From cnns to transformers," *Journal of Computer Technology and Software*, vol. 4, no. 8, 2025.

[5] M. G. S. Murshed *et al.*, "Machine learning at the network edge: A survey," *ACM Comput. Surv.*, vol. 54, no. 8, Oct. 2021. [Online]. Available: https://doi.org/10.1145/3469029

[6] S. Jiang *et al.*, "Edge large language models: a comprehensive survey," *CCF Transactions on Pervasive Computing and Interaction*, pp. 1–30, 2026.

[7] I. Lamaakal *et al.*, "Tinyml vs llms: A survey of extreme scales in machine learning," in *2025 International Conference on Electrical Systems Automation (ICESA)*, 2025, pp. 1–6.

[8] Y. Chen *et al.*, "A survey of accelerator architectures for deep neural networks," *Engineering*, vol. 6, no. 3, pp. 264–274, 2020. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2095809919306356

[9] Arm, "Arm® Ethos™-U85 NPU Technical Reference Manual," https://developer.arm.com/documentation/102685/0000?lang=en, 2025, 2026-02-23.

[10] PyTorch Foundation, "Arm Ethos-U NPU Backend," https://docs.pytorch.org/executorch/0.7/backends-arm-ethos-u.html, 2026, execuTorch documentation. Accessed 2026-06-15.

[11] B. Jacob *et al.*, "Quantization and training of neural networks for efficient integer-arithmetic-only inference," in *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2018, pp. 2704–2713.

[12] A. Gholami *et al.*, "A survey of quantization methods for efficient neural network inference," in *Low-Power Computer Vision: Improve the Efficiency of Artificial Intelligence*. Chapman and Hall/CRC, 2022, pp. 291–326. [Online]. Available: https://doi.org/10.1201/9781003162810-13

[13] H. Wu *et al.*, "Integer quantization for deep learning inference: Principles and empirical evaluation," 2020. [Online]. Available: https://arxiv.org/abs/2004.09602

[14] Y. Bondarenko, M. Nagel, and T. Blankevoort, "Understanding and overcoming the challenges of efficient transformer quantization," in *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*. Online and Punta Cana, Dominican Republic: Association for Computational Linguistics, Nov. 2021, pp. 7947–7969. [Online]. Available: https://aclanthology.org/2021.emnlp-main.627

[15] Z. Yuan *et al.*, "PTQ4ViT: Post-training quantization for vision transformers with twin uniform quantization," in *European Conference on Computer Vision*, 2022.

[16] Y. Lin *et al.*, "FQ-ViT: Post-training quantization for fully quantized vision transformer," in *Proceedings of the Thirty-First International Joint Conference on Artificial Intelligence*. International Joint Conferences on Artificial Intelligence Organization, 2022, pp. 1173–1179. [Online]. Available: https://www.ijcai.org/proceedings/2022/164

[17] Y.-S. Tai, M.-G. Lin, and A.-Y. A. Wu, "Tsptq-vit: Two-scaled post-training quantization for vision transformer," in *ICASSP 2023 - 2023 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 2023, pp. 1–5.

[18] Z. Li and Q. Gu, "I-vit: Integer-only quantization for efficient vision transformer inference," in *2023 IEEE/CVF International Conference on Computer Vision (ICCV)*, 2023, pp. 17 019–17 029.

[19] M. Rakka *et al.*, "A review of state-of-the-art mixed-precision neural network frameworks," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 46, no. 12, pp. 7793–7812, 2024.

[20] S. Ashkboos *et al.*, "QuaRot: Outlier-free 4-bit inference in rotated llms," in *Advances in Neural Information Processing Systems*, vol. 37, 2024. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2024/hash/b5b939436789f76f08b9d0da5e81af7c-Abstract-Conference.html

[21] Z. Liu *et al.*, "Spinquant: Llm quantization with learned rotations," 2025. [Online]. Available: https://arxiv.org/abs/2405.16406

[22] A. Dosovitskiy *et al.*, "An image is worth 16x16 words: Transformers for image recognition at scale," in *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*. OpenReview.net, 2021. [Online]. Available: https://openreview.net/forum?id=YicbFdNTTy

[23] Z. Liu *et al.*, "Swin transformer: Hierarchical vision transformer using shifted windows," in *Proceedings of the IEEE/CVF International Conference on Computer Vision*, 2021, pp. 10 012–10 022.

[24] D. P. Samuel Williams, Andrew Waterman, "Roofline : an insightful visual performance model for multicore architectures," *Communications of the ACM - A Direct Path to Dependable Software.*, vol. 52, no. 4, pp. 65–76, 2009.

[25] V. Sze *et al.*, "Efficient processing of deep neural networks: A tutorial and survey," *Proceedings of the IEEE*, vol. 105, no. 12, pp. 2295–2329, 2017.

[26] PyTorch Foundation, "Arm Ethos-U NPU Backend," https://docs.pytorch.org/executorch/1.0/embedded-arm-ethos-u.html, 2026, execuTorch documentation. Accessed 2026-06-04.

[27] PyTorch, "Practical quantization in pytorch," https://pytorch.org/blog/quantization-in-practice/, 2021, accessed 2026-06-04.

[28] J. Lang, Z. Guo, and S. Huang, "A comprehensive study on quantization techniques for large language models," in *2024 4th International Conference on Artificial Intelligence, Robotics, and Communication (ICAIRC)*, 2024, pp. 224–231.

[29] S. Ashkboos *et al.*, "Slicegpt: Compress large language models by deleting rows and columns," in *The Twelfth International Conference on Learning Representations*. OpenReview.net, May 2024. [Online]. Available: https://openreview.net/forum?id=vXxardq6db

[30] Y. Bengio, N. Léonard, and A. Courville, "Estimating or propagating gradients through stochastic neurons for conditional computation," *CoRR*, vol. abs/1308.3432, 2013. [Online]. Available: https://arxiv.org/abs/1308.3432

[31] J. Li, F. Li, and S. Todorovic, "Efficient riemannian optimization on the stiefel manifold via the cayley transform," in *International Conference on Learning Representations*. OpenReview.net, 2020. [Online]. Available: https://openreview.net/forum?id=HJxV-ANKDH

[32] V. J. Reddi *et al.*, "Mlperf inference benchmark," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, 2020, pp. 446–459.

[33] V. Sze *et al.*, "How to evaluate deep neural network processors: Tops/w (alone) considered harmful," *IEEE Solid-State Circuits Magazine*, vol. 12, no. 3, pp. 28–41, 2020.

[34] M. Horowitz, "1.1 computing's energy problem (and what we can do about it)," in *2014 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*, 2014, pp. 10–14.

[35] N. Ranjan and A. Savakis, "Lrp-qvit: Mixed-precision vision transformer quantization using layer importance score," in *2025 25th International Conference on Digital Signal Processing (DSP)*, 2025, pp. 1–5.

[36] Y. Liang *et al.*, "MCUFormer: Deploying vision transformers on microcontrollers with limited memory," in *Advances in Neural Information Processing Systems*. Curran Associates, Inc., 2023, pp. 8501–8512.

[37] J. Yang *et al.*, "TinyFormer: Efficient transformer design and deployment on tiny devices," *arXiv preprint arXiv:2311.01759*, 2023.

[38] O. Andersson and PyTorch Contributors, "Arm backend: Add matmul decomposition pass," https://github.com/pytorch/executorch/pull/17004, 2026, execuTorch pull request #17004, merged commit `a06697e`. Accessed 2026-06-15.

[39] PyTorch Foundation, "Arm Ethos-U NPU Backend," https://docs.pytorch.org/executorch/1.0/backends-arm-ethos-u.html, 2026, execuTorch documentation. Accessed 2026-06-15.

[40] PyTorch Contributors, "ExecuTorch Arm Ethos-U backend requirements," https://raw.githubusercontent.com/pytorch/executorch/a06697e/backends/arm/requirements-arm-ethos-u.txt, 2026, specifies `ethos-u-vela == 4.5.0`. Accessed 2026-06-15.

[41] Google AI Edge, "Model explorer," https://github.com/google-ai-edge/model-explorer/, 2025, accessed: 2026-05-12.

[42] PyTorch Contributors, "torchvision.models.vit_b_16," https://docs.pytorch.org/vision/main/models/generated/torchvision.models.vit_b_16.html, 2025, accessed: 2026-05-12.

[43] ——, "torchvision.models.swin_t," https://docs.pytorch.org/vision/main/models/generated/torchvision.models.swin_t.html, 2025, accessed: 2026-05-12.

[44] SPCL, "QuaRot: Outlier-Free 4-Bit Inference in Rotated LLMs," https://github.com/spcl/QuaRot, 2024, gitHub repository, accessed 2026-05-22.

[45] Meta AI and facebookresearch, "SpinQuant: LLM Quantization with Learned Rotations," https://github.com/facebookresearch/SpinQuant, 2024, gitHub repository, accessed 2026-05-22.

[46] PyTorch Foundation, "TorchAO Documentation," https://docs.pytorch.org/ao/stable/index.html, 2026, accessed 2026-06-15.

[47] ——, "IntxFakeQuantizer," https://docs.pytorch.org/ao/stable/api_reference/generated/torchao.quantization.qat.IntxFakeQuantizer.html, 2026, torchAO API documentation. Accessed 2026-06-15.

[48] Hugging Face, "Ilsvrc/imagenet-1k dataset," https://huggingface.co/datasets/ILSVRC/imagenet-1k, 2025, accessed: 2026-05-12.

Appendices

INSTITUTIONEN FÖR DATAVETENSKAP | LUNDS TEKNISKA HÖGSKOLA | PRESENTERAD 2026-06-11

EXAMENSARBETE Quantization Techniques for Memory-bound Transformers on Ethos-U Hardware Accelerators
STUDENTER Märta Holmquist, Emma Kujala
HANDLEDARE Flavius Gruian (LTH)
EXAMINATOR Sven Robertz (LTH)

Kvantisering + rotationer = sant?

POPULÄRVETENSKAPLIG SAMMANFATTNING **Märta Holmquist, Emma Kujala**

Det blir alltmer vanligt att stora och beräkningstunga maskininlärningsmodeller ska köras lokalt på väldigt små enheter. *Kvantisering* används för göra dessa modeller mindre, snabbare och mer energieffektiva utan att tappa för mycket precision. Detta arbete undersöker ett sätt att förbättra kvantisering genom att introducera rotationer.

Många kraftfulla AI-modeller körs i dag i molnet, men i en del fall kan exempelvis krav på svarstid, integritet eller tillgång till nätverk göra det fördelaktigt att köra modellen direkt på en enhet. Detta kallas edge-inferens och kan utnyttjas i bland annat fordon, mobiler, kameror, sensorer och annan inbyggd elektronik.

Utmaningen med detta är att små edge-enheter har begränsat minne, låg energibudget och mindre beräkningskraft. Därför behöver modellerna optimeras innan de kan köras på sådan hårdvara. En vanlig metod för detta är kvantisering, där modellens tal approximeras till färre bitar i enklare, diskreta format. På så sätt minskas både minnesanvändning och energiförbrukning, men med potentiella uppoffringar i modellens noggrannhet.

Kvantisering fungerar sämre när modellens aktiveringar innehåller enstaka mycket stora värden, så kallade outliers. Dessa extremvärden kan tvinga kvantiseringen att täcka ett större intervall, vilket gör att de vanligaste värdena får färre tal i intervallet att representeras av och därför avrundas mer. För att minska detta problem kan man använda rotationer. En rotation ändrar inte informationen i modellen, utan uttrycker den i ett nytt koordinatsystem - på så sätt kan extremvärden spridas över flera dimensioner i stället för att dominera en enskild kanal. Det kan göra aktiveringarna jämnare och enklare att kvantisera utan att modellens noggrannhet försämras lika mycket. Olika rotationer är olika bra på att åstadkomma detta, och i detta arbete har vi undersökt slumpmässiga rotationer samt försökt hitta optimala rotationer med hjälp av maskininlärningsalgoritmer.

Vi har även undersökt effekterna av att endast kvantisera delar av modellen, då vissa komponenter är mer känsliga för minskad precision, samt hur detta kan kombineras med rotationer för att balansera minnesanvändning, hastighet och precision på Arm Ethos-U NPU:er.

Resultaten visar att rotationer och partiell kvantisering kan förbättra modellens noggrannhet till nivåer mycket nära den icke-kvantiserade modellen, med en mindre påverkan på minnesanvändning och effektivitet. Rotationer funna via maskininlärning var resurskrävande att ta fram och nya verktyg hade behövts för att förbättra och enklare generalisera denna process till andra modeller. Sammantaget visar studien att teknikerna kan vara mycket effektiva i att bevara precision, men att processerna för att hitta vilka delar av en modell som gynnas mest av delvis kvantisering eller rotationer, samt för att ta fram optimala rotationer, behöver effektiviseras.