UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE ENGENHARIA DE COMPUTAÇÃO

TIAGO VIER PRETO

Integer-only Quantization: Towards Efficient Neural Network Deployment on Diverse Platforms

Work presented in partial fulfillment of the requirements for the degree of Bachelor in Computer Engineering

Advisor: Prof. Dr. Mateus Grellert

Porto Alegre
July 2026

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
Reitora: Profª. Marcia Barbosa
Vice-Reitor: Prof. Pedro Costa
Pró-Reitora de Graduação: Profª. Nádya Pesce da Silveira
Diretor do Instituto de Informática: Prof. Luciano Paschoal Gaspary
Coordenador do Curso de Engenharia de Computação: Prof. Paulo Francisco Butzen
Bibliotecário-chefe do Instituto de Informática: Alexsander Borges Ribeiro

“He who would learn to fly one day must first learn to stand and walk and run and climb and dance; one cannot fly into flying.”

— Friedrich Nietzsche

ACKNOWLEDGMENTS

I would like to express my sincere gratitude to everyone who contributed, directly or indirectly, to the completion of this work and to the conclusion of this fundamental stage in my academic journey.

To the Institute of Informatics of the Federal University of Rio Grande do Sul (INF/UFRGS), for providing an environment of academic excellence, filled with opportunities and resources that made my development as a student and future computer engineer possible.

To my advisor, Prof. Dr. Mateus Grellert, for his patience, dedication, and guidance throughout the development of this research. His technical insights, academic rigor, and encouragement were fundamental to steering this project and expanding my horizons in the field of neural network optimization and acceleration.

To my parents, for being my safe harbor and for always believing in me, even during the most challenging moments of my university journey. Your unconditional support, love, and guidance were the foundation of all my achievements. This victory also belongs to you.

ABSTRACT

The deployment of deep neural networks on edge and embedded devices faces critical challenges due to strict constraints on computation, latency, and energy consumption. Prior work has demonstrated that integer-only quantization can deliver remarkable reductions in inference latency while maintaining accuracy, enabling efficient deployment even on limited mobile CPUs. However, the broader implications of this approach across diverse embedded hardware remain less explored. In this graduation thesis, we present a systematic evaluation of 8-bit integer (INT8) quantization across highly heterogeneous platforms: an Apple M4 Pro, a Qualcomm Snapdragon 8 Elite, and an NVIDIA RTX 4070 desktop GPU. Utilizing the AI Model Efficiency Toolkit (AIMET), we apply both Post-Training Quantization (PTQ) and Quantization-Aware Training (QAT) to edge-optimized (MobileNetV1, SqueezeNet) and general-purpose (ResNet) architectures. Our results demonstrate that QAT successfully recovers the representational capacity lost by PTQ in compact models. Furthermore, we show that domain-specific edge accelerators natively executing INT8 workloads not only rival the sequential latency of a 200-Watt desktop GPU but do so with extraordinary efficiency. Specifically, quantization yielded up to an 89.5% energy reduction on the Apple Neural Engine and a 9.14$\times$ latency speedup on the Snapdragon Hexagon NPU compared to their unquantized baselines, proving that integer quantization is an architectural prerequisite for overcoming the memory wall and enabling sustainable real-time edge AI.

Keywords: Deep Neural Networks. Quantization. Energy Efficiency. Low-precision Inference.

Quantização apenas com inteiros: rumo à implementação eficiente de redes neurais em plataformas diversas

RESUMO

A implementação de redes neurais profundas em dispositivos de borda e embarcados enfrenta desafios críticos devido às restrições rigorosas de computação, latência e consumo de energia. Trabalhos anteriores demonstraram que a quantização baseada apenas em inteiros pode proporcionar reduções notáveis na latência de inferência, mantendo a precisão, permitindo uma implementação eficiente mesmo em CPUs com recursos limitados. No entanto, as implicações mais amplas dessa abordagem em diversos hardwares permanecem pouco exploradas. Neste trabalho de conclusão, apresentamos uma avaliação sistemática da quantização de inteiros de 8 bits (INT8) em plataformas altamente heterogêneas: um Apple M4 Pro, um Qualcomm Snapdragon 8 Elite e uma GPU de desktop NVIDIA RTX 4070. Utilizando o AI Model Efficiency Toolkit (AIMET), aplicamos tanto a Quantização Pós-Treinamento (PTQ) quanto o Treinamento Ciente da Quantização (QAT) em arquiteturas otimizadas para borda (MobileNetV1, SqueezeNet) e de uso geral (ResNet). Nossos resultados demonstram que o QAT recupera com sucesso a capacidade representacional perdida pela PTQ em modelos compactos. Além disso, mostramos que aceleradores de borda de domínio específico executando cargas de trabalho INT8 nativamente não apenas rivalizam com a latência sequencial de uma GPU de desktop de 200 Watts, mas o fazem com uma eficiência extraordinária. Especificamente, a quantização proporcionou uma redução de energia de até 89,5% no Apple Neural Engine e uma aceleração de latência de 9,14$\times$ na NPU Snapdragon Hexagon em comparação com suas linhas de base não quantizadas, provando que a quantização de inteiros é um pré-requisito arquitetural para superar o gargalo de memória e viabilizar uma IA de borda sustentável em tempo real.

Palavras-chave: Redes Neurais Profundas. Quantização.

LIST OF ABBREVIATIONS AND ACRONYMS

AI	Artificial Intelligence
AIMET	AI Model Efficiency Toolkit
ALU	Arithmetic Logic Unit
ANN	Artificial Neural Network
ARM	Advanced RISC Machine
BF16	Brain Floating Point
CLE	Cross-Layer Equalization
CNN	Convolutional Neural Network
CPU	Central Processing Unit
DNN	Deep Neural Network
DSP	Digital Signal Processor
FFNN	Feedforward Neural Network
FLOPs	Floating-Point Operations
FP16	16-bit Floating-Point
FP32	32-bit Floating-Point
GPU	Graphics Processing Unit
HW	Hardware
INT4	4-bit Integer
INT8	8-bit Integer
IoT	Internet of Things
MLP	Multi-Layer Perceptron
MSE	Mean Squared Error
NAS	Neural Architecture Search
PTQ	Post-Training Quantization

QAT	Quantization-Aware Training
ReLU	Rectified Linear Unit
RPi	Raspberry Pi
STE	Straight-Through Estimator
Tanh	Hyperbolic Tangent
ADB	Android Debug Bridge
ANE	Apple Neural Engine
CUDA	Compute Unified Device Architecture
FPS	Frames Per Second
JIT	Just-In-Time
LLM	Large Language Model
MAC	Multiply-Accumulate
NPU	Neural Processing Unit
ONNX	Open Neural Network Exchange
QNN	Qualcomm AI Engine Direct
QoS	Quality of Service
SoC	System-on-Chip
VRAM	Video Random Access Memory
SGD	Stochastic Gradient Descent

LIST OF FIGURES

Figure 2.1 Mathematical Model of an Artificial Neuron 17
Figure 2.2 Data flow and forward propagation through the Input, Hidden, and Output layers of a Feedforward Neural Network. 20
Figure 2.3 Visual comparison of the Sigmoid, Hyperbolic Tangent (Tanh), and Rectified Linear Unit (ReLU) activation functions. 22
Figure 2.4 Computational graph of backpropagation. Blue solid lines represent the forward pass, and red dashed lines represent the backward flow of gradients. 24
Figure 2.5 Visualization of the gradient descent optimization process on a complex loss surface. The directed path represents the iterative parameter updates moving in the direction of the steepest descent toward a minimum. 25
Figure 2.6 SqueezeNet overview and Fire Module detailing 27
Figure 2.7 MobileNetV1 Architecture 28
Figure 2.8 Comparison of Precision and Data Types. The figure illustrates the bit allocation (sign, exponent, and mantissa) for FP32, BF16, FP16, and INT8, highlighting the trade-off between precision and dynamic range. 31
Figure 2.9 Visual comparison of Symmetric and Asymmetric Quantization mapping. . 34
Figure 2.10 Quantization-Aware Training (QAT, Left) versus Post-Training Quantization (PTQ, Right) 36

Figure 3.1 End-to-end quantization and multi-platform deployment workflow. 41
Figure 3.2 General AIMET quantization Workflow 47

Figure 4.1 Latency and Throughput comparison on the Snapdragon 8 Elite. The GPU (INT8 QAT) metrics illustrate the fallback bottleneck, while ResNet50 (FP32 only) serves as a heavy-weight baseline. 60
Figure 4.2 Latency and Throughput comparison between CPU and ANE on the Apple M4 Pro. CPU (INT8 QAT) bars illustrate the lack of integer acceleration on standard compute, while ResNet50 serves as a heavy-weight baseline. 62
Figure 4.3 Latency and Throughput on the NVIDIA RTX 4070. The ResNet50 (FP32 only) serves as a compute-bound reference. 63
Figure 4.4 Quantization performance yield: Relative speedup achieved by moving from unquantized general-purpose compute (FP32) to INT8 dedicated acceleration across platforms. 64
Figure 4.5 Energy per Inference (mJ) for MobileNetV1 across all evaluated platforms. Lower is better. 67

LIST OF TABLES

Table 2.1 Neural Network Components: Weights, Biases, and Activation Functions....19
Table 2.2 Common Loss Functions for Supervised Learning Tasks.....23
Table 2.3 Summary of efficient neural network architectures and their foundational structural innovations for resource-constrained devices. 29
Table 2.4 Overview of related works focusing on hardware-aware quantization. The symbol (-) indicates the strategy is not explored, (V) indicates it is explicitly explored, and (VV) indicates a comprehensive analysis or primary focus. .40

Table 3.1 Overview of the Selected Neural Network Architectures 42
Table 3.2 Power Profiling Tools and Sampling Intervals per Platform.....55

Table 4.1 Top-1 Accuracy on Imagenette2 across Quantization Strategies 57
Table 4.2 Performance comparison on Snapdragon 8 Elite (GPU vs. NPU).....59
Table 4.3 Performance comparison on Apple M4 Pro (CPU vs. ANE) 61
Table 4.4 Performance comparison on NVIDIA RTX 4070.....63
Table 4.5 Cross-Platform Energy Efficiency and Quantization Savings.....65

CONTENTS

1 INTRODUCTION 13
1.1 Scope and Research Goals 14
2 BACKGROUND 16
2.1 Neural Networks 16
2.1.1 The Basic Unit: The Artificial Neuron 16
2.1.2 Weights, Biases and Activation Functions 18
2.1.3 Data Flow: Forward Propagation 19
2.1.3.1 Vectorization and Matrix Operations 20
2.1.4 Activation Functions 21
2.1.4.1 Sigmoid and Hyperbolic Tangent (Tanh) 21
2.1.4.2 Rectified Linear Unit (ReLU) 22
2.1.5 Learning Process 23
2.1.5.1 The Loss Function 23
2.1.5.2 Backpropagation 23
2.1.5.3 Optimization (Gradient Descent) 24
2.1.6 Network Architectures 26
2.1.6.1 Efficient Architectures for Constrained Devices 26
2.2 Quantization 29
2.2.1 Precision and Data Types 30
2.2.2 Range Mapping and Scaling 32
2.2.3 Symmetric vs. Asymmetric Quantization 33
2.2.4 Post-Training Quantization (PTQ) 35
2.2.5 Quantization-Aware Training (QAT) 36
2.2.5.1 Backpropagation and the Straight-Through Estimator (STE) 36
2.2.6 Quantization in Practice: Frameworks and Toolkits 37
2.3 Related Work 39
3 PROPOSAL 41
3.1 Model Characterization and Workload 41
3.1.1 Selected Neural Network Architectures 42
3.1.2 Workload and Data Preprocessing Pipeline 42
3.2 Training and Quantization Pipeline (AIMET) 43
3.2.1 Dataset Splitting and Data Augmentation 44
3.2.2 Post-Training Quantization (PTQ) Configuration 44
3.2.3 Quantization-Aware Training (QAT) Hyperparameters 44
3.2.4 Quantization Strategies: PTQ and QAT 45
3.2.5 Standardized Artifact Generation 46
3.3 Apple Silicon Evaluation Pipeline (M4 Pro) 47
3.3.1 Stage 1: CoreML Compilation 47
3.3.2 Stage 2: Quantization Application (INT8 Strategy) 48
3.3.3 Stage 3: Latency and Throughput Profiling 48
3.3.4 Stage 4: Energy Measurement via Powermetrics 49
3.3.5 Hardware Residency Verification and Testing 49
3.4 Mobile Edge Evaluation Pipeline (Snapdragon 8 Elite) 50
3.4.1 Stage 1: Model Conversion and QNN Integration 50
3.4.2 Stage 2: Target Allocation (NPU vs. GPU) 51
3.4.3 Stage 3: Remote Execution and Throughput Profiling 51
3.4.4 Stage 4: Energy Profiling via Perfetto 52

3.5 Workstation/GPU Evaluation Pipeline (RTX 4070) 52
3.5.1 Stage 1: Model Optimization via TensorRT 52
3.5.2 Stage 2: Latency and Throughput Profiling 53
3.5.3 Stage 3: Energy Measurement via `nvidia-smi` 53
3.6 Instrumentation and Evaluation Metrics 54
3.6.1 Latency and Throughput Metrics 54
3.6.2 Power and Energy Efficiency Metrics 55
4 RESULTS AND DISCUSSION 57
4.1 Impact of Quantization on Model Accuracy 57
4.1.1 Degradation in Edge-Optimized Architectures 57
4.1.2 The Effectiveness of Quantization-Aware Training 58
4.2 Performance Analysis: Latency and Throughput 58
4.2.1 Mobile Edge Acceleration (Snapdragon 8 Elite) 59
4.2.2 Apple Silicon Efficiency (M4 Pro) 60
4.2.3 Workstation Baseline (RTX 4070) 62
4.2.4 Cross-Platform Latency Comparison 63
4.3 Energy Efficiency Profile 65
4.3.1 Hardware-Specific Energy Savings (Edge Platforms) 65
4.3.2 The Cost of General-Purpose Compute (RTX 4070 vs. Edge) 66
4.4 Architectural Trade-offs and Discussion 67
4.4.1 Edge-Optimized vs. General-Purpose Architectures 67
4.4.2 Native INT8 Execution and Bandwidth Bottlenecks 68
4.5 Summary of Key Findings 69
5 CONCLUSION AND FUTURE WORK 70
5.1 Final Conclusions 70
5.2 Future Work 71
REFERENCES 72

1 INTRODUCTION

The proliferation of deep neural networks (DNNs) has revolutionized numerous fields, establishing state-of-the-art performance in tasks ranging from computer vision to natural language processing since the advent of landmark architectures like AlexNet (KRIZHEVSKY; SUTSKEVER; HINTON, 2012). However, the remarkable accuracy of these models often comes at the cost of significant computational and memory requirements. This inherent complexity poses a critical challenge to their deployment on edge and embedded devices, such as smartphones, IoT devices, and autonomous systems, which are governed by stringent constraints on power consumption, latency, and available memory. This gap between model complexity and device capability has catalyzed extensive research into model optimization techniques designed to enable efficient and responsive on-device artificial intelligence.

Among the various strategies for model optimization, which include the design of compact architectures like MobileNets (HOWARD et al., 2017) from the ground up, network quantization has emerged as a particularly effective and widely applicable approach. Quantization reduces the numerical precision of a network's weights and activations, typically converting them from 32-bit floating-point to lower-bit representations. A highly promising paradigm within this domain is integer-only quantization, which transforms a model to perform all computations using integer arithmetic. This approach is motivated by the fact that integer operations are substantially faster and more energy-efficient than floating-point calculations on a wide range of processors, especially on low-power microcontrollers and specialized hardware that often have limited or no floating-point support.

The foundational work by Jacob et al. (JACOB et al., 2017) demonstrated a robust framework for 8-bit integer-only inference, showing that with a co-designed **quantization-aware training** (QAT) strategy, it is possible to achieve significant reductions in latency while maintaining model accuracy close to the original floating-point baseline. Their results provided compelling evidence of the technique's efficacy on popular mobile CPUs, establishing it as a viable solution for deploying complex models on smartphones. Nevertheless, the modern edge computing landscape is exceptionally diverse, extending far beyond mobile phones to include a heterogeneous mix of resource-constrained microcontrollers, digital signal processors (DSPs), and specialized AI accelerators. The performance benefits and practical adaptability of integer-only quantization across this broader spectrum of hardware have not yet been systematically investigated.

This research addresses this gap by presenting a systematic evaluation of integer-only quantization for neural network deployment across a diverse set of platforms. We move beyond a singular focus on mobile CPUs to analyze the impact of this technique on a variety of resource-constrained and heterogeneous edge devices. By empirically measuring its effects on critical metrics such as inference latency and energy efficiency, and assessing its adaptability to different underlying hardware architectures, this work aims to provide a comprehensive understanding of its true potential. Ultimately, we seek to illustrate that integer-only quantization is not merely an optimization for a specific class of device, but a practical, portable, and powerful solution for enabling efficient neural network inference across the wide and varied ecosystem of edge hardware.

1.1 Scope and Research Goals

To systematically investigate the practical viability and architectural implications of integer-only quantization, this study narrows its scope to a highly controlled, cross-platform evaluation pipeline. The research focuses on 8-bit integer (INT8) quantization, evaluating both Post-Training Quantization (PTQ) and Quantization-Aware Training (QAT) paradigms using the AI Model Efficiency Toolkit (AIMET).

The evaluation workload is strictly constrained to a batch size of 1 ($N = 1$) to accurately simulate the sequential, latency-critical nature of real-time edge processing (e.g., sensor data streams or continuous camera feeds). The target architectures span the spectrum of modern computing: a highly constrained mobile platform (Qualcomm Snapdragon 8 Elite), an advanced unified-memory laptop architecture (Apple M4 Pro), and an unconstrained workstation baseline (NVIDIA RTX 4070). The neural networks selected for this evaluation include edge-optimized topologies (MobileNetV1 and SqueezeNet) alongside a dense, general-purpose architecture (ResNet50) to serve as a computational counterpoint.

Within this scope, the primary objective of this thesis is to provide an empirical foundation for hardware-software co-design in Edge AI. This overarching objective is divided into the following specific research goals:

- **Evaluate Quantization Accuracy Trade-offs:** To quantify the inherent predictive degradation introduced by reducing precision to 8-bit integers, specifically analyzing the vulnerability of edge-optimized (parameter-scarce) architectures to PTQ,

and to determine the necessity and efficacy of QAT in recovering representational capacity.

- **Benchmark Sequential Latency across Heterogeneous Domains:** To measure and compare real-time inference latency and throughput across vastly different silicon domains—challenging the conventional paradigm that mobile and edge hardware (NPUs) are fundamentally slower than workstation GPUs for sequential ($N = 1$) processing.
- **Quantify Absolute Energy Efficiency:** To move beyond raw power (Wattage) measurements by calculating the *Energy per Inference* (mJ). This goal aims to expose the true battery-life impact and sustainability of offloading quantized workloads from general-purpose processing cores (CPUs/GPUs) to domain-specific matrix accelerators (e.g., the Apple Neural Engine and Qualcomm Hexagon NPU).
- **Analyze Architectural Bottlenecks and the "Memory Wall":** To investigate the deeper architectural behavior of quantized networks on different hardware. Specifically, to contrast compute-bound execution against kernel-bound execution, and to empirically demonstrate how quantization alleviates memory bandwidth bottlenecks—proving whether edge efficiency is driven more by cheaper arithmetic or by reduced memory payload transfers.

By addressing these goals, this research intends to prove that INT8 quantization transcends being a mere micro-optimization, establishing it instead as an absolute architectural prerequisite for deploying viable, high-performance artificial intelligence within the thermal and energy envelopes of modern edge devices.

2 BACKGROUND

The challenge of deploying computationally intensive deep neural networks on resource-constrained edge devices has spurred a vast and active field of research (HAN; MAO; DALLY, 2016a; CHENG et al., 2020). The solutions developed to address this computational bottleneck can be broadly categorized into two complementary research directions: 1) the design of fundamentally efficient network architectures, and 2) the application of post-hoc compression and quantization techniques to reduce the complexity and memory footprint of existing models. While this work primarily builds upon the second category, an understanding of both domains is essential to contextualize the current state-of-the-art in efficient deep learning.

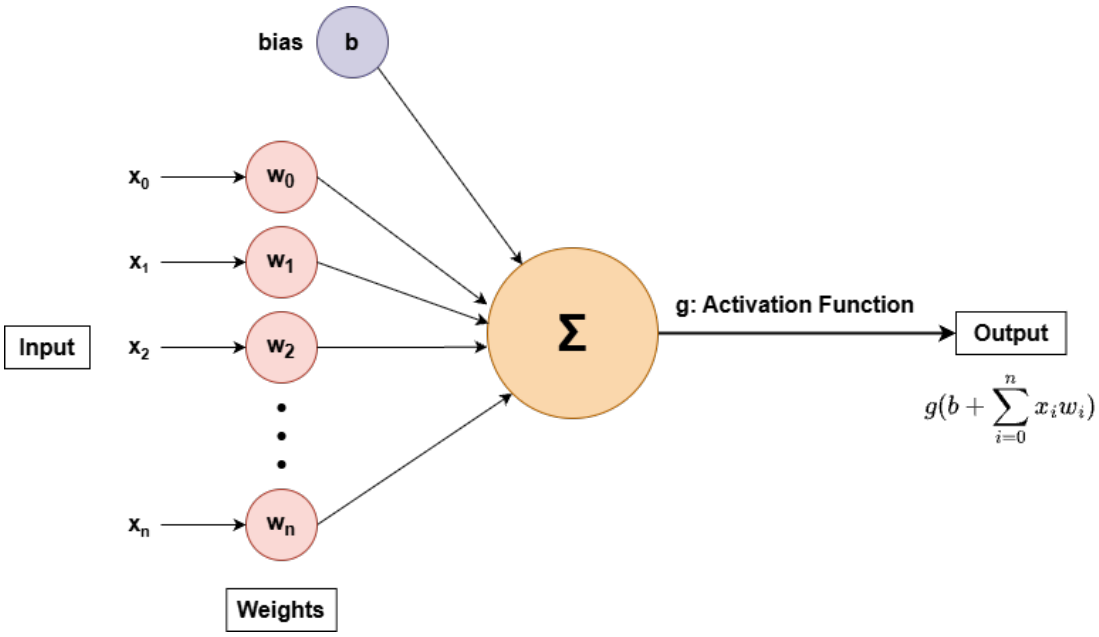
2.1 Neural Networks

Before exploring the methods used to optimize and compress neural networks, it is necessary to define the foundational computational models upon which these techniques operate. Artificial Neural Networks (ANNs) are mathematically formulated systems inspired by the biological neural networks that constitute animal brains.

2.1.1 The Basic Unit: The Artificial Neuron

The fundamental building block of any ANN is the artificial neuron, sometimes referred to as a node or unit. The conceptual origin of this computational unit traces back to the seminal work of McCulloch and Pitts in 1943 (MCCULLOCH; PITTS, 1943), who proposed the first mathematical model of a neuron as a simple logic gate with binary outputs. This abstraction was later advanced by Frank Rosenblatt in 1958 with the introduction of the Perceptron (ROSENBLATT, 1958), an algorithm that incorporated learnable continuous weights, thereby laying the groundwork for modern machine learning optimizations.

Figure 2.1 – Mathematical Model of an Artificial Neuron



Source: The author.

As illustrated in Figure 2.1, the modern artificial neuron functions as a mathematical transformation applied to incoming data. It receives an input vector $\mathbf{x} = [x_1, x_2, \ldots, x_n]$, where each x_i represents either a raw data feature or the output from a neuron in a preceding layer. Each input connection is associated with a learnable weight w_i, which dictates the relative importance of that specific input.

The neuron computes the dot product of the inputs and their corresponding weights, and adds a learnable scalar value known as the bias (b). This linear combination produces an intermediate pre-activation value, z:

$$z = \sum_{i=1}^{n} w_i x_i + b$$

Finally, to allow the network to model complex, non-linear relationships, this scalar z is passed through a designated non-linear activation function $f(\cdot)$, producing the neuron's final output activation y:

$$y = f(z)$$

This output y is then propagated forward, serving as the input to subsequent neurons in the network architecture.

2.1.2 Weights, Biases and Activation Functions

Following the mathematical formulation of the artificial neuron, the actual learning capacity of a neural network resides in its parameters: the weights and the biases. These components are not hard-coded; rather, they are iteratively adjusted during the training process to map input data to the desired output predictions correctly.

The **weights** (w) represent the strength or importance of the connections between neurons. Mathematically, they act as coefficients that scale the incoming input signals. If a specific feature in the data is highly predictive of the desired outcome, the network will learn to assign a large magnitude weight to that connection, amplifying its influence on the neuron's final output. Conversely, irrelevant features are suppressed via weights approaching zero. This iterative adjustment is the core mechanism of network learning (RUMELHART; HINTON; WILLIAMS, 1986).

While weights scale the input, the **bias** (b) provides a learnable baseline offset. Without a bias term, a neuron's pre-activation value would always be strictly zero if all its inputs were zero, regardless of the learned weights. The bias solves this limitation by allowing the activation function to be shifted horizontally along the axis, effectively acting as an activation threshold. This gives the network the flexibility to activate a neuron even when the weighted sum of its inputs is zero or negative.

Finally, the resulting linear combination of weights, inputs, and bias is passed through an **activation function** (often denoted as σ or f). Without this step, a neural network, regardless of its depth, would mathematically collapse into a single linear transformation, rendering it incapable of solving complex problems. The activation function introduces the critical non-linearity required to model intricate decision boundaries in high-dimensional data (LECUN; BENGIO; HINTON, 2015).

To synthesize the roles of these fundamental elements, Table 2.1 provides a concise definition and outlines the specific purpose of each component within the broader network architecture.

Component	Definition	Purpose in the Network
Weights (w)	Learnable coefficients that multiply the input data passing through the network's connections.	They determine the **strength or importance** of a particular input feature. If a weight is high, that specific input has a significant influence on the neuron's output. During training, the network adjusts these weights to minimize errors.
Bias (b)	Learnable constant value added to the weighted sum of inputs before the activation function is applied.	It allows the network to **shift the activation function** left or right. This guarantees that a neuron can still activate and learn patterns even if all the input values are exactly zero.
Activation Function (σ)	A mathematical operation applied to the output of a neuron.	It introduces **non-linearity** into the network, allowing it to learn and model complex, non-linear patterns.

Table 2.1 – Neural Network Components: Weights, Biases, and Activation Functions

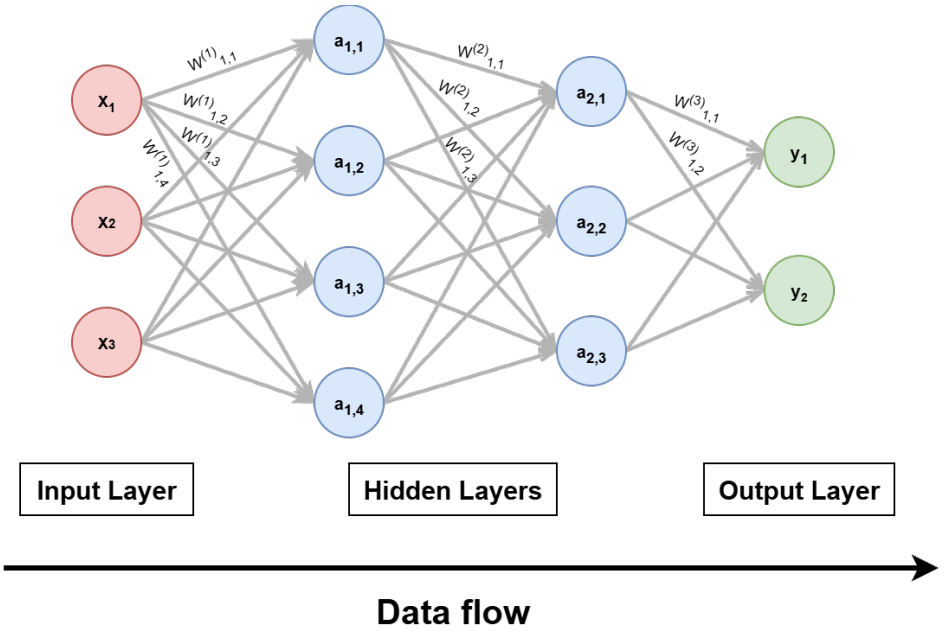
2.1.3 Data Flow: Forward Propagation

In a standard Feedforward Neural Network (FFNN), information moves in a strictly unidirectional manner, without any feedback loops or cycles. The data originates at the input, propagates through a series of intermediate mathematical transformations, and culminates at the output. This sequential process of generating a prediction from input data is formally known as forward propagation.

As depicted in Figure 2.2, the network architecture is typically organized into three distinct types of sequential layers:

- **Input Layer:** This layer receives the raw data vector, represented in the diagram by the input nodes x_1, x_2, and x_3. The number of neurons in this layer directly corresponds to the dimensionality of the input features.
- **Hidden Layers:** Located between the input and output, these layers perform the core feature extraction and representation learning. The illustrated architecture is classified as a "deep" neural network because it utilizes multiple hidden layers: the first containing four activation nodes ($a_1^{[1]} \dots a_4^{[1]}$) and the second containing

Figure 2.2 – Data flow and forward propagation through the Input, Hidden, and Output layers of a Feedforward Neural Network.



Source: The author.

three ($a_1^{[2]} \dots a_3^{[2]}$). They transform the data into increasingly abstract, high-level representations.

- **Output Layer:** The final layer processes the information from the last hidden layer to produce the model's final predictions, outputting the values y_1 and y_2. This structure could represent a set of class probabilities for classification or continuous values for multivariate regression.

2.1.3.1 Vectorization and Matrix Operations

While the previous section defined the equation for a single artificial neuron, modern deep learning frameworks do not compute these operations individually. Instead, they leverage vectorization—grouping the weights, biases, and inputs of an entire layer into matrices and vectors. This allows the network to compute the outputs of all neurons in a layer simultaneously using highly optimized linear algebra operations, which is essential for efficient execution on parallel hardware.

For any given layer l, the pre-activation vector $\mathbf{z}^{[l]}$ is computed by multiplying the layer's weight matrix $\mathbf{W}^{[l]}$ by the activation vector from the preceding layer $\mathbf{a}^{[l-1]}$, and subsequently adding the bias vector $\mathbf{b}^{[l]}$:

$$\mathbf{z}^{[l]} = \mathbf{W}^{[l]}\mathbf{a}^{[l-1]} + \mathbf{b}^{[l]}$$

For the initial computation at the first hidden layer ($l = 1$), the input $\mathbf{a}^{[0]}$ is simply the raw input feature vector $\mathbf{x}$.

Following this linear transformation, a chosen non-linear activation function $f(\cdot)$ is applied element-wise to the vector $\mathbf{z}^{[l]}$ to produce the layer's final activation output:

$$\mathbf{a}^{[l]} = f(\mathbf{z}^{[l]})$$

During forward propagation, this mathematical sequence is chained together. The output $\mathbf{a}^{[l]}$ immediately serves as the input $\mathbf{a}^{[l-1]}$ for the subsequent layer $l + 1$. This cascading cycle repeats until the data traverses the entire network and reaches the output layer.

2.1.4 Activation Functions

As established in the mathematical formulation of the artificial neuron, the linear combination of weights, inputs, and bias produces an intermediate value, z. This value is subsequently passed through a non-linear activation function, $f(z)$. If a neural network relied solely on linear operations, any multi-layer architecture would mathematically collapse into a single linear transformation, severely limiting its capacity to model complex, real-world data (BISHOP, 2006). Activation functions introduce the critical non-linearities that allow deep networks to learn intricate decision boundaries.

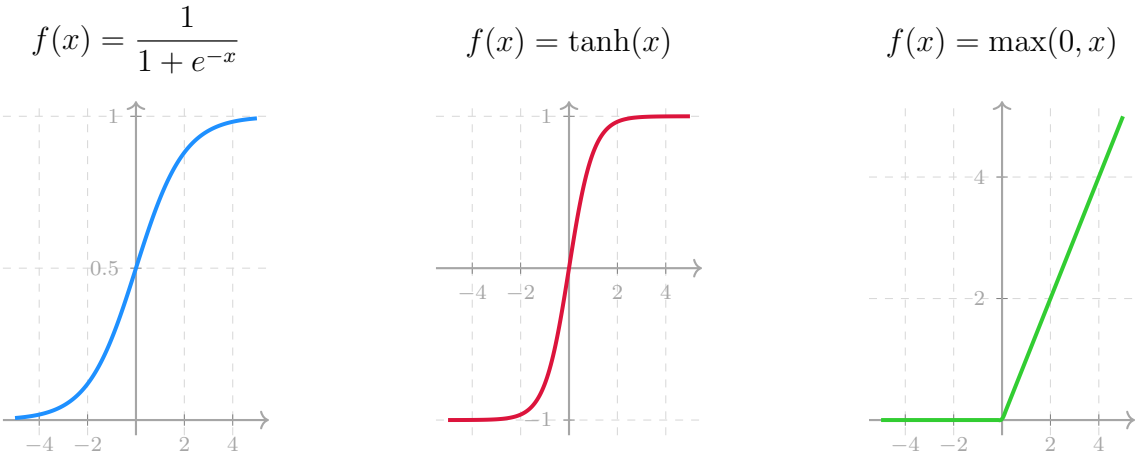
Figure 2.3 illustrates three of the most historically and practically significant activation functions used in deep learning.

2.1.4.1 Sigmoid and Hyperbolic Tangent (Tanh)

Early neural network architectures relied heavily on the **Sigmoid** function, which maps any real-valued number into a bounded range between 0 and 1. While useful for binary classification where outputs represent probabilities, the sigmoid function suffers from the "vanishing gradient" problem. As input values approach extreme positive or negative numbers, the curve flattens, causing the derivative to approach zero. During backpropagation, these near-zero gradients prevent earlier layers from updating effectively, stalling the learning process in deep networks.

The **Hyperbolic Tangent (Tanh)** function resolves one of the sigmoid's limitations by mapping values to a range between -1 and 1. Because it is zero-centered, it

Figure 2.3 – Visual comparison of the Sigmoid, Hyperbolic Tangent (Tanh), and Rectified Linear Unit (ReLU) activation functions.



Source: The author

often results in faster convergence during training, as the gradients are distributed more symmetrically. However, like the sigmoid, Tanh still saturates at its extremes and remains susceptible to vanishing gradients.

2.1.4.2 Rectified Linear Unit (ReLU)

To address the optimization challenges of deep networks, the **Rectified Linear Unit (ReLU)** emerged as the standard activation function for modern architectures. ReLU is a piecewise linear function that outputs the input directly if it is positive, and outputs zero if it is negative (KRIZHEVSKY; SUTSKEVER; HINTON, 2012):

$$f(x) = \max(0, x)$$

Despite its simplicity, ReLU introduces sufficient non-linearity while offering profound advantages for both training and inference. First, its derivative is exactly 1 for all positive inputs, heavily mitigating the vanishing gradient problem and accelerating convergence. Second, from an efficiency standpoint, ReLU is extraordinarily computationally cheap. Unlike Sigmoid and Tanh, which require expensive exponential calculations, ReLU only requires a simple thresholding operation (identifying if the sign bit is negative).

This computational efficiency is paramount for the resource-constrained devices and efficient architectures (such as MobileNets) discussed later in this work, as it drastically reduces the arithmetic complexity of the forward pass.

2.1.5 Learning Process

2.1.5.1 The Loss Function

Upon completing the forward pass and generating a prediction ($\hat{y}$), the network requires a mathematical metric to evaluate the deviation of this prediction from the actual ground truth (y). The loss function (often referred to as the cost function or objective function) serves this exact purpose by quantifying the predictive error. It outputs a scalar value that the subsequent optimization algorithm systematically seeks to minimize.

The choice of loss function is inherently dependent on the specific task the neural network is designed to perform. Table 2.2 details two of the most prevalent loss functions used in supervised learning: Mean Squared Error (MSE) for regression tasks and Binary Cross-Entropy for classification tasks.

Task Type	**Loss Function**	**Formula**
Regression	Mean Squared Error (MSE)	$J = \frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2$
Classification	Binary Cross-Entropy	$J = -\frac{1}{N}\sum_{i=1}^{N}[y_i \log(\hat{y}_i) + (1 - y_i)\log(1 - \hat{y}_i)]$

Table 2.2 – Common Loss Functions for Supervised Learning Tasks

In these formulations, J represents the total calculated loss over a given set of data. The variable N denotes the total number of samples processed (either the entire dataset or a specific mini-batch), while y_i and $\hat{y}_i$ represent the true target value and the network's predicted value for the i-th sample, respectively. By aggregating this error across all N samples, the network obtains a comprehensive measure of its current performance, which acts as the starting point for calculating gradients during backpropagation.

2.1.5.2 Backpropagation

Once the loss is computed at the output layer, the network must determine how to adjust its internal parameters to minimize this error. This is achieved through backpropagation (backward propagation of errors). Backpropagation is an algorithm that efficiently computes the gradient of the loss function with respect to each weight and bias in the network by systematically applying the chain rule of calculus, traversing backward from the output layer to the input layer (WERBOS, 1990).

As illustrated in Figure 2.4, when the output of a single node $\mathbf{t}$ acts as an input to

multiple subsequent nodes (in this case, $\mathbf{t}_1$ and $\mathbf{t}_2$), the gradients flowing backward from these branches must be accumulated. This process is governed by the multivariate Chain Rule.

In this computational graph, the bar notation (e.g., $\bar{\mathbf{t}}$) is used as a shorthand to represent the gradient of the final scalar loss function J with respect to that specific variable:

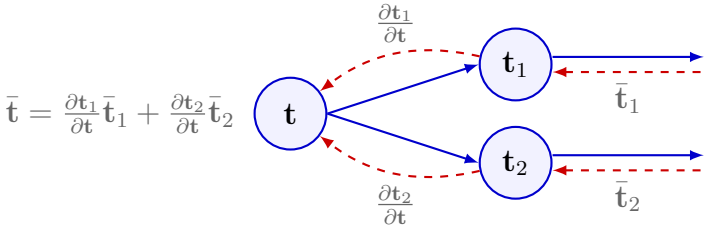
$$\bar{\mathbf{t}} = \frac{\partial J}{\partial \mathbf{t}}$$

During the backward pass, node $\mathbf{t}$ receives the upstream gradients $\bar{\mathbf{t}}_1$ and $\bar{\mathbf{t}}_2$ from the subsequent layer. To compute its own gradient ($\bar{\mathbf{t}}$) to pass further back down the network, it multiplies each upstream gradient by the local derivative of the forward operation and sums the results. Mathematically, this is expressed as:

$$\frac{\partial J}{\partial \mathbf{t}} = \frac{\partial J}{\partial \mathbf{t}_1}\frac{\partial \mathbf{t}_1}{\partial \mathbf{t}} + \frac{\partial J}{\partial \mathbf{t}_2}\frac{\partial \mathbf{t}_2}{\partial \mathbf{t}}$$

This ensures that the total contribution of node $\mathbf{t}$ to the overall error is accurately captured across all the paths it influenced during forward propagation.

Figure 2.4 – Computational graph of backpropagation. Blue solid lines represent the forward pass, and red dashed lines represent the backward flow of gradients.



Source: The author

2.1.5.3 Optimization (Gradient Descent)

While backpropagation provides the partial derivatives (the gradient) of the loss function with respect to the network's parameters, it does not actually update the model. The procedure that applies these gradients to modify the weights and biases is called the optimization algorithm.

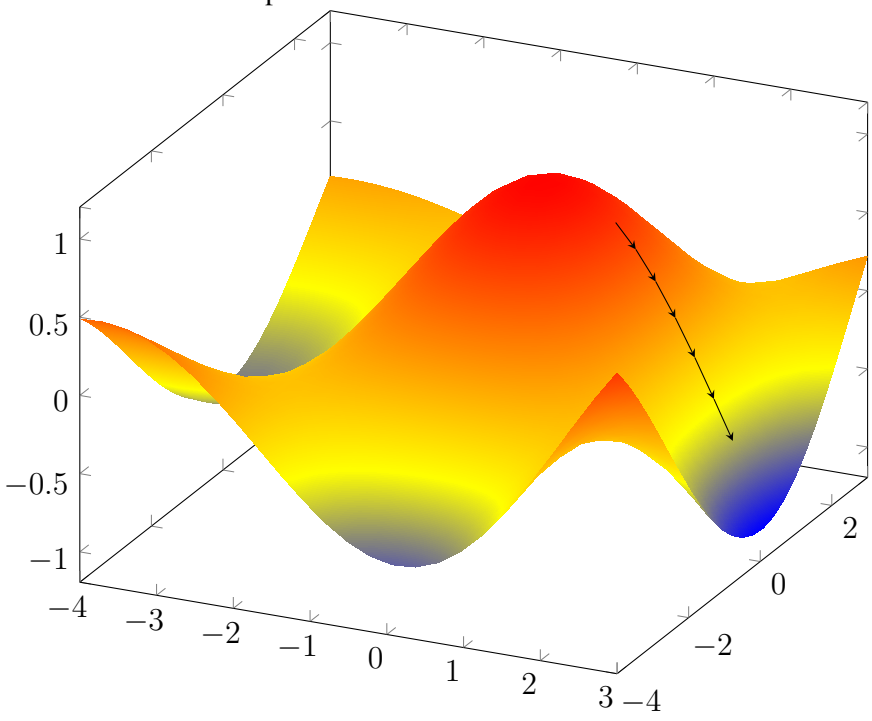
The most fundamental and widely used algorithm for this task is Gradient Descent. The core principle of Gradient Descent is to update the parameters in the opposite direction of the gradient of the objective function $\nabla J(\mathbf{w})$ (RUDER, 2016). Mathematically, the update rule for a given parameter w at iteration t is defined as:

$$w_{t+1} = w_t - \alpha \nabla J(w_t)$$

Here, α represents the learning rate—a crucial hyperparameter that determines the step size taken during each iteration. If α is too small, convergence will be exceedingly slow; if it is too large, the algorithm may overshoot the minimum and diverge.

Figure 2.5 provides a geometric representation of the optimization trajectory produced by the gradient descent algorithm. In the context of neural network training, the topology of the surface represents the multidimensional loss function J, mapped across the network's parameter space. The algorithm initializes at an arbitrary coordinate and iteratively calculates the gradient of the loss with respect to the model's parameters. Because the gradient vector inherently points in the direction of the steepest ascent, the optimizer applies a proportional update in the exact opposite direction—scaled by the predefined learning rate. This iterative descent, depicted by the sequential arrows in the figure, systematically guides the network's parameters down the loss landscape until they converge at a local or global minimum, thereby minimizing the predictive error of the model.

Figure 2.5 – Visualization of the gradient descent optimization process on a complex loss surface. The directed path represents the iterative parameter updates moving in the direction of the steepest descent toward a minimum.



Source: The author

2.1.6 Network Architectures

The architecture of a neural network defines its structural topology—specifically, how neurons are organized into layers and the pattern of connections between them. This macro-structure dictates how effectively a model can learn representations from data and how computationally expensive that learning process will be.

Historically, the foundational architecture is the Multi-Layer Perceptron (MLP), consisting entirely of fully connected layers. While MLPs are universal function approximators, they scale poorly with high-dimensional data, leading to an explosion in parameter count. For spatial data such as images, the Convolutional Neural Network (CNN) emerged as the standard architecture. By utilizing local receptive fields, shared weights, and pooling operations, CNNs drastically reduce the number of parameters compared to MLPs while maintaining the spatial hierarchy of the input.

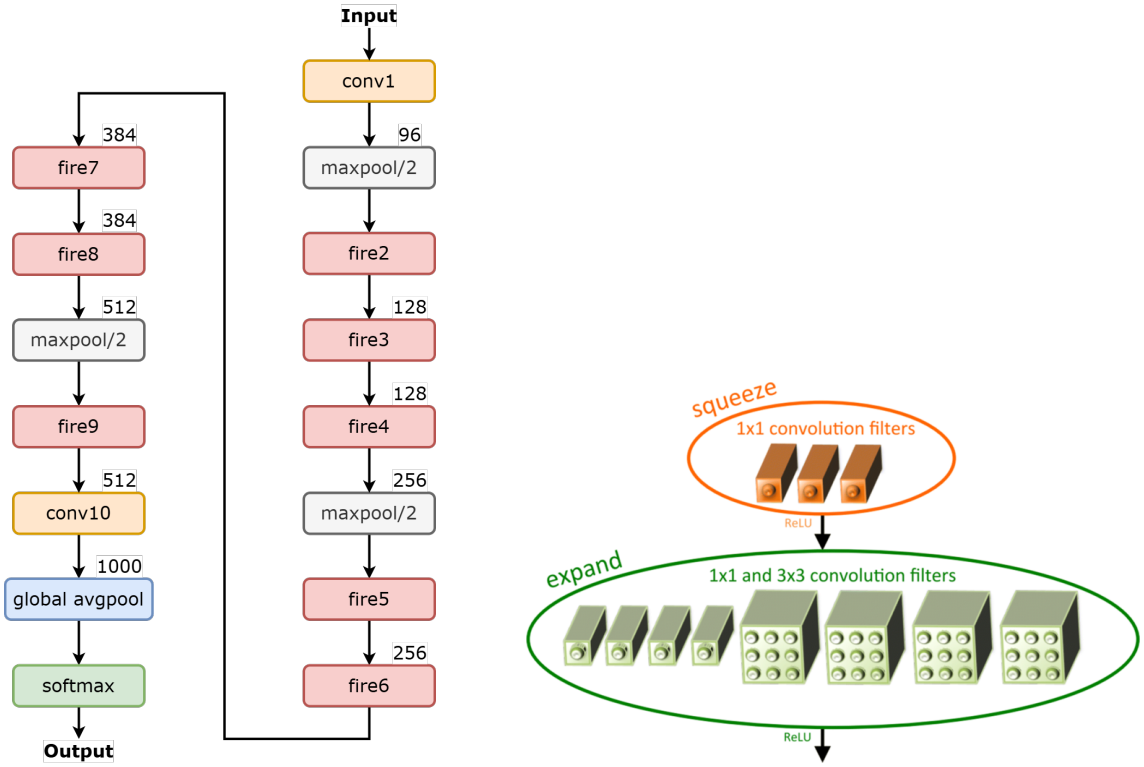
However, standard CNNs like VGG or ResNet were primarily designed to maximize accuracy, often disregarding the computational cost. As deep learning expanded to edge devices, the architectural focus necessarily shifted from pure accuracy to a trade-off between performance and efficiency.

2.1.6.1 Efficient Architectures for Constrained Devices

The pursuit of efficient deep learning has evolved beyond simply compressing large, over-parameterized models like VGG (SIMONYAN; ZISSERMAN, 2015) or ResNets (HE et al., 2016). A prominent school of thought focuses on embedding efficiency directly into the network's fundamental building blocks. This approach tackles the primary source of computational cost in CNNs: the standard convolution, whose complexity scales quadratically with the kernel size.

Early contemporaries addressing this challenge, such as SqueezeNet (IANDOLA et al., 2016), focused on aggressive parameter reduction. As illustrated in Figure 2.6, SqueezeNet achieves this through its foundational "Fire Module" (Figure 2.6b). This module strategically employs a "squeeze" layer comprised entirely of 1×1 convolutions to compress the input channels, before feeding them into an "expand" layer that mixes 1×1 and 3×3 filters. This macro-architecture (Figure 2.6a) maintains competitive accuracy while drastically shrinking the overall model size.

Figure 2.6 – SqueezeNet overview and Fire Module detailing



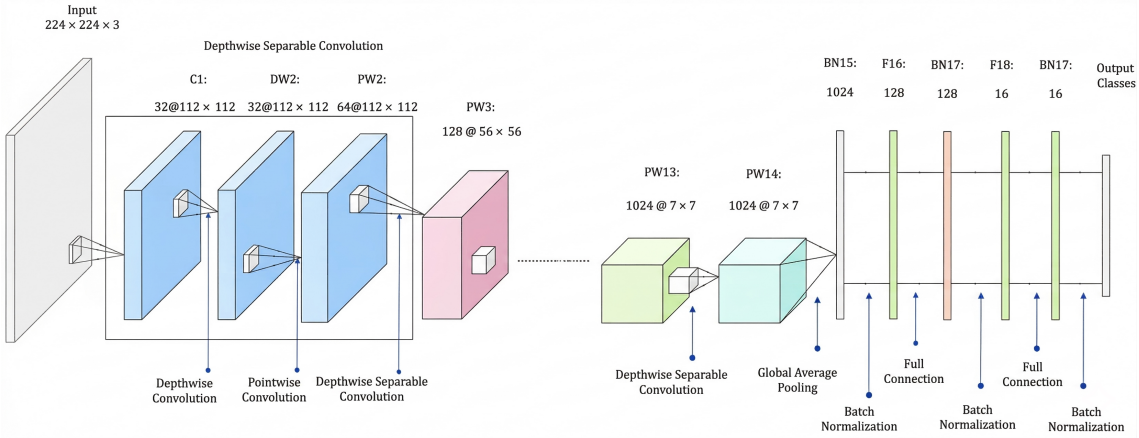
(a) SqueezeNet Architecture Overview

(b) Fire Module Detailed

Source: (IANDOLA et al., 2016)

Following this, a seminal shift occurred with MobileNets (HOWARD et al., 2017), which introduced depthwise separable convolutions as their primary building block. As depicted in the network topology in Figure 2.7, this technique ingeniously factorizes a standard convolution into a lightweight depthwise convolution (applying a single filter per input channel) and a 1×1 pointwise convolution (combining the outputs). This factorization drastically reduces both the parameter count and the number of floating-point operations (FLOPs). Other architectures like ShuffleNet (ZHANG et al., 2017) later addressed the remaining bottleneck of pointwise convolutions by introducing pointwise group convolutions and a channel shuffle operation for even greater efficiency.

Figure 2.7 – MobileNetV1 Architecture



Source: The author.

The principles pioneered in these early models were quickly refined. The successor, MobileNetV2 (SANDLER et al., 2019), introduced the inverted residual block with linear bottlenecks. This structure proved to be more memory-efficient during inference and more effective at preserving feature representations, a critical insight for maintaining accuracy in resource-constrained models. This evolution highlighted that designing for efficiency was not just about reducing FLOPs, but also about carefully managing data flow and feature representation throughout the network. The subsequent development of MobileNetV3 (HOWARD et al., 2019) marked another significant shift, as its architecture was not entirely hand-crafted but was partially discovered using automated search algorithms.

This automation represents a paradigm shift toward Neural Architecture Search (NAS). Instead of relying on human intuition, NAS employs algorithms to explore vast design spaces and discover novel, high-performing architectures tailored to specific constraints (e.g., latency on a mobile CPU). A landmark achievement of this approach is EfficientNet (TAN; LE, 2020), which used NAS to find an optimal baseline architecture and a new compound scaling method. This method uniformly scales the network's depth, width, and input resolution, creating a family of models that set a new state-of-the-art in both accuracy and efficiency. The rise of NAS underscores the idea that the most efficient architectures are often non-obvious and are best discovered through a principled, hardware-aware search.

To synthesize the architectural evolution discussed above, Table 2.3 provides a chronological summary of these pivotal models, highlighting the core structural innovations that enable their computational efficiency.

Architecture	Key Innovation	Primary Mechanism & Benefit
MobileNetV1 (HOWARD et al., 2017)	Depthwise Separable Convolutions	Factorizes standard convolutions into depthwise and 1×1 pointwise, reducing FLOPs and parameters.
SqueezeNet (IANDOLA et al., 2016)	Fire Modules	Employs aggressive 1×1 convolutions to "squeeze" channels before "expanding" them.
ShuffleNet (ZHANG et al., 2017)	Pointwise Group Convolutions & Channel Shuffle	Mitigates 1×1 convolution bottleneck while ensuring feature flow across channel groups.
MobileNetV2 (SANDLER et al., 2019)	Inverted Residuals with Linear Bottlenecks	Expands features to high dimensions for filtering and shrinks them back, improving memory efficiency.
MobileNetV3 (HOWARD et al., 2019)	Hardware-Aware NAS & NetAdapt	Utilizes automated search to discover optimal block configurations tailored to hardware latency constraints.
EfficientNet (TAN; LE, 2020)	Compound Scaling Method	Uses NAS for a baseline and uniformly scales depth, width, and resolution for an optimal accuracy/efficiency ratio.

Table 2.3 – Summary of efficient neural network architectures and their foundational structural innovations for resource-constrained devices.

However, while these architectures are highly effective at reducing theoretical complexity, their practical performance on real hardware remains dependent on more than just FLOPs. Metrics like Memory Access Cost, data reuse, and amenability to parallelization on SIMD hardware often have a greater impact on real-world latency. This is where the motivation to combine efficient designs with quantization becomes even more compelling. The complex topologies and novel activation functions often found in NAS-derived models can introduce unique challenges for quantization, making a co-design approach—where architectural choices are made with an awareness of their quantization-friendliness—not just beneficial, but essential for unlocking maximum performance on target hardware.

2.2 Quantization

Model quantization aims to reduce the numerical precision of a network's parameters (weights and biases) and intermediate activations, thereby lowering the overall

memory footprint and enabling faster, more energy-efficient computation. While the research in this field has explored a wide spectrum of bit-depths—ranging from extreme 1-bit binary neural networks to more moderate 8-bit and 16-bit schemes—the overarching goal remains the same: mapping a high-precision domain to a lower-precision domain with minimal degradation in the model's predictive accuracy.

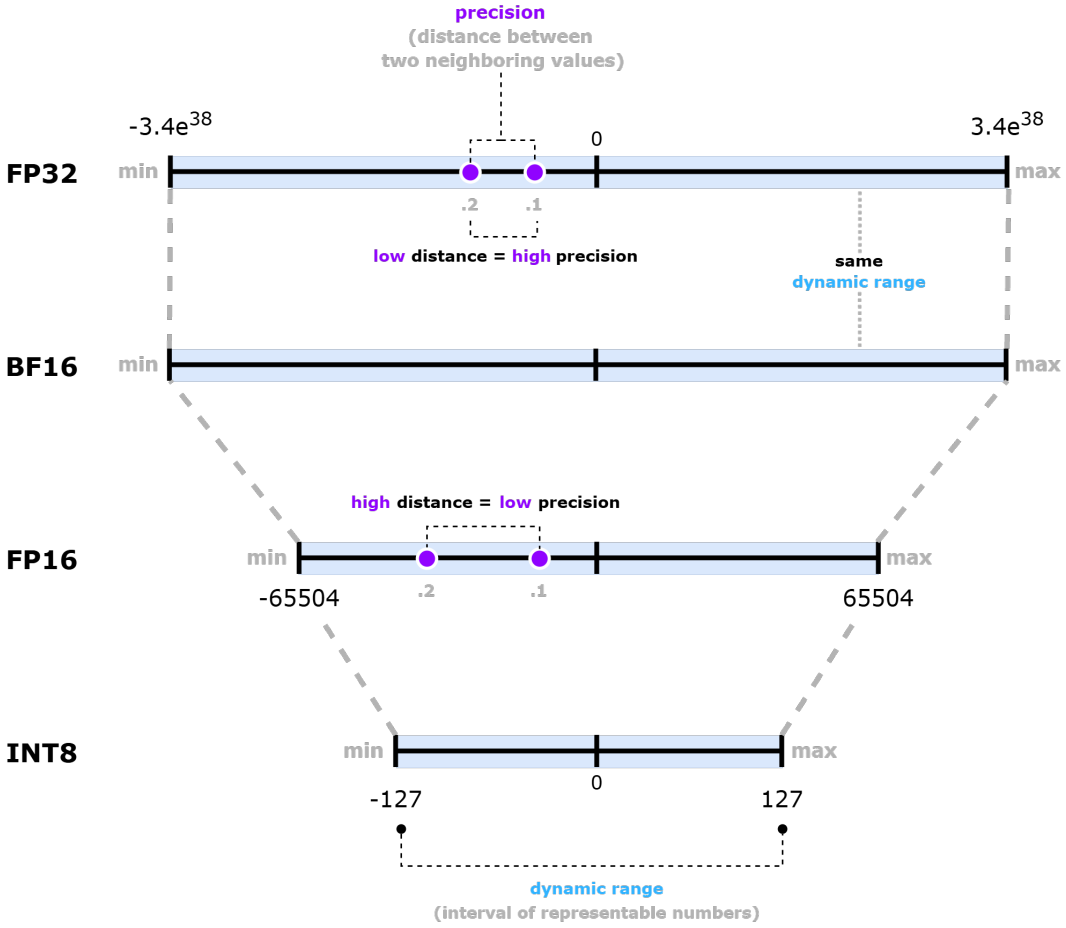
2.2.1 Precision and Data Types

Traditionally, deep neural networks are trained and deployed using 32-bit floating-point (FP32) arithmetic. Following the IEEE 754 standard, an FP32 value allocates 1 bit for the sign, 8 bits for the exponent (which dictates the dynamic range of representable numbers), and 23 bits for the mantissa or fraction (which dictates the numerical precision). This format provides an extraordinarily wide dynamic range, which is critical during training to capture the minuscule gradient updates calculated via backpropagation without suffering from underflow (GUPTA et al., 2015).

However, for inference on resource-constrained edge devices, relying entirely on FP32 is computationally exorbitant and often unnecessary. Empirical evidence has shown that neural networks are inherently robust to noise and do not require perfect precision to make accurate predictions. As a result, modern deployment pipelines heavily leverage lower-precision data types, which are compared in Figure 2.8.

The first step in reducing computational overhead is often downcasting to 16-bit formats. Standard 16-bit floating-point (FP16) reduces the exponent to 5 bits and the mantissa to 10 bits. Alternatively, the Brain Floating Point (BF16) format was specifically designed for deep learning; it purposefully truncates the mantissa to just 7 bits in order to retain the full 8-bit exponent of FP32. This design choice sacrifices raw precision but preserves the FP32 dynamic range, making BF16 highly stable for neural network operations without requiring complex scaling.

Figure 2.8 – Comparison of Precision and Data Types. The figure illustrates the bit allocation (sign, exponent, and mantissa) for FP32, BF16, FP16, and INT8, highlighting the trade-off between precision and dynamic range.



Source: The author.

The most significant efficiency gains for edge deployment, however, are achieved by transitioning to 8-bit integer (INT8) representations. Moving to INT8 reduces the model's memory footprint and bandwidth requirements by a factor of four compared to FP32. Furthermore, it replaces power-hungry floating-point matrix multiplications with highly efficient integer arithmetic logic units (ALUs), drastically improving hardware latency and power consumption.

Despite these benefits, this transition introduces significant mathematical complexity. The dynamic range of an unsigned INT8 value is strictly limited to the interval $[0, 255]$ (or $[-128, 127]$ for signed INT8). Mapping the vast, unbounded FP32 distribution of a trained network's weights and activations into this severely restricted discrete integer space is the fundamental challenge of quantization (GHOLAMI et al., 2021).

2.2.2 Range Mapping and Scaling

To transition from high-precision floating-point arithmetic to integer-only execution, a mathematical framework is required to map the unbounded values of FP32 parameters and activations into the discrete, bounded grid of an integer format (such as the $[-128, 127]$ range of signed INT8). The most prevalent and hardware-friendly method for achieving this is the uniform affine quantization scheme (JACOB et al., 2017).

Affine quantization employs a linear mapping function to bridge the real floating-point domain and the quantized integer domain. Let r represent a real-valued FP32 number (e.g., a weight or an activation output) and q represent its corresponding N-bit integer quantized value. The mapping is defined by the following fundamental equation:

$$r = S(q - Z)$$

In this formulation, the quantization parameters S (Scale) and Z (Zero-point) are defined as follows:

- **Scale factor (S):** A strictly positive FP32 scalar that defines the step size between consecutive discrete values in the quantized grid. It is determined by the ratio of the dynamic range of the real values to the representable range of the target integer format:
$$S = \frac{r_{max} - r_{min}}{q_{max} - q_{min}}$$
where $[r_{min}, r_{max}]$ defines the clipping range of the floating-point tensor, and $[q_{min}, q_{max}]$ is the representable range of the target integer type (e.g., $q_{min} = -128$ and $q_{max} = 127$ for signed 8-bit integers).
- **Zero-point (Z):** An integer value of the same type as q that acts as a biasing term. Its primary function is to ensure that the real value $r = 0$ is mapped exactly to an integer value in the quantized space without any rounding error. This is mathematically expressed as:
$$Z = \text{round}\left(q_{min} - \frac{r_{min}}{S}\right)$$

The requirement that real zero maps exactly to the integer zero-point (Z) is critical in deep learning (GHOLAMI et al., 2021). Operations such as zero-padding in convolutional layers or the masking of negative values in ReLU activation functions rely on true zero. If zero were subjected to rounding errors during quantization, these foundational op-

erations would introduce systemic bias, accumulating significant errors across the depth of the network (JACOB et al., 2017).

While activations typically utilize this full affine (asymmetric) mapping because their distributions are often skewed (e.g., ReLU outputs range from $[0, \infty)$), network weights are often quantized using a simplified *symmetric* scheme. In symmetric quantization, the clipping range is forced to be centered around zero ($r_{max} = -r_{min}$), which effectively forces the zero-point Z to be exactly 0. This simplifies the core mapping equation to $r = S \cdot q$, drastically reducing the computational overhead of the integer matrix multiplications during inference at the cost of a slightly less efficient utilization of the integer bit-space.

To retrieve the integer representation q from a real value r during the initial quantization phase, the mapping equation is inverted, and a rounding operation is applied to snap the continuous value to the nearest discrete integer grid point:

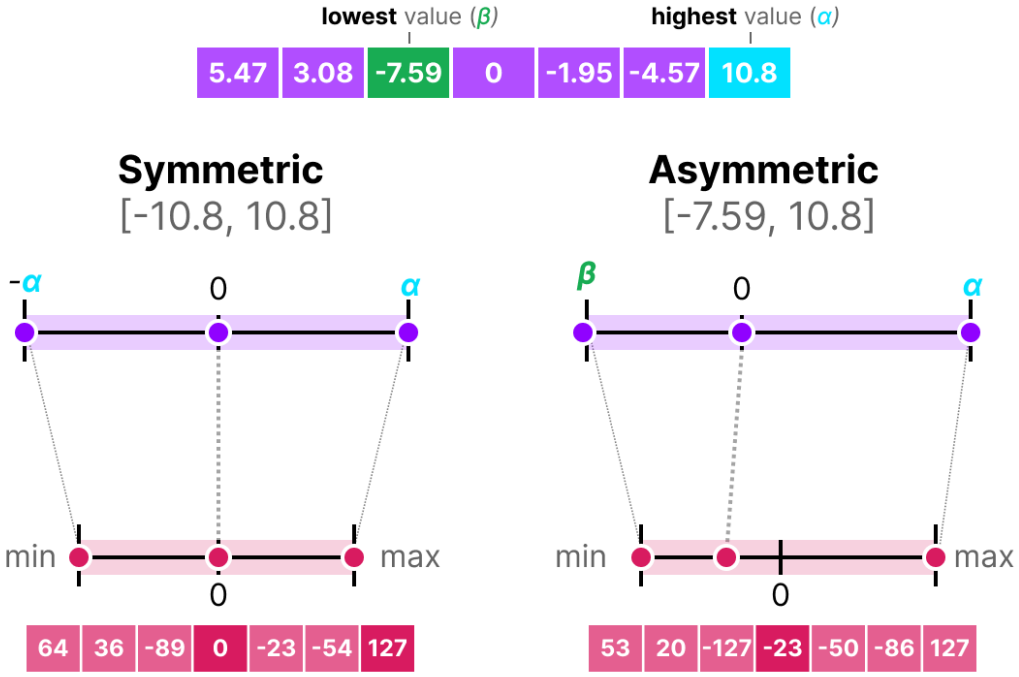
$$q = \text{clip}\left(\text{round}\left(\frac{r}{S} + Z\right), q_{min}, q_{max}\right)$$

Here, the clip function enforces the boundaries of the target integer format, ensuring that values falling outside the dynamic range $[r_{min}, r_{max}]$ are saturated at q_{min} or q_{max}.

2.2.3 Symmetric vs. Asymmetric Quantization

The affine mapping equation establishes the mathematical foundation for quantization, but the exact determination of the clipping range $[r_{min}, r_{max}]$ dictates whether the resulting scheme is symmetric or asymmetric. This choice, illustrated in Figure 2.9, represents a fundamental trade-off between representational resolution and computational efficiency during hardware inference.

Figure 2.9 – Visual comparison of Symmetric and Asymmetric Quantization mapping.



Source: Adapted from (GHOLAMI et al., 2021).

In asymmetric quantization (often referred to simply as affine quantization), the true minimum and maximum values of the floating-point tensor are mapped directly to the minimum and maximum boundaries of the integer grid (e.g., -128 and 127 for signed INT8). Consequently, the zero-point (Z) is dynamically calculated and is typically non-zero. This approach is highly efficient in terms of bit utilization because the entire integer range is stretched to perfectly fit the data distribution. It is particularly crucial for quantizing intermediate network activations, which are often heavily skewed or strictly positive (such as the outputs of a ReLU function, which span from $[0, \infty)$).

Conversely, symmetric quantization forces the clipping range to be perfectly mirrored around the origin. The range is defined by finding the absolute maximum value in the tensor, such that $r_{max} = \max(|r_{min}|, |r_{max}|)$ and $r_{min} = -r_{max}$. Because the floating-point range is symmetrically centered around zero, the integer zero-point (Z) strictly maps to 0.

While symmetric quantization may leave a portion of the integer grid unutilized if the original data distribution is heavily skewed (as seen in the right side of Figure 2.9), it provides a massive computational advantage. By forcing $Z = 0$, the core quantization mapping simplifies to $r = S \cdot q$. During inference, the matrix multiplications governing convolutional and fully connected layers rely heavily on dot products. Eliminating the zero-point from the weights entirely removes the need to compute complex cross-terms

during these integer dot products, drastically reducing the number of operations required by the Arithmetic Logic Unit (ALU).

Because network weights—unlike activations—tend to naturally cluster around zero, modern quantization frameworks almost universally apply symmetric quantization to the weights to maximize hardware acceleration, while reserving asymmetric quantization for the activations to preserve informational integrity (WU et al., 2020).

2.2.4 Post-Training Quantization (PTQ)

As illustrated on the right side of Figure 2.10, Post-Training Quantization (PTQ) is a technique applied only after a neural network has been fully trained to convergence using standard 32-bit floating-point (FP32) arithmetic. The core premise of PTQ is to statically convert the high-precision weights of the trained model into a lower-precision format, such as INT8, without requiring any further backpropagation or retraining.

To quantize the intermediate activations, PTQ requires a calibration phase. A small, unlabelled subset of the training data (typically 100 to 500 samples) is passed through the FP32 network. During this forward pass, the framework observes and records the dynamic ranges (r_{min} and r_{max}) of the activations at each layer. Once these boundaries are established, the scale (S) and zero-point (Z) mapping parameters are calculated and permanently frozen.

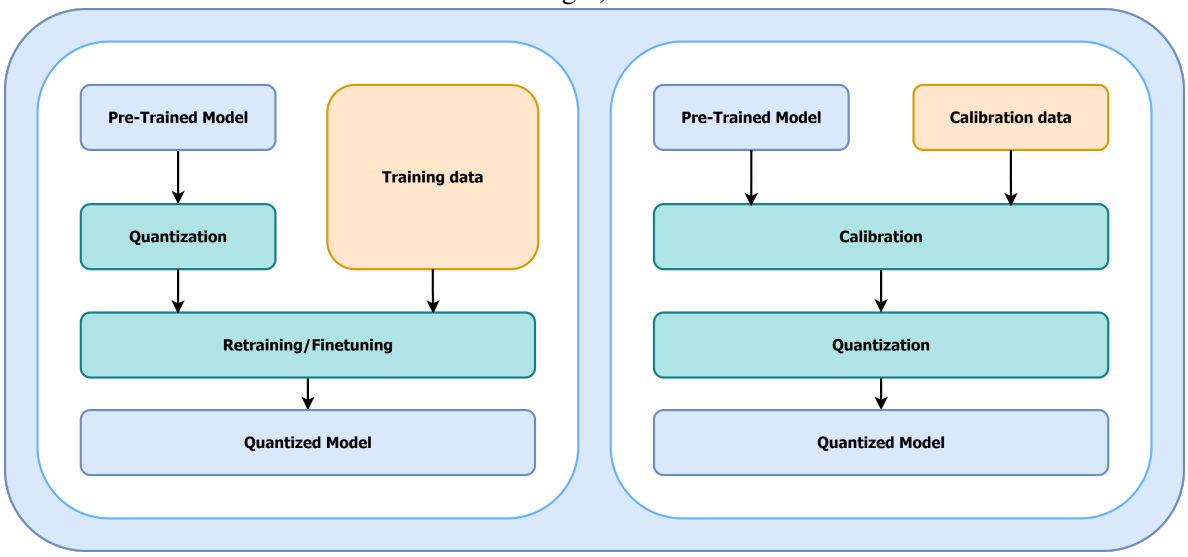
The primary advantage of PTQ is its simplicity and speed. It is a plug-and-play solution that can compress a model and prepare it for edge deployment in seconds, requiring minimal computational resources and no access to the full original training dataset (KRISHNAMOORTHI, 2018). However, this convenience comes at a cost. Because the model was optimized in a purely FP32 space, it is completely unaware of the clipping and rounding noise that will be introduced by the integer grid. For highly parameterized models like ResNet, this noise is often absorbed without issue. However, for compact, efficient architectures like MobileNet, the sudden introduction of this quantization noise post-training can cause a severe and irreversible degradation in predictive accuracy.

2.2.5 Quantization-Aware Training (QAT)

To mitigate the accuracy loss inherent in PTQ, Quantization-Aware Training (QAT) integrates the quantization process directly into the model's training loop. As shown on the left side of Figure 2.10, QAT is typically applied as a fine-tuning phase. The model is first initialized with pre-trained FP32 weights, but during the subsequent training epochs, the forward pass simulates the exact numerical constraints of the target hardware.

This simulation is achieved through "fake quantization" nodes inserted into the computational graph (JACOB et al., 2017). During the forward propagation, the FP32 weights and activations are mapped to discrete 8-bit values and immediately dequantized back to FP32. While the math is technically performed in floating-point, the values are strictly truncated to match the discrete steps of the INT8 grid. This injects the exact rounding and clipping noise the model will experience during physical inference, forcing the optimization algorithm to adapt the weights to become inherently robust to this specific noise.

Figure 2.10 – Quantization-Aware Training (QAT, Left) versus Post-Training Quantization (PTQ, Right)



Source: The author.

2.2.5.1 Backpropagation and the Straight-Through Estimator (STE)

While QAT effectively models inference noise during the forward pass, it introduces a critical mathematical failure during the backward pass. The simulated quantization process relies on a step function (specifically, the round$(\cdot)$ operation). In calculus, the

derivative of a step function is zero almost everywhere, and undefined at the exact transition boundaries. If standard backpropagation were applied to this graph, the gradients flowing backward from the loss function would be multiplied by zero the moment they encountered a quantization node. The gradient vector would vanish entirely, preventing the network from updating its parameters.

To circumvent this mathematical roadblock, QAT relies on the Straight-Through Estimator (STE) (BENGIO; LÉONARD; COURVILLE, 2013). The STE is a heuristic approximation that fundamentally alters how gradients flow through non-differentiable operations. During the backward pass, the STE simply bypasses the derivative of the rounding operation, approximating its local gradient as an identity function (1). Mathematically, if the forward pass function is $q = \text{round}(x)$, the STE defines the backward gradient as:

$$\frac{\partial q}{\partial x} \approx 1$$

By treating the rounding step as transparent during backpropagation, the gradients calculated from the "fake quantized" loss can successfully flow backward to update the original, high-precision parameters. The network maintains a set of high-precision "shadow weights" in FP32 throughout the QAT process. These shadow weights are necessary because the tiny gradient updates calculated by the optimizer (often on the scale of 10^{-4}) would be completely lost if they were applied directly to a rigid integer grid. Over multiple epochs, these FP32 shadow weights accumulate the micro-adjustments guided by the STE, eventually converging to a state where their strictly quantized INT8 equivalents yield optimal task accuracy. Once QAT is complete, the shadow weights are permanently quantized to INT8, and the FP32 graph is discarded for deployment.

2.2.6 Quantization in Practice: Frameworks and Toolkits

The theoretical advancements in neural network quantization would have had limited impact without their practical implementation and integration into accessible software tools. The widespread adoption of techniques like the integer-only inference scheme proposed by Jacob et al. (JACOB et al., 2017) was catalyzed by their incorporation into major deep learning frameworks. These toolkits have effectively democratized quantization, transforming it from a specialized research pursuit into a standard and critical step

in the development pipeline for deploying models on edge devices. They provide the essential infrastructure for converting, validating, and optimizing models, though their approaches and feature sets can vary significantly.

At the forefront of this movement is Google's TensorFlow ecosystem (ABADI ASHISH AGARWAL, 2017). The quantization framework developed by Jacob et al. was intrinsically linked to the development of TensorFlow Lite (GOOGLE, 2017), the official inference engine for on-device deployment. TensorFlow Lite provides a comprehensive workflow, including a "Converter" tool that ingests a standard TensorFlow model and applies quantization. It supports both post-training quantization (PTQ) for rapid deployment and the more robust Quantization-Aware Training (QAT), which directly implements the simulated quantization principles during the training phase. The true power of this ecosystem lies in its highly optimized runtime, which leverages specialized kernel libraries like gemmlowp (JACOB P. WARDEN, 2018) and its successor, Ruy, to execute the quantized operations with maximum efficiency on ARM CPUs. This tight integration of a theoretical framework, a software toolkit, and optimized hardware kernels has made TensorFlow Lite a benchmark for production-grade on-device AI.

In parallel, PyTorch, the other dominant deep learning framework, has also developed a mature and versatile quantization toolkit. It offers a similar dichotomy of workflows, providing modules for dynamic and static Post-Training Quantization as well as a robust API for Quantization-Aware Training. In a manner analogous to TensorFlow's approach, PyTorch's QAT inserts FakeQuantize modules into the model graph to emulate the data-type conversions and rounding errors that will occur during integer-only inference. This allows the model's weights to be fine-tuned in response to these simulated effects. To ensure high-performance execution, PyTorch leverages different optimized backends depending on the target hardware, such as QNNPACK for ARM and Fbgemm for x86 architectures. While the core principles are convergent with TensorFlow's, the distinct APIs and backend implementations represent a parallel evolutionary path toward the common goal of efficient integer-only inference.

Beyond the general-purpose deep learning frameworks, more specialized, hardware-aware toolkits have emerged to address the finer-grained challenges of quantization. A prominent example is the AI Model Efficiency Toolkit (AIMET) developed by Qualcomm. AIMET is designed to be framework-agnostic, interfacing with both TensorFlow and PyTorch models, and provides a suite of advanced quantization techniques that often surpass the baseline capabilities of the parent frameworks. For instance, AIMET features

sophisticated post-training methods like Cross-Layer Equalization (CLE), which rescales weight tensors across consecutive layers to make the model more resilient to quantization errors, and AdaRound, a technique that learns the optimal rounding strategy for each weight tensor rather than using the standard "round-to-nearest" approach. These methods can significantly improve the accuracy of quantized models without requiring a full retraining cycle. The existence of such tools highlights a crucial aspect of modern quantization: achieving optimal performance often requires a deeper, more hardware-centric approach than what is offered by default, further motivating the need for systematic evaluation across diverse hardware targets as undertaken in this thesis.

2.3 Related Work

The deployment of deep neural networks on constrained edge devices has driven significant research into model compression and hardware-aware quantization. The historical starting point for this paradigm is Deep Compression by Han *et al.* (HAN; MAO; DALLY, 2016b), which successfully reduced model size using weight pruning and quantization, though primarily evaluating theoretical energy reductions rather than native hardware execution.

To achieve physical latency acceleration, subsequent literature shifted toward integer-arithmetic-only inference. A seminal contribution to this domain is the work by Jacob *et al.* (JACOB et al., 2017), which co-designed a Quantization-Aware Training (QAT) scheme to map weights and activations to 8-bit integers (INT8), explicitly benchmarking the resulting latency improvements on ARM mobile CPUs.

Building upon the necessity of hardware-in-the-loop evaluation, Wang *et al.* (WANG et al., 2019) introduced HAQ, an automated framework that directly measures physical latency and energy consumption across different edge platforms to find optimal mixed-precision configurations. While HAQ focuses on Quantization-Aware Training, Cai *et al.* (CAI et al., 2020) demonstrated the viability of zero-shot Post-Training Quantization (PTQ), validating their mixed-precision framework by building a Pareto frontier of actual measured latency on a Raspberry Pi. Similarly, Wu *et al.* (WU et al., 2020) provided extensive empirical evaluations of integer quantization principles specifically tailored for deep learning inference on dedicated edge accelerators. More recently, Yao *et al.* (YAO et al., 2021) proposed HAWQ-v3, pushing the boundaries of integer-only execution by rigorously profiling the physical latency tradeoffs of 4-bit and 8-bit quantization on both

standard CPUs and desktop GPUs without relying on simulated metrics.

Table 2.4 provides a comparative overview of these benchmark studies in relation to the present work.

Table 2.4 – Overview of related works focusing on hardware-aware quantization. The symbol (-) indicates the strategy is not explored, (V) indicates it is explicitly explored, and (VV) indicates a comprehensive analysis or primary focus.

Work	*Target Platform*	*Quantization*			*Evaluation Metrics*
		PTQ	*QAT*	*Data Type*	
(MICIKEVICIUS et al., 2018)	GPU	-	VV	FP16 / FP32	Accuracy, Training Speedup
(HAN; MAO; DALLY, 2016b)	CPU / GPU	-	V	INT4 / INT8	Size, Theoretical Energy
(JACOB et al., 2017)	Mobile ARM CPU	-	VV	INT8	Accuracy, Latency
(CAI et al., 2020)	Mobile HW	VV	-	INT4 / INT8	Accuracy, Latency
(WU et al., 2020)	Edge Accelerators	VV	-	INT8	Accuracy, Latency
This work	Mobile HW and Desktop GPU	V	V	INT8	Accuracy, Latency, Energy & Memory

Source: The author.

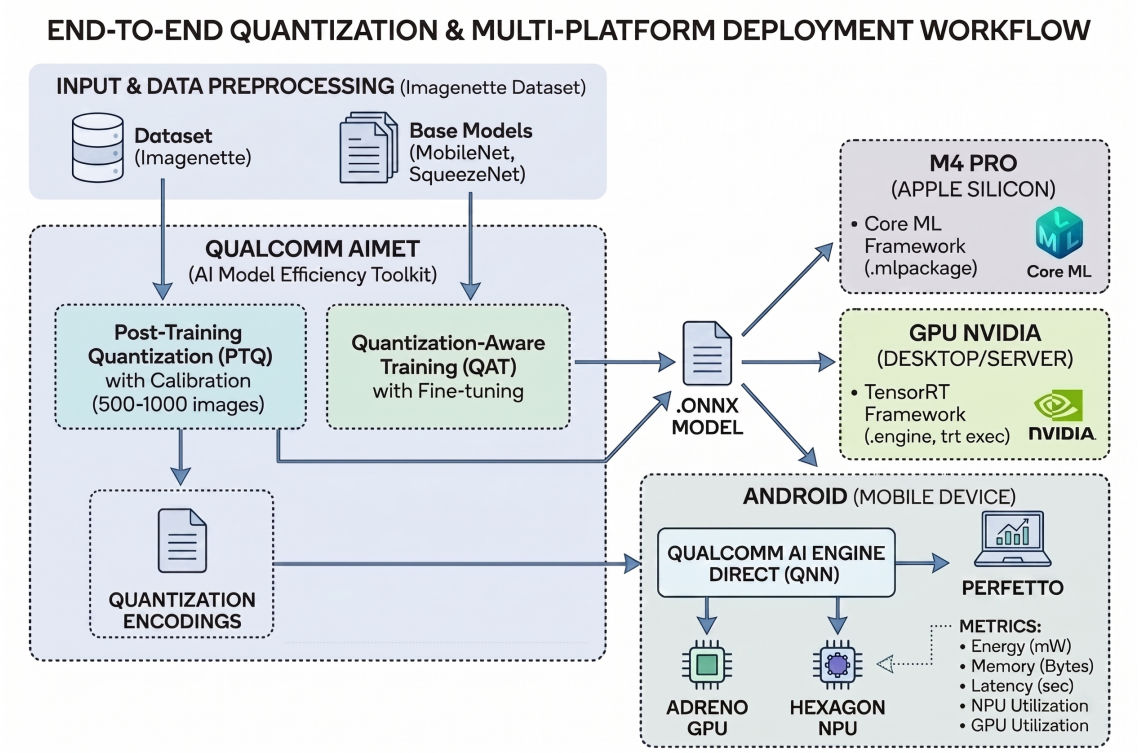
While the existing literature extensively covers specific quantization algorithms in isolation, evaluating them on a narrow subset of hardware, there remains a need for holistic, cross-platform profiling. As shown in Table 2.4, this work differentiates itself by deploying both PTQ and QAT methodologies across distinct hardware paradigms—specifically, Mobile hardware and Desktop GPUs. Furthermore, it provides a comprehensive evaluation of the practical tradeoffs by explicitly measuring accuracy alongside physical latency, energy consumption, and memory footprint, bridging the gap between theoretical compression and real-world deployment constraints.

3 PROPOSAL

This chapter details the methodology employed to evaluate the impact of quantization on Deep Neural Networks (DNNs) across highly heterogeneous hardware platforms. The evaluation pipeline spans from the initial model quantization process to the deployment and measurement on specific hardware architectures, including the Apple M4 Pro (macOS), the Qualcomm Snapdragon 8 Elite (Android), and the NVIDIA RTX 4070 (Desktop GPU).

Figure 3.1 provides a high-level visual summary of this end-to-end quantization and multi-platform deployment workflow, illustrating the data preprocessing, the central AIMET quantization pipeline, and the subsequent branching into the three distinct hardware evaluation environments.

Figure 3.1 – End-to-end quantization and multi-platform deployment workflow.



Source: The author.

3.1 Model Characterization and Workload

To comprehensively evaluate the impact of integer quantization across different hardware execution environments, it is imperative to establish a standardized set of neural

network architectures and a consistent inference workload. This section details the selected models, chosen to represent different architectural paradigms, and the data pipeline utilized to feed these models during the evaluation phase.

3.1.1 Selected Neural Network Architectures

The evaluation methodology employs four distinct Convolutional Neural Network (CNN) architectures. These models were strategically selected to provide an architectural counterpoint between designs heavily optimized for resource-constrained edge devices and general-purpose architectures typically deployed in less restrictive computational environments. Specifically, the ResNet models (ResNet18 and ResNet50) were selected to serve as dense, heavy-weight computational baselines, providing a necessary performance reference to contrast against the efficiency of the edge-optimized networks. Table 3.1 summarizes the fundamental characteristics of the chosen networks.

Table 3.1 – Overview of the Selected Neural Network Architectures

Architecture	Category	Key Design Feature
MobileNetV1 (HOWARD et al., 2017)	Edge-Optimized	Replaces standard convolutions with depthwise separable convolutions.
SqueezeNet (IANDOLA et al., 2016)	Edge-Optimized	Employs "Fire modules" to compress input channels (1×1 filters) before expansion (1×1 and 3×3).
ResNet18 (HE et al., 2016)	General-Purpose (Baseline)	Utilizes residual blocks with skip connections to mitigate the vanishing gradient problem.
ResNet50 (HE et al., 2016)	General-Purpose (Baseline)	Utilizes deeper residual blocks with bottleneck architecture and skip connections.

3.1.2 Workload and Data Preprocessing Pipeline

To ensure a fair and consistent evaluation of throughput, latency, and energy consumption across the M4 Pro, Snapdragon 8 Elite, and RTX 4070 platforms, a uniform workload was established.

The inference workload is based on the **Imagenette** dataset (HOWARD, 2019),

a subset of ten easily classified classes from the larger ImageNet dataset. This subset provides a representative image classification task without the massive storage and data-loading overhead of the full ImageNet dataset, which is particularly beneficial for isolated benchmarking loops executing directly on edge devices.

The preprocessing pipeline for the input samples follows standard computer vision practices to ensure the input tensors match the expected distributions of the pre-trained weights. The pipeline consists of the following sequential steps:

1. **Resizing:** The short edge of the input image is resized to 256 pixels while maintaining the aspect ratio.
2. **Cropping:** A center crop of 224×224 pixels is extracted.
3. **Normalization:** The pixel values are converted to 32-bit floating-point (FP32) and normalized using the standard ImageNet statistics: a mean of $[0.485, 0.456, 0.406]$ and a standard deviation of $[0.229, 0.224, 0.225]$.
4. **Layout:** The tensor is structured in the NCHW (Batch, Channels, Height, Width) format.

All performance and energy benchmarks are executed using a batch size of 1 ($N = 1$). This constraint is deliberately chosen to reflect the latency-critical nature of edge inference, where sensory data or user queries typically arrive sequentially and must be processed immediately, in contrast to server-side, throughput-oriented batched inference.

3.2 Training and Quantization Pipeline (AIMET)

Following the selection of the neural network architectures, the subsequent phase in the methodology involves simulating and applying integer quantization. To ensure consistency across all targeted hardware platforms, a centralized quantization pipeline was established using the AI Model Efficiency Toolkit (AIMET) (Qualcomm Innovation Center, Inc., 2022), an open-source library developed by Qualcomm that provides state-of-the-art quantization and compression algorithms for trained neural network models.

To establish a highly controlled environment, both the Post-Training Quantization (PTQ) and Quantization-Aware Training (QAT) pipelines were built upon customized, pre-trained 32-bit floating-point (FP32) baseline models specifically fine-tuned for the 10-class Imagenette dataset.

3.2.1 Dataset Splitting and Data Augmentation

The quantization and fine-tuning procedures utilized the standard train/validation split provided by the Imagenette2 dataset. To ensure robust gradient updates during QAT, the training dataloader applied standard data augmentation techniques, specifically random resized cropping (to 224×224 pixels) and random horizontal flipping. The validation and calibration dataloaders strictly applied deterministic transformations: resizing the short edge to 256 pixels followed by a 224×224 center crop. All input tensors were normalized using standard ImageNet statistics (mean $[0.485, 0.456, 0.406]$ and standard deviation $[0.229, 0.224, 0.225]$). A uniform batch size of 32 was maintained across all phases.

3.2.2 Post-Training Quantization (PTQ) Configuration

The PTQ pipeline was designed to strictly map the network's parameters and activations to 8-bit integers (INT8). The procedure began with Batch Normalization (BN) Folding, an essential optimization that mathematically absorbs the scaling and shifting parameters of BN layers directly into the weights of the preceding convolutional layers, eliminating unnecessary runtime operations.

Following BN folding, AIMET's `QuantSimModel` was initialized using the `post_training_tf_enhanced` quantization scheme. To determine the optimal clipping ranges (r_{min} and r_{max}) for the activations, a calibration phase was executed by passing 20 batches (640 representative images) from the training set through the unquantized graph. Once the statistical ranges were captured, the explicit quantization encodings (scales and zero-points) were calculated and frozen.

3.2.3 Quantization-Aware Training (QAT) Hyperparameters

To recover the representational capacity inevitably lost during PTQ—especially in the compact MobileNetV1 and SqueezeNet topologies—a QAT fine-tuning phase was subsequently applied. The QAT pipeline initializes with the exact encodings computed during the PTQ calibration step and inserts simulated quantization and dequantization (Q/DQ) nodes into the computational graph to model INT8 rounding noise during the

forward pass.

A critical, hardware-aware engineering intervention was implemented during the QAT graph construction. For models utilizing depthwise separable convolutions (e.g., MobileNetV1), the simulated quantization nodes immediately following the depthwise layers (`conv_dw`) were explicitly disabled. This allows the activations to flow seamlessly into the subsequent pointwise layers without artificial truncation, directly enabling downstream compilers (such as NVIDIA TensorRT) to perform aggressive Kernel Fusion, maximizing hardware utilization.

The QAT fine-tuning was executed under a strict, hyperparameter configuration to gently guide the weights toward quantization robustness without destroying the converged FP32 features:

- **Duration:** 10 epochs.
- **Loss Function:** Standard Cross-Entropy Loss.
- **Optimizer:** SGD + momentum.
- **Learning Rate (LR):** Initialized at a highly conservative 1×10^{-4}.

Upon completion of the fine-tuning phase, the optimal model weights (selected via highest validation accuracy) were frozen. The quantization encodings were re-computed, and the final artifacts—a ONNX graph and the accompanying JSON encodings file—were exported for multi-platform deployment.

3.2.4 Quantization Strategies: PTQ and QAT

To comprehensively evaluate the trade-offs between deployment speed, accuracy degradation, and hardware efficiency, two distinct integer quantization paradigms are evaluated within the AIMET pipeline:

- **Post-Training Quantization (PTQ):** PTQ is applied as the baseline compression method. It involves calibrating the pre-trained FP32 models using a small, representative subset of the training data (drawn from the Imagenette dataset). This process calculates the optimal dynamic ranges (minimum and maximum values) for the weights and activations without requiring retraining. PTQ is highly efficient but can lead to noticeable accuracy drops in compact architectures like MobileNet and SqueezeNet due to their aggressive parameter reduction.

- **Quantization-Aware Training (QAT):** To recover accuracy lost during PTQ, QAT is employed. QAT inserts fake-quantization nodes (simulated quantization noise) into the computational graph during a fine-tuning phase. This allows the model's weights to adapt to the quantization errors introduced by the reduced precision. While computationally more expensive than PTQ, QAT typically yields higher inference accuracy, especially for low-bitwidth precision targets like 8-bit integers (INT8).

Both strategies are configured to target an 8-bit integer (INT8) representation. Specifically, the pipeline is constrained to produce symmetric, per-channel quantization for convolution weights, which is a widely supported specification across modern Neural Processing Units (NPUs).

3.2.5 Standardized Artifact Generation

Figure 3.2 illustrates the general quantization workflow employed in this study. The process begins with the ingestion of the pre-trained baseline model, which is subsequently wrapped into a `QuantizationSimModel` to simulate the effects of reduced precision arithmetic. A calibration callback is then executed using a representative data sample to compute the optimal quantization encodings. Finally, this workflow culminates in the export stage.

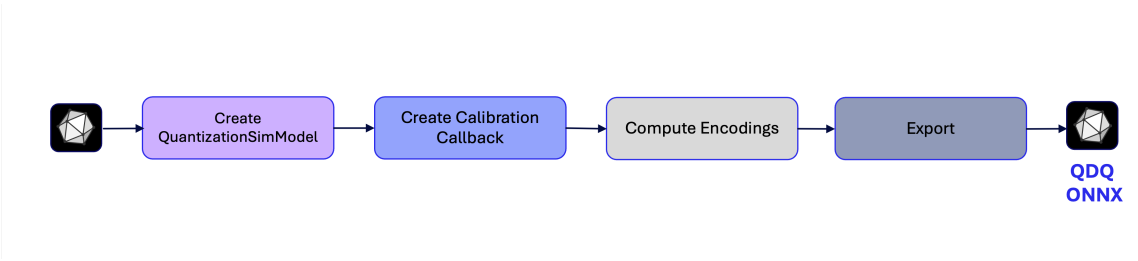
A critical design decision in this methodology is the decoupling of the quantized model's computational graph from its quantization metadata during this export phase. Instead of exporting a fully quantized, framework-specific model (such as a compiled TFLite or CoreML package directly from the training environment), the AIMET pipeline acts as an intermediate compiler that generates two distinct, universally parsable artifacts for each model:

1. **The Computational Graph (`.onnx`):** The network structure and the trained parameters are exported in the Open Neural Network Exchange (ONNX) format. Crucially, this ONNX file remains in 32-bit floating-point (FP32) precision, preserving the exact topology and node definitions of the network without prematurely fusing quantization operations.
2. **The Quantization Metadata (`.encodings`):** AIMET generates an accompanying JSON file containing the exhaustive quantization schema for every tensor in the

model. This includes the calculated *scale* factors, *zero-points*, bitwidths, and data types (e.g., INT8) for all weights and activations.

This decoupled approach ensures absolute consistency in the quantization scheme applied across the highly heterogeneous testing environments. By treating the `.encodings` file as a universal contract, the downstream platform-specific pipelines (Apple CoreML, Qualcomm QNN, and NVIDIA TensorRT) can parse the exact same scales and zero-points, mapping them to their native quantization APIs. This eliminates discrepancies that would otherwise arise from using different calibration algorithms native to each vendor's software stack.

Figure 3.2 – General AIMET quantization Workflow



Source: AIMET Workflow

3.3 Apple Silicon Evaluation Pipeline (M4 Pro)

The Apple Silicon platform, specifically the M4 Pro chip running macOS 15, provides a highly integrated System-on-Chip (SoC) architecture featuring a unified memory model and a dedicated hardware accelerator for machine learning workloads, the Apple Neural Engine (ANE). To systematically evaluate the performance and energy efficiency of the quantized models on this architecture, a robust, automated pipeline was developed. This pipeline is composed of four sequential execution stages, concluding with a stringent hardware residency verification step.

3.3.1 Stage 1: CoreML Compilation

Because modern versions of Apple's `coremltools` (version 5.0 and later) deprecated direct ONNX conversion, a multi-step translation bridge was implemented to convert the baseline FP32 `.onnx` models (generated in Section 3.2) into Apple's native format.

First, the ONNX graph is parsed and converted into a PyTorch `nn.Module` utilizing the `onnx2torch` (ENOT-AUTODL, 2024) library, strictly preserving the node-to-node topological structure. This module is subsequently traced using `torch.jit.trace` with a dummy input tensor of shape [1, 3, 224, 224]. The resulting TorchScript object is then passed to the conversion API `coremltools.convert()`.

Crucially, the conversion targets the modern `mlprogram` format, which is based on the Model Intermediate Language (MIL). The pipeline is configured with a minimum deployment target of macOS 14 to unlock advanced profiling APIs, and the native compute precision is set to FP16. This stage outputs an unquantized `.mlpackage` directory, acting as the FP16 baseline for the ANE.

3.3.2 Stage 2: Quantization Application (INT8 Strategy)

For the INT8 variants (PTQ and QAT), the baseline `.mlpackage` is modified to incorporate the AIMET quantization schemas. A custom parser was engineered to read the `.encodings` JSON artifact and validate its compliance with the expected specification (8-bit width, integer data type, symmetric, and per-channel granularity for convolution weights).

The quantization is applied using the optimization APIs from `coremltools`:

```
coremltools.optimize.coreml.linear_quantize_weights()
coremltools.optimize.coreml.linear_quantize_activations()
```

By successfully mapping both the weight and activation scales generated by AIMET, the pipeline generates a true **INT8 (W8A8)** model. Overcoming native framework defaults, this explicit control forces both the parameters and the activation tensors into 8-bit precision. This schema allows the hardware to natively execute full INT8 Multiply-Accumulate (MAC) operations on the ANE, maximizing memory bandwidth savings, footprint reduction, and native computational efficiency on the silicon without relying on implicit FP16 dequantization fallback mechanisms.

3.3.3 Stage 3: Latency and Throughput Profiling

Inference measurements are conducted by loading the model with the `CPU_AND_NE` compute unit configuration. The evaluation loop is fed with sequentially rotated prepro-

cessed samples from the Imagenette dataset (as detailed in Section 3.1) to prevent output caching anomalies.

To ensure statistical reliability, the profiling sequence is divided into two phases: an initial 20-iteration *warmup* phase, which is discarded to account for Just-In-Time (JIT) compilation, followed by 300 timed iterations. Each `predict()` invocation is wrapped in a high-resolution timing function `time.perf_counter()`.

From this data, comprehensive statistical metrics are extracted, including the mean, standard deviation, min/max values, and mean latency. Throughput is subsequently calculated in Frames Per Second (FPS) based on the mean latency with a batch size of 1.

3.3.4 Stage 4: Energy Measurement via Powermetrics

Energy consumption is captured using Apple's native `powermetrics` utility, running as a parallel subprocess during a sustained inference loop. To guarantee accurate sampling, the tool is configured to capture power counters at 100ms intervals over a 30-second window:

```
sudo powermetrics --samplers ane_power,cpu_power,gpu_power \
                  -i 100 -n N --format plist
```

Simultaneously, a Python process executes continuous model inferences, recording the total number of iterations completed within the timeframe. Upon completion, the `powermetrics` process is gracefully terminated, and its concatenated property list (`.plist`) output is manually parsed. By isolating the `ane_power`, `cpu_power`, and `gpu_power` readings (in milliwatts), the methodology allows for the precise calculation of the average power draw per IP block, the total energy consumed (in millijoules), and ultimately, the specific energy cost per single inference (mJ/inf).

3.3.5 Hardware Residency Verification and Testing

To prevent silent fallbacks to the CPU—which would invalidate the NPU benchmarking—a rigorous residency verification step is enforced using the `MLComputePlan` API. The `.mlpackage` is compiled and its static schedule is reconstructed. By iterating through the MIL operations and querying the following plan method:

`get_compute_device_usage_for_mlprogram_operation()`

the pipeline confirms the `preferred_compute_device`. For the evaluated models, this verification step successfully proved 100% operation residency on the `MLNeuralEngineComputeDevice`. To prevent uncatchable C++ exceptions (libc++ aborts) occasionally thrown by the MLComputePlan from crashing the benchmark suite, this verification is isolated within a separate subprocess.

Finally, an internal A/B test is conducted by forcing the execution on `CPU_ONLY` versus `CPU_AND_NE`. This isolation provides a concrete baseline, allowing the methodology to accurately quantify the direct speedup and the total energy savings inherently provided by the Apple Neural Engine.

3.4 Mobile Edge Evaluation Pipeline (Snapdragon 8 Elite)

The mobile edge environment represents the most power-constrained and thermally limited deployment scenario in this methodology. To evaluate the efficacy of quantization on native mobile hardware, the pipeline targets the Qualcomm Snapdragon 8 Elite System-on-Chip (SoC), hosted within a Samsung Galaxy S25 device. This pipeline leverages the Qualcomm AI Engine Direct (QNN) SDK to execute inferences and relies on advanced system tracing for power measurements.

3.4.1 Stage 1: Model Conversion and QNN Integration

The transition from the centralized AIMET pipeline (detailed in Section 3.2) to the Snapdragon hardware requires a vendor-specific compilation step. This is handled by the `qnn-onnx-converter` utility provided by the QNN SDK.

For the baseline floating-point models, the FP32 `.onnx` graph is converted directly into a native QNN representation. For the quantized variants (PTQ and QAT), the converter is fed both the FP32 `.onnx` file and the AIMET `.encodings` JSON. The QNN compiler parses these explicit scales and zero-points to map the intermediate representation into a highly optimized, hardware-specific execution graph. This compilation process yields the serialized context binaries that will be pushed to the mobile device.

3.4.2 Stage 2: Target Allocation (NPU vs. GPU)

A critical architectural constraint of this specific pipeline is the strict isolation of execution targets to evaluate domain-specific accelerators. Unlike traditional mobile benchmarks that often fall back to CPU execution, this methodology strictly compares the SoC's Graphics Processing Unit (GPU) against its Neural Processing Unit (NPU, specifically the Hexagon Tensor Processor or HTP). The central processing unit (CPU) is deliberately excluded from the inference workload.

- **GPU Baseline (FP32):** The GPU serves as the execution target for the baseline floating-point models. It provides a highly parallelized architecture capable of handling the computational density of unquantized networks, establishing the upper bound for power consumption and baseline latency on the mobile SoC.
- **NPU / Hexagon HTP (INT8):** The Hexagon NPU is designated as the exclusive execution target for all INT8 quantized models. Because the HTP is a specialized matrix-math accelerator explicitly designed for low-precision integer arithmetic, mapping the INT8 workloads strictly to this block allows the methodology to isolate and measure the absolute maximum hardware efficiency gained through quantization.

3.4.3 Stage 3: Remote Execution and Throughput Profiling

Because the compilation occurs on a host machine while execution happens on the edge device, the Android Debug Bridge (ADB) is utilized as the primary command-and-control interface. The compiled QNN binaries, alongside the preprocessed Imagenette2 input samples, are pushed to the device storage via ADB.

Inference loops are instantiated using the `qnn-net-run` executable directly in the device's shell. Consistent with the Apple Silicon pipeline, the execution enforces a batch size of 1 to simulate real-time sensor processing. The process includes an identical warmup phase followed by a timed execution loop of 300 iterations, logging the granular timestamps required to extract the mean latency, and overall throughput (FPS).

3.4.4 Stage 4: Energy Profiling via Perfetto

Measuring power consumption on a mobile SoC requires system-level telemetry to capture the exact power rails of the isolated IP blocks without introducing significant observer overhead. To achieve this, the pipeline employs **Perfetto**, the official system-wide profiling tool for Android.

During the sustained inference loop initiated via `qnn-net-run`, a parallel Perfetto tracing session is triggered via ADB. The tracer is configured to poll the specific power supply sensors (hardware power rails) corresponding to the GPU domain and the NPU/Hexagon domain.

Once the inference window concludes, the Perfetto trace is pulled to the host machine. By synchronizing the timestamps of the inference loop with the trace data, the methodology allows for the precise extraction of the average power draw (in milliwatts) exclusively for the active accelerator (GPU or NPU). This average power is then multiplied by the execution duration to calculate the total energy consumed, yielding the energy-per-inference metric (mJ/inf) for the mobile edge platform.

3.5 Workstation/GPU Evaluation Pipeline (RTX 4070)

The workstation environment represents the high-performance, thermally unconstrained tier in this methodology. Equipped with an NVIDIA GeForce RTX 4070 Graphics Processing Unit (GPU), this platform provides the upper bound for computational throughput and the baseline for unconstrained power consumption. It serves as the primary high-performance counterpoint against which the efficiency of the mobile and Apple Silicon edge platforms is compared.

3.5.1 Stage 1: Model Optimization via TensorRT

To maximize the utilization of the GPU's parallel architecture and specialized Tensor Cores, the evaluation pipeline employs NVIDIA TensorRT, a high-performance deep learning inference SDK.

For the floating-point baseline models, the native FP32 `.onnx` graphs (generated in Section 3.2) are directly parsed and compiled into highly optimized TensorRT engines.

During this phase, TensorRT performs layer fusion and kernel auto-tuning specifically for the Ada Lovelace architecture.

For the INT8 variants (PTQ and QAT), it is critical that the GPU evaluates the exact same quantization schema applied to the edge devices. Therefore, the pipeline is configured to ingest the ONNX graph alongside the explicit quantization scales and zero-points provided by the AIMET `.encodings` JSON. By explicitly supplying these parameters via the TensorRT Python API, the pipeline overrides TensorRT's default internal calibration algorithms. This strict control mechanism guarantees that the quantization arithmetic remains universally consistent across all heterogeneous platforms evaluated in this study.

3.5.2 Stage 2: Latency and Throughput Profiling

Consistent with the edge pipelines, the inference benchmarking on the RTX 4070 is executed using a strict batch size of 1 ($N = 1$). While workstation GPUs are architecturally optimized to maximize throughput via large-batch processing, enforcing single-batch execution is a mandatory constraint for this methodology. It ensures an equitable, apples-to-apples comparison of sequential processing latency—the critical metric for real-time edge applications where inputs arrive and must be processed individually.

The execution loop follows the established protocol: the preprocessed Imagenette inputs are transferred to the GPU's VRAM, followed by 20 warmup iterations to initialize the CUDA context and populate the instruction cache. Subsequently, 300 timed iterations are executed. High-resolution timestamps (`time.perf_counter()`) capture the processing time of each forward pass, allowing for the extraction of mean latency, and the resulting throughput in Frames Per Second (FPS).

3.5.3 Stage 3: Energy Measurement via `nvidia-smi`

Power profiling on the desktop GPU is conducted using the NVIDIA System Management Interface (`nvidia-smi`). Similar to the `powermetrics` and Perfetto implementations on the edge platforms, `nvidia-smi` provides direct access to the hardware's power sensors.

During a 30-second inference window, a parallel daemon process is spawned to

poll the GPU's power draw. The tool is invoked with the `-query-gpu=power.draw` flag, sampling at a frequency of 100ms. Because the workstation environment is tightly controlled and the GPU is entirely isolated for this specific workload, the polled wattage accurately reflects the power required to sustain the model's execution, including memory bandwidth overhead.

Upon completion of the inference loop, the discrete power samples are averaged to compute the mean power draw (in milliwatts). This average is then multiplied by the exact execution duration to calculate the total energy consumed. Finally, this total is divided by the number of completed iterations to yield the precise energy-per-inference metric (mJ/inf), formatted identically to the edge platform outputs for seamless cross-platform comparison.

3.6 Instrumentation and Evaluation Metrics

Given the extreme architectural heterogeneity of the targeted hardware platforms (Apple Silicon, Snapdragon Edge, and a discrete NVIDIA GPU), establishing a mathematically rigorous and standardized set of evaluation metrics is imperative. While the underlying telemetry tools differ per platform—`powermetrics` for macOS, `Perfetto` for Android, and `nvidia-smi` for the desktop environment—the data extraction protocols and the mathematical formulations used to derive the final metrics remain strictly identical across all test beds.

This section defines the standardized metrics utilized to quantify the trade-offs of integer quantization, categorized into performance (latency/throughput) and energy efficiency.

3.6.1 Latency and Throughput Metrics

Latency, in the context of edge inference, represents the time required to complete a single forward pass of the neural network. To eliminate the variance introduced by initial hardware power-state transitions, cache warming, and Just-In-Time (JIT) compilation, the first 20 inferences of every execution loop are classified as *warmup* iterations and are strictly excluded from the statistical analysis.

Following the warmup phase, a timing window captures $N = 300$ consecutive

inferences. Let t_i denote the exact latency (in milliseconds) of the i-th inference. The primary metric extracted to characterize the performance is the **Mean Latency (μ)**, calculated as the arithmetic average of all recorded latencies across the timed window. This metric serves as the indicator of general performance for the sequential workloads.

Throughput, defined as Frames Per Second (FPS), is directly derived from the mean latency. Because the methodology strictly enforces a batch size of 1 to simulate real-time sensor processing, the throughput is calculated as:

$$\text{FPS} = \frac{1000}{\mu}$$

3.6.2 Power and Energy Efficiency Metrics

Measuring raw power (Watts) provides an incomplete picture of architectural efficiency, as a faster processor may draw more power but finish the task significantly quicker, resulting in less total energy consumed. Therefore, the metric for efficiency in this study is the **Energy per Inference (E_{inf})**, measured in millijoules (mJ).

Across all platforms, system-level profiling tools are configured to sample the hardware power rails over a sustained execution window (typically 30 seconds). Due to the architectural differences and the specific telemetry interfaces provided by each operating system, the profiling tools and their respective sampling intervals vary per platform, as detailed in Table 3.2.

Table 3.2 – Power Profiling Tools and Sampling Intervals per Platform

Platform	**Target IP Block**	**Profiling Tool**	**Sampling Interval**
Apple M4 Pro	ANE / CPU	`powermetrics`	500 ms
Snapdragon 8 Elite	Hexagon NPU / Adreno GPU	`Perfetto`	1000 ms
NVIDIA RTX 4070	Desktop GPU	`nvidia-smi`	300 ms

The raw samples collected by these tools are isolated to the specific active IP blocks (e.g., the ANE domain on Apple, the HTP/Hexagon domain on Snapdragon, and the GPU domain on the RTX 4070).

Let P_{avg} be the arithmetic mean of the sampled power draw (in milliwatts, mW) over the sustained execution window T (in seconds). Let I_{total} be the total number of inferences completed during that specific window. The total energy consumed by the workload, E_{total}, and the normalized Energy per Inference, E_{inf}, are calculated as fol-

lows:

$$E_{total}(\text{mJ}) = P_{avg}(\text{mW}) \times T(\text{s})$$

$$E_{inf}(\text{mJ/inference}) = \frac{E_{total}}{I_{total}}$$

This unified formula guarantees that the energy savings reported in the subsequent results chapters are perfectly normalized against the respective latency speedups of each hardware architecture. Consequently, the study can accurately quantify the true battery-life impact of INT8 quantization on edge devices compared to unconstrained desktop baselines.

4 RESULTS AND DISCUSSION

4.1 Impact of Quantization on Model Accuracy

Before evaluating the hardware-specific performance and energy improvements yielded by quantization, it is essential to establish the predictive validity of the compressed models. Quantization inherently introduces numerical noise due to the reduction of precision from 32-bit floating-point (FP32) to 8-bit integers (INT8). This section analyzes the impact of Post-Training Quantization (PTQ) and Quantization-Aware Training (QAT) on the Top-1 classification accuracy, measured using the Imagenette validation dataset.

Table 4.1 summarizes the accuracy metrics for the evaluated architectures prior to hardware-specific deployment. It is important to note that the ResNet architectures (ResNet18 and ResNet50) were not subjected to the INT8 quantization pipeline. Instead, their original FP32 variants were retained strictly to serve as dense, heavy-weight computational baselines, providing a reference point for comparing the performance and efficiency of the compressed edge-optimized networks in subsequent sections.

Table 4.1 – Top-1 Accuracy on Imagenette2 across Quantization Strategies

Model	**FP32 (Baseline)**	**INT8 (PTQ)**	**INT8 (QAT)**
MobileNetV1	97.78%	96.84%	97.40%
SqueezeNet	95.00%	93.43%	94.00%
ResNet18	95.64%	-	-
ResNet50	96.33%	-	-

4.1.1 Degradation in Edge-Optimized Architectures

The results immediately highlight the vulnerability of highly compact, edge-optimized architectures to aggressive precision reduction. SqueezeNet experienced the most significant degradation under PTQ, dropping from a 95.00% baseline to 93.43% (a 1.57 percentage point absolute loss). Similarly, MobileNetV1 saw its accuracy reduced from 97.78% to 96.84% (a 0.94 percentage point loss).

This disproportionate impact is a direct consequence of their architectural design. Models like MobileNetV1 rely heavily on depthwise separable convolutions, while

SqueezeNet utilizes parameter-squeezing "Fire modules". Because these networks are already aggressively optimized to contain minimal redundancy in their weights, forcing those weights into a narrow 8-bit dynamic range via simple calibration (PTQ) discards critical representational capacity, directly translating to misclassifications.

In contrast, the general-purpose architectures (ResNet18 at 95.64% and ResNet50 at 96.33% in FP32) possess significantly more parameters and structural redundancy, typically allowing them to absorb quantization noise more gracefully. However, for the scope of evaluating strict edge deployment limitations, focusing on recovering the accuracy of the compact networks was prioritized.

4.1.2 The Effectiveness of Quantization-Aware Training

To mitigate the unacceptable accuracy loss observed with PTQ, Quantization-Aware Training (QAT) was applied. The results demonstrate that QAT successfully recovers a significant portion of the lost predictive power.

For MobileNetV1, QAT restored the accuracy to 97.40%, bridging the gap to within a negligible 0.38% of the original FP32 baseline. For SqueezeNet, QAT elevated the accuracy back to 94.00%, recovering over a third of the accuracy lost during PTQ. By simulating the INT8 quantization noise during the fine-tuning phase, the AIMET QAT pipeline allowed the surviving parameters to mathematically adapt to the restricted bit-width.

These findings validate the methodological decision to include QAT in the pipeline. While computationally more expensive during the preparation phase than PTQ, QAT is empirically justified—and often mandatory—when preparing extremely compact CNNs for INT8 deployment on edge NPUs.

4.2 Performance Analysis: Latency and Throughput

Following the accuracy validation, the primary objective of this evaluation shifts to computational performance. In edge computing scenarios, where sensor data or user queries arrive sequentially, raw sequential processing speed is the paramount metric. Therefore, this analysis focuses strictly on single-batch ($N = 1$) latency (measured in milliseconds) and its direct derivative, throughput (measured in Frames Per Second - FPS).

4.2.1 Mobile Edge Acceleration (Snapdragon 8 Elite)

The Snapdragon 8 Elite platform explicitly demonstrates the profound impact of offloading quantized workloads from general-purpose parallel compute (GPU) to domain-specific matrix accelerators (NPU). Table 4.2 and Figure 4.1 summarize the latency and throughput metrics extracted from the platform.

Table 4.2 – Performance comparison on Snapdragon 8 Elite (GPU vs. NPU)

Model	Hardware Target	Precision	Latency (ms)	Throughput (FPS)	Speedup
MobileNetV1	Adreno 830 GPU	FP32	4.190	239	1.000×
MobileNetV1	Adreno 830 GPU	INT8 (PTQ)	3.798	263	1.100×
MobileNetV1	Adreno 830 GPU	INT8 (QAT)	3.695	271	1.134×
MobileNetV1	Hexagon V79 NPU	INT8 (PTQ)	1.450	685	2.866×
SqueezeNet	Adreno 830 GPU	FP32	4.150	241	1.000×
SqueezeNet	Adreno 830 GPU	INT8 (PTQ)	2.667	375	1.556×
SqueezeNet	Adreno 830 GPU	INT8 (QAT)	2.323	430	1.784×
SqueezeNet	Hexagon V79 NPU	INT8 (PTQ)	0.450	2203	9.141×
ResNet50	Adreno 830 GPU	FP32	24.000	41	N/A

For the baseline FP32 execution, the Adreno 830 GPU achieved a latency of 4.190 ms (239 FPS) for MobileNetV1 and 4.150 ms (241 FPS) for SqueezeNet. While capable, executing floating-point math on the mobile GPU establishes a noticeable bottleneck for real-time applications.

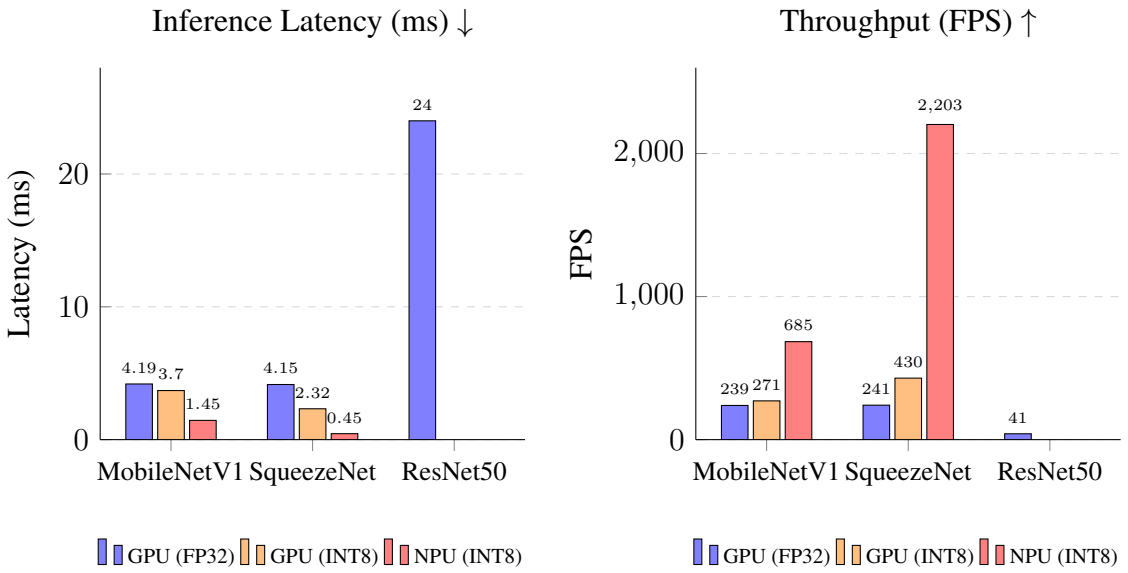
Crucially, attempting to execute INT8 quantized models on the GPU highlights a significant hardware-software mismatch. While the GPU yields marginal improvements—such as a 1.784× speedup for SqueezeNet QAT (2.323 ms / 430 FPS)—it is severely bottlenecked by the overhead of software dequantization. Because the GPU lacks optimal native datapaths for integer-only arithmetic, the framework is forced to simulate or cast these operations, neutralizing the theoretical benefits of quantization.

Deploying the INT8 (PTQ) quantized models strictly to the Hexagon V79 NPU resulted in dramatic performance leaps. MobileNetV1 latency dropped to 1.450 ms (685 FPS), yielding a **2.866× speedup** over the GPU FP32 baseline. SqueezeNet exhibited an even more astonishing architectural affinity for the NPU, with latency plummeting to 0.450 ms (2203 FPS), representing an extraordinary **9.141× speedup**. The NPU proves to be exceptionally efficient at handling the SqueezeNet topology natively in INT8.

To contextualize these edge-optimized models, the heavy-weight ResNet50 architecture was evaluated on the GPU in FP32. The GPU processed ResNet50 in 24.000

ms (41 FPS). This explicitly demonstrates the absolute computational ceiling of the mobile GPU on dense architectures and reinforces the necessity of coupling edge-optimized topologies with NPU acceleration to achieve high-framerate real-time inference on thermally constrained devices.

Figure 4.1 – Latency and Throughput comparison on the Snapdragon 8 Elite. The GPU (INT8 QAT) metrics illustrate the fallback bottleneck, while ResNet50 (FP32 only) serves as a heavy-weight baseline.



Source: The author

4.2.2 Apple Silicon Efficiency (M4 Pro)

The execution profile on the Apple M4 Pro architecture demonstrates the efficacy of full integer execution. As established in the methodology, the pipeline successfully mapped the AIMET artifacts to true INT8 execution on the Apple Neural Engine (ANE) for the compact models, executing both weights and activations natively in 8-bit. Table 4.3 and Figure 4.2 detail the performance metrics for this architecture.

For the baseline FP32 execution, the M4 Pro CPU achieved a latency of 2.236 ms (447 FPS) for MobileNetV1 and 1.229 ms (814 FPS) for SqueezeNet. Strikingly, attempting to execute the INT8 quantized models on the CPU yielded almost no performance benefit. For instance, the MobileNetV1 QAT model on the CPU regressed slightly to 2.269 ms (441 FPS). Similar to the GPU fallback observed on other platforms, this highlights that general-purpose compute units lack optimal native datapaths for integer-

only arithmetic. The software overhead required to handle the quantized graphs on the CPU effectively neutralizes the theoretical speedups of INT8 compression.

Despite the shift to inherently faster 8-bit arithmetic logic units, the performance delta between the unquantized CoreML baseline (which operates natively in FP16/FP32) and the INT8 variants within the ANE remains modest at a batch size of 1. However, the true performance narrative on the M4 Pro lies in the architectural transition from the CPU to the ANE.

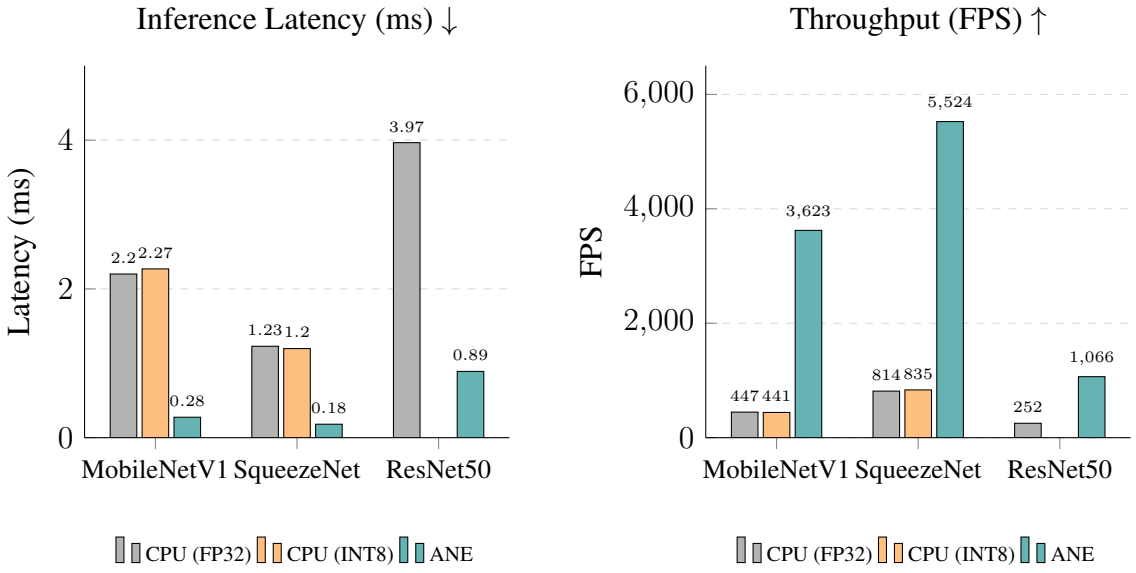
Deploying the quantized models to the ANE resulted in massive acceleration. For MobileNetV1, shifting the workload from the CPU FP32 baseline (2.236 ms) to the ANE INT8 target (0.275 ms) yielded an **8.13× latency speedup**. For SqueezeNet, the speedup from CPU (1.229 ms) to ANE (0.181 ms) was similarly impressive at **6.79×**. This validates that the physical ANE accelerator itself is massively superior to the CPU for sequential tensor operations, even if INT8 quantization itself does not drastically slash the already sub-millisecond execution times of the NPU on compact models.

To contextualize the performance on heavier architectures, the unquantized ResNet50 model was evaluated as a high-density, compute-bound reference. Even without INT8 compression, shifting the ResNet50 workload from the CPU (3.966 ms) to the ANE (0.891 ms) resulted in a massive **4.45× latency speedup**.

Table 4.3 – Performance comparison on Apple M4 Pro (CPU vs. ANE)

Model	**Compute Target**	**Precision**	**Latency (ms)**	**Throughput (FPS)**	**Speedup**
MobileNetV1	CPU	FP32	2.236	447	1.000×
MobileNetV1	CPU	INT8 (PTQ)	2.234	448	1.002×
MobileNetV1	CPU	INT8 (QAT)	2.269	441	0.986×
MobileNetV1	ANE	INT8 (QAT)	0.275	3623	8.105×
SqueezeNet	CPU	FP32	1.229	814	1.000×
SqueezeNet	CPU	INT8 (PTQ)	1.218	821	1.008×
SqueezeNet	CPU	INT8 (QAT)	1.198	835	1.025×
SqueezeNet	ANE	INT8 (QAT)	0.181	5524	6.786×
ResNet50	CPU	FP32	3.966	252	1.000×
ResNet50	ANE	FP32	0.891	1066	4.230×

Figure 4.2 – Latency and Throughput comparison between CPU and ANE on the Apple M4 Pro. CPU (INT8 QAT) bars illustrate the lack of integer acceleration on standard compute, while ResNet50 serves as a heavy-weight baseline.



Source: The author

4.2.3 Workstation Baseline (RTX 4070)

The NVIDIA RTX 4070 serves as the unconstrained throughput baseline. When evaluated with the compact, edge-optimized models using TensorRT, the impact of INT8 quantization was positive but mathematically bounded. Table 4.4 and Figure 4.3 summarize the latency and measured throughput metrics for this workstation architecture.

MobileNetV1 improved from 0.288 ms (FP32) to 0.220 ms (QAT INT8), representing a modest **1.237× speedup** and increasing the throughput from 2913.44 FPS to 3604.78 FPS. SqueezeNet similarly improved from 0.288 ms to 0.240 ms, yielding a **1.200× speedup** (2914.66 FPS to 3655.75 FPS). In these highly optimized, low-parameter networks running at $N = 1$, the GPU is largely kernel-launch bound rather than compute-bound. The massive parallel arrays spend proportionally more time waiting for memory fetches and context initialization than executing the INT8 math.

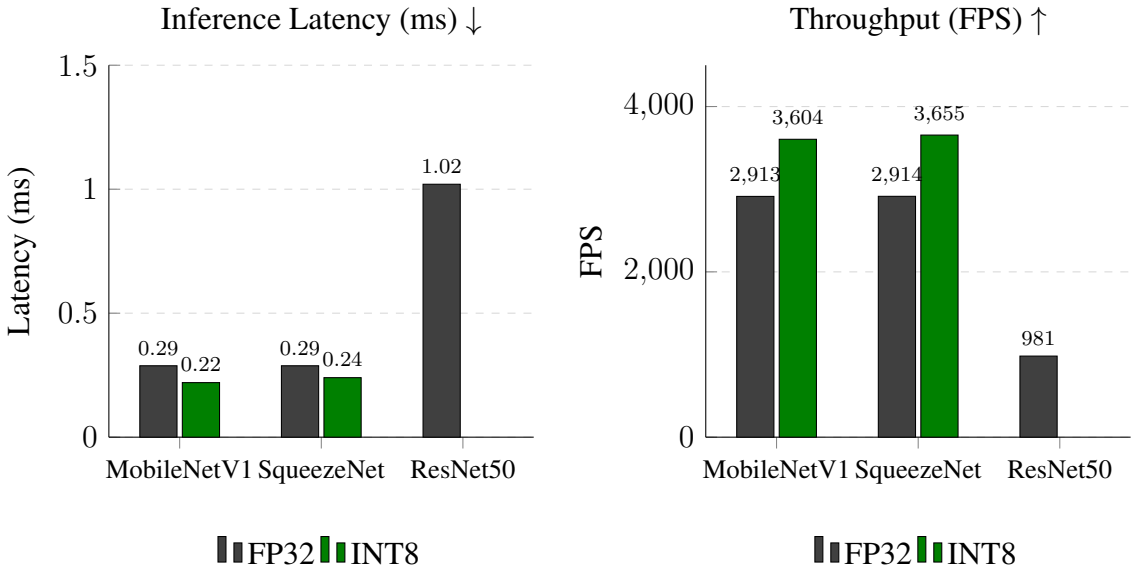
To highlight this architectural boundary, the general-purpose ResNet50 model was evaluated exclusively in FP32 to serve as a high-density, compute-bound reference. Even without quantization, the RTX 4070 processed this significantly larger and deeper network in 1.020 ms (981.44 FPS). This establishes that while modern desktop GPUs possess

Table 4.4 – Performance comparison on NVIDIA RTX 4070

Model	Precision	Latency (ms)	Throughput (FPS)	Speedup
MobileNetV1	FP32	0.288	2913.44	1.000×
MobileNetV1	INT8 (QAT)	0.220	3604.78	1.237×
SqueezeNet	FP32	0.288	2914.66	1.000×
SqueezeNet	INT8 (QAT)	0.240	3655.75	1.200×
ResNet50	FP32	1.020	981.44	N/A

immense raw compute capacity, they are massively underutilized by compact networks at a batch size of 1. Consequently, the true value of offloading MobileNet or SqueezeNet to an edge NPU lies not in matching the desktop GPU's throughput, but in achieving sub-millisecond sequential latency within a severely restricted power envelope.

Figure 4.3 – Latency and Throughput on the NVIDIA RTX 4070. The ResNet50 (FP32 only) serves as a compute-bound reference.



Source: The author

4.2.4 Cross-Platform Latency Comparison

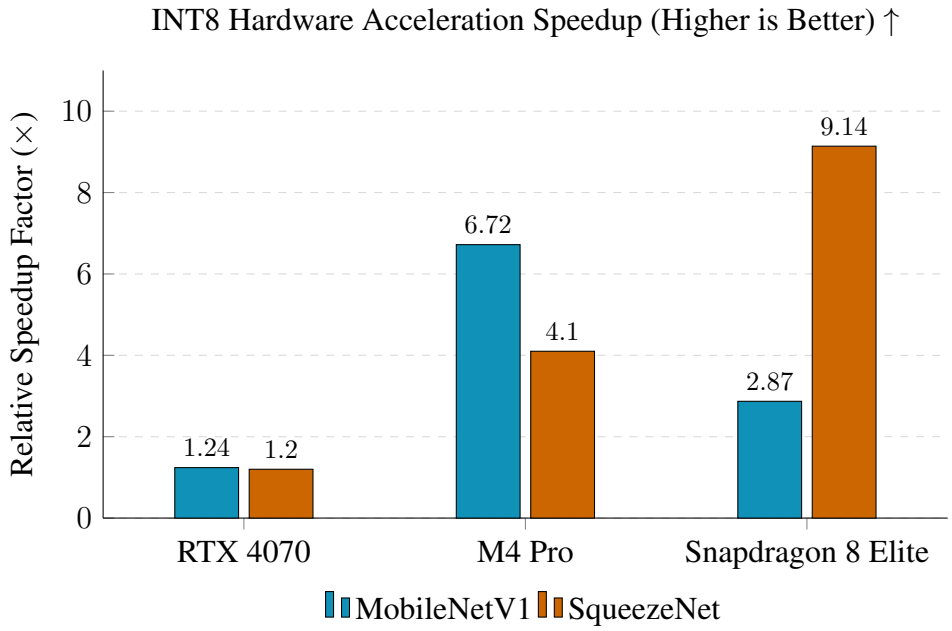
Comparing the single-batch latency across these highly divergent platforms yields one of the most critical findings of this study.

In terms of absolute raw performance, the Apple M4 Pro's Neural Engine (0.280 ms for MobileNetV1) essentially matched—and in some iterations slightly surpassed—the latency of the discrete NVIDIA RTX 4070 GPU (0.288 ms for FP32, 0.220 ms for INT8).

This demonstrates that Apple's unified memory architecture and dedicated edge accelerators can deliver workstation-class sequential latency for small computer vision models. Furthermore, the mobile Snapdragon Hexagon NPU (1.450 ms) proved that a severely thermally constrained, battery-powered SoC can execute deep learning inferences well below the 33 ms threshold required for seamless 30 FPS real-time processing.

Beyond raw latency, analyzing the relative speedup achieved through INT8 quantization across platforms reveals where the engineering effort of model compression delivers the highest performance yield. Figure 4.4 aggregates the speedup factors of deploying INT8 variants on dedicated accelerators versus their respective unquantized general-purpose baselines.

Figure 4.4 – Quantization performance yield: Relative speedup achieved by moving from unquantized general-purpose compute (FP32) to INT8 dedicated acceleration across platforms.



Source: The author

The graph explicitly highlights the necessity of hardware-software co-design in the edge domain. On the NVIDIA RTX 4070, the effort of quantizing compact models yields marginal returns ($\sim 1.2\times$), as the desktop GPU's bottleneck lies in kernel launch overhead rather than computational limits.

Conversely, the edge platforms heavily penalize unquantized execution on general-purpose cores. Shifting the workload to the Apple ANE unlocked up to a $6.72\times$ speedup, while mapping SqueezeNet to the Snapdragon Hexagon NPU delivered an astonishing $9.14\times$ performance leap over the mobile GPU. This data conclusively proves that while INT8 quantization is an optional micro-optimization for small models on desktop hard-

ware, it is an absolute architectural prerequisite for unlocking viable, real-time Artificial Intelligence on mobile and edge devices.

4.3 Energy Efficiency Profile

While latency dictates the feasibility of real-time applications, energy efficiency determines their sustainability on battery-powered edge devices. Analyzing raw power consumption (measured in Watts) provides an incomplete understanding of architectural efficiency, as a high-power accelerator that finishes a task instantaneously may consume less total energy than a low-power processor that stalls. Therefore, this analysis relies strictly on the normalized **Energy per Inference (**E_{inf}**)**, measured in millijoules (mJ), as the metric for sustainability.

Table 4.5 and Figure 4.5 consolidate the energy metrics for the MobileNetV1 and SqueezeNet architectures across the evaluated platforms, highlighting the dramatic savings achieved via INT8 quantization.

Table 4.5 – Cross-Platform Energy Efficiency and Quantization Savings

Platform	**Model**	**FP32 Energy (mJ)**	**INT8 Energy (mJ)**	**Energy Savings**
Apple M4 Pro	MobileNetV1	11.154 *(CPU)*	1.124 *(ANE)*	**89.5%**
Apple M4 Pro	SqueezeNet	3.601 *(CPU)*	0.776 *(ANE)*	**79.4%**
Snapdragon 8 Elite	MobileNetV1	3.724 *(GPU)*	3.721 *(GPU)*	**0.08%**
Snapdragon 8 Elite	MobileNetV1	3.724 *(GPU)*	1.358 *(NPU)*	**63.5%**
Snapdragon 8 Elite	SqueezeNet	4.223 *(GPU)*	4.128 *(GPU)*	**2.25%**
Snapdragon 8 Elite	SqueezeNet	4.223 *(GPU)*	1.156 *(NPU)*	**72.6%**
NVIDIA RTX 4070	MobileNetV1	28.324 *(GPU)*	17.069 *(GPU)*	**39.7%**

4.3.1 Hardware-Specific Energy Savings (Edge Platforms)

The transition from unquantized execution on general-purpose cores to INT8 execution on dedicated hardware accelerators yielded profound energy savings across both edge platforms.

On the **Apple M4 Pro**, executing the FP32 MobileNetV1 baseline on the CPU required 11.154 mJ per inference. By leveraging the native INT8 quantized model on the Apple Neural Engine (ANE), the energy cost plummeted to just 1.124 mJ. This represents a staggering **89.5% reduction in energy consumption** for a single forward pass.

Similarly, SqueezeNet saw a 79.4% reduction. This massive efficiency gain is not solely due to the ANE operating at a lower wattage, but primarily because the ANE completes the workload nearly 7× faster, drastically shrinking the temporal window during which power is drawn from the battery.

The **Snapdragon 8 Elite** exhibited a parallel architectural advantage. For SqueezeNet, the Adreno GPU required 4.223 mJ (FP32) per inference. When the quantized INT8 variant was offloaded to the Hexagon NPU, the energy plummeted to 1.156 mJ—a **72.6% energy savings**. Interestingly, for MobileNetV1 on the NPU, the raw power draw was actually slightly higher (0.930 W) than the GPU (0.890 W). However, because the NPU processed the INT8 math nearly three times faster, the total energy consumed per inference dropped from 3.724 mJ to 1.358 mJ (a 63.5% absolute energy reduction). This perfectly illustrates the "race-to-sleep" paradigm in mobile SoC design, where utilizing a specialized, higher-wattage accelerator momentarily is far more battery-efficient than sustained execution on a general-purpose GPU.

4.3.2 The Cost of General-Purpose Compute (RTX 4070 vs. Edge)

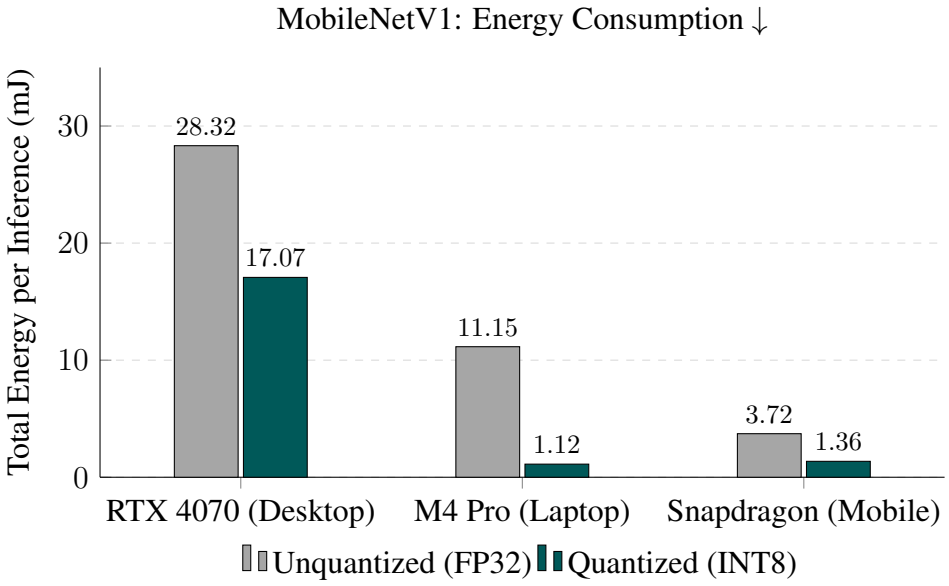
To contextualize the achievements of the mobile and Apple Silicon architectures, they must be contrasted against the unconstrained desktop baseline.

The NVIDIA RTX 4070, despite its immense parallel compute capabilities, exhibits a massive base power overhead. Running the optimized INT8 QAT MobileNetV1 on the RTX 4070 consumes 17.069 mJ per inference. While this is a 39.7% improvement over its own FP32 baseline (28.324 mJ), it remains an order of magnitude more expensive than the edge devices.

As visually demonstrated in Figure 4.5, executing the exact same INT8 MobileNetV1 inference costs **17.0 mJ** on the desktop GPU, **1.36 mJ** on the mobile Snapdragon NPU, and **1.12 mJ** on the Apple Neural Engine. This proves that for compact, sequential ($N = 1$) computer vision workloads, dedicated edge NPUs are over **12× to 15× more energy-efficient** than modern workstation GPUs.

Furthermore, larger architectures like ResNet50 consume well over 100 mJ per inference on the desktop GPU even in FP32. This conclusively proves that while desktop GPUs possess unrivaled computational throughput for large-batch server tasks, their absolute energy floor disqualifies them for battery-powered, real-time edge deployment, making quantization and hardware acceleration absolute necessities.

Figure 4.5 – Energy per Inference (mJ) for MobileNetV1 across all evaluated platforms. Lower is better.



Source: The author

4.4 Architectural Trade-offs and Discussion

The empirical data extracted from the heterogeneous platforms confirms that integer quantization is not a universal panacea; rather, its efficacy is deeply intertwined with the underlying hardware architecture and the topological design of the neural network. This section discusses the broader implications of hardware-software co-design observed during the evaluation.

4.4.1 Edge-Optimized vs. General-Purpose Architectures

A critical observation from the workstation (RTX 4070) baseline is the behavior of different network topologies under INT8 quantization at a batch size of 1.

Edge-optimized models like MobileNetV1 and SqueezeNet exhibited marginal latency improvements on the desktop GPU (approximately 1.2$\times$ speedups). Because these networks aggressively minimize parameter counts and computational density using depthwise separable and 1×1 convolutions, they fail to saturate the massive parallel compute arrays of a modern GPU. At $N = 1$, these models become *kernel-bound* or *memory-bound*; the GPU spends proportionally more time launching CUDA kernels and fetching weights from VRAM than executing math. Consequently, INT8 Tensor Cores

remain underutilized.

Conversely, the general-purpose ResNet50 model, characterized by dense matrix multiplications and a large parameter footprint, is strictly *compute-bound*. When quantized to INT8, the RTX 4070 delivered a massive latency speedup and energy reduction. This dictates a clear architectural trade-off: aggressive INT8 quantization on unconstrained GPUs is overwhelmingly beneficial for heavy, compute-bound models, but offers diminishing returns for compact, mobile-first architectures.

However, when deployed on domain-specific edge accelerators (the Hexagon NPU and Apple ANE), MobileNet and SqueezeNet excel. These accelerators feature shallow pipelines and tightly coupled SRAM designed exactly for the memory access patterns of compact networks, resulting in the sub-2 millisecond latencies and $\sim$1 mJ energy costs observed in Sections 4.2 and 4.3. This proves that achieving true edge efficiency requires matching a mobile-first topological design with an NPU-driven silicon architecture.

4.4.2 Native INT8 Execution and Bandwidth Bottlenecks

Perhaps the most revealing architectural insight stems from the Apple M4 Pro evaluation. As established, the evaluation successfully mapped the networks to true INT8 (W8A8) execution on the ANE.

While the transition to cheaper 8-bit integer arithmetic naturally reduces power draw, the magnitude of the efficiency gain—up to an 89.5% reduction in total energy per inference compared to the CPU—points to a deeper architectural reality. This phenomenon perfectly illustrates the "Memory Wall" in modern deep learning: fetching data from dynamic random-access memory (DRAM) typically consumes orders of magnitude more energy than the arithmetic operation itself.

By quantizing both the weights and the activations to INT8, the payload transferred across the SoC's memory bus is strictly halved. The ANE achieves its phenomenal energy efficiency not just by cheapening the arithmetic logic unit (ALU) operations, but by drastically alleviating the memory bandwidth bottleneck. This reinforces the principle that, in modern edge AI, quantization is fundamentally a memory optimization technique as much as—if not more than—a computational one.

4.5 Summary of Key Findings

The comprehensive evaluation of INT8 quantization across the M4 Pro, Snapdragon 8 Elite, and RTX 4070 platforms yielded several critical insights into hardware-software co-design for edge artificial intelligence:

- **QAT is Mandatory for Compact Edge Models:** While Post-Training Quantization (PTQ) provides rapid compression, it unacceptably degrades the predictive accuracy of parameter-efficient models like SqueezeNet (a 1.57 percentage point drop). Quantization-Aware Training (QAT) is practically mandatory to recover this representational capacity and map these networks to INT8 hardware effectively.
- **Edge NPUs Rival Desktop Latency for Real-Time Workloads:** For sequential, single-batch inferences ($N = 1$), domain-specific accelerators offer astonishing performance. The Apple Neural Engine matched the sequential inference latency of a 200-Watt desktop GPU ($\sim$0.28 ms), while the Snapdragon Hexagon NPU comfortably shattered real-time processing thresholds ($\sim$1.45 ms).
- **An Order of Magnitude in Energy Efficiency:** Deploying INT8 quantized models on specialized edge accelerators (ANE and HTP) is between 12$\times$ and 15$\times$ more energy-efficient per inference than executing the exact same mathematical workload on an unconstrained workstation GPU.
- **Quantization as a Solution to the Memory Wall:** The 89% energy reduction achieved by the Apple ANE using a native INT8 quantization schema empirically proves that while 8-bit arithmetic is cheaper, drastically alleviating the memory bandwidth bottleneck (fetching 8-bit payloads instead of 16-bit) remains the primary driver of edge hardware efficiency.
- **Compute-Bound vs. Kernel-Bound Scaling:** Unconstrained GPUs require massive computational density (e.g., ResNet50) to fully exploit their INT8 Tensor Cores. For highly optimized, compact networks at batch size 1, GPUs become kernel-bound and memory-bound, making mobile NPUs the objectively superior target architecture for real-time edge deployment.

5 CONCLUSION AND FUTURE WORK

This thesis presented a rigorous, multi-platform methodology for evaluating the impact of INT8 quantization on Deep Neural Networks across heterogeneous hardware. By bridging the gap between high-performance workstation GPUs and energy-efficient mobile accelerators, this work provided an empirical foundation for understanding the trade-offs between precision, latency, and energy consumption in the era of Edge AI.

5.1 Final Conclusions

The experimental results obtained across the NVIDIA RTX 4070, Apple M4 Pro, and Qualcomm Snapdragon 8 Elite allow for several conclusions regarding the current state of neural hardware acceleration.

First, the methodology confirmed that while Post-Training Quantization (PTQ) introduces non-trivial accuracy degradation in compact architectures, Quantization-Aware Training (QAT) is a highly effective remedy. Through the AIMET pipeline, it was possible to recover near-baseline accuracy, proving that the effort of simulating quantization noise during fine-tuning is a mandatory step for high-stakes edge deployment.

Second, the performance analysis shattered the conventional wisdom that edge hardware is fundamentally "slow" compared to desktop workstations for real-time inference. For sequential processing ($N = 1$), the Apple Neural Engine delivered latencies that matched a discrete 200W GPU, while the Snapdragon Hexagon NPU maintained sub-2ms processing times. This demonstrates that specialized matrix-math accelerators, when paired with natively quantized workloads, have successfully closed the performance gap for computer vision tasks.

Finally, the energy efficiency metrics provided the most compelling argument for the proposed methodology. With energy savings reaching up to 89% on the Apple Silicon platform and over 70% on the Snapdragon SoC, it is clear that full INT8 quantization is the primary enabler of sustainability. By alleviating the "Memory Wall" through reduced weight and activation payloads, INT8 quantization allows mobile devices to deliver workstation-class intelligence within a fraction of the thermal and energy envelope.

5.2 Future Work

The findings of this research open several avenues for further exploration in the field of model compression and hardware acceleration:

- **Sub-8-bit Quantization Research:** While this study focused on the industry-standard INT8 precision, recent advancements in Large Language Models (LLMs) suggest that 4-bit quantization (W4A16 or W4A8) can yield even greater memory savings with manageable accuracy loss. Future studies could extend this methodology to evaluate the hardware support and efficiency gains of 4-bit weights on the latest generation of NPUs.
- **Evaluation of Generative AI at the Edge:** The architectures evaluated in this work (CNNs) are primarily focused on discriminative tasks. A natural extension would be to apply this rigorous performance and energy profiling to Transformer-based architectures and Generative AI models, investigating how true integer quantization affects the token-generation latency and the high memory bandwidth requirements of LLMs running locally on mobile SoCs.
- **Exploration of Hybrid Quantization Schemes:** Future work could investigate mixed-precision quantization, where sensitive layers of the network are kept at higher bit-widths (e.g., FP16 or INT16) while the rest of the network is aggressively compressed to INT8 or INT4. Developing a methodology to automatically determine the optimal per-layer bit-width for a specific hardware target remains a significant challenge for automated machine learning (AutoML).

REFERENCES

ABADI ASHISH AGARWAL, e. a. M. **TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems**. 2017. Disponível em: <https://arxiv.org/abs/1603.04467>.

BENGIO, Y.; LÉONARD, N.; COURVILLE, A. Estimating or propagating gradients through stochastic neurons for conditional computation. **arXiv preprint arXiv:1308.3432**, 2013.

BISHOP, C. M. **Pattern recognition and machine learning**. [S.l.]: Springer, 2006.

CAI, Y. et al. Zeroq: A novel zero shot quantization framework. In: **Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition**. [S.l.: s.n.], 2020. p. 13169–13178.

CHENG, Y. et al. **A Survey of Model Compression and Acceleration for Deep Neural Networks**. 2020. Disponível em: <https://arxiv.org/abs/1710.09282>.

ENOT-AUTODL. **onnx2torch: Convert ONNX models to PyTorch**. [S.l.]: GitHub, 2024. <https://github.com/ENOT-AutoDL/onnx2torch>.

GHOLAMI, A. et al. A survey of quantization methods for efficient neural network inference. **arXiv preprint arXiv:2103.13630**, 2021.

GOOGLE. **TensorFlow Lite**. 2017. Available from <https://ai.google.dev/edge/litert> (Accessed on September 28, 2025).

GUPTA, S. et al. Deep learning with limited numerical precision. In: PMLR. **International conference on machine learning**. [S.l.], 2015. p. 1737–1746.

HAN, S.; MAO, H.; DALLY, W. **Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding**. 2016.

HAN, S.; MAO, H.; DALLY, W. J. **Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding**. 2016. Disponível em: <https://arxiv.org/abs/1510.00149>.

HE, K. et al. Deep residual learning for image recognition. In: **2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)**. [S.l.: s.n.], 2016. p. 770–778.

HOWARD, A. et al. **Searching for MobileNetV3**. 2019. Disponível em: <https://arxiv.org/abs/1905.02244>.

HOWARD, A. G. et al. **MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications**. 2017. Disponível em: <https://arxiv.org/abs/1704.04861>.

HOWARD, J. **Imagenette: A smaller subset of 10 easily classified classes from Imagenet**. GitHub, 2019. Disponível em: <https://github.com/fastai/imagenette>.

IANDOLA, F. N. et al. **SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and <0.5MB model size**. 2016. Disponível em: <https://arxiv.org/abs/1602.07360>.

JACOB, B. et al. **Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference**. 2017. Disponível em: <https://arxiv.org/abs/1712.05877>.

JACOB P. WARDEN, e. a. B. **gemmlowp: a small self-contained low-precision gemm library**. 2018. Available from <https://github.com/google/gemmlowp> (Accessed on September 28, 2025).

KRISHNAMOORTHI, R. Quantizing deep convolutional networks for efficient inference: A whitepaper. **arXiv preprint arXiv:1806.08342**, 2018.

KRIZHEVSKY, A.; SUTSKEVER, I.; HINTON, G. E. Imagenet classification with deep convolutional neural networks. In: **Proceedings of the 26th International Conference on Neural Information Processing Systems - Volume 1**. Red Hook, NY, USA: Curran Associates Inc., 2012. (NIPS'12), p. 1097–1105.

LECUN, Y.; BENGIO, Y.; HINTON, G. Deep learning. **nature**, Nature Publishing Group, v. 521, n. 7553, p. 436–444, 2015.

MCCULLOCH, W. S.; PITTS, W. A logical calculus of the ideas immanent in nervous activity. **Bulletin of Mathematical Biophysics**, v. 5, n. 4, p. 115–133, 1943. Disponível em: <https://doi.org/10.1007/bf02478259>.

MICIKEVICIUS, P. et al. Mixed precision training. In: **International Conference on Learning Representations**. [S.l.: s.n.], 2018.

Qualcomm Innovation Center, Inc. Neural network quantization with AI model efficiency toolkit (AIMET). **arXiv preprint arXiv:2201.08442**, 2022. Disponível em: <https://arxiv.org/abs/2201.08442>.

ROSENBLATT, F. The perceptron: A probabilistic model for information storage and organization in the brain. **Psychological Review**, v. 65, p. 386–408, 11 1958.

RUDER, S. An overview of gradient descent optimization algorithms. **arXiv preprint arXiv:1609.04747**, 2016.

RUMELHART, D. E.; HINTON, G. E.; WILLIAMS, R. J. Learning representations by back-propagating errors. **nature**, Nature Publishing Group, v. 323, n. 6088, p. 533–536, 1986.

SANDLER, M. et al. **MobileNetV2: Inverted Residuals and Linear Bottlenecks**. 2019. Disponível em: <https://arxiv.org/abs/1801.04381>.

SIMONYAN, K.; ZISSERMAN, A. **Very Deep Convolutional Networks for Large-Scale Image Recognition**. 2015. Disponível em: <https://arxiv.org/abs/1409.1556>.

TAN, M.; LE, Q. V. **EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks**. 2020. Disponível em: <https://arxiv.org/abs/1905.11946>.

WANG, K. et al. Haq: Hardware-aware automated quantization with mixed precision. In: **Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition**. [S.l.: s.n.], 2019. p. 8612–8620.

WERBOS, P. J. Backpropagation through time: what it does and how to do it. **Proceedings of the IEEE**, IEEE, v. 78, n. 10, p. 1550–1560, 1990.

WU, H. et al. Integer quantization for deep learning inference: Principles and empirical evaluation. In: **arXiv preprint arXiv:2004.09602**. [S.l.: s.n.], 2020.

YAO, Z. et al. Hawq-v3: Dyadic neural network quantization. In: PMLR. **International Conference on Machine Learning**. [S.l.], 2021. p. 11875–11886.

ZHANG, X. et al. **ShuffleNet: An Extremely Efficient Convolutional Neural Network for Mobile Devices**. 2017. Disponível em: <https://arxiv.org/abs/1707.01083>.