POLITECNICO DI TORINO

Master’s degree in electronic engineering

Master’s degree thesis

End-to-End Capsule Network Optimization for Edge-Based Indoor Localization: From Quantization to Hardware Implement ation

Supervisor

Prof. Mihai Teodor Lazarescu

Candidate

Ramin Nazari

July 2025

Abstract

Indoor human localization has gained significant importance because of its wide range of applications, including smart environments, health monitoring and security systems. However, still one of the main challenges is having an accurate indoor positioning system due to complex environments, signal interference, and the demand for real-time processing. While traditional approaches like Recurrent Neural Network (RNNs) are often used for time-series sensor data, they can suffer from high resource consumption and difficulty in modeling complex relational patterns. To address this challenges, this thesis presents a solution using Capsule Network (CAPSNET) with capacitive sensors for indoor localization and the motivation of using CAPSNETS lies in their inherent ability to model hierarchical part-whole relationships which allows to better interpret complex and overlapping temporal patterns from sensors compared to conventional architectures so it helps to achieve high accuracy by preserving the rich capacity of CAPSNETS comes at the cost of high computational and memory requirements, making them so difficult to deploy on real-time, resource-constrained edge devices like Field-Programmable Gate Arrays (FPGA).

The experimental setup was designed to mimic a realistic indoor environment, specifically an empty 3 m × 3 m room. Capacitive sensors were placed at chest height, with each sensor featuring a 16 cm × 16 cm sensing plate operating in load mode. These sensors capture the capacitance changes induced by human proximity, providing three readings per second (3 Hz). Then an indirect measurement of the capacitance has been performed by measuring the frequency of a 555 timerbased relaxation oscillator. The capacitive coupling between the human body and the sensors allows distance to be inferred from the measured capacitance. In addition, a reference system based on ultrasound anchors was employed to track the person's exact location with ±2 cm accuracy at 15 Hz, providing ground truth data for performance evaluation.

This work introduces a second key contribution: a complete optimization and implementation workflow, featuring a custom mixed-precision quantization framework based on Post Training Quantization (PTQ) method and a subsequent High-Level Synthesis (HLS) implementation. The custom framework was developed to handle the unique numerical sensitivity of the CAPSNET'S dynamic routing algorithm, which requires a highly accurate quantization approach. This end-to-end process successfully compressed the model by a factor of 4.56x (from 59.72 KB to 13.1 KB) with negligible accuracy degradation (less than 0.5% on test set), demonstrating the feasibility of deploying this advanced architecture in practical applications. This thesis demonstrates the first hardware-ready PTQ Solution for dynamic routing capsules on temporal data and provides a complete methodology for designing and implementing a novel deep learning model for efficient and accurate indoor localization on edge devices.

Despite these successful results of our work, several key challenges due to complexity and high sensitivity of CAPSNETS remain. The high numerical sensitivity of the dynamic routing algorithm, which requires maintaining high-precision (18-20 bit) activations to preserve accuracy, poses a significant hurdle for further optimization. Future work should

focus on Quantization-Aware Training (QAT), which could make the model inherently more robust to aggressive, low-bit quantization because it allows the model to be trained on simulated quantized data so the model learns to adapt and handle the constraints of low-precision data. Furthermore, a complete hardware resource and power analysis on a target FPGA is required to validate real-world efficiency. Exploring architectural enhancements to the CAPSNET itself, such as alternative routing algorithms, also presents a promising avenue for improving both performance and efficiency. This thesis provides a complete methodology for designing, optimizing, and implementing a novel deep learning model for efficient and accurate indoor localization on edge devices.

Contents

1. CHAPTER 1: Introduction

1.1. Background and Motivation

The rapid increase of Internet of Things (IoT) devices has ushered in an era of intelligent environments, where spaces can perceive and react to human presence and activity. From smart homes that adjust lighting and temperature based on occupancy to assisted living facilities that monitor the well-being of elderly residents, the ability to accurately and non-intrusively locate individuals indoors has become a key technology. This capability, known as Indoor Localization, fuels a wide range of applications, including security systems, energy management, and personalized healthcare.

While outdoor localization has been effectively solved by Global Navigation Satellite Systems (GNSS) like GPS, these signals are unreliable indoors. This has driven significant research into alternative technologies. However, many existing solutions face a critical trade-off. High-accuracy systems like Ultra-Wideband (UWB) often require expensive, specialized structure, while wide-ranging technologies like Wi-Fi and Bluetooth suffer from lower accuracy due to signal interference in complex indoor environments [1]. Furthermore, many approaches require the person to carry a specific device, which is not always practical or desirable.

A parallel challenge lies in the deployment of intelligent algorithms that process sensor data. The rise of deep learning has provided powerful tools for interpreting complex patterns, but these models are often computationally and memory-intensive. This makes it difficult to deploy the low-cost, low-power edge devices that are typical in IoT applications. The need to send data to the cloud for processing introduces latency, privacy concerns, and reliance on network connectivity. Therefore, there is a pressing need for solutions that can run sophisticated models efficiently at the edge.

1.2. Problem Statement

The central problem addressed in this thesis is the accurate, real-time indoor localization of a human subject using a computationally efficient deep learning model suitable for deployment on resource-constrained edge devices. This challenge is multifaceted:

1. Sensing Modality: A non-intrusive, device-free sensing method is required that can provide rich data for localization.

2. Model Architecture: A model is needed that can effectively learn complex temporal and spatial relationships from sensor data to achieve high accuracy, surpassing traditional models that may discard critical information.

3. Computational Efficiency: The chosen model, despite its potential complexity, must be heavily optimized to meet the strict memory and computational budgets of edge hardware without a significant loss in performance.

4. Hardware Implementation: A clear and verifiable pathway must be established to translate the optimized software model into a functional hardware accelerator.

1.3. Proposed Solution and Contributions

This thesis presents an end-to-end workflow that addresses these challenges by combining a novel deep learning architecture with a specialized optimization and implementation pipeline.

The proposed solution is based on a Capsule Network (CAPSNET) that processes time-series data from non-intrusive capacitive sensors. The motivation for using CAPSNETS lies in their inherent ability to model hierarchical part-whole relationships, which we hypothesize is ideal for interpreting the complex temporal patterns of human movement.

To overcome the computational cost of CAPSNETS, this work introduces a custom, mixed-precision quantization framework. This framework uses a smart search algorithm to drastically reduce the model's size and computational requirements while preserving its high accuracy. Finally, the optimized model is implemented as a hardware accelerator using High-Level Synthesis (HLS), demonstrating a complete path from algorithm design to a verifiable hardware representation.

The key contributions of this work are:

1. A novel application of CAPSNETS to a time-series regression task for indoor localization.

2. A custom-built, mixed-precision quantization framework specifically designed to handle the numerical sensitivities of the CAPSNET architecture.

3. A complete and verified end-to-end workflow from model conception and training to software optimization and bit-accurate hardware implementation.

2. CHAPTER 2: Literature Review

This chapter provides a review of the existing body of work relevant to this thesis. It covers the primary technologies used for indoor localization, the application of deep learning to time-series data, the architectural principles of CAPSNETS, and techniques for model optimization and hardware acceleration.

2.1. Indoor Localization Technologies

The field of indoor localization is rich with diverse approaches. Radio Frequency (RF) based systems are the most common, utilizing signals that are already prevalent in many environments [2]. Wi-Fi-based localization, which often uses Received Signal Strength Indication (RSSI) fingerprinting, is a popular low-cost method but is susceptible to multipath fading and environmental changes [2]. Bluetooth Low Energy (BLE) beacons offer another low-power alternative, but their accuracy can be similarly affected by environmental factors [2]. Ultra-Wideband (UWB) technology provides centimeter-level accuracy by measuring the time-of-flight of radio signals, but it typically requires the deployment of a dedicated and often costly infrastructure of anchors [2].

In contrast to these device-based approaches, device-free localization aims to detect and track subjects without requiring them to carry any electronic device. These methods often rely on sensing disturbances in ambient signals. Capacitive sensing, the modality used in this thesis, falls into this category. It works by measuring changes in the electric field caused by the presence and movement of a human body, which has different dielectric properties than the surrounding air. Research by G. Subbicini (2023) and others has demonstrated its potential for creating rich, informative data suitable for activity recognition and localization tasks [3].

2.2. Deep Learning for Time-Series Localization

Once sensor data is collected, machine learning is used to map the time-series signals to a position. While traditional algorithms like Support Vector Machines (SVM) have been used, deep learning models have become state-of-the-art. Recurrent Neural Networks (RNNs), particularly Long Short-Term Memory (LSTM) and Gated Recurrent Unit (GRU) networks, are a natural fit for sequential data, as their internal states allow them to model temporal dependencies. 1D Convolutional Neural Networks (CNNs) are also effective at extracting local features and patterns from time-series data and are often more computationally efficient than RNNs. In fact, direct comparisons on the capacitive sensor dataset found that a Temporal Convolutional Network (TCN) could achieve higher accuracy with significantly fewer resources than a baseline 1D-CNN, highlighting the

potential for non-recurrent architectures [4]. These models, however, can sometimes struggle to capture higher-order, hierarchical relationships within the data without becoming very large and complex.

2.3. Capsule Networks (CAPSNETS)

The core idea behind CAPSNETS is that a neural network should explicitly model hierarchical part-whole relationships using vector outputs which have its roots in earlier research on transforming auto-encoders [5]. This foundational concept was later developed into the more concrete architecture featuring a 'dynamic routing' mechanism, which was introduced by Sabour, Frosst, and Hinton [6] as an alternative to traditional CNNs for computer vision. Their key innovation is the "capsule," a group of neurons whose output is a vector, not a scalar. The length of the vector represents the probability of a feature's existence, while its orientation encodes the feature's properties (e.g., pose, texture). Capsules are connected by "dynamic routing," an iterative process where low-level capsules send their output to high-level capsules with which they "agree." This mechanism allows CAPSNETS to robustly model part-whole relationships. While their success has been primarily in image recognition, their application to other domains is an emerging area of research. This thesis explores their potential for time-series analysis, hypothesizing that their ability to model hierarchical relationships is well-suited for interpreting the composite patterns of human movement.

2.4. Model Optimization and Hardware Acceleration

The deployment of deep learning models on resource-constrained edge devices is a significant challenge. Quantization is a key optimization technique that reduces the numerical precision of a model's weights and activations (e.g., from 32-bit floating-point to 8-bit integer), thereby decreasing memory footprint and computational cost. Post-Training Quantization (PTQ) is a popular method where a pre-trained model is quantized without retraining, a technique thoroughly detailed in influential works on efficient network deployment [7]. However, for sensitive models like CAPSNETS, a trade-off that has been well-documented in studies on efficient inference [8]. This has motivated the development of mixed-precision techniques, where different parts of the model are quantized to different bit-widths.

To run these models efficiently, custom hardware accelerators are often designed, particularly on Field-Programmable Gate Arrays (FPGAs). High-Level Synthesis (HLS) has become a popular design methodology, allowing hardware to be described in high-level languages like C++ and then automatically synthesized into a hardware description language (HDL) like Verilog or VHDL. This significantly accelerates the hardware design cycle compared to manual HDL coding [9].

2.5. Research Gap

While the literature contains extensive research on each of these individual areas, there is a lack of work addressing the complete end-to-end workflow for a novel and sensitive architecture like a CAPSNET. Specifically, the research gaps in this thesis aims to fill is the development and verification of a complete pipeline that takes a time-series CAPSNETS from initial design, through specialized mixed-precision quantization, to a bit-accurate and verifiable HLS implementation for edge deployment.

3. CHAPTER 3: Methodology

3.1. Capsule Network Architecture for Localization

While Recurrent Neural Networks (RNNs) like LSTMs are a standard choice for time-series analysis, they were not selected for this task. The primary hypothesis of this work is that the subtle, overlapping temporal patterns from the capacitive sensors correspond to part-whole relationships in a person's movement. A CAPSNET, with its dynamic routing mechanism, is theoretically better suited to model these hierarchical dependencies explicitly. An LSTM, while excellent at capturing temporal sequences, might model these relationships implicitly but less robustly, potentially requiring more data to learn. Therefore, the CAPSNET was chosen to test its unique architectural advantages on this specific type of sensor data.

General Concept: CAPSNETS are a type of neural network architecture designed to better model hierarchical relationships between features and their properties. Unlike traditional neurons that output a single scalar value, capsules output a vector (or a set of vectors). The length of this vector represents the probability of a feature's existence, while its orientation encodes the feature's instantiation parameters (e.g., properties like shape, timing, or intensity of a pattern in time series).

Dynamic Routing: A key mechanism in CAPSNETS is dynamic routing (often "routing-by-agreement"). Instead of pooling layers found in many CNNs (which can discard precise positional information), dynamic routing allows lower-level capsules to iteratively decide how to send their output to higher-level capsules. A higher-level capsule becomes active if multiple lower-level capsules "agree" by sending similar predictions, effectively allowing the network to learn part-whole relationships robustly.

Potential for Time Series: For time series data, CAPSNETS offer the potential to capture not just the presence of temporal patterns but also their specific characteristics and how they combine to form more complex sequences. Their aim for equivariance (where transformations in the input lead to predictable changes in the output representation) rather than strict invariance may help in better generalization from smaller datasets compared to some conventional approaches like CNNs or LSTMs.

3.2. Mixed-Precision Post-Training Quantization Framework

Model quantization is a critical optimization technique that reduces the numerical precision of a model's weights and activations. This process significantly decreases the model's memory footprint and computational requirements, leading to faster inference speeds and lower power consumption, which is crucial for deploying complex deep learning models on resource-constrained edge devices.

Our CAPSNET model, while effective, is resource-intensive. Therefore, quantization is a primary objective. However, the core mechanisms of the network, including the dynamic routing algorithm and the squash activation function, are highly sensitive to numerical precision so here is the point that custom mixed-precision quantization idea plays a crucial role rather than uniform precision. To address these challenges, this project utilizes a custom, specialized PTQ framework that employs a mixed-precision strategy. The framework was developed using the TensorFlow library, with standard layers implemented as wrappers around native keras layers to inject quantization logic. The core idea of the framework and multi-step search methodology is inspired by the specialized framework for quantizing CAPSNETS first proposed by Marchisio et al..[10].

Instead of uniform quantization, the framework's core is a smart search algorithm that finds the optimal NIB and NFB for each tensor, guided by a user-defined loss tolerance and memory budget. The architecture consists of several key functional blocks:

a. Quantization Simulation Primitives: At the core of the framework are primitive functions that simulate the effects of fixed-point quantization on floating-point values. This block is highly configurable, supporting distinct rounding strategies.

b. Quantized Model Construction: The framework constructs a new, fully quantizable version of the model using custom layers and wrappers. Standard keras layers are placed inside a custom wrapper, while specialized layers were developed from the ground up.

c. Model Builder from Configuration: This block functions as a model factory, receiving a quantization configuration and building the corresponding ready-to-evaluate model instance.

d. Calibration Block: This crucial initial block analyzes the pre-trained full-precision model to extract the baseline clipping range for all weights and activations.

e. Smart Search Block: This is the principal block, organizing the optimization. It implements an efficient search strategy to find the optimal NFB for each layer that minimizes memory footprint while satisfying user-defined constraints for loss tolerance and memory budget. The search strategy involves multiple phases, including an initial uniform search, a layer-wise search on weights to meet a memory budget, and a final refinement step on activations that includes a recovery phase if the performance tolerance is violated.

Limitations of the Post-Training Quantization (PTQ) Approach: It is important to note that Post-Training Quantization, while fast and effective for moderate compression, has

significant limitations, especially under aggressive quantization schemes. The PTQ process optimizes a model that has already been trained in a full-precision environment and is therefore naive to quantization errors. When attempting extreme compression, such as reducing precision to very low bit-widths (e.g., 1-2 bits), PTQ often leads to a catastrophic drop in accuracy and an increase in loss. The pre-trained weights are simply unable to function correctly after such a drastic reduction in their numerical range.

In this situation, Quantization-Aware Training (QAT) becomes the more reliable and necessary choice. QAT simulates the quantization noise directly within the training loop. This allows the model to learn and adapt to the constraints of low-precision arithmetic, effectively recovering the performance drop by adjusting its weights to be inherently robust to quantization. While PTQ was a practical choice for this project's goal of efficiently optimizing an existing full-precision model, QAT would be the crucial methodology for any future work aiming to achieve ultra-low-bit-width implementations [11].

3.3. High-Level Synthesis (HLS) Strategy for Hardware Acceleration

The primary objective of this phase is to implement the fully quantized CAPSNET as a hardware accelerator using High-Level Synthesis. The implementation will use the specific fixed-point bit-widths (NIB and NFB) data type called ap-fixed and the number of fractional and integer bits for activations and weights are determined by our custom quantization framework to create an efficient hardware design that minimizes resource usage and latency.

High-Level Synthesis (HLS) Design Limitations: The hardware implementation, while bit-accurate, is not without its own constraints. The use of a uniform ap_fixed<32, 16> accumulator type for all layers was a design decision made for simplicity and to guarantee no overflow. However, this is a suboptimal approach, a more granular analysis could identify layers where a smaller accumulator would suffice, leading to further savings in FPGA logic resources. Furthermore, this work focuses on C/RTL co-simulation for verification and does not extend to post-synthesis resource and power analysis on a specific FPGA target, which would be required for a full production deployment.

4. CHAPTER 4: Implementation and Experimental Setup

4.1. Dataset and Preprocessing

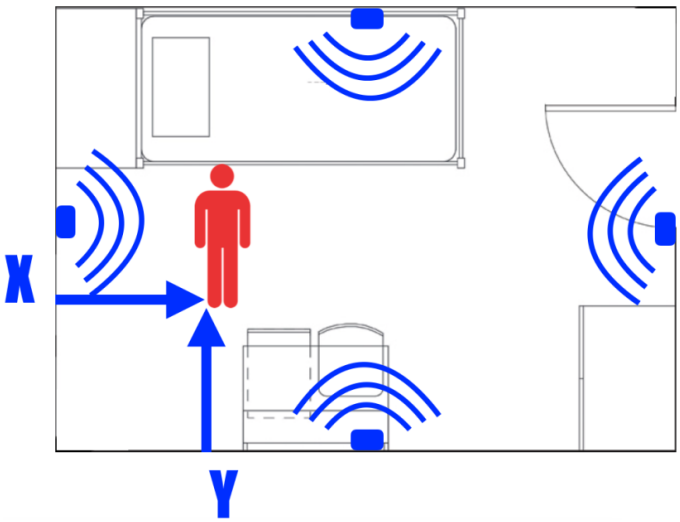


Fig. 4.1. Four capacitive sensors centered on the walls of a 3 m × 3 m virtual room in the lab trace the position of a person moving in the space

The dataset used for this project is sourced from an experiment conducted in a 3 m × 3 m laboratory space.

- Data Source: The data originates from four single-plate, load-mode capacitive sensors positioned at the center of the virtual walls of the space. These sensors record changes in capacitance correlated with a person's position, with the data being labeled with the person's ground-truth (X, Y) coordinates. The data was recorded at a sampling rate of 3 Hz.

- Preprocessing: The raw sensor data underwent a two-stage preprocessing procedure to reduce noise and isolate relevant signals. This included:
  - A median filter with a 50-second input window.
  - A subsequent low-pass filter with a transition band between 0.3 Hz and 0.4 Hz.

- Data Split: The preprocessed dataset was split chronologically to ensure that the model is tested on data that occurs later in time than the training data. Following the methodology described by G. Subbicini (2023) [12], the data was partitioned into three sets:
  - Training Set: 60%
  - Validation Set: 20%
  - Test Set: 20%

- Input/Output Specification for the Model:
  - Input: sequences of 15 time steps × 4 features.
  - Output: sequences of 1 time step × 2 features.

4.2. Model Implementation and Hyperparameter Optimization

4.2.1. Model Architecture

The implemented model consists of a sequence of convolutional layers followed by custom capsule layers and dense layers for final regression.

- Standard Layers: The model begins with two standard 1D convolutional layers, each using the Rectified Linear Unit (ReLU) activation function, for initial feature extraction and after primary capsule layer and digit capsule layer, it is followed by a flatten layer and two dense layers. The first two dense layers also use a ReLU activation function, while the final output layer has a linear activation to map the capsule features to the (X, Y) coordinate outputs.
- PrimaryCaps Layer:
  - Role: This layer acts as the bridge between the standard feature maps of the convolutional layers and the capsule domain. It takes the detected features from the Conv1D layers and groups them into low-level capsules. The output of each capsule is passed through a non-linear "squash" activation function. This function normalizes the magnitude of an input vector to a length between 0 and 1, representing a probability, while preserving its direction. The formula is given by:

$$v = \frac{||s||^2}{1 + ||s||^2} \frac{s}{||s||}$$

  - Parameters: The layer is defined by hyperparameters such as num_capsule_caps1 (the number of primary capsules to create) and dim_capsule_caps1 (the dimensionality of each capsule's output vector).
  - Sensitive Operations: Each primary capsule's output is passed through a "squash" function. This non-linear activation normalizes the capsule's vector length to represent probability. It involves calculating the vector's squared norm, square root, and vector-wise division, all of which are sensitive to numerical precision.
- DigitCaps Layer (Secondary Capsule Layer):
  - Role: This is the core reasoning layer of the network. It receives input from all primary capsules and aims to identify more complex, higher-level patterns or part-whole relationships in the data. The coupling between the primary capsules (layer L) and the digit capsules (layer L+1) is determined by the iterative

dynamic routing algorithm. For a set number of routing iterations (in this work, routings=3)

- Parameters: Its key parameters are num_capsule_caps2 (the number of high-level capsules), dim_capsule_caps2 (their vector dimensionality), and routings (the number of iterations for the dynamic routing algorithm).
- Sensitivity and a Deeper Look at Dynamic Routing: This layer's sensitivity comes from its dynamic routing algorithm. This is not a simple feed-forward connection; it is an iterative process where the following steps occur for each of the (routings=3) iterations:
- Prediction Vectors (û): The output vectors of the primary capsules are first transformed by trainable weight matrices to produce "prediction vectors" or "votes", denoted as û. Each û is a primary capsule's prediction for the output of a higher-level DigitCap.

$$\hat{u}_{j|i} = W_{ij} u_i$$

- Logits and Coupling Coefficients (b, c): For each prediction vector û, a temporary logit value, b, is maintained. These logits, which represent the log prior probabilities of a primary capsule being coupled with a DigitCap, are initialized to zero. They are iteratively updated based on the agreement between û and the current DigitCap output. The actual strength of the connection is determined by a coupling coefficient c_{ij}, calculated using the softmax function on a temporary logit value b_{ij}, which is initialized to zero.

$$c_{ij} = \frac{\exp(b_{ij})}{\sum_k \exp(b_{ik})}$$

- Weighted Sum (s): The total input to each DigitCap is a weighted sum (s) of all the prediction vectors (û), it receives from the layer below, with each vote being weighted by its corresponding coupling coefficient c.

$$s_j = \sum_i c_{ij} \hat{u}_{j|i}$$

- Final Output Vector (v): Finally, the output of each DigitCap, v_j is produced by applying the non-linear squash function to its weighted sum vector (s_j). This squashed vector v_j is then used to update the agreement and logits for the next routing iteration. The iterative nature of this process updating b, calculating c via softmax, forming the weighted sum s, and squashing to get v_j, meaning that small quantization errors can be amplified with each routing iteration, potentially leading to incorrect convergence.

- Logit Update (Routing by Agreement): For the next routing iteration, the logits are updated based on the agreement (dot product) between the prediction $\hat{u}_{j|i}$ and the output vector v_j This reinforces connections where capsules agree.

$$b_{ij} \leftarrow b_{ij} + \hat{u}_{j|i} \cdot v_j$$

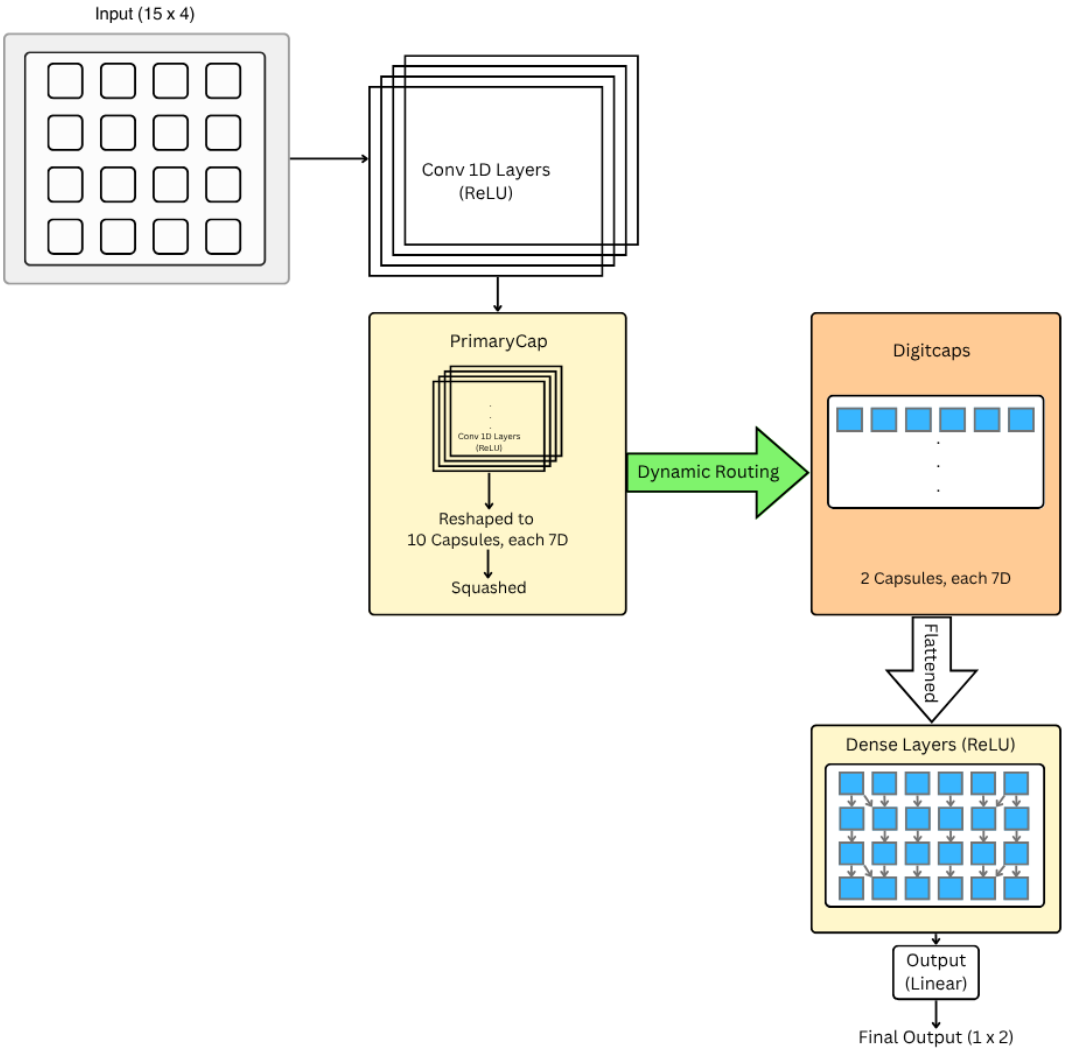


Fig. 4.2. The figure shows the architecture of capsule network

Instrumentation for Calibration: To enable the extraction of internal activation values for quantization calibration, the custom capsule layers in the full-precision model were implemented with a Calibration Flag. When this flag is set to True, the layer's forward pass (call method) is modified to return a dictionary containing the values of all internal activation tensors (e.g., $\hat{u}$, b, c, s) instead of just the final output tensor. This mechanism is the essential hook that allows the calibration block to access and analyze the complete distribution of the model's internal states.

4.2.2. Optimizations, Training and Tuning

The model was trained using the Nadam (Nesterov-accelerated Adaptive Moment Estimation) optimizer. This optimizer is used for its robustness and efficiency. It enhances the widely used Adam optimizer by incorporating Nesterov Accelerated Gradient (NAG), which allows for a more effective "lookahead" step when updating the model's weights. This combination of adaptive learning rates from Adam and improved momentum from Nesterov often leads to faster convergence and a more stable training process, making it a strong choice for complex architectures like the CAPSNET [13]. The advanced optimization and tuning process also included several regularization techniques to combat overfitting and improve generalization.

- L2 Regularization: This technique, also known as weight decay, adds a penalty to the model's loss function that is proportional to the square of the magnitude of the model's weights. This discourages the learning of overly large weight parameters, which reduces model complexity and makes it less sensitive to noise in the training data, thereby improving its ability to generalize [14]. The L2 regularization rate was a key hyperparameter in the search.
- Dropout Regularization Technique: Dropout is a regularization technique where a fraction of neurons is randomly ignored during each training update. This prevents neurons from becoming overly specialized and co-dependent on each other, forcing the network to learn more robust and redundant features [15]. This significantly reduces overfitting and improves the model's ability to generalize from the training data to unseen samples. The dropout rate was a key hyperparameter tuned during the search.
- Self-Knowledge Distillation (Self-KD): To enhance performance, Self-KD was used. This is a training strategy where the model acts as its own teacher. This training strategy, based on the principles of knowledge distillation introduced by Hinton et al. [16]. Typically, an averaged version of the model's own past weights (using an Exponential Moving Average) generates "soft" targets. The model is then trained to match both these soft targets and the ground-truth "hard" labels. This technique acts as a powerful regularizer, forcing the model to produce more consistent predictions, which improves generalization and reduces overfitting, especially with complex architectures like CAPSNETS. For this work, default parameters of alpha=0.7 and ema_decay=0.999 were used.

The search space explored the following parameters:

Parameter	Search Space
Number of Filters	[8, 16, 32]
Kernel Size	[2, 3, 5]
Capsule 1 Dimension	[3, 5, 7]
Number of Capsule 1	[7, 10, 12]
Capsule 2 dimension	[3, 5, 7]
Number of Capsule 2	[2]
Dense 1 and 2 Units	[8, 16, 32]
DropOut Rate	[0.1, 0.2, 0.3, 0.4, 0.5]
L2 Regularization Rate	[1e-5, 1e-4, 1e-3, 1e-2]
Routings	3 (Fixed)
Learning Rate	0.001 (Fixed)

Table 4.1: search space used for training and tuning the model

For hyperparameter tuning, a random search strategy was chosen over a grid search. Given the large, 14,580-combination search space, an exhaustive grid search would be computationally prohibitive. As argued by Bergstra and Bengio (2012), random search is empirically and theoretically more efficient in high-dimensional spaces, having a higher probability of finding optimal or near-optimal parameter sets within a fixed computational budget. The 50 trials represent a practical compromise between exploration of the search space and available computational resources [17].

Training Configuration: 50 trials were randomly sampled and evaluated. Each trial was executed 3 times for a maximum of 800 epochs, with a batch size of 32 and an early stopping patience of 50 on the validation loss (MSE).

4.2.3. Final Full-Precision Model Results and Performance

The hyperparameter search yielded Trial #11 as the best-performing configuration.

- **Model Summary:**

Layer (Type)	Output Shape	Number Of Parameters
Input layer (Input Layer)	(None, 15, 4)	0
Conv1 (Conv1D Layer)	(None, 14, 16)	144
Dropout_1 (Dropout Layer)	(None, 14, 16)	0
Conv2 (Conv1D Layer)	(None, 13, 16)	528
Dropout_2 (Dropout Layer)	(None, 13, 16)	0
Primarycap_Conv1D (Conv1D Layer)	(None, 12, 70)	2,310
Primarycap_Reshape (Reshape Layer)	(None, 120, 7)	0
Primarycap_Squash (Squash Layer)	(None, 120, 7)	0
Digitcaps (DigitCaps Layer)	(None, 2, 7)	11,760
Flatten_Caps (Flatten Layer)	(None, 14)	0
Dense_1 (Dense Layer)	(None, 16)	240
Dropout_3 (Dropout Layer)	(None, 16)	0
Dense_2 (Dense Layer)	(None, 16)	272
Dropout_4 (Dropout Layer)	(None, 16)	0
Output (Dense Layer)	(None, 2)	34

Table 4.2: shows the summary of the best model after training and tuning process

- Full-Precision Model Specifications: The final selected model has 15,288 trainable parameters and a size of 59.72 KB. Its detailed performance is shown in Table 4.4.

- **Best Set of Hyperparameters Found:**

Parameter	Value
Number of Filters	16
Kernel Size	2
Capsule 1 Dimension	7
Number of Capsule 1	10
Capsule 2 dimension	7
Number of Capsule 2	2
Dense 1 and 2 Units	16
DropOut Rate	0.1
L2 Regularization Rate	1e-4

Table 4.3: The table shows the best set of hyperparameter after tunning procedure

- **Performance of the Final Full-Precision Model**

Dataset	Loss (MSE) (m^2)	MAE (m)	RMSE ($^{m}/_{s^2}$)
Training	0.0695	0.1952	0.2636
Validation	0.0943	0.2408	0.3071
Test	0.1066	0.2504	0.3265

Table 4.4: The table shows the performance of trained and tuned model based on mean squared error (MSE), mean absolute error (MAE) and root mean squared error (RMSE) metrics

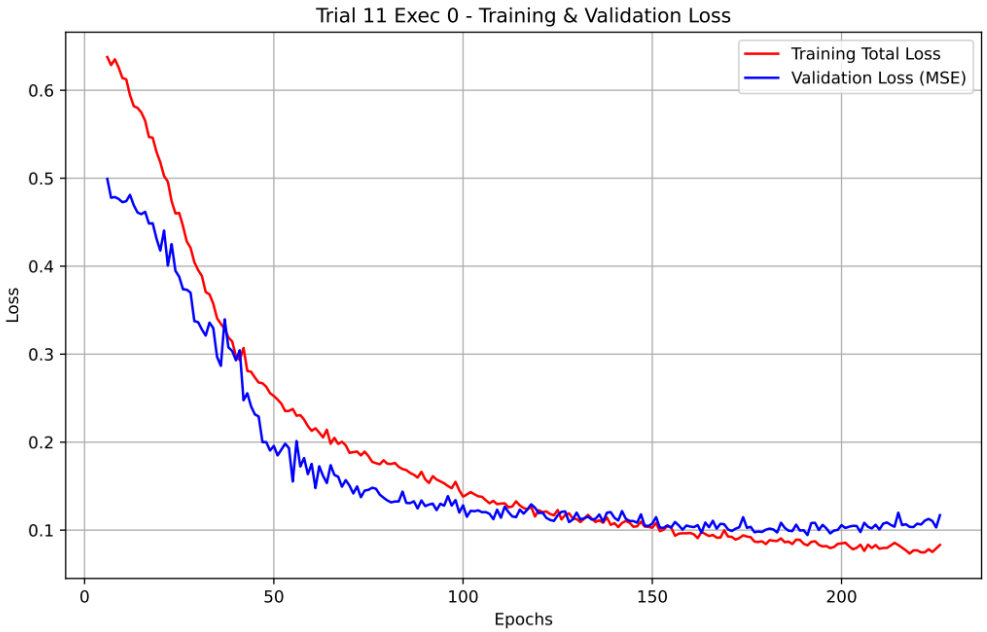


Fig. 4.3. The figure shows the learning curve of the model under training with the best set of hyperparameter found by tuning strategy

- Model Export for Quantization: Following the evaluation, the best-performing model was exported to .keras format. This was done to import the trained model into our quantization framework, where it will undergo a calibration process to extract the dynamic range and determine appropriate clipping thresholds for its weights and activations.

4.3. Quantization Framework Implementation

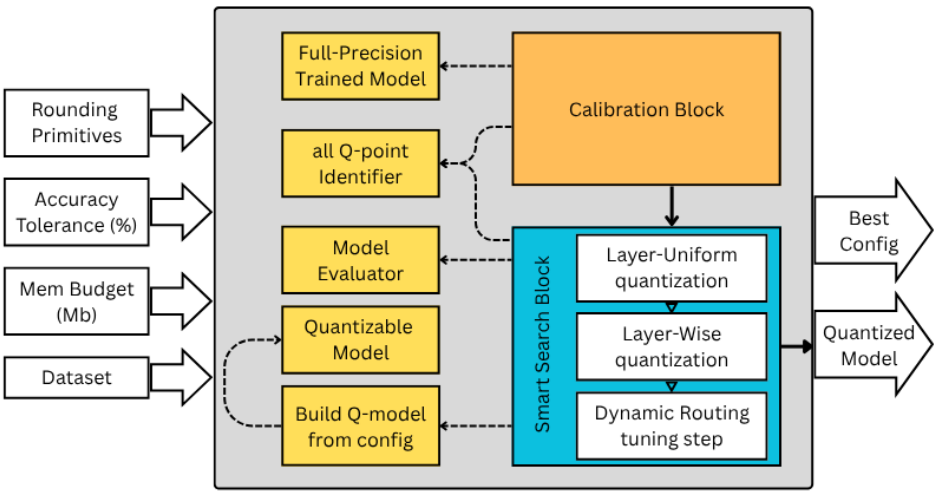


Fig. 4.4. The figure shows a simplified view of quantization framework

4.3.1. Quantizable Model Architecture

The foundation of the quantization framework is the ability to reconstruct the full-precision model into a fully quantizable equivalent. This was achieved through a combination of custom layers and wrappers built upon tf.keras.layers.Layer.

- Custom Layers for Capsule Network: To handle the unique structure of the Capsule Network, its specialized layers were re-implemented as custom keras layers with added hooks for quantization.
  - The PrimaryCaps layer was designed to apply quantization to its internal convolutional operation and to the final squashed output.
  - The squash function itself was implemented as a standalone, modular keras layer. This separation was crucial for isolating this highly sensitive non-linear operation and applying a specific quantization configuration to its output.
  - The DigitCaps layer was the most complex, requiring multiple "quantization points" to be exposed. The implementation allows for quantization to be individually applied to each internal tensor of the dynamic routing algorithm: the input prediction vectors (û), the routing logits (b), the softmax-derived coupling coefficients (c), and the final weighted sum (s) before squashing.
- Wrapper for Standard Layers: For standard layers like dense and Conv1D, a generic QuantizationWrapper was created. This wrapper takes a standard layer as input and applies the specified quantization logic to its output tensor during the forward pass, leaving the original layer's internal operations unchanged.

4.3.2. Quantization Configuration and Primitives

The quantization applied at each point is controlled by a flexible configuration system:

- Quantization Primitives: The core of the simulation is a set of primitive functions that apply fixed-point quantization. These functions take a floating-point tensor and quantize it based on a clipping range and a scale factor, the latter of which is determined by NFB.

- Configuration Object (config): Before any layer is executed in the quantized model, a config object is created for each point of quantization (both layer outputs and internal tensors). This object bundles all necessary information: the clipping range, the NIB/NFB values, and a reference to the specific quantization class (the primitive method) to be used.

- Configuration Flow: Custom layers were designed to accept a dictionary of these config objects during their construction, allowing them to apply the correct

quantization to each internal part. The QuantizationWrapper for standard layers similarly accepts a single config object to quantize the layer's output.

4.3.3. Framework Execution Flow and Calibration

The end-to-end process begins with the trained full-precision model and proceeds through identification and calibration before the search begins.

1. Receive Full-Precision Model: The framework takes the trained model as its primary input.
2. Identify Quantization Points: A block called get_tensor_identifier traverses the model architecture to identify all tensors that are subject to quantization. This includes the outputs (Activations) of all standard and custom layers, as well as the specially exposed internal tensors of the capsule layers and the layer's weights. This creates a master list of quantization points called all_q_points, which serves as a map for the entire framework.
3. Execute Calibration Block: The framework then proceeds to the calibration stage.
    - The calibration block receives the all_q_points list and the full-precision model. It first iterates through the model's layers to extract the clipping ranges for all weights using a min/max method, as this can be done statically.
    - Next, it calibrates the activations. The model is run in inference mode with the calibration flag set to true on a small batch (30) of calibration data. This allows the framework to capture the outputs of all standard layers and the internal activations of the custom layers. Due to its complexity, the DigitCaps layer's activations are handled separately from the standard activations.
    - The collected ranges are organized into a dictionary called calibration_data with three distinct parts: weights, activations, and dr_activations (for the dynamic routing internals). This dictionary is then saved as a JSON file, providing a complete set of initial clipping ranges for the search algorithm.

4.3.4. Calibration Challenges and Refinements

A significant challenge arose during the implementation of the calibration process. The initial approach for determining activation clipping ranges was to use the simple min/max values observed from the small calibration batch. This standard method led to a catastrophic drop in performance, with the quantized model's MSE increasing dramatically.

The first debugging attempt was to replace the min/max strategy with percentile-based clipping, which is typically more robust to outliers. However, this also failed to restore the model's accuracy. Further investigation revealed that the internal states of the DigitCaps

layer were uniquely sensitive. Specifically, the dynamic routing logits, b, had an extremely narrow activation range, on the order of -0.00079 to 0.0015. Manually forcing a wider clipping range for these specific tensors yielded some improvement, but the MSE was still far from the target.

The root cause was ultimately identified which was due to small batch size used for calibration, so it was insufficient to capture the true dynamic range of activations across the entire dataset. While most values within the dynamic routing algorithm are between 0 and 1, occasional "spikes" in activation values occurred that the small sample batch did not contain. These spikes, when clipped, caused significant quantization error. The final, effective solution was to manually analyze the full range and set a much wider clipping range of -3.6 to 3.6 for these sensitive tensors, which successfully accounted for the spikes and restored model performance perfectly. A possible automated solution for future work would be to calibrate using a significantly larger number of batches from the calibration dataset to ensure these rare but critical activation values are observed.

4.3.5. Significant Challenge in Extraction of DigitCaps Intermediate Activations

A persistent challenge during extraction of the digitcaps internal activations for calibration process was due to Keras TensorFlow workflow limitation. The issue stems from how Keras TensorFlow build and manage the computation graph of a model. When a model is defined using Keras functional API, it creates a symbolic graph where each layer has predefined inputs and outputs so this graph enforces shape and type consistency, as well as deterministic forward propagation therefore even if the layer was forced by calibration flag to return a dictionary of all internal activations of layer, the TensorFlow forced the layer to return only the standard output and ignored the flag in the full model context.

To fix the issue, a separate probe model created using same layer and inputs to extract the intermediate activations with expected shapes and output information.

4.3.6. Smart Search Execution Details

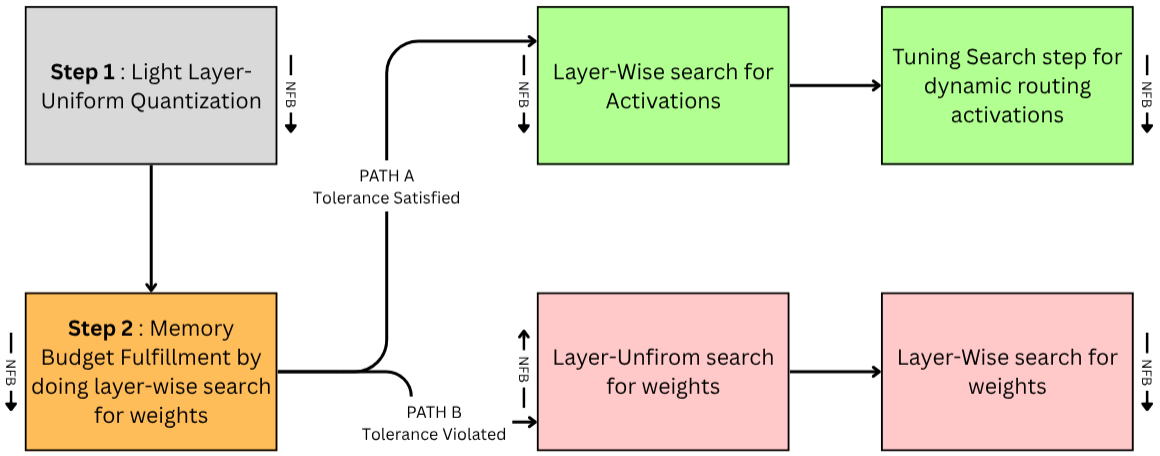


Fig. 4.5. The figure shows a simplified flow diagram of search algorithm

Once calibration is completed, the search block uses the generated calibration data to find the optimal bit-widths. The search loop is facilitated by a set of helper blocks:

- Model Builder: Constructs a quantized model instance based on a given bit-width configuration.
- Size Calculator: Computes the memory footprint of the constructed model.
- Evaluator: Measures the performance (MSE) of the model on the validation set.

The smart search block uses these helpers to iteratively test new configurations against the constraints defined in the search configuration, following a detailed, multi-step strategy. The primary search parameters that guide this process are the user-defined values for INITIAL_SEARCH_NFB (initial number of fractional bits), MIN_SEARCH_NFB (minimum number of fractional bits), loss tolerance, and memory budget.

1. Step 1 - Initial Uniform Search: The process begins with a "uniform light search" Starting with all layers set to the INITIAL_SEARCH_NFB, the algorithm uniformly reduces the NFB for all layers simultaneously. This initial, coarse-grained step quickly finds a baseline NFB configuration that utilizes only a small fraction (e.g., 5%) of the allowed performance degradation specified by the loss tolerance. This provides an aggressive but high-performance starting point for the next step.
2. Step 2 - Memory Budget Satisfaction (Layer-Wise Search on Weights): The next goal is to meet the specified memory budget. Starting from the NFB configuration

found in Step 1, the algorithm performs a targeted, layer-wise search for the weights. It iteratively reduces the NFB for the weights of individual layers, likely prioritizing larger or less sensitive layers until the model's total size is less than or equal to the memory budget. When final refinement and tolerance satisfaction after the memory budget is met, the algorithm evaluates the model's current performance against the total loss tolerance. The strategy then diverges into one of two paths:

- Path A (Tolerance is already satisfied): If the model's performance is still within the acceptable loss tolerance after Step 2, the algorithm proceeds to a layer-wise search for the activations. It attempts to further reduce the NFB for individual activation layers, seeking additional optimization opportunities while ensuring the performance does not drop below the tolerance threshold.
- Path B (Tolerance is violated): If meeting the memory budget in Step 2 caused the performance to drop below the acceptable tolerance, a recovery phase is initiated. The algorithm first performs a uniform search by incrementally increasing the NFB across layers until the loss tolerance is satisfied again. Once performance is recovered, it may proceed with a more cautious layer-wise search to refine the configuration, finding a final balance that respects both the performance and memory constraints.

At the conclusion of this process, the search block outputs the best set of NFB and NIB values for each layer's weights, its output activations, and the internal activations of the custom capsule layers. This configuration represents the optimal balance found between model compression and predictive accuracy.

4.3.7. Overfitting Problem Due to Aggressive Quantization

Another problem appears when we try to apply aggressive quantization on model, for example if we set MIN_SEARCH_NFB=2 it may satisfy the tolerance and memory budget but due to overfitting on our dataset (test set) that the quantized model is evaluated on it in search algorithm, it may work not really accurate on other unseen datasets, practically configuration with MIN_SEARCH_NFB =2 was set and it could force even NFB for DigitCaps internal activations to more aggressive NFB=2,3 and 4 while satisfying loss tolerance but when the quantized model was evaluated on training and validation set, loss drop was more than what we expected and it seems that’s because of high sensitivity of capsule network on numerical values.

4.3.8. Smart Search Configuration

The custom quantization framework was executed on the best-performing full-precision model with the following configuration:

- Performance Constraint:

    - loss tolerance = 2%. The search was configured to find a quantization scheme where the validation MSE would not degrade by more than 2%.
- Size Constraint:
    - memory budget = 0.01 MB. The target size for the final quantized model was set to 10 KB.
- Search Space for Fractional Bits:
    - INITIAL_SEARCH_NFB = 16. The starting precision for the search.
    - MIN_SEARCH_NFB = 6. The lowest precision allowed during the search.

4.4. High Level Synthesis (HLS) Implementation and Verification

As a final step, two key sets of data were extracted from the optimized quantized model to prepare for hardware implementation:

- The final quantized weights for all layers were exported directly into a C++ header file (.h).
- The output activations of each layer were captured for the first sample of the test dataset. These will serve as "golden vectors" for debugging and verification during the HLS process.

The HLS implementation is designed to be highly modular and configurable.

- Configuration Header File: A central C++ header file defines all static parameters, including layer shapes and, most importantly, the specific ap_fixed data types for all weights and activations, hard-coding the optimal NIB and NFB values.
- Fixed-Point Data Types: The entire design is based on the ap_fixed arbitrary precision fixed-point data type to precisely match the mixed-precision scheme.
- Uniform Accumulator Type: To prevent overflow, a single, robust accumulator type, ap_fixed<32, 16, AP_RND, AP_SAT>, was used for all layers.
- Modular Functional Structure: Each neural network layer (e.g., Conv1D, DigitCaps) is implemented as a separate C++ function.
- Top-Level Function: A top-level function serves as the main entry point, calling all layer functions in sequence to define the dataflow for the entire accelerator.

4.4.1. A Notable Implementation Challenge

A notable implementation challenge involved the non-linear functions required by the capsule layers, namely the exponent function for the softmax operation and the square root function for the squash operation. The initial approach was to implement these using resource-efficient Look-Up Tables (LUTs), which were pre-computed in Python and

exported as C++ header files. However, this LUT-based approximation introduced numerical errors that resulted in a tangible degradation of the model's MSE during HLS simulation. Consequently, a design decision was made to prioritize accuracy over resource optimization. The final implementation uses the standard, more accurate exponent (hls::exp) and square root (hls::sqrt) functions from the HLS math library, ensuring bit-accuracy with the software model.

4.4.2. High Level Synthesis (HLS) Optimization Strategies

Several optimization strategies were employed to ensure numerical stability and an efficient hardware implementation.

- Fixed-Point Mode Configuration (ap_rnd and ap_sat): The ap_fixed data types were configured with specific rounding (ap_rnd) and saturation (ap_sat) modes. Selecting a specific rounding mode ensures predictable quantization behavior that matches the software simulation. Using saturation mode (ap_sat) prevents catastrophic "wrap-around" errors from overflowing by clamping the result at the maximum/minimum representable value.
- Loop Pipelining and Unrolling (PIPELINE and UNROLL): #pragma HLS PIPELINE was applied to enable instruction-level parallelism, and #pragma HLS UNROLL was used to create multiple parallel hardware units for concurrent execution of loop iterations.
- Memory Access Optimization (ARRAY_PARTITION): #pragma HLS ARRAY_PARTITION was applied to the weight arrays, partitioning them into smaller blocks so the hardware can read multiple values simultaneously in a single clock cycle.
- Top-Level Architecture Optimization (DATAFLOW and M_AXI): #pragma HLS INTERFACE m_axi was used to create a standard AXI4-Master bus interface for efficient data transfer with external memory. #pragma HLS DATAFLOW was applied to enable task-level parallelism, treating the layer functions as concurrent processes in a deep hardware pipeline.

4.4.3. Layer-wise Verification Strategy

To manage complexity and facilitate debugging, a layer-by-layer verification strategy was employed. Instead of implementing and verifying the entire model in the beginning, each neural network layer was implemented and tested as a discrete module. The C++ testbench was designed to support this approach. For each layer, the testbench would:

1. Read the corresponding "golden" input vector (i.e., the output of the previous layer, saved from the quantized software model related to first sample).
2. Execute the HLS function for only the current layer under verification.
3. Read the "golden" output vector for that specific layer.

4. Perform a bit-accurate comparison between the HLS layer's output and its golden vector to ensure a perfect match.

Only after a layer was individually verified it was integrated into the top-level design. This incremental process, comparing intermediate "golden vectors" at each stage, was critical for quickly isolating and fixing numerical mismatches, ensuring that the final, fully-formed model was correct by construction [18].

5. CHAPTER 5: Results and Analysis

5.1. Final Quantization Configuration

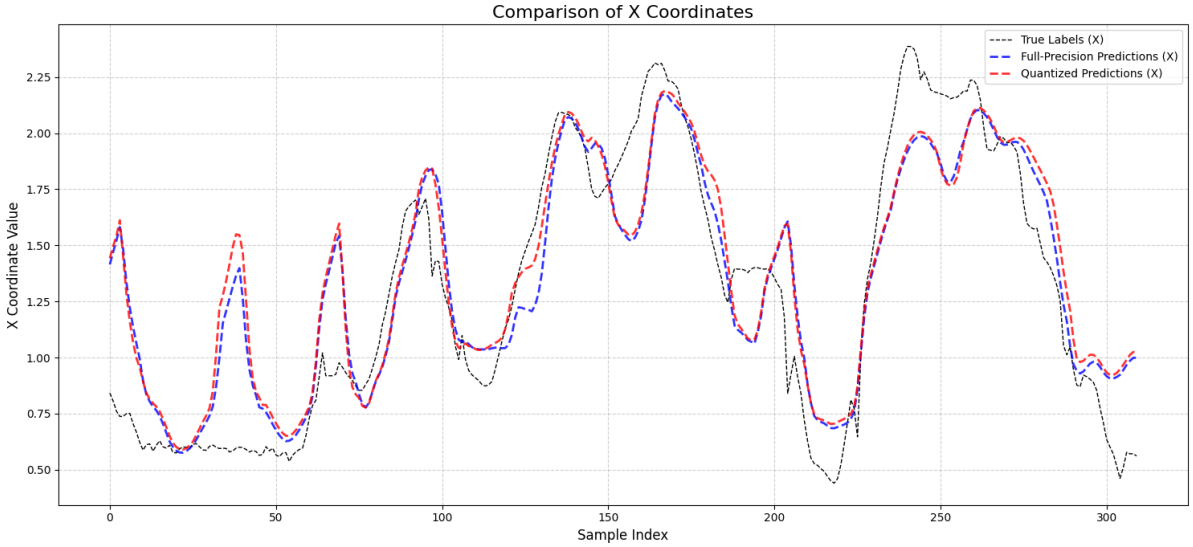
Layer / Tensor Group	Type	NIB	NFB	Total Bits
Convolutional & Dense Weights				
conv1, conv2 kernels & biases	Weight	1	6-7	8-9 bits
primarycap_conv1d kernel & bias	Weight	1-2	6	8-9 bits
dense_1, dense_2 kernels & biases	Weight	1-2	6-7	8-10 bits
Capsule Layer Weights				
digitcaps/W (Routing Weights)	Weight	1	6	8 bits
All Activations				
Layer Outputs (conv_output, etc.)	Activation	1-3	16	18-20 bits
primarycap_squash_output	Activation	1	16	18 bits
digitcaps Internal Activations (u^,b,c,s,v)	Activation	3	16	20 bits

Table 5.1: This table shows the final set of number of integer and fractional bits (NIB and NFB) found by framework for weights and activations of the quantized model

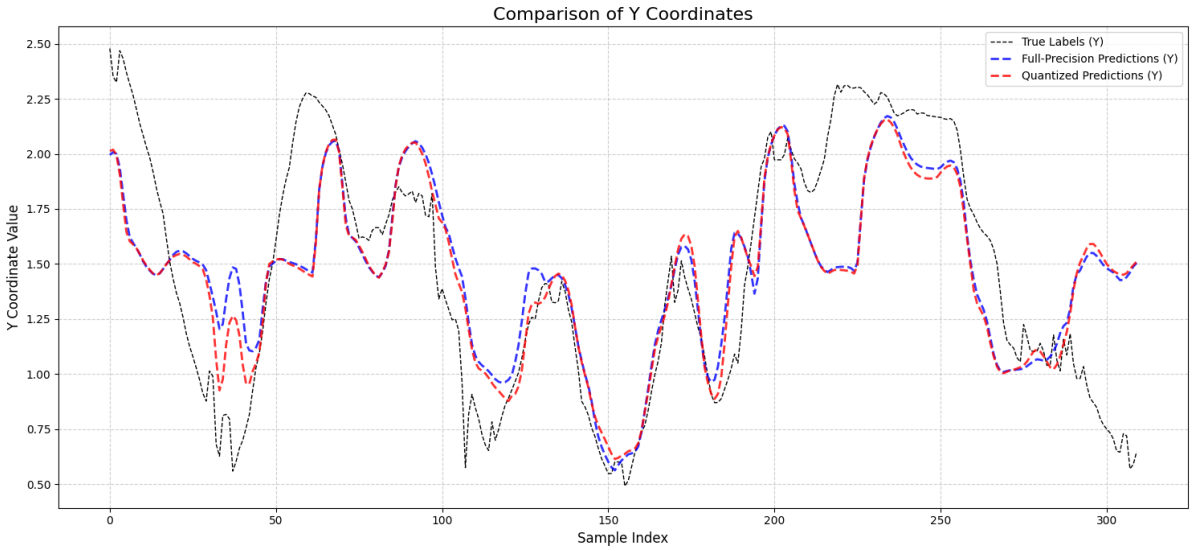
5.2. Performance and Size Comparison

Metric	Original Full-Precision Model	Final Quantized Model	Change
Model Size	59.72 KB	**13.1 KB**	**4.56x Compression**
Test Loss (MSE)	0.1066	0.1071	**+0.47% Degradation**
Validation Loss (MSE)	0.0943	0.0960	+1.8% Degradation
Train Loss (MSE)	0.0695	0.0651	-6.3% Improvement

Table 5.2: shows the comparison of performance and size between original full-precision model and final quantized model

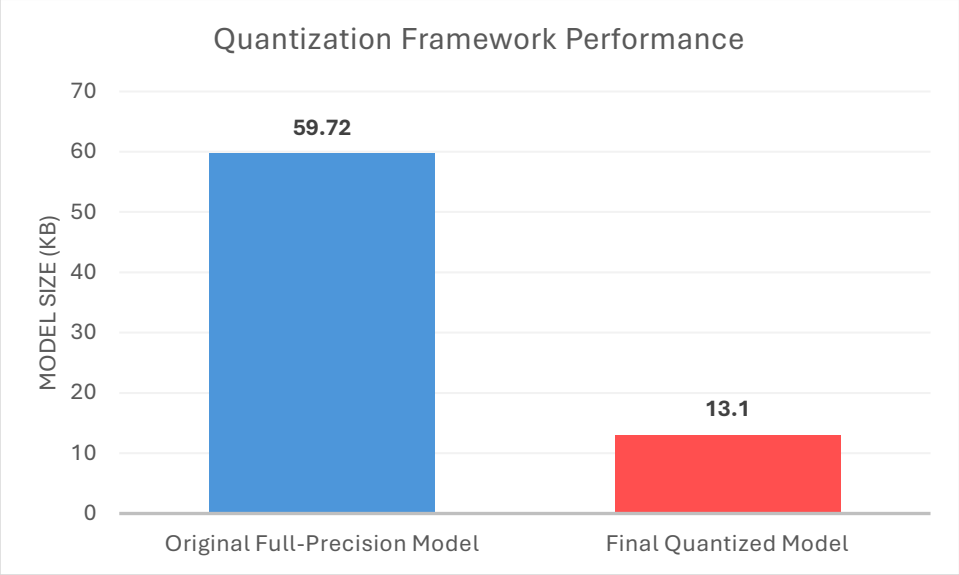


(a)

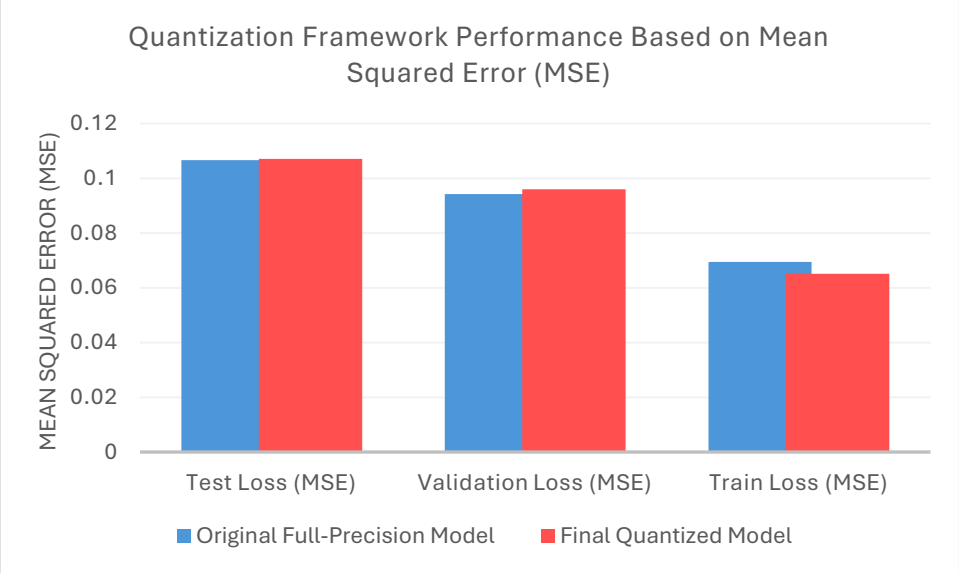


(b)

Fig. 5.1. The figure shows true labels, full-precision model predictions and quantized model predictions on test dataset for (a) first output feature which is X coordinate, (b) second output feature which is Y coordinate



(a)



(b)

Fig. 5.2. Comparison of the original full-precision and final quantized models. (a) Model size reduction. (b) Training, validation and test set MSE degradation.

5.3. Analysis of Quantization Results

The final results in the "Performance and Size Comparison" table clearly illustrate the trade-offs managed by the quantization framework. The smart search algorithm successfully followed its programmed logic to navigate the constraints. Specifically, after step 2 of the search satisfied the aggressive 10 KB memory budget, the resulting model's validation MSE degradation exceeded the 2% tolerance.

Consequently, the algorithm correctly initiated path B, the recovery phase. It began to uniformly increase the precision (NFB) of the layers until the performance was brought back within the 2% tolerance threshold. This resulted in a final model size of 13.1 KB, slightly over the initial budget. This outcome demonstrates the framework's primary design principle: prioritizing the user-defined performance constraint over the memory budget when the two are in conflict. The final model represents the smallest possible size that still meets the critical accuracy requirements.

It can be seen the performance of framework is quite satisfying and after compressing the size of model by factor 4.56x, the increase in loss (MSE) on test dataset is almost negligible (<0.5%).

5.4. Sensitivity to Number of Fractional and Integer Bits

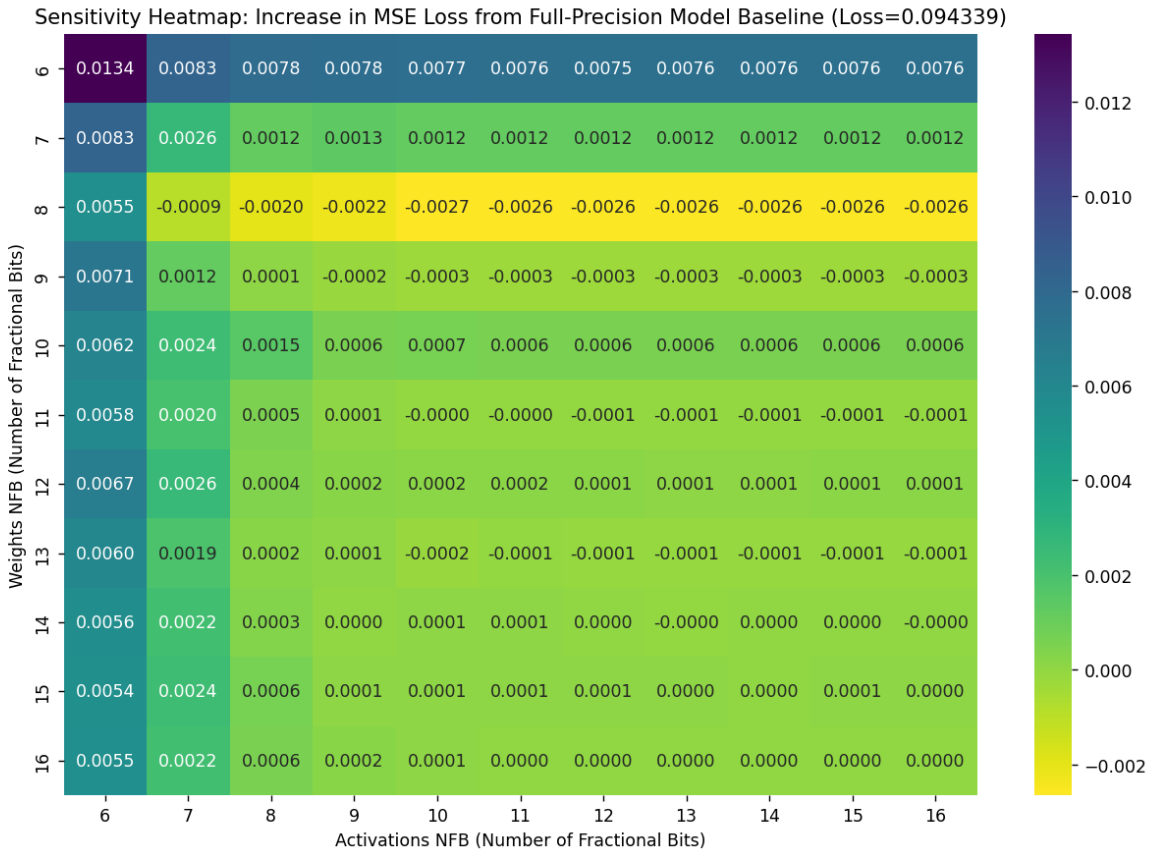


Fig. 5.3. The Heatmap shows the sensitivity of quantized model to number of fractional bits (NFB) of weights and layers output (Activations) based on mean squared error (MSE) metric with respect to full-precision model performance baseline

It's easy to see how the performance of the CapsNet based on MSE degrades when the framework try to force the model to decrease the NFB of weights and activations, but also we can see some unexpected behavior in some points that even if the NFB is fewer, the model works better even with respect to the full-precision model and the probable reason

behind is that sometimes the effects of clipping the weights and activations work like a regularization for the quantized model.

5.5. High Level Synthesis (HLS) Performance Verification

A critical success metric for this work was ensuring that the hardware implementation did not introduce any numerical degradation compared to the optimized software model. The HLS C/RTL co-simulation confirmed this. The MSE of the final HLS model on the test, validation, and train sets was virtually identical to the quantized software model. In fact, the HLS model showed almost the same in MSE on all datasets (e.g., a test set MSE of 0.107 in HLS versus 0.107 in software), with minor differences attributable to rounding nuances in the C++ fixed-point libraries. Also, figure 5.1 clearly shows that the predictions on whole test dataset in HLS is almost same as Python, This confirms that the transition from the Python-based quantization framework to the C++ HLS implementation was successful, achieving a truly bit-accurate hardware representation with almost no loss of performance.

6. CHAPTER 6: Discussion, Conclusion, and Future Work

6.1. Discussion of Results and Generalizability

The results successfully demonstrate that a CAPSNET can be effectively trained for the localization task and, critically, can be aggressively optimized for edge deployment with negligible accuracy degradation. The final bit-accurate HLS model, with a test MSE of 0.107, confirms the viability of the entire end-to-end workflow, from abstract software concept to verified hardware design.

However, a discussion on generalizability is crucial when considering the transition from this successful proof-of-concept to a robust, real-world application. Such a transition introduces challenges beyond the scope of this initial implementation. For instance, environmental variances such as changes in ambient humidity, temperature, or the introduction of new furniture can alter the baseline capacitance of the sensors. A production-grade system would need to incorporate adaptive algorithms or periodic recalibration routines to remain accurate under such changing conditions.

Furthermore, the current model architecture is designed to interpret the signals corresponding to a single entity. Scaling this system to handle more complex and dynamic scenarios, particularly those involving multiple people, would demand significant architectural modifications. This would likely require incorporating novel mechanisms capable of differentiating, isolating, and tracking multiple simultaneous signal sources, representing a substantial leap in model and system complexity.

6.2. Conclusion

This report details the successful development, optimization, and hardware implementation of a CAPSNET for time series prediction. The primary objective was to design an effective model for a complex task and subsequently make it efficient enough for practical deployment on resource-constrained edge devices, a critical challenge for resource-intensive architectures like CAPSNETS.

The key achievements are:

- Successful Model Development: A CAPSNET architecture was successfully implemented and trained for time series data. Through a rigorous hyperparameter search involving 50 trials, an optimal floating-point model was developed, achieving a baseline test MSE of 0.106.

- Advanced Software Optimization: A custom, mixed-precision quantization framework was created to compress the model. The framework's smart search strategy successfully navigated the trade-off between model size and accuracy, reducing the model's memory footprint by ~4.6x (from 59.7 KB to 13.1 KB). This

was achieved with a negligible performance degradation of less than 0.5% on the test set, staying well within the predefined 2% loss tolerance (>0.5% on test set).

- Bit-Accurate Hardware Implementation: The optimized software model was successfully translated into a hardware accelerator using HLS. The HLS design was architected for performance using pipelining, parallelism, and standard memory interfaces. C/RTL co-simulation verified that the final hardware implementation is bit-accurate with the quantized software model, confirming its correctness.

In summary, this work presents a complete and successful workflow from algorithm conception to hardware verification. By achieving significant model compression with minimal loss in accuracy, we have demonstrated that it is feasible to deploy complex and sensitive models like CapsNets on edge devices, paving the way for more intelligent applications in low-power environments.

6.3. Future Work

Building on the successful results of this thesis, several promising directions for future work can be identified:

- Exploration of Quantization-Aware Training (QAT): While the PTQ framework yielded excellent results, employing QAT could potentially allow for even more aggressive compression. By simulating quantization effects during the training process, the model can learn to be more robust to lower bit-widths, possibly allowing for a reduction in the precision of activations without violating the loss tolerance.

- Hardware Resource and Power Analysis: The current work focused on functional verification through HLS. A crucial next step would be to perform a full synthesis and implementation on a target FPGA device. This would allow for a detailed analysis of resource utilization (LUTs, DSPs, BRAM), clock frequency, latency, and, most importantly, power consumption, providing concrete data on the real-world efficiency of the accelerator.

- Architectural Enhancements to the Capsule Network: Further research could explore modifications to the CapsNet architecture itself. This might include investigating different routing algorithms, exploring attention mechanisms within the capsule layers, or designing more efficient capsule structures specifically for time-series data to potentially improve both accuracy and computational efficiency.

7. List of Symbols

Capsule Network Architecture

- u_i - The output vector from a primary capsule i in the lower layer (layer L).
- W_{ij} - The trainable weight matrix used to transform the output of a primary capsule i into a prediction for a digit capsule j.
- $\hat{u}_{j|i}$ - The "prediction vector" or "vote" that capsule i sends to capsule j, calculated as $\hat{u}_{j|i} = W_{ij}u_i$.
- b_{ij} - The temporary logit representing the log prior probability that capsule i should be coupled with capsule j. These are updated iteratively during the dynamic routing process.
- c_{ij} - The coupling coefficient that determines the strength of the connection between capsule i and capsule j. It is calculated by applying the softmax function to the logits b_{ij}.
- s_j - The total input vector to a digit capsule j, calculated as a weighted sum of all prediction vectors it receives from the layer below, according to the formula $s_j = \sum_i c_{ij}\hat{u}_{j|i}$.
- v_j - The final output vector of a digit capsule j, which is produced by applying the non-linear "squash" activation function to the input vector s_j.
- routings - A hyperparameter that defines the number of iterations for the dynamic routing algorithm which is fixed to a value of 3.

Quantization Framework

- PTQ - Post-Training Quantization; an optimization method where a pre-trained model is quantized without retraining.
- QAT - Quantization-Aware Training; a method that simulates quantization noise during the training loop, making the model more robust to low-precision arithmetic.
- NIB - Number of Integer Bits; used to define the integer part of a fixed-point number.
- NFB - Number of Fractional Bits; used to define the fractional part of a fixed-point number and determine its precision.

High-Level Synthesis (HLS)

- ap_fixed - The arbitrary precision fixed-point data type from the HLS library used to precisely match the mixed-precision scheme. The accumulator type was defined as ap_fixed<32, 16>.
- ap_rnd - The rounding mode specified for the ap_fixed data type to ensure predictable quantization behavior.

- ap_sat - The saturation mode specified for the ap_fixed data type to prevent overflow errors by clamping values at their representable maximum or minimum.

Regularization and Training

- Self-KD - Self-Knowledge Distillation; a training strategy where the model acts as its own teacher to improve generalization and reduce overfitting.
- alpha - A parameter for the Self-KD process, set to a default of 0.7.
- ema_decay - The Exponential Moving Average decay parameter for the Self-KD process, set to a default of 0.999.

8. REFERENCES

[1]. Ye, Q., Fan, X., Fang, G., et al. (2020). CapsLoc: A Robust Indoor Localization System with WiFi Fingerprinting Using Capsule Networks. IEEE International Conference on Communications (ICC).

[2]. H. S. Obeidat, W. Shuaieb, E. A. Al-qudah, and O. Obeidat, "A Review of Indoor Localization Techniques and Wireless Technologies," *Wireless Personal Communications*, vol. 119, pp. 289-327, 2021.

[3]. Subbicini, G., Lavagno, L., & Lazarescu, M. T. (2023). Enhanced Exploration of Neural Network Models for Indoor Human Monitoring. 9th International Workshop on Advances in Sensors and Interfaces (IWASI).

[4]. Xi, E., Bing, S., & Jin, Y. (2017). Capsule Network Performance on Complex Data. arXiv preprint arXiv:1712.03480.

[5]. G. E. Hinton, A. Krizhevsky, and S. D. Wang, "Transforming auto-encoders," in *International Conference on Artificial Neural Networks (ICANN)*, Berlin, Heidelberg, 2011, pp. 44-51.

[6]. Sabour, S., Frosst, N., & Hinton, G. E. (2017). Dynamic Routing Between Capsules. Advances in Neural Information Processing Systems (NIPS).

[7]. Jacob, B., Kligys, S., Chen, B., et al. (2018). Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference. Conference on Computer Vision and Pattern Recognition (CVPR).

[8]. Krishnamoorthi, R. (2018). Quantizing deep convolutional networks for efficient inference: A whitepaper. arXiv preprint arXiv:1806.08342.

[9]. Nane, R., Sima, V. M., & Bertels, K. (2016). A Survey and Evaluation of FPGA High-Level Synthesis Tools. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems.

[10]. 1Marchisio, A., Bussolino, B., Colucci, A., et al. (2020). Q-CapsNets: A Specialized Framework for Quantizing Capsule Networks. IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP).

[11]. Nagel, M., Fournarakis, M., Amjad, R. A., et al. (2021). A White Paper on Neural Network Quantization. arXiv preprint arXiv:2106.08295.

[12]. Tariq, O. B., Lazarescu, M. T., & Lavagno, L. (2020). Neural Networks for Indoor Human Activity Reconstructions. IEEE Sensors Journal.

[13]. T. Dozat, "Incorporating Nesterov Momentum into Adam," in *Proceedings of the 4th International Conference on Learning Representations (ICLR) Workshop*, 2016.

[14]. Krogh and J. A. Hertz, "A simple weight decay can improve generalization," in *Advances in Neural Information Processing Systems (NIPS)*, vol. 4, 1991, pp. 950-957.

[15]. N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: a simple way to prevent neural networks from overfitting," *The Journal of Machine Learning Research*, vol. 15, no. 1, pp. 1929-1958, 2014.

[16]. Y. Zhang, T. Xiang, T. M. Hospedales, and H. Lu, "Be Your Own Teacher: Improve the Performance of Convolutional Neural Networks via Self Distillation," in

Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV), 2019, pp. 3713-3722.Relevance:

[17]. J. Bergstra and Y. Bengio, "Random search for hyper-parameter optimization," Journal of Machine Learning Research, vol. 13, pp. 281-305, Feb. 2012.

[18]. Gupta, S., Akesh, B., & Mohanty, S. P. (2022). For-HLS: A Formal-Methods-Inspired High-Level Synthesis Methodology for Robust AI-Chip Synthesis. IEEE Transactions on Circuits and Systems I: Regular Papers.

9. ACKNOWLEDGEMENTS

I would like to deeply appreciate Professor Mihai Teodor Lazarescu for his invaluable guidance, support, and encouragement throughout this path. His expertise and insightful feedback have been fundamental to the development and successful completion of this work.

I am also sincerely thankful to my parents for their constant love, patience, and support, which have been a source of strength and motivation throughout my academic journey.

10. APPENDIX. A Related Codes:

10.1. Python Codes:

10.1.1. CAPSNET Custom Layers:

```
import tensorflow as tf
from tensorflow.keras import layers, initializers
from tensorflow.keras import backend as K

class OriginalLength(layers.Layer): # Renamed
    def __init__(self, name=None, **kwargs):
        super(OriginalLength, self).__init__(name=name or "original_length", **kwargs)
    def call(self, inputs, **kwargs):
        if K.ndim(inputs) > 3:
            return K.sqrt(K.sum(K.square(inputs), -2, keepdims=False) + K.epsilon())
        else:
            return K.sqrt(K.sum(K.square(inputs), -1, keepdims=False) + K.epsilon())
    def compute_output_shape(self, input_shape):
        if len(input_shape) > 3: return input_shape[:-2] + input_shape[-1:]
        elif len(input_shape) == 3 : return input_shape[:-1]
        return input_shape[:-1]
    def get_config(self):
        config = super(OriginalLength, self).get_config()
        return config

class OriginalMask(layers.Layer): # Renamed
    def __init__(self, name=None, **kwargs):
        super(OriginalMask, self).__init__(name=name or "original_mask", **kwargs)
    def call(self, inputs, **kwargs):
        if type(inputs) is list:
            assert len(inputs) == 2
            input_tensor, mask_labels = inputs # Corrected variable name
        else:
            input_tensor = inputs # Corrected variable name
            if K.ndim(input_tensor) != 3:
                raise ValueError(f"Input to Mask layer must have rank 3 (None, num_capsule,
dim_vector), got rank {K.ndim(input_tensor)}")
            x = K.sqrt(K.sum(K.square(input_tensor), -1) + K.epsilon())
            mask_labels = K.one_hot(indices=K.argmax(x, 1), num_classes=tf.shape(x)[1]) #
mask_labels is the one-hot mask

        mask_expanded = K.expand_dims(mask_labels, -1) # Use mask_labels for expand
        # Use input_tensor for multiply, which is the first element if inputs is a list
        current_input_tensor = input_tensor if not type(inputs) is list else inputs[0]
        masked = tf.multiply(current_input_tensor, tf.cast(mask_expanded,
dtype=current_input_tensor.dtype))
        return K.batch_flatten(masked)

    def compute_output_shape(self, input_shape):
        tensor_shape = input_shape[0] if isinstance(input_shape, list) else input_shape
        if len(tensor_shape) != 3:
             if tensor_shape[1] is not None and tensor_shape[2] is not None:
                return tuple([tensor_shape[0], tensor_shape[1] * tensor_shape[2]])
             return tuple([tensor_shape[0], None])
        return tuple([tensor_shape[0], tensor_shape[1] * tensor_shape[2]])
    def get_config(self):
        config = super(OriginalMask, self).get_config()
        return config

#--------------------------------------------------------------------

def squash_function(vectors, axis=-1): # This helper function name can remain the same
    s_squared_norm = K.sum(K.square(vectors), axis, keepdims=True)
    scale = s_squared_norm / (1 + s_squared_norm + K.epsilon()) / (K.sqrt(s_squared_norm +
K.epsilon()) + K.epsilon())
    return scale * vectors

class OriginalSquashActivation(layers.Layer): # Renamed
```

```
    def __init__(self, axis=-1, name=None, **kwargs):
        super(OriginalSquashActivation, self).__init__(name=name or
"original_squash_activation", **kwargs)
        self.axis = axis

    def call(self, inputs):
        return squash_function(inputs, axis=self.axis) # Uses the helper

    def compute_output_shape(self, input_shape):
        return input_shape

    def get_config(self):
        config = super(OriginalSquashActivation, self).get_config()
        config.update({'axis': self.axis})
        return config

##-------------------------------------------------------------------

# Renamed PrimaryCap function to OriginalPrimaryCapFunction for clarity
def OriginalPrimaryCapFunction(inputs, dim_capsule, n_channels, kernel_size, strides,
padding):
    total_filters = dim_capsule * n_channels
    output = layers.Conv1D(filters=total_filters, kernel_size=kernel_size, strides=strides,
padding=padding,
                           activation='relu', name='primarycap_conv1d')(inputs)
    if output.shape[1] is None or output.shape[1] == 0:
        raise ValueError(f"PrimaryCap Conv1D resulted in zero output steps...")
    num_output_steps = output.shape[1]
    target_shape = (num_output_steps * n_channels, dim_capsule)
    outputs = layers.Reshape(target_shape=target_shape, name='primarycap_reshape')(output)
    # PrimaryCap should use the OriginalSquashActivation
    outputs = OriginalSquashActivation(axis=-1, name='primarycap_squash')(outputs)
    return outputs

#-------------------------------------------------------------------
class OriginalCapsuleLayer(layers.Layer):
    def __init__(self, num_capsule, dim_capsule, num_routing=3,
                 kernel_initializer='glorot_uniform',
                 name=None,
                 **kwargs):
        super(OriginalCapsuleLayer, self).__init__(name=name or "original_capsule_layer",
**kwargs)
        self.num_capsule = num_capsule
        self.dim_capsule = dim_capsule
        self.num_routing = num_routing
        self.kernel_initializer = initializers.get(kernel_initializer)
        self.internal_squash_layer = OriginalSquashActivation(axis=-2,
name=f"{self.name}_internal_squash_op")

    def build(self, input_shape):
        input_shape = tf.TensorShape(input_shape)
        assert len(input_shape) == 3, "Input Tensor should have shape=[None,
input_num_capsule, input_dim_capsule]"
        self.input_num_capsule = input_shape[1]
        self.input_dim_capsule = input_shape[2]
        self.W = self.add_weight(shape=(1, self.input_num_capsule, self.num_capsule,
                                        self.dim_capsule, self.input_dim_capsule),
                                 initializer=self.kernel_initializer, name='W',
trainable=True)
        self.built = True

    def call(self, inputs, training=None, return_internals_for_calibration=False): #
Respect the argument
        # Optional: Keep one print to see when this specific instance's call is traced and
with which flag
        # print(f"OCL_CALL ({self.name}): `return_internals_for_calibration` =
{return_internals_for_calibration}")

        batch_size = tf.shape(inputs)[0]
        inputs_expand = K.expand_dims(K.expand_dims(inputs, 2), -1)
```

```
120.         inputs_tiled = K.tile(inputs_expand, [1, 1, self.num_capsule, 1, 1])
121.         W_tiled = tf.tile(self.W, [batch_size, 1, 1, 1, 1])
122.         inputs_hat = tf.matmul(W_tiled, inputs_tiled,
name=f"{self.name}_inputs_hat_matmul")
123.         inputs_hat_stopped = K.stop_gradient(inputs_hat)
124.         b = tf.zeros(shape=[batch_size, self.input_num_capsule, self.num_capsule, 1, 1],
dtype=inputs.dtype)
125.         assert self.num_routing > 0, 'The num_routing should be > 0.'
126.
127.         inputs_hat_for_calib_final = inputs_hat
128.         b_for_calib_final = b
129.         c_for_calib_final = None
130.         weighted_sum_s_for_calib_final = None
131.         squashed_v_loop_for_calib_final = None
132.         final_outputs_vj_primary = None
133.
134.         for i in range(self.num_routing):
135.             c = tf.nn.softmax(b, axis=2, name=f"{self.name}_routing_coeffs_c_iter{i}")
136.             current_inputs_hat_for_sum = inputs_hat if i == self.num_routing - 1 else
inputs_hat_stopped
137.             weighted_predictions = tf.multiply(c, current_inputs_hat_for_sum,
name=f"{self.name}_weighted_preds_iter{i}")
138.             weighted_sum_s = tf.reduce_sum(weighted_predictions, axis=1, keepdims=True,
name=f"{self.name}_weighted_sum_s_iter{i}")
139.             squashed_v_loop = self.internal_squash_layer(weighted_sum_s)
140.             final_outputs_vj_primary = squashed_v_loop
141.
142.             if i == self.num_routing - 1:
143.                 inputs_hat_for_calib_final = current_inputs_hat_for_sum
144.                 b_for_calib_final = b
145.                 c_for_calib_final = c
146.                 weighted_sum_s_for_calib_final = weighted_sum_s
147.                 squashed_v_loop_for_calib_final = squashed_v_loop
148.
149.             if i < self.num_routing - 1:
150.                 outputs_vj_tiled = tf.tile(squashed_v_loop, [1, self.input_num_capsule, 1,
1, 1], name=f"{self.name}_outputs_vj_tiled_iter{i}")
151.                 agreement = tf.matmul(inputs_hat_stopped, outputs_vj_tiled,
transpose_a=True, name=f"{self.name}_agreement_iter{i}")
152.                 b = tf.add(b, agreement, name=f"{self.name}_update_b_iter{i}")
153.
154.         final_outputs_squeezed = tf.squeeze(final_outputs_vj_primary, [1, -1],
name=f"{self.name}_final_output_squeeze")
155.
156.         if return_internals_for_calibration:
157.             if self.num_routing == 1: # Ensure final calib tensors are set if loop ran once
158.                 if c_for_calib_final is None: c_for_calib_final =
tf.nn.softmax(b_for_calib_final, axis=2)
159.                 if weighted_sum_s_for_calib_final is None: weighted_sum_s_for_calib_final
= tf.reduce_sum(tf.multiply(c_for_calib_final, inputs_hat_for_calib_final), axis=1,
keepdims=True)
160.                 if squashed_v_loop_for_calib_final is None:
squashed_v_loop_for_calib_final = self.internal_squash_layer(weighted_sum_s_for_calib_final)
161.
162.             ret_dict = {
163.                 "main_output": final_outputs_squeezed,
164.                 "inputs_hat": inputs_hat_for_calib_final,
165.                 "routing_logits_b": b_for_calib_final,
166.                 "routing_coeffs_c": c_for_calib_final,
167.                 "weighted_sum_s": weighted_sum_s_for_calib_final,
168.                 "squashed_v_loop_internals": squashed_v_loop_for_calib_final
169.             }
170.             # print(f"DEBUG_OCL_CALL ({self.name}): In
'return_internals_for_calibration=True' path. Returning DICTIONARY.") # Optional debug
171.             return ret_dict
172.         else:
173.             # print(f"DEBUG_OCL_CALL ({self.name}): In
'return_internals_for_calibration=False' path. Returning single tensor.") # Optional debug
174.             return final_outputs_squeezed
175.
```

```
    def compute_output_shape(self, input_shape):
        # Reverted to original: defines the shape of the main output only
        input_shape = tf.TensorShape(input_shape)
        return tf.TensorShape([input_shape[0], self.num_capsule, self.dim_capsule])

    def get_config(self):
        config = super(OriginalCapsuleLayer, self).get_config()
        config.update({
            'num_capsule': self.num_capsule, 'dim_capsule': self.dim_capsule,
            'num_routing': self.num_routing,
            'kernel_initializer': initializers.serialize(self.kernel_initializer)
        })
        return config

```

10.1.2. Full-Precision Capsule Network (CAPSNET) with input shape (15,4):

```
import tensorflow as tf
from tensorflow import keras
from keras import layers, models, initializers # Assuming Keras components are needed
from keras.regularizers import l2
from My_CapsNet_Layers_new import ( OriginalCapsuleLayer,
                                    OriginalPrimaryCapFunction,
                                    OriginalSquashActivation,
                                    OriginalLength,
                                     OriginalMask,
)

def CPSmodel_ir_flattened(input_shape, n_class, routings, nb_filters, kernel_size,
                          dim_capsule_caps1, num_capsule_caps1, dim_capsule_caps2,
                          dense_unit, dropout_rate, l2_reg):
    """
    A Capsule Network Model (FP32 definition) modified for regression by Flattening output capsules.
    This version uses the explicitly named "Original" custom layers.

    :param input_shape: data shape, 3d, [time steps, features] -> e.g., (15, 4)
    :param nb_filters: number of filters for the Conv1D layers
    :param kernel_size: kernel size for the Conv1D layers
    :param n_class: number of output capsules (e.g., 2, matching final output dim)
    :param routings: number of routing iterations for CapsuleLayer
    :param dim_capsule_caps1: dimension of capsules in PrimaryCap layer
    :param num_capsule_caps1: number of capsule types/channels in PrimaryCap layer
    :param dim_capsule_caps2: dimension of capsules in the final CapsuleLayer (DigitCaps)
    :param dense_unit: number of units for the Dense layers in the decoder part
    :param dropout_rate: dropout rate applied after Conv and Dense layers
    :param l2_reg: L2 regularization factor
    :return: A Keras Model for evaluation/training.
    """
    x = layers.Input(shape=input_shape, name="input_layer") # Explicit name for input layer

    # Layer 1: First Conv1D layer
    conv1 = layers.Conv1D(filters=nb_filters, kernel_size=kernel_size, strides=1,
                          padding='valid', activation='relu', name='conv1', # Name used for identification
                          kernel_regularizer=l2(l2_reg))(x)
    conv1_dropout = layers.Dropout(dropout_rate, name='dropout_1')(conv1) # Explicit name
    # print(f"Output 1st Conv (conv1_dropout): {conv1_dropout.shape}")

    # Layer 1b: Second Conv1D layer
    conv2 = layers.Conv1D(filters=nb_filters, kernel_size=kernel_size, strides=1,
                          padding='valid', activation='relu', name='conv2', # Name used for identification
                          kernel_regularizer=l2(l2_reg))(conv1_dropout)
    conv2_dropout = layers.Dropout(dropout_rate, name='dropout_2')(conv2) # Explicit name
    # print(f"Output 2nd Conv (conv2_dropout): {conv2_dropout.shape}")

    # Layer 2: Primary Capsule Layer
```

```
    # OriginalPrimaryCapFunction will internally create layers like 'primarycap_conv1d' and 'primarycap_squash'
    # Ensure those internal names are consistent if you need to access them by name.
    primarycaps_output_tensor = OriginalPrimaryCapFunction(
        conv2_dropout,
        dim_capsule=dim_capsule_caps1,
        n_channels=num_capsule_caps1,
        kernel_size=kernel_size, strides=1,
        padding='valid'
    )
    # print(f"PrimaryCaps Output (primarycaps_output_tensor): {primarycaps_output_tensor.shape}")

    # Layer 3: Digit Capsule Layer (CapsuleLayer)
    # Uses the renamed OriginalCapsuleLayer
    digitcaps = OriginalCapsuleLayer( # Using renamed class
        num_capsule=n_class,
        dim_capsule=dim_capsule_caps2,
        num_routing=routings,
        name='digitcaps' # Crucial name for layer identification
    )(primarycaps_output_tensor)
    # print(f"DigitCaps Output (digitcaps): {digitcaps.shape}")

    flattened_caps = layers.Flatten(name='flatten_caps')(digitcaps) # Explicit name
    # print(f"Flattened Capsule Output: {flattened_caps.shape}")

    # Decoder/Regression Part
    dense1 = layers.Dense(dense_unit, activation='relu',
                          kernel_initializer=initializers.glorot_uniform(),
                          kernel_regularizer=l2(l2_reg),
                          name='dense_1')(flattened_caps) # Explicit name
    dense1_dropout = layers.Dropout(dropout_rate, name='dropout_3')(dense1) # Explicit name
    # print(f"Output 1st Dense (dense1_dropout): {dense1_dropout.shape}")

    dense2 = layers.Dense(dense_unit, activation='relu',
                          kernel_initializer=initializers.glorot_uniform(),
                          kernel_regularizer=l2(l2_reg),
                          name='dense_2')(dense1_dropout) # Explicit name
    dense2_dropout = layers.Dropout(dropout_rate, name='dropout_4')(dense2) # Explicit name
    # print(f"Output 2nd Dense (dense2_dropout): {dense2_dropout.shape}")

    # Final Output Layer
    final_output = layers.Dense(n_class, activation='linear', name="output")(dense2_dropout) # Explicit name
    # print(f"Final Output shape (final_output): {final_output.shape}")

    model = models.Model(inputs=x, outputs=final_output, name="CapsuleNetwork_Flattened_FP32")
    return model

```

10.1.3. Rounding Primitives:

```
import tensorflow as tf
from tensorflow.keras import layers
```

```
import numpy as np # For type checking if needed

# --- Core Quantization Logic (with @tf.custom_gradient) ---
# These functions now take scale_factor, clip_min, clip_max as Python numbers
# and cast them to tensors internally. num_integer_bits is for context/metadata.

@tf.custom_gradient
def _quantize_truncate_logic(x, num_integer_bits_py, scale_factor_py, clip_min_py, clip_max_py):
    # x is the input tensor
    # Other args are Python scalars passed from the Layer's call method

    scale_factor = tf.cast(scale_factor_py, dtype=x.dtype)
    clip_min_tf = tf.cast(clip_min_py, dtype=x.dtype)
    clip_max_tf = tf.cast(clip_max_py, dtype=x.dtype)

    x_scaled = x * scale_factor
    x_rounded_scaled = tf.floor(x_scaled)
    x_quantized = x_rounded_scaled / scale_factor
    x_clipped = tf.clip_by_value(x_quantized, clip_min_tf, clip_max_tf)

    def grad(dy):
        # Gradient only for x. Other args are constants to this op.
        return dy, None, None, None, None
    return x_clipped, grad

@tf.custom_gradient
def _quantize_round_to_nearest_logic(x, num_integer_bits_py, scale_factor_py, clip_min_py, clip_max_py):
    scale_factor = tf.cast(scale_factor_py, dtype=x.dtype)
    clip_min_tf = tf.cast(clip_min_py, dtype=x.dtype)
    clip_max_tf = tf.cast(clip_max_py, dtype=x.dtype)

    x_scaled = x * scale_factor
    x_rounded_scaled = tf.floor(x_scaled + tf.cast(0.5, dtype=x.dtype))
    x_quantized = x_rounded_scaled / scale_factor
    x_clipped = tf.clip_by_value(x_quantized, clip_min_tf, clip_max_tf)
    def grad(dy):
        return dy, None, None, None, None
    return x_clipped, grad

@tf.custom_gradient
def _quantize_stochastic_logic(x, num_integer_bits_py, scale_factor_py, clip_min_py, clip_max_py):
    scale_factor = tf.cast(scale_factor_py, dtype=x.dtype)
    clip_min_tf = tf.cast(clip_min_py, dtype=x.dtype)
    clip_max_tf = tf.cast(clip_max_py, dtype=x.dtype)

    x_scaled = x * scale_factor
    x_floor_scaled = tf.floor(x_scaled)
    prob_round_up = x_scaled - x_floor_scaled

    rand_values = tf.random.uniform(shape=tf.shape(x_scaled), dtype=x.dtype)
    should_round_up = tf.less(rand_values, prob_round_up)

    x_stochastic_rounded_scaled = tf.where(should_round_up, x_floor_scaled + tf.cast(1.0, dtype=x.dtype), x_floor_scaled)
    x_quantized = x_stochastic_rounded_scaled / scale_factor
    x_clipped = tf.clip_by_value(x_quantized, clip_min_tf, clip_max_tf)
    def grad(dy):
        return dy, None, None, None, None
    return x_clipped, grad

# --- Keras Layer Wrappers for Quantization Primitives ---

class QuantizeTruncateLayer(layers.Layer):
    def __init__(self, num_integer_bits, num_fractional_bits, clip_min, clip_max, name=None, **kwargs):
        super(QuantizeTruncateLayer, self).__init__(name=name, **kwargs)
        self.num_integer_bits = int(num_integer_bits)
```

```
 68.         self.num_fractional_bits = int(num_fractional_bits)
 69.         self.clip_min = float(clip_min)
 70.         self.clip_max = float(clip_max)
 71.         # Precompute scale_factor as a Python float
 72.         self.scale_factor_py = 2.0**self.num_fractional_bits
 73.
 74.     def call(self, inputs):
 75.         # Pass Python scalars (attributes and precomputed scale_factor) to the logic
function
 76.         return _quantize_truncate_logic(inputs,
 77.                                         self.num_integer_bits,
 78.                                         self.scale_factor_py,
 79.                                         self.clip_min,
 80.                                         self.clip_max)
 81.
 82.     def get_config(self):
 83.         config = super(QuantizeTruncateLayer, self).get_config()
 84.         config.update({
 85.             "num_integer_bits": self.num_integer_bits,
 86.             "num_fractional_bits": self.num_fractional_bits,
 87.             "clip_min": self.clip_min,
 88.             "clip_max": self.clip_max,
 89.         })
 90.         return config
 91.
 92. class QuantizeRoundToNearestLayer(layers.Layer):
 93.     def __init__(self, num_integer_bits, num_fractional_bits, clip_min, clip_max,
name=None, **kwargs):
 94.         super(QuantizeRoundToNearestLayer, self).__init__(name=name, **kwargs)
 95.         self.num_integer_bits = int(num_integer_bits)
 96.         self.num_fractional_bits = int(num_fractional_bits)
 97.         self.clip_min = float(clip_min)
 98.         self.clip_max = float(clip_max)
 99.         self.scale_factor_py = 2.0**self.num_fractional_bits
100.
101.     def call(self, inputs):
102.         return _quantize_round_to_nearest_logic(inputs,
103.                                                 self.num_integer_bits,
104.                                                 self.scale_factor_py,
105.                                                 self.clip_min,
106.                                                 self.clip_max)
107.
108.     def get_config(self):
109.         config = super(QuantizeRoundToNearestLayer, self).get_config()
110.         config.update({
111.             "num_integer_bits": self.num_integer_bits,
112.             "num_fractional_bits": self.num_fractional_bits,
113.             "clip_min": self.clip_min,
114.             "clip_max": self.clip_max,
115.         })
116.         return config
117.
118. class QuantizeStochasticLayer(layers.Layer):
119.     def __init__(self, num_integer_bits, num_fractional_bits, clip_min, clip_max,
name=None, **kwargs):
120.         super(QuantizeStochasticLayer, self).__init__(name=name, **kwargs)
121.         self.num_integer_bits = int(num_integer_bits)
122.         self.num_fractional_bits = int(num_fractional_bits)
123.         self.clip_min = float(clip_min)
124.         self.clip_max = float(clip_max)
125.         self.scale_factor_py = 2.0**self.num_fractional_bits
126.
127.     def call(self, inputs):
128.         return _quantize_stochastic_logic(inputs,
129.                                           self.num_integer_bits,
130.                                           self.scale_factor_py,
131.                                           self.clip_min,
132.                                           self.clip_max)
133.
134.     def get_config(self):
```

```
        config = super(QuantizeStochasticLayer, self).get_config()
        config.update({
            "num_integer_bits": self.num_integer_bits,
            "num_fractional_bits": self.num_fractional_bits,
            "clip_min": self.clip_min,
            "clip_max": self.clip_max,
        })
        return config

```

10.1.4. Quantizable Model:

```
import tensorflow as tf
from tensorflow.keras import layers, models, initializers
from tensorflow.keras.regularizers import l2
from tf_quantization_primitives import QuantizeRoundToNearestLayer,QuantizeStochasticLayer,QuantizeTruncateLayer # Example quantization function
# Assume your quantized custom layers are in 'tf_quantized_custom_layers.py'
# You would uncomment these lines in your actual project:
# from tf_quantized_custom_layers import PrimaryCapsuleLayer, CapsuleLayer, SquashActivation, _apply_quant_from_config
# from tf_quantization_primitives import quantize_truncate_ste # Or your chosen default

from tf_quantized_custom_layers import QuantizedLayerWrapper, QuantizedPrimaryCapsuleLayer, QuantizedCapsuleLayer
from tf_quantized_custom_layers import _create_quantizer_from_config
from tf_quantized_custom_layers import QuantizedSquashActivation

from tf_quantized_custom_layers import QuantizedPrimaryCapsuleLayer, QuantizedCapsuleLayer, QuantizedSquashActivation

def CPSmodel_ir_flattened_quantized(
    input_shape, n_class, routings, nb_filters, kernel_size,
    dim_capsule_caps1, num_capsule_caps1, dim_capsule_caps2,
    dense_unit, dropout_rate, l2_reg,
    default_quantization_layer_class,
    quantization_configs=None,
    training=None
    ):
    """
    A Quantized Capsule Network Model built with a fully static, serializable,
    and loadable graph architecture.
    """
    x_input = layers.Input(shape=input_shape, name="input_layer")
    q_configs = quantization_configs or {}

    # --- Convolutional Base ---
    conv1_quantizer = _create_quantizer_from_config(q_configs.get('conv1_output'), default_quantization_layer_class, "conv1_quant")
    conv1 = QuantizedLayerWrapper(
        layers.Conv1D(filters=nb_filters, kernel_size=kernel_size, strides=1,
                      padding='valid', activation='relu', name='conv1',
                      kernel_regularizer=l2(l2_reg)),
        quantizer=conv1_quantizer,
        name='wrapped_conv1'
    )(x_input)
    conv1_dropout = layers.Dropout(dropout_rate, name='dropout_1')(conv1)

    conv2_quantizer = _create_quantizer_from_config(q_configs.get('conv2_output'), default_quantization_layer_class, "conv2_quant")
    conv2 = QuantizedLayerWrapper(
        layers.Conv1D(filters=nb_filters, kernel_size=kernel_size, strides=1,
                      padding='valid', activation='relu', name='conv2',
                      kernel_regularizer=l2(l2_reg)),
        quantizer=conv2_quantizer,
        name='wrapped_conv2'
    )(conv1_dropout)
```

```
    conv2_dropout = layers.Dropout(dropout_rate, name='dropout_2')(conv2)

    # --- Capsule Layers (Create quantizers and pass instances) ---

    # Primary Capsule Layer
    prim_caps_conv_q = _create_quantizer_from_config(q_configs.get('primary_caps_conv_output'), default_quantization_layer_class, "prim_caps_conv_q")
    prim_caps_squash_q_config = q_configs.get('primary_caps_squash_output')
    prim_caps_squash_q = _create_quantizer_from_config(prim_caps_squash_q_config, default_quantization_layer_class, "prim_caps_squash_q")
    prim_caps_squash_act = QuantizedSquashActivation(axis=-1, output_quantizer=prim_caps_squash_q, name=f"primary_caps_squash")

    primarycaps_output = QuantizedPrimaryCapsuleLayer(
        dim_capsule=dim_capsule_caps1, n_channels=num_capsule_caps1,
        kernel_size=kernel_size, strides=1, padding='valid', name='primary_caps',
        kernel_regularizer=l2(l2_reg),
        conv_output_quantizer=prim_caps_conv_q,
        squash_activation=prim_caps_squash_act
    )(conv2_dropout)

    # Digit Capsule Layer
    digit_caps_internal_configs = q_configs.get('digit_caps_internal', {})
    digit_caps_quantizer_dict = {
        key: _create_quantizer_from_config(conf, default_quantization_layer_class, f"digit_caps_{key}_q")
        for key, conf in digit_caps_internal_configs.items()
    }
    digit_caps_squash_q = digit_caps_quantizer_dict.pop('final_squashed_outputs_v', None) # Get the squash quantizer
    digit_caps_squash_layer = QuantizedSquashActivation(axis=-2, output_quantizer=digit_caps_squash_q, name="digit_caps_internal_squash")

    digitcaps_output = QuantizedCapsuleLayer(
        num_capsule=n_class, dim_capsule=dim_capsule_caps2,
        num_routing=routings, name='digitcaps',
        quantizer_dict=digit_caps_quantizer_dict,
        internal_squash_layer=digit_caps_squash_layer
    )(primarycaps_output)

    flattened_caps = layers.Flatten(name='flatten_caps')(digitcaps_output)

    # --- Decoder/Regression Part ---
    dense1_quantizer = _create_quantizer_from_config(q_configs.get('dense1_output'), default_quantization_layer_class, "dense1_quant")
    dense1 = QuantizedLayerWrapper(
        layers.Dense(dense_unit, activation='relu', name='dense_1', kernel_initializer=initializers.glorot_uniform(), kernel_regularizer=l2(l2_reg)),
        quantizer=dense1_quantizer,
        name='wrapped_dense1'
    )(flattened_caps)
    dense1_dropout = layers.Dropout(dropout_rate, name='dropout_3')(dense1)

    dense2_quantizer = _create_quantizer_from_config(q_configs.get('dense2_output'), default_quantization_layer_class, "dense2_quant")
    dense2 = QuantizedLayerWrapper(
        layers.Dense(dense_unit, activation='relu', name='dense_2', kernel_initializer=initializers.glorot_uniform(), kernel_regularizer=l2(l2_reg)),
        quantizer=dense2_quantizer,
        name='wrapped_dense2'
    )(dense1_dropout)
    dense2_dropout = layers.Dropout(dropout_rate, name='dropout_4')(dense2)

    # Final Output Layer
    final_out_quantizer = _create_quantizer_from_config(q_configs.get('final_model_output'), default_quantization_layer_class, "final_out_quant")
    final_output = QuantizedLayerWrapper(
        layers.Dense(n_class, activation='linear', name="output"),
```

```
        quantizer=final_out_quantizer,
        name='wrapped_output'
    )(dense2_dropout)

    model = models.Model(inputs=x_input, outputs=final_output,
name="Quantized_CapsuleNetwork_Flattened")
    return model

```

10.1.5. Quantization Framework Main Function:

```
import tensorflow as tf
import numpy as np
import os
import copy  # For deep copying model states or configurations
import json  # For saving/loading calibration results
import csv  # For loading your CSV data
from tensorflow.keras import backend as K  # For K.is_keras_tensor

# --- Custom Module Imports ---
# Ensure these files are in the same directory or accessible via sys.path
from tf_quantization_primitives import (
    QuantizeRoundToNearestLayer,
    QuantizeTruncateLayer,
    QuantizeStochasticLayer,
)
from tf_quantized_custom_layers import (
    QuantizedPrimaryCapsuleLayer,
    QuantizedCapsuleLayer,
    # _apply_quant_from_config # Used internally by quantized layers/model
)
from CPSmodel_ir_flattened_quantized import CPSmodel_ir_flattened_quantized
from My_CapsNet_Layers_new import (
    OriginalCapsuleLayer,
    OriginalSquashActivation,
    OriginalLength,
    OriginalMask,
)
from tf_quantized_custom_layers import _create_quantizer_from_config,
QuantizedLayerWrapper, save_weights_to_csv

# --- NEW IMPORTS FOR HEATMAP ---
import matplotlib.pyplot as plt
import seaborn as sns


# --- HELPER MODEL CLASS FOR PROBING DigitCaps INTERNALS ---
##### This prob model function take the original digitcaps layer and returns a model that
can probe the internals of the DigitCaps layer.
class DigitCapsInternalsProbe(tf.keras.Model):
    def __init__(self, original_digitcaps_layer_ref,
name="digitcaps_internals_probe_model", **kwargs):
        super().__init__(name=name, **kwargs)
        ocl_config = original_digitcaps_layer_ref.get_config()
        ocl_config['name'] =
f"{original_digitcaps_layer_ref.name}_probe_internal_instance"
        self.probe_caps_layer = OriginalCapsuleLayer.from_config(ocl_config)
        self.original_weights_values = original_digitcaps_layer_ref.get_weights()
        self._probe_weights_set = False
        self.internal_dict_keys_in_order = [
            "main_output", "inputs_hat", "routing_logits_b",
            "routing_coeffs_c", "weighted_sum_s", "squashed_v_loop_internals"
        ]

    def call(self, inputs, training=None):
        if not self.probe_caps_layer.built:
            self.probe_caps_layer.build(inputs.shape)
        if not self._probe_weights_set:
```

```
            if self.original_weights_values and self.probe_caps_layer.weights:
                self.probe_caps_layer.set_weights(self.original_weights_values)
                self._probe_weights_set = True
        #### Set the config for the probe layer to return internals for calibration (Set the clibration flag)
        internal_outputs_dict = self.probe_caps_layer(inputs, training=training, return_internals_for_calibration=True)

        if isinstance(internal_outputs_dict, dict):
            ordered_tensors_for_output = []
            for key in self.internal_dict_keys_in_order:
                if key in internal_outputs_dict:
                    ordered_tensors_for_output.append(internal_outputs_dict[key])
                else:
                    # Fallback for missing keys to maintain output structure
                    ordered_tensors_for_output.append(tf.keras.layers.Lambda(lambda x: tf.zeros_like(x))(inputs))
            return ordered_tensors_for_output
        else:
            # Handle case where return_internals_for_calibration might not return a dict
            ref_tensor = inputs
            if isinstance(internal_outputs_dict, tf.Tensor) and K.is_keras_tensor(internal_outputs_dict):  # Check if it's a Keras tensor
                ref_tensor = internal_outputs_dict
            return [tf.keras.layers.Lambda(lambda x: tf.zeros_like(x))(ref_tensor) for _ in self.internal_dict_keys_in_order]

    def compute_output_shape(self, input_shape):
        input_shape = tf.TensorShape(input_shape)  # Ensure it's a TensorShape
        batch_size = input_shape[0]  # Can be None
        current_input_num_capsule = input_shape[1]  # Can be None
        # These should be defined during __init__ based on original_digitcaps_layer_ref
        num_output_caps = self.probe_caps_layer.num_capsule
        dim_output_caps = self.probe_caps_layer.dim_capsule
        # Shapes for the internal tensors
        main_out_shape = tf.TensorShape([batch_size, num_output_caps, dim_output_caps])
        inputs_hat_shape = tf.TensorShape([batch_size, current_input_num_capsule, num_output_caps, dim_output_caps, 1])
        b_shape = tf.TensorShape([batch_size, current_input_num_capsule, num_output_caps, 1, 1])
        c_shape = tf.TensorShape([batch_size, current_input_num_capsule, num_output_caps, 1, 1])
        s_shape = tf.TensorShape([batch_size, 1, num_output_caps, dim_output_caps, 1])
        v_loop_shape = tf.TensorShape([batch_size, 1, num_output_caps, dim_output_caps, 1])
        return [main_out_shape, inputs_hat_shape, b_shape, c_shape, s_shape, v_loop_shape]

# --- Paths and Constants ---
SCRIPT_DIRECTORY = os.path.dirname(os.path.abspath(__file__))
FP32_MODEL_PATH = os.path.join(SCRIPT_DIRECTORY, "best_CapsNet_Deep_model_unscaled_eval_full.keras")
CALIBRATION_RESULTS_PATH = os.path.join(SCRIPT_DIRECTORY, "CapsNet_Deep_calibration_results.json")
save_path = os.path.join(SCRIPT_DIRECTORY, "final_quantized_model.keras")

MODEL_ARCH_PARAMS = {
    'input_shape': (15, 4), 'n_class': 2, 'routings': 3, 'nb_filters': 16,
    'kernel_size': 2, 'dim_capsule_caps1': 7, 'num_capsule_caps1': 10,
    'dim_capsule_caps2': 7, 'dense_unit': 16, 'dropout_rate': 0.1, 'l2_reg': 0.0001
}
INPUT_SEQUENCE_LENGTH = MODEL_ARCH_PARAMS['input_shape'][0]
OUTPUT_SEQUENCE_LENGTH = 1

CALIBRATION_BATCH_SIZE = 32
CALIBRATION_NUM_BATCHES = 30
TEST_BATCH_SIZE = 32
LOSS_TOLERANCE_PERCENT = 2
MEMORY_BUDGET_MB = 0.01
ROUNDING_SCHEMES_TO_TRY = {
    "truncate": QuantizeTruncateLayer,
```

```
    "round_to_nearest": QuantizeRoundToNearestLayer,
    "stochastic": QuantizeStochasticLayer
}
INITIAL_SEARCH_FRACTIONAL_BITS = 16
MIN_SEARCH_FRACTIONAL_BITS = 6

PARENT_DIR_CSV = "c:/Users/RN/PycharmProjects/pytorch-esn-master/Time_Series_QCapsNet - new1/Datasets/"
FILE_VALIDATION_CSV = "CapEXP3_validation_dataset.csv"
FILE_TESTING_CSV = "CapEXP3_testing_dataset.csv"
FILE_TRAINING_CSV = "CapEXP3_training_dataset.csv"

# --- HELPER FUNCTIONS ---
def load_csv_data_from_path(path, file_name):
    file_path = os.path.join(path, file_name)
    try:
        with open(file_path, 'r') as f:
            data_list = list(csv.reader(f))
        return np.asarray(data_list, dtype=np.float32)
    except FileNotFoundError as e:
        print(f"Error: Dataset file not found: {file_path}. {e}")
        raise
    except Exception as e:
        print(f"Error processing CSV file {file_path}: {e}")
        raise

def create_tf_dataset_from_array(data_array: np.ndarray, output_array: np.ndarray,
                                 input_sequence_length: int, output_sequence_length: int,
                                 batch_size: int = 1, shuffle=False):
    if output_sequence_length != 1:
        raise ValueError("This function is designed for output_sequence_length=1.")
    min_data_len = (batch_size if shuffle else 1) - 1 + input_sequence_length
    if len(data_array) < min_data_len:
        print(f"Warning: Not enough data for input_sequence_length {input_sequence_length}. Required: {min_data_len}.")
        return tf.data.Dataset.from_tensor_slices((np.zeros((0, input_sequence_length, data_array.shape[-1] if data_array.ndim > 1 else 1)),
                                                   np.zeros((0, output_array.shape[-1] if output_array.ndim > 1 else 1)))).batch(batch_size)
    targets_start_index = input_sequence_length
    num_sequences = len(data_array) - input_sequence_length + 1
    if len(output_array) < targets_start_index + num_sequences - 1:
        print(f"Warning: Target array too short ({len(output_array)}) for the number of sequences ({num_sequences}) from data_array ({len(data_array)}).")
        num_sequences = max(0, len(output_array) - targets_start_index + 1)
        if num_sequences == 0:
            return tf.data.Dataset.from_tensor_slices((np.zeros((0, input_sequence_length, data_array.shape[-1] if data_array.ndim > 1 else 1)),
                                                       np.zeros((0, output_array.shape[-1] if output_array.ndim > 1 else 1)))).batch(batch_size)
    dataset = tf.keras.utils.timeseries_dataset_from_array(
        data=data_array[:num_sequences + input_sequence_length - 1],
        targets=output_array[targets_start_index: targets_start_index + num_sequences],
        sequence_length=input_sequence_length, sequence_stride=1, shuffle=shuffle, batch_size=batch_size)
    if shuffle:
        dataset = dataset.shuffle(buffer_size=max(1000, num_sequences // 10), reshuffle_each_iteration=True)
    return dataset

def load_fp32_model(model_path):
    custom_objects_for_fp32 = {
        'OriginalCapsuleLayer': OriginalCapsuleLayer,
        'OriginalSquashActivation': OriginalSquashActivation,
        'OriginalMask': OriginalMask, 'OriginalLength': OriginalLength
    }
    try:
        if not os.path.exists(model_path):
            raise FileNotFoundError(f"Model file not found: {model_path}")
```

```
        model = tf.keras.models.load_model(model_path, custom_objects=custom_objects_for_fp32, compile=False)
        print(f"Successfully loaded FP32 model from: {model_path}")
        return model
    except Exception as e:
        print(f"Error loading FP32 model: {e}.")
        raise

def get_tensor_identifiers(fp32_model):
    print("Generating tensor identifiers for quantization...")
    weight_keys = []
    for layer in fp32_model.layers:
        if hasattr(layer, 'trainable_weights') and layer.trainable_weights:
            for weight_var in layer.trainable_weights:
                base_name = weight_var.name
                unique_key = f"{layer.name}/{base_name}"
                weight_keys.append(unique_key)
    weight_keys = sorted(list(set(weight_keys)))

    act_keys_standard_layers = ['conv1_output', 'conv2_output', 'dense1_output', 'dense2_output', 'final_model_output']
    primary_cap_conceptual_name = 'primarycap'
    act_keys_primary_cap = [
        f'{primary_cap_conceptual_name}_conv1d_output',
        f'{primary_cap_conceptual_name}_squash_output'
    ]

    caps_layer_name_fp32 = 'digitcaps'
    caps_tuple_conceptual_keys = [
        f'{caps_layer_name_fp32}_layer_output',
        f'{caps_layer_name_fp32}_internal_inputs_hat',
        f'{caps_layer_name_fp32}_internal_routing_logits_b',
        f'{caps_layer_name_fp32}_internal_routing_coeffs_c',
        f'{caps_layer_name_fp32}_internal_weighted_sum_s',
        f'{caps_layer_name_fp32}_internal_squashed_v_loop'
    ]

    activation_keys_conceptual = list(set(act_keys_standard_layers + act_keys_primary_cap + caps_tuple_conceptual_keys))
    activation_keys_conceptual = sorted(list(set(activation_keys_conceptual)))

    dr_activation_keys_conceptual = sorted(list(set([
        f'{caps_layer_name_fp32}_internal_routing_logits_b',
        f'{caps_layer_name_fp32}_internal_routing_coeffs_c',
        f'{caps_layer_name_fp32}_internal_weighted_sum_s',
        f'{caps_layer_name_fp32}_internal_squashed_v_loop'
    ])))

    all_q_point_keys_map = {
        'weights': weight_keys,
        'activations': activation_keys_conceptual,
        'dr_activations': dr_activation_keys_conceptual,
        'caps_tuple_conceptual_keys': caps_tuple_conceptual_keys
    }
    print(f"Identified {len(all_q_point_keys_map['weights'])} unique weight keys.")
    print(f"Identified {len(all_q_point_keys_map['activations'])} conceptual activation keys.")
    return all_q_point_keys_map

def perform_calibration(fp32_model, calibration_dataset, all_q_point_keys_map, num_batches_to_use):
    print(f"Starting calibration using {num_batches_to_use} batches...")
    calibration_results = {}
    print("Calibrating weights...")
    for constructed_weight_key in all_q_point_keys_map['weights']:
        try:
            layer_name, base_weight_name = constructed_weight_key.split('/', 1)
            fp32_layer_for_weight = fp32_model.get_layer(name=layer_name)
            target_weight_tensor = None
            for w_var in fp32_layer_for_weight.trainable_weights:
```

```
                if w_var.name == base_weight_name:
                    target_weight_tensor = w_var
                    break
            if target_weight_tensor is None:
                print(f"  Warning (Weight Calib): Could not find weight tensor for key '{constructed_weight_key}' using base name '{base_weight_name}' in layer '{layer_name}'. Skipping.")
                continue
        except Exception as e:
            print(f"  Error accessing weight for key '{constructed_weight_key}': {e}. Skipping.")
            continue
        min_val, max_val = tf.reduce_min(target_weight_tensor).numpy(), tf.reduce_max(target_weight_tensor).numpy()
        max_abs_val = max(abs(min_val), abs(max_val), 1e-9)
        num_int_bits = int(np.floor(np.log2(max_abs_val))) + 2 if max_abs_val >= 0.5 else 1
        calibration_results[constructed_weight_key] = {'clip_min': float(min_val), 'clip_max': float(max_val), 'num_integer_bits': int(num_int_bits), 'is_weight': True}
    print(f"Calibrated {len([k for k,v in calibration_results.items() if v.get('is_weight')])} weight tensors.")

    print("Calibrating activations...")
    probe_model_inputs = fp32_model.input # feed the same input as the original model
    probe_model_outputs_map = {}
    ### extracting the activations of the standard layers
    standard_layer_map = {'conv1_output': 'conv1', 'conv2_output': 'conv2', 'dense1_output': 'dense_1', 'dense2_output': 'dense_2', 'final_model_output': 'output'}
    for conceptual_key, layer_name_in_fp32 in standard_layer_map.items():
        try:
            probe_model_outputs_map[conceptual_key] = fp32_model.get_layer(layer_name_in_fp32).output
        except Exception as e:
            print(f"Warning (Act Calib Probe Setup - Std): Failed for '{conceptual_key}' (layer '{layer_name_in_fp32}'): {e}")
    ### extracting the activations of the primary capsule layer, it's available in standard layers
    primary_cap_conceptual_name = 'primarycap'
    primary_cap_internal_layer_map = {
        f'{primary_cap_conceptual_name}_conv1d_output': f'{primary_cap_conceptual_name}_conv1d',
        f'{primary_cap_conceptual_name}_squash_output': f'{primary_cap_conceptual_name}_squash'
    }
    for conceptual_key, internal_layer_name in primary_cap_internal_layer_map.items():
        try:
            probe_model_outputs_map[conceptual_key] = fp32_model.get_layer(internal_layer_name).output
        except Exception as e:
            print(f"Warning (Act Calib Probe Setup - PrimCap): Failed for '{conceptual_key}' (layer '{internal_layer_name}'): {e}")

    caps_layer_name_fp32 = 'digitcaps'
    caps_tuple_conceptual_keys = all_q_point_keys_map.get('caps_tuple_conceptual_keys')
    if caps_tuple_conceptual_keys and len(caps_tuple_conceptual_keys) == 6:
        try:
            ## take the original DigitCaps layer reference
            original_digitcaps_layer_ref = fp32_model.get_layer(caps_layer_name_fp32)
            ## use original digitcaps input as input to the probe model
            input_to_caps_tensor_symbolic = original_digitcaps_layer_ref.input
            ## create a separate probe model based on the original DigitCaps layer
            digitcaps_prober = DigitCapsInternalsProbe(original_digitcaps_layer_ref)
            ## call the prob model by feeding with the original DigitCaps input
            list_of_internal_symbolic_tensors = digitcaps_prober(input_to_caps_tensor_symbolic)
            ## map the outputs to the conceptual keys
            if isinstance(list_of_internal_symbolic_tensors, list) and len(list_of_internal_symbolic_tensors) == len(caps_tuple_conceptual_keys):
                for i, conceptual_key_dc in enumerate(caps_tuple_conceptual_keys):
```

```
                    probe_model_outputs_map[conceptual_key_dc] =
list_of_internal_symbolic_tensors[i]
                print(f"Successfully mapped {len(caps_tuple_conceptual_keys)} DigitCaps
internals for probing.")
            else:
                print(f"ERROR: DigitCapsInternalsProbe did not return expected list for
mapping. Got: {type(list_of_internal_symbolic_tensors)}")
        except Exception as e:
            print(f"Error during DigitCapsInternalsProbe setup: {e}")
            import traceback
            traceback.print_exc()
    else:
        print(f"Warning: 'caps_tuple_conceptual_keys' issue for DigitCaps. Keys:
{caps_tuple_conceptual_keys}")
    ## --- Prepare the final probe model outputs list for activations of standard layers
and digitcaps internals ---
    target_probe_keys_conceptual = sorted(list(set(all_q_point_keys_map['activations'] +
all_q_point_keys_map.get('dr_activations', []))))
    final_probe_tensors_list, final_probe_keys_ordered = [], []
    for key in target_probe_keys_conceptual:
        if key in probe_model_outputs_map:
            final_probe_tensors_list.append(probe_model_outputs_map[key])
            final_probe_keys_ordered.append(key)
        else:
            print(f"Warning (Final Probe List): Conceptual key '{key}' not found in
probe_model_outputs_map. Will be defaulted.")
    ### Start calibration of activations by detecting the min/max values
    if final_probe_tensors_list:
        try:
            calibration_probe_model = tf.keras.Model(inputs=probe_model_inputs,
outputs=final_probe_tensors_list, name="Calibration_Probe_Model")
            print(f"Calibration probe model created with {len(final_probe_keys_ordered)}
outputs.")
            tracker = {k: {'min': tf.Variable(np.inf, dtype=tf.float32), 'max':
tf.Variable(-np.inf, dtype=tf.float32)} for k in final_probe_keys_ordered}
            pb = 0
            for x_batch, _ in calibration_dataset.take(num_batches_to_use):
                if tf.shape(x_batch)[0] == 0:
                    continue
                if pb == 0 or (pb + 1) % 10 == 0 or (pb + 1) == num_batches_to_use:
                    print(f" Calibrating acts: batch {pb + 1}/{num_batches_to_use}")
                outputs_batch = calibration_probe_model(x_batch, training=False)
                if not isinstance(outputs_batch, list):
                    outputs_batch = [outputs_batch]
                if len(outputs_batch) != len(final_probe_keys_ordered):
                    print(f"ERROR: Probe output mismatch during batch run. Expected
{len(final_probe_keys_ordered)}, got {len(outputs_batch)}. Skipping batch.")
                    continue
                for i, k_act in enumerate(final_probe_keys_ordered):
                    current_tensor_out = outputs_batch[i]
                    if tf.size(current_tensor_out) > 0:
                        current_tensor_out = tf.cast(current_tensor_out, tf.float32)
                        tracker[k_act]['min'].assign(tf.minimum(tracker[k_act]['min'],
tf.reduce_min(current_tensor_out)))
                        tracker[k_act]['max'].assign(tf.maximum(tracker[k_act]['max'],
tf.reduce_max(current_tensor_out)))
                pb += 1
            for k_act, r_vals in tracker.items():
                min_val_np, max_val_np = r_vals['min'].numpy(), r_vals['max'].numpy()
                if np.isinf(min_val_np) or np.isinf(max_val_np) or np.isnan(min_val_np) or
np.isnan(max_val_np) or min_val_np >= max_val_np:
                    min_val_np, max_val_np = -1.0, 1.0
                    print(f"Warning: Invalid or non-finite range for activation '{k_act}'.
Min={r_vals['min'].numpy()}, Max={r_vals['max'].numpy()}. Defaulting to [-1.0, 1.0].")
                max_abs_val = max(abs(min_val_np), abs(max_val_np), 1e-9)
                nib = int(np.floor(np.log2(max_abs_val))) + 2 if max_abs_val >= 0.5 else 1
                calibration_results[k_act] = {'clip_min': float(min_val_np), 'clip_max':
float(max_val_np), 'num_integer_bits': nib, 'is_weight': False}
            print(f"Calibrated {len(tracker)} activation points found by probe model.")
        except Exception as e:
```

```
344.             print(f"ERROR during calibration_probe_model execution: {e}")
345.             import traceback
346.             traceback.print_exc()
347.     else:
348.         print("ERROR: No valid activation outputs for probe model construction.")
349.
350.     for k_act_conceptual in target_probe_keys_conceptual:
351.         if k_act_conceptual not in calibration_results:
352.             calibration_results[k_act_conceptual] = {'clip_min': -1.0, 'clip_max': 1.0,
'num_integer_bits': 1, 'is_weight': False}
353.             print(f"Warning: Activation key '{k_act_conceptual}' was missing post-probing
and has been defaulted.")
354.
355.     print("Calibration data collection finished.")
356.     try:
357.         with open(CALIBRATION_RESULTS_PATH, 'w') as f:
358.             json.dump(calibration_results, f, indent=4, sort_keys=True)
359.         print(f"Calibration results saved to {CALIBRATION_RESULTS_PATH}")
360.     except Exception as e:
361.         print(f"Error saving calibration results: {e}")
362.     return calibration_results
363.
364. def build_quantized_model_from_config(fp32_model_ref, model_arch_params,
current_quant_configs, default_rounding_layer_class):
365.     print("Building quantized model with current configurations...")
366.     q_model = CPSmodel_ir_flattened_quantized(
367.         **model_arch_params,
368.         default_quantization_layer_class=default_rounding_layer_class,
369.         quantization_configs=current_quant_configs,
370.         training=False
371.     )
372.
373.     # --- Create a map of all FP32 weights for easy lookup ---
374.     fp32_weights_map = {}
375.     for layer in fp32_model_ref.layers:
376.         # Use layer.weights to get all weights, not just trainable
377.         for weight_var in layer.weights:
378.             # The key is the layer name + the base weight name (e.g., 'kernel:0')
379.             fp32_weights_map[f"{layer.name}/{weight_var.name.split('/')[-1]}"] =
weight_var.numpy()
380.
381.     # --- Iterate through the new Quantized Model and assign weights ---
382.     assigned_count = 0
383.     total_q_weights = len(q_model.trainable_weights)
384.
385.     for q_layer in q_model.layers:
386.         if not q_layer.trainable:
387.             continue
388.
389.         # Determine the target layer for weight lookup
390.         target_fp32_layer_name = ""
391.         weights_to_assign = []
392.
393.         if isinstance(q_layer, QuantizedLayerWrapper):
394.             # If it's our wrapper, the target is the layer INSIDE it.
395.             target_fp32_layer_name = q_layer.layer_to_wrap.name
396.             weights_to_assign = q_layer.layer_to_wrap.trainable_weights
397.
398.         elif isinstance(q_layer, QuantizedPrimaryCapsuleLayer):
399.             # Special handling for the primary capsule layer's internal Conv1D
400.             # The FP32 model has 'primarycap_conv1d', the Q model has it inside
'primary_caps'.
401.             target_fp32_layer_name = 'primarycap_conv1d'
402.             weights_to_assign = q_layer.conv1d.trainable_weights
403.
404.         elif isinstance(q_layer, QuantizedCapsuleLayer):
405.             # For the main capsule layer, the names match directly.
406.             target_fp32_layer_name = q_layer.name
407.             weights_to_assign = q_layer.trainable_weights
408.
```

```
409.         else:
410.             # For other layers like Dropout, Flatten, etc. (which have no weights)
411.             continue
412.
413.         if not target_fp32_layer_name:
414.             continue
415.
416.         # Now assign weights using the determined target name
417.         for q_w_var in weights_to_assign:
418.             # Construct the key to find the corresponding FP32 weight
419.             base_weight_name = q_w_var.name.split('/')[-1]
420.             fp32_lookup_key = f"{target_fp32_layer_name}/{base_weight_name}"
421.
422.             fp32_w_np = fp32_weights_map.get(fp32_lookup_key)
423.
424.             if fp32_w_np is None:
425.                 print(f"Warning: FP32 weight not found for key: {fp32_lookup_key}.
Skipping assignment.")
426.                 continue
427.
428.             # Check if this weight needs to be quantized
429.             w_conf = current_quant_configs.get(fp32_lookup_key)
430.             if w_conf and w_conf.get('is_weight') and
w_conf.get('quantization_function_tf_class'):
431.                 # This weight has a quantization configuration, apply it
432.                 quant_class = w_conf['quantization_function_tf_class']
433.                 q_op = quant_class(
434.                     num_integer_bits=w_conf['num_integer_bits'],
435.                     num_fractional_bits=w_conf['num_fractional_bits'],
436.                     clip_min=w_conf['clip_min'],
437.                     clip_max=w_conf['clip_max']
438.                 )
439.                 q_w_var.assign(q_op(tf.constant(fp32_w_np, dtype=tf.float32)).numpy())
440.             else:
441.                 # No quantization for this weight, just assign the FP32 value
442.                 q_w_var.assign(fp32_w_np)
443.
444.             assigned_count += 1
445.
446.     print(f"Finished building q_model. Weights assigned/quantized:
{assigned_count}/{total_q_weights} trainable variables in q_model.")
447.     return q_model
448.
449. def evaluate_quantized_model_loss(q_model, test_dataset):
450.     print("Evaluating quantized model (MSE on UNCALED data)...")
451.     if not isinstance(test_dataset, tf.data.Dataset) or
tf.data.experimental.cardinality(test_dataset).numpy() == 0:
452.         print("Warning: Test dataset is invalid or empty. Cannot evaluate.")
453.         return float('inf')
454.
455.     all_preds_list, all_true_list = [], []
456.     for x_batch, y_batch_true in test_dataset:
457.         if tf.shape(x_batch)[0] == 0:
458.             continue
459.         pred_batch = q_model(x_batch, training=False)
460.         all_preds_list.append(pred_batch.numpy())
461.         all_true_list.append(y_batch_true.numpy())
462.
463.     if not all_true_list or not all_preds_list:
464.         print("Warning: No data processed from test_dataset for evaluation.")
465.         return float('inf')
466.
467.     true_np = np.concatenate(all_true_list, axis=0)
468.     preds_np = np.concatenate(all_preds_list, axis=0)
469.
470.     min_len = min(len(true_np), len(preds_np))
471.     true_np = true_np[:min_len]
472.     preds_np = preds_np[:min_len]
473.
474.     if not preds_np.size:
```

```
475.         print("Warning: Prediction array effectively empty after alignment.")
476.         return float('inf')
477.
478.     mse_loss = tf.reduce_mean(tf.square(tf.cast(true_np.flatten(), tf.float32) -
tf.cast(preds_np.flatten(), tf.float32))).numpy()
479.     print(f"Evaluation: MSE Loss (UNSCALED data) = {mse_loss:.6f}")
480.     if np.isnan(mse_loss) or np.isinf(mse_loss):
481.         print(f"Warning: MSE Loss is NaN or Inf ({mse_loss}). Returning very high loss.")
482.         return float('inf')
483.     return mse_loss
484.
485. def evaluate_fp32_model_loss(fp32_model, test_dataset):
486.     print("Evaluating FP32 model (MSE on UNSCALED data)...")
487.     if not isinstance(test_dataset, tf.data.Dataset) or
tf.data.experimental.cardinality(test_dataset).numpy() == 0:
488.         print("Warning: Test dataset is invalid or empty for FP32 eval.")
489.         return float('inf')
490.
491.     all_preds_list, all_true_list = [], []
492.     for x_batch, y_batch_true in test_dataset:
493.         if tf.shape(x_batch)[0] == 0:
494.             continue
495.         pred_batch = fp32_model(x_batch, training=False)
496.         all_preds_list.append(pred_batch.numpy())
497.         all_true_list.append(y_batch_true.numpy())
498.
499.     if not all_true_list or not all_preds_list:
500.         print("Warning: No data processed from test_dataset for FP32 evaluation.")
501.         return float('inf')
502.
503.     true_np = np.concatenate(all_true_list, axis=0)
504.     preds_np = np.concatenate(all_preds_list, axis=0)
505.
506.     min_len = min(len(true_np), len(preds_np))
507.     true_np = true_np[:min_len]
508.     preds_np = preds_np[:min_len]
509.
510.     if not preds_np.size:
511.         print("Warning: FP32 Prediction array effectively empty after alignment.")
512.         return float('inf')
513.
514.     mse_loss = tf.reduce_mean(tf.square(tf.cast(true_np.flatten(), tf.float32) -
tf.cast(preds_np.flatten(), tf.float32))).numpy()
515.     print(f"FP32 Model Evaluation: MSE Loss (UNSCALED data) = {mse_loss:.6f}")
516.     if np.isnan(mse_loss) or np.isinf(mse_loss):
517.         print(f"Warning: FP32 MSE Loss is NaN or Inf ({mse_loss}). Returning very high
loss.")
518.         return float('inf')
519.     return mse_loss
520.
521.
522. def get_num_parameters_for_weights(fp32_model_ref, robust_weight_keys_list):
523.     num_params_map = {}
524.     fp32_robust_key_to_var_map = {}
525.     for layer in fp32_model_ref.layers:
526.         if hasattr(layer, 'trainable_weights') and layer.trainable_weights:
527.             for weight_var in layer.trainable_weights:
528.                 key = f"{layer.name}/{weight_var.name}"
529.                 fp32_robust_key_to_var_map[key] = weight_var
530.     for r_key in robust_weight_keys_list:
531.         tensor_var = fp32_robust_key_to_var_map.get(r_key)
532.         if tensor_var is not None:
533.             num_params_map[r_key] = tf.size(tensor_var).numpy()
534.         else:
535.             print(f"Warning (NumParams): Weight key '{r_key}' not found in FP32 model
map.")
536.             num_params_map[r_key] = 0
537.     return num_params_map
538.
539. def calculate_memory_footprint(current_quant_configs, weight_keys_list, num_params_map):
```

```
    total_bits = 0
    for key in weight_keys_list:
        config = current_quant_configs.get(key)
        if config and config.get('is_weight', False) and 'num_integer_bits' in config and
'num_fractional_bits' in config:
            num_params = num_params_map.get(key, 0)
            bits_per_param = config['num_integer_bits'] + config['num_fractional_bits']
            if bits_per_param <= 0:
                continue
            total_bits += num_params * bits_per_param
    return total_bits / (8 * 1024 * 1024)  # Convert bits to MegaBytes

# --- SENSITIVITY HEATMAP GENERATION FUNCTION ---
def generate_nfb_sensitivity_heatmap(
    fp32_model_ref, model_arch_params, all_q_points_info, calibration_data_loaded,
    test_dataset, baseline_loss, nfb_range, default_rounding_scheme_class):
    """
    Performs a sensitivity analysis by varying the Number of Fractional Bits (NFB)
    for weights and activations and plots the resulting model loss as a heatmap.

    Args:
        fp32_model_ref: The reference FP32 model instance.
        model_arch_params: Dictionary of model architecture parameters.
        all_q_points_info: Dictionary containing lists of weight and activation keys.
        calibration_data_loaded: The dictionary with calibration data (clip ranges, NIB).
        test_dataset: The dataset to use for evaluating the model loss.
        baseline_loss: The pre-calculated loss of the FP32 model.
        nfb_range: A list or range of NFB values to test (e.g., range(6, 17)).
        default_rounding_scheme_class: The quantization function class to use.
    """
    print("\n\n=== Starting NFB Sensitivity Heatmap Generation ===")

    # We will vary NFB for weights (y-axis) and activations (x-axis)
    weight_nfb_range = list(nfb_range)
    activation_nfb_range = list(nfb_range)

    # Matrix to store the loss results.
    loss_matrix = np.zeros((len(weight_nfb_range), len(activation_nfb_range)))

    # Get all weight and activation keys from the info dictionary
    weight_keys = all_q_points_info.get('weights', [])
    activation_keys = all_q_points_info.get('activations', [])
    dr_activation_keys = all_q_points_info.get('dr_activations', [])
    all_activation_keys = list(set(activation_keys + dr_activation_keys))

    # For updating the nested digitcaps config
    simple_to_conceptual_map_digitcaps = {
        'inputs_hat': all_q_points_info['caps_tuple_conceptual_keys'][1],
        'routing_logits_b': all_q_points_info['caps_tuple_conceptual_keys'][2],
        'routing_coeffs_c': all_q_points_info['caps_tuple_conceptual_keys'][3],
        'weighted_sum_s': all_q_points_info['caps_tuple_conceptual_keys'][4],
        'final_squashed_outputs_v': all_q_points_info['caps_tuple_conceptual_keys'][5],
        'squashed_outputs_v_routing_loop':
all_q_points_info['caps_tuple_conceptual_keys'][5]
    }

    # --- Main loop for sensitivity analysis ---
    for i, w_nfb in enumerate(weight_nfb_range):
        for j, a_nfb in enumerate(activation_nfb_range):
            print(f"\n--- Testing: Weights NFB = {w_nfb}, Activations NFB = {a_nfb} ---")

            # Create a fresh quantization configuration for this specific run
            temp_quant_configs = copy.deepcopy(calibration_data_loaded)

            # 1. Configure all weights with the current weight NFB
            for key in weight_keys:
                if key in temp_quant_configs:
                    temp_quant_configs[key]['num_fractional_bits'] = w_nfb
                    temp_quant_configs[key]['quantization_function_tf_class'] =
default_rounding_scheme_class
```

```
 607.
 608.             # 2. Configure all activations with the current activation NFB
 609.             for key in all_activation_keys:
 610.                 if key in temp_quant_configs:
 611.                     temp_quant_configs[key]['num_fractional_bits'] = a_nfb
 612.                     temp_quant_configs[key]['quantization_function_tf_class'] =
default_rounding_scheme_class
 613.
 614.             # 3. Update the special 'digit_caps_internal' nested dictionary
 615.             digit_caps_nested_config = {}
 616.             for simple_k, conceptual_k in simple_to_conceptual_map_digitcaps.items():
 617.                 if conceptual_k in temp_quant_configs:
 618.                     digit_caps_nested_config[simple_k] = temp_quant_configs[conceptual_k]
 619.             temp_quant_configs['digit_caps_internal'] = digit_caps_nested_config
 620.
 621.             # 4. Build and evaluate the quantized model with this configuration
 622.             q_model_candidate = build_quantized_model_from_config(
 623.                 fp32_model_ref, model_arch_params, temp_quant_configs,
default_rounding_scheme_class)
 624.
 625.             current_loss = float('inf')
 626.             if q_model_candidate:
 627.                 # Use your existing evaluation function
 628.                 current_loss = evaluate_quantized_model_loss(q_model_candidate,
test_dataset)
 629.
 630.             print(f"Resulting MSE Loss: {current_loss:.6f}")
 631.             loss_matrix[i, j] = current_loss
 632.
 633.     # --- Plotting the heatmap ---
 634.     print("\n--- Plotting Sensitivity Heatmap ---")
 635.     plt.figure(figsize=(12, 10))
 636.
 637.     # We can show the absolute loss or the increase from baseline
 638.     # Let's show the increase in loss for better color contrast
 639.     loss_increase_matrix = loss_matrix - baseline_loss
 640.
 641.     sns.heatmap(loss_increase_matrix, annot=True, fmt=".4f", cmap="viridis_r",
 642.                 xticklabels=activation_nfb_range,
 643.                 yticklabels=weight_nfb_range)
 644.
 645.     plt.xlabel("Activations NFB (Number of Fractional Bits)")
 646.     plt.ylabel("Weights NFB (Number of Fractional Bits)")
 647.     plt.title(f"Sensitivity Heatmap: Increase in MSE Loss from Full-Precision Model
Baseline (Loss={baseline_loss:.6f})")
 648.     plt.show()
 649.
 650. # --- MAIN Q-CAPSNETS ALGORITHM FUNCTION ---
 651. def run_q_capsnets_framework(
 652.     fp32_model_ref, model_arch_parameters, all_q_points_info, calibration_data_loaded,
 653.     num_params_per_weight_map, test_dataset,
 654.     fp32_model_loss_metric, loss_tolerance_percentage, memory_budget_mb,
 655.     default_rounding_scheme_class, initial_search_frac_bits, min_search_frac_bits):
 656.
 657.     print("\n=== Starting Q-CapsNets Framework Algorithm (Loss-based) ===")
 658.     # NEW: Calculate target loss (higher is worse, so quantized loss must be <=
target_loss_max)
 659.     target_loss_max = fp32_model_loss_metric * (1.0 + (loss_tolerance_percentage / 100.0))
 660.     allowed_loss_increase = target_loss_max - fp32_model_loss_metric
 661.     # Step 1 target is a tighter constraint: allow only a small fraction of the total
allowed loss increase
 662.     step1_target_loss_max = fp32_model_loss_metric + (0.05 * allowed_loss_increase)  #
Allow only 5% of the increase
 663.
 664.     print(f"FP32 Loss: {fp32_model_loss_metric:.6f}")
 665.     print(f"Max Tolerated Quantized Loss: {target_loss_max:.6f} (Tolerance:
{loss_tolerance_percentage}%)")
 666.     print(f"Step 1 Max Tolerated Loss: {step1_target_loss_max:.6f}")
 667.     print(f"Memory Budget: {memory_budget_mb:.5f} MB")
 668.
```

```
669.
670.     current_quant_configs = copy.deepcopy(calibration_data_loaded)
671.
672.     simple_to_conceptual_map_digitcaps = {
673.         'inputs_hat': all_q_points_info['caps_tuple_conceptual_keys'][1],
674.         'routing_logits_b': all_q_points_info['caps_tuple_conceptual_keys'][2],
675.         'routing_coeffs_c': all_q_points_info['caps_tuple_conceptual_keys'][3],
676.         'weighted_sum_s': all_q_points_info['caps_tuple_conceptual_keys'][4],
677.         'final_squashed_outputs_v': all_q_points_info['caps_tuple_conceptual_keys'][5],
678.         'squashed_outputs_v_routing_loop':
all_q_points_info['caps_tuple_conceptual_keys'][5]
679.     }
680.
681.     conceptual_layer_weight_keys = [
682.         [k for k in all_q_points_info['weights'] if k.startswith("conv1/")],
683.         [k for k in all_q_points_info['weights'] if k.startswith("conv2/")],
684.         [k for k in all_q_points_info['weights'] if k.startswith("primarycap_conv1d/")],
685.         [k for k in all_q_points_info['weights'] if k.startswith("digitcaps/")],
686.         [k for k in all_q_points_info['weights'] if k.startswith("dense_1/")],
687.         [k for k in all_q_points_info['weights'] if k.startswith("dense_2/")],
688.         [k for k in all_q_points_info['weights'] if k.startswith("output/")]]
689.     conceptual_layer_weight_keys = [lwk_group for lwk_group in
conceptual_layer_weight_keys if lwk_group]
690.
691.
692.     print("\n--- STEP 1: Uniform Quantization Search (Dynamic NFB Search) ---")
693.     best_n_frac_step1 = min_search_frac_bits
694.     loss_at_best_n_frac_step1 = float('inf')  # Initialize with a high loss
695.     found_satisfactory_nfb_s1 = False
696.
697.     for candidate_nfb in range(initial_search_frac_bits, min_search_frac_bits - 1, -1):
698.         print(f"  Step 1: Testing uniform NFB = {candidate_nfb}...")
699.         temp_s1_configs = copy.deepcopy(calibration_data_loaded)
700.         for key_group_name in ['weights', 'activations', 'dr_activations']:
701.             for key in all_q_points_info.get(key_group_name, []):
702.                 if key in temp_s1_configs:
703.                     temp_s1_configs[key]['num_fractional_bits'] = candidate_nfb
704.                     temp_s1_configs[key]['quantization_function_tf_class'] =
default_rounding_scheme_class
705.         digit_caps_nested_config_s1_temp = {}
706.         for simple_k, conceptual_k in simple_to_conceptual_map_digitcaps.items():
707.             if conceptual_k in temp_s1_configs:
708.                 digit_caps_nested_config_s1_temp[simple_k] = temp_s1_configs[conceptual_k]
709.         temp_s1_configs['digit_caps_internal'] = digit_caps_nested_config_s1_temp
710.         q_model_s1_candidate = build_quantized_model_from_config(
711.             fp32_model_ref, model_arch_parameters, temp_s1_configs,
default_rounding_scheme_class)
712.
713.         current_loss_s1_candidate = float('inf')
714.         if q_model_s1_candidate:
715.             current_loss_s1_candidate =
evaluate_quantized_model_loss(q_model_s1_candidate, test_dataset)
716.         print(f"    Step 1: NFB = {candidate_nfb}, Loss =
{current_loss_s1_candidate:.6f}")
717.
718.         if current_loss_s1_candidate <= step1_target_loss_max:
719.             print("    Step 1: Candidate NFB meets the target loss criteria.")
720.             best_n_frac_step1 = candidate_nfb
721.             loss_at_best_n_frac_step1 = current_loss_s1_candidate
722.             found_satisfactory_nfb_s1 = True
723.             print(f"  Step 1: Found satisfactory NFB = {best_n_frac_step1} with Loss =
{loss_at_best_n_frac_step1:.6f}")
724.             break  # Found the highest NFB that meets the tight Step 1 loss target
725.
726.     if not found_satisfactory_nfb_s1:
727.         print(f"  Step 1: No NFB met the target loss {step1_target_loss_max:.6f}.")
728.         best_n_frac_step1 = min_search_frac_bits  # Default to min if nothing met the
criteria
729.         print(f"  Step 1: Defaulting to NFB = {best_n_frac_step1}. Re-evaluating loss for
this NFB if not already last.")
```

```
730.         if not (candidate_nfb == min_search_frac_bits and found_satisfactory_nfb_s1 is
False):  # if last_tested_loss_s1 is not for min_search_frac_bits
731.             temp_s1_configs = copy.deepcopy(calibration_data_loaded)
732.             for key_group_name in ['weights', 'activations', 'dr_activations']:
733.                 for key in all_q_points_info.get(key_group_name, []):
734.                     if key in temp_s1_configs:
735.                         temp_s1_configs[key]['num_fractional_bits'] = best_n_frac_step1
736.                         temp_s1_configs[key]['quantization_function_tf_class'] =
default_rounding_scheme_class
737.             digit_caps_nested_config_s1_temp = {}
738.             for simple_k, conceptual_k in simple_to_conceptual_map_digitcaps.items():
739.                 if conceptual_k in temp_s1_configs:
740.                     digit_caps_nested_config_s1_temp[simple_k] =
temp_s1_configs[conceptual_k]
741.             temp_s1_configs['digit_caps_internal'] = digit_caps_nested_config_s1_temp
742.             q_model_s1_default = build_quantized_model_from_config(
743.                 fp32_model_ref, model_arch_parameters, temp_s1_configs,
default_rounding_scheme_class)
744.             loss_at_best_n_frac_step1 = evaluate_quantized_model_loss(q_model_s1_default,
test_dataset) if q_model_s1_default else float('inf')
745.
746.         print(f"  Step 1: Using NFB = {best_n_frac_step1} with Loss =
{loss_at_best_n_frac_step1:.6f}")
747.
748.     print(f"--- STEP 1 Finished --- Best Uniform Frac Bits: {best_n_frac_step1}, Loss:
{loss_at_best_n_frac_step1:.6f}")
749.
750.     for key_group_name in ['weights', 'activations', 'dr_activations']:
751.         for key in all_q_points_info.get(key_group_name, []):
752.             if key in current_quant_configs:
753.                 current_quant_configs[key]['num_fractional_bits'] = best_n_frac_step1
754.                 current_quant_configs[key]['quantization_function_tf_class'] =
default_rounding_scheme_class
755.     digit_caps_nested_config_current = {}
756.     for simple_k, conceptual_k in simple_to_conceptual_map_digitcaps.items():
757.         if conceptual_k in current_quant_configs:
758.             digit_caps_nested_config_current[simple_k] =
current_quant_configs[conceptual_k]
759.     current_quant_configs['digit_caps_internal'] = digit_caps_nested_config_current
760.
761.     print("\n--- STEP 2: Memory Budget Fulfillment ---")
762.     configs_for_model_memory = copy.deepcopy(current_quant_configs)
763.     n_frac_w_base = best_n_frac_step1
764.     found_mem_bits = False
765.     final_s2_weight_nfb_map = {}
766.
767.     while n_frac_w_base >= min_search_frac_bits:
768.         current_total_weight_bits = 0
769.         temp_iter_weight_nfb_map = {}
770.         for lidx, layer_weight_keys_group in enumerate(conceptual_layer_weight_keys):
771.             current_layer_target_nfb = max(min_search_frac_bits, n_frac_w_base - lidx)
772.             for weight_key in layer_weight_keys_group:
773.                 temp_iter_weight_nfb_map[weight_key] = current_layer_target_nfb
774.                 if weight_key not in configs_for_model_memory:
775.                     print(f"    ERROR (Step 2 Calc): Weight key '{weight_key}' not found
in current_quant_configs. Using default NIB=1.")
776.                     num_integer_bits = 1
777.                 else:
778.                     num_integer_bits =
configs_for_model_memory[weight_key]['num_integer_bits']
779.                 num_params = num_params_per_weight_map.get(weight_key, 0)
780.                 current_total_weight_bits += num_params * (num_integer_bits +
current_layer_target_nfb)
781.         current_memory_mb = current_total_weight_bits / (8 * 1024 * 1024)
782.         if current_memory_mb <= memory_budget_mb:
783.             final_s2_weight_nfb_map = temp_iter_weight_nfb_map
784.             found_mem_bits = True
785.             print(f"  Step 2: Found Weight NFBs meeting budget (base NFB={n_frac_w_base}).
Est. Memory: {current_memory_mb:.4f}MB")
786.             break
```

```
787.         n_frac_w_base -= 1
788.
789.     if not found_mem_bits:
790.         print(f"  Step 2: No weight NFB combination met the memory budget
{memory_budget_mb:.2f}MB with the decreasing rule starting from NFB={best_n_frac_step1}.")
791.         print(f"  Step 2: Defaulting to weights derived from base
NFB={min_search_frac_bits} and decreasing rule.")
792.         for lidx, layer_weight_keys_group in enumerate(conceptual_layer_weight_keys):
793.             current_layer_target_nfb = max(min_search_frac_bits, min_search_frac_bits -
lidx)
794.             for weight_key in layer_weight_keys_group:
795.                 final_s2_weight_nfb_map[weight_key] = current_layer_target_nfb
796.
797.     for weight_key, nfb_val in final_s2_weight_nfb_map.items():
798.         if weight_key in configs_for_model_memory:
799.             configs_for_model_memory[weight_key]['num_fractional_bits'] = nfb_val
800.         else:
801.             print(f"    Warning (Step 2 Apply): Weight key '{weight_key}' not in
configs_for_model_memory to update NFB.")
802.
803.     digit_caps_nested_config_mem = {}
804.     for simple_k, conceptual_k in simple_to_conceptual_map_digitcaps.items():
805.         if conceptual_k in configs_for_model_memory:
806.             digit_caps_nested_config_mem[simple_k] =
configs_for_model_memory[conceptual_k]
807.     configs_for_model_memory['digit_caps_internal'] = digit_caps_nested_config_mem
808.
809.     step2_weight_bits_summary_actual = {k: v['num_fractional_bits'] for k, v in
configs_for_model_memory.items() if v.get('is_weight')}
810.     print(f"  Step 2: Actual weight NFBs being used for model_memory:
{step2_weight_bits_summary_actual}")
811.
812.     print("  Step 2: Building model_memory...")
813.     model_memory = build_quantized_model_from_config(
814.         fp32_model_ref, model_arch_parameters, configs_for_model_memory,
default_rounding_scheme_class)
815.
816.     loss_model_memory = float('inf')
817.     if model_memory:
818.         loss_model_memory = evaluate_quantized_model_loss(model_memory, test_dataset)
819.
820.     final_memory_mb = calculate_memory_footprint(configs_for_model_memory,
all_q_points_info['weights'], num_params_per_weight_map)
821.     print(f"--- STEP 2 Finished --- model_memory: Loss={loss_model_memory:.6f}, Actual
Mem={final_memory_mb:.4f}MB")
822.
823.     model_satisfied = None
824.     configs_satisfied = None
825.     model_loss_final_path_b = None
826.     configs_loss_final_path_b = None
827.
828.     if loss_model_memory <= target_loss_max:
829.         print(f"\n--- Path A Taken: model_memory loss ({loss_model_memory:.6f}) <= target
loss ({target_loss_max:.6f}) ---")
830.         configs_step3a = copy.deepcopy(configs_for_model_memory)
831.         model_step3a = model_memory
832.         loss_step3a = loss_model_memory
833.
834.         print("\n--- STEP 3A: Layer-Wise Quantization of Activations ---")
835.         activation_keys_to_tune = all_q_points_info['activations']
836.
837.         for act_key in activation_keys_to_tune:
838.             if act_key not in configs_step3a or configs_step3a[act_key].get('is_weight',
False):
839.                 continue
840.             original_nfb_act = configs_step3a[act_key]['num_fractional_bits']
841.             print(f"  Step 3A: Tuning activation '{act_key}', current NFB =
{original_nfb_act} (target_loss_max: {target_loss_max:.6f})")
842.
```

```
            for candidate_nfb in range(original_nfb_act - 1, min_search_frac_bits - 1, -
1):
                temp_configs_3a_iter = copy.deepcopy(configs_step3a)
                temp_configs_3a_iter[act_key]['num_fractional_bits'] = candidate_nfb
                is_digitcaps_internal_key = False
                for simple_k, conceptual_k in simple_to_conceptual_map_digitcaps.items():
                    if conceptual_k == act_key:
                        is_digitcaps_internal_key = True
                        if 'digit_caps_internal' not in temp_configs_3a_iter:
                            temp_configs_3a_iter['digit_caps_internal'] = {}
                        if simple_k not in temp_configs_3a_iter['digit_caps_internal'] or
\
                           not
isinstance(temp_configs_3a_iter['digit_caps_internal'].get(simple_k), dict):
                            temp_configs_3a_iter['digit_caps_internal'][simple_k] =
copy.deepcopy(temp_configs_3a_iter[act_key])
                        else:
                            temp_configs_3a_iter['digit_caps_internal'][simple_k]['num_fractional_bits'] = candidate_nfb
                        break
                if is_digitcaps_internal_key:
                    current_digit_caps_nested_config = {}
                    for sk, ck in simple_to_conceptual_map_digitcaps.items():
                        if ck in temp_configs_3a_iter:
                            current_digit_caps_nested_config[sk] =
temp_configs_3a_iter[ck]
                    temp_configs_3a_iter['digit_caps_internal'] =
current_digit_caps_nested_config

                temp_model_3a_iter = build_quantized_model_from_config(
                    fp32_model_ref, model_arch_parameters, temp_configs_3a_iter,
default_rounding_scheme_class)

                loss_temp_3a_iter = float('inf')
                if temp_model_3a_iter:
                    loss_temp_3a_iter = evaluate_quantized_model_loss(temp_model_3a_iter,
test_dataset)

                if loss_temp_3a_iter <= target_loss_max:
                    configs_step3a = temp_configs_3a_iter
                    model_step3a = temp_model_3a_iter
                    loss_step3a = loss_temp_3a_iter
                    print(f"    Step 3A: Accepted NFB={candidate_nfb} for '{act_key}'. New
Loss: {loss_step3a:.6f}")
                else:
                    print(f"    Step 3A: Rejected NFB={candidate_nfb} for '{act_key}'
(Loss {loss_temp_3a_iter:.6f} > {target_loss_max:.6f}). Keeping previous NFB.")
                    break

        print(f"--- STEP 3A Finished --- Final Loss after Act Quant: {loss_step3a:.6f}")
        configs_step4a = copy.deepcopy(configs_step3a)
        model_step4a = model_step3a
        loss_step4a = loss_step3a

        print("\n--- STEP 4A: Dynamic Routing Quantization ---")
        dr_activation_keys_to_tune = all_q_points_info['dr_activations']
        for dr_act_key in dr_activation_keys_to_tune:
            if dr_act_key not in configs_step4a or
configs_step4a[dr_act_key].get('is_weight', False):
                continue
            original_nfb_dr_act = configs_step4a[dr_act_key]['num_fractional_bits']
            print(f"  Step 4A: Tuning DR activation '{dr_act_key}', current NFB =
{original_nfb_dr_act} (target_loss_max: {target_loss_max:.6f})")

            for candidate_nfb in range(original_nfb_dr_act - 1, min_search_frac_bits - 1,
-1):
                temp_configs_4a_iter = copy.deepcopy(configs_step4a)
                temp_configs_4a_iter[dr_act_key]['num_fractional_bits'] = candidate_nfb
                for simple_k, conceptual_k in simple_to_conceptual_map_digitcaps.items():
                    if conceptual_k == dr_act_key:
```

```
 899.                         if 'digit_caps_internal' not in temp_configs_4a_iter:
 900.                             temp_configs_4a_iter['digit_caps_internal'] = {}
 901.                         if simple_k not in temp_configs_4a_iter['digit_caps_internal'] or
\
 902.                            not
isinstance(temp_configs_4a_iter['digit_caps_internal'].get(simple_k), dict):
 903.                             temp_configs_4a_iter['digit_caps_internal'][simple_k] =
copy.deepcopy(temp_configs_4a_iter[dr_act_key])
 904.                         else:
 905.
temp_configs_4a_iter['digit_caps_internal'][simple_k]['num_fractional_bits'] = candidate_nfb
 906.                         break
 907.                 current_digit_caps_nested_config_4a = {}
 908.                 for sk, ck in simple_to_conceptual_map_digitcaps.items():
 909.                     if ck in temp_configs_4a_iter:
 910.                         current_digit_caps_nested_config_4a[sk] = temp_configs_4a_iter[ck]
 911.                 temp_configs_4a_iter['digit_caps_internal'] =
current_digit_caps_nested_config_4a
 912.
 913.                 temp_model_4a_iter = build_quantized_model_from_config(
 914.                     fp32_model_ref, model_arch_parameters, temp_configs_4a_iter,
default_rounding_scheme_class)
 915.
 916.                 loss_temp_4a_iter = float('inf')
 917.                 if temp_model_4a_iter:
 918.                     loss_temp_4a_iter = evaluate_quantized_model_loss(temp_model_4a_iter,
test_dataset)
 919.
 920.                 if loss_temp_4a_iter <= target_loss_max:
 921.                     configs_step4a = temp_configs_4a_iter
 922.                     model_step4a = temp_model_4a_iter
 923.                     loss_step4a = loss_temp_4a_iter
 924.                     print(f"    Step 4A: Accepted NFB={candidate_nfb} for DR
'{dr_act_key}'. New Loss: {loss_step4a:.6f}")
 925.                 else:
 926.                     print(f"    Step 4A: Rejected NFB={candidate_nfb} for DR
'{dr_act_key}' (Loss {loss_temp_4a_iter:.6f} > {target_loss_max:.6f}). Keeping previous NFB.")
 927.                     break
 928.
 929.         print(f"--- STEP 4A Finished --- Final Loss after DR Quant: {loss_step4a:.6f}")
 930.         model_satisfied = model_step4a
 931.         configs_satisfied = copy.deepcopy(configs_step4a)
 932.         loss_satisfied = loss_step4a  # Final loss for model_satisfied
 933.
 934.         final_mem_satisfied = calculate_memory_footprint(configs_satisfied,
all_q_points_info['weights'], num_params_per_weight_map)
 935.         print(f"Path A Result: model_satisfied Loss={loss_satisfied:.6f},
Mem={final_mem_satisfied:.4f}MB")
 936.
 937.     else:  # Path B: loss_model_memory > target_loss_max
 938.         print(f"\n--- Path B Taken: model_memory loss ({loss_model_memory:.6f}) > target
loss ({target_loss_max:.6f}) ---")
 939.         print("--- STEP 3B: Finding Weight NFBs to meet loss target ---")
 940.
 941.         activation_nfb_for_path_b = initial_search_frac_bits  # Use a high-quality NFB for
activations
 942.         print(f"  Step 3B: Setting activations to NFB={activation_nfb_for_path_b} (using
initial_search_frac_bits as high-quality baseline).")
 943.
 944.         configs_3b = copy.deepcopy(calibration_data_loaded)
 945.         for key_group_name_act in ['activations', 'dr_activations']:
 946.             for act_key in all_q_points_info.get(key_group_name_act, []):
 947.                 if act_key in configs_3b:
 948.                     configs_3b[act_key]['num_fractional_bits'] = activation_nfb_for_path_b
 949.                     configs_3b[act_key]['quantization_function_tf_class'] =
default_rounding_scheme_class
 950.
 951.         digit_caps_nested_config_3b_acts = {}
 952.         for simple_k, conceptual_k in simple_to_conceptual_map_digitcaps.items():
 953.             if conceptual_k in configs_3b:
```

```
                digit_caps_nested_config_3b_acts[simple_k] = configs_3b[conceptual_k]
        configs_3b['digit_caps_internal'] = digit_caps_nested_config_3b_acts

        found_uniform_weights_nfb_met_target = False
        uniform_nfb_for_weights_target_met = -1
        best_loss_phase1 = float('inf')
        configs_at_best_loss_phase1 = copy.deepcopy(configs_3b)

        print(f"  Step 3B (Phase 1): Searching for uniform weight NFB to meet
target_loss_max {target_loss_max:.6f} (Activations fixed at NFB={activation_nfb_for_path_b})")

        for candidate_uniform_w_nfb in range(min_search_frac_bits,
initial_search_frac_bits + 1):  # Iterate upwards for weights
            print(f"    Step 3B: Testing uniform NFB={candidate_uniform_w_nfb} for all
weights...")
            temp_configs_uniform_w = copy.deepcopy(configs_3b)
            for weight_key in all_q_points_info['weights']:
                if weight_key in temp_configs_uniform_w:
                    temp_configs_uniform_w[weight_key]['num_fractional_bits'] =
candidate_uniform_w_nfb
                    if 'quantization_function_tf_class' not in
temp_configs_uniform_w[weight_key]:

temp_configs_uniform_w[weight_key]['quantization_function_tf_class'] =
default_rounding_scheme_class

            temp_digit_caps_nested_config = {}
            for simple_k, conceptual_k in simple_to_conceptual_map_digitcaps.items():
                if conceptual_k in temp_configs_uniform_w:
                    temp_digit_caps_nested_config[simple_k] =
temp_configs_uniform_w[conceptual_k]
            temp_configs_uniform_w['digit_caps_internal'] = temp_digit_caps_nested_config

            model_uniform_w = build_quantized_model_from_config(
                fp32_model_ref, model_arch_parameters, temp_configs_uniform_w,
default_rounding_scheme_class)
            loss_uniform_w = evaluate_quantized_model_loss(model_uniform_w, test_dataset)
if model_uniform_w else float('inf')
            print(f"      Loss with uniform NFB={candidate_uniform_w_nfb} for weights
(Activations NFB={activation_nfb_for_path_b}): {loss_uniform_w:.6f}")

            if loss_uniform_w < best_loss_phase1:
                best_loss_phase1 = loss_uniform_w
                configs_at_best_loss_phase1 = copy.deepcopy(temp_configs_uniform_w)

            if loss_uniform_w <= target_loss_max:
                print(f"    Step 3B: Uniform NFB={candidate_uniform_w_nfb} for weights MET
target_loss_max ({target_loss_max:.6f}) with Loss={loss_uniform_w:.6f}")
                uniform_nfb_for_weights_target_met = candidate_uniform_w_nfb
                configs_3b = temp_configs_uniform_w
                found_uniform_weights_nfb_met_target = True
                break

        if not found_uniform_weights_nfb_met_target:
            print(f"  Step 3B (Phase 1): No uniform weight NFB (up to
{initial_search_frac_bits}) met target_loss_max {target_loss_max:.6f} with activations at
NFB={activation_nfb_for_path_b}.")
            print(f"  Using best configuration found in Phase 1
(Loss={best_loss_phase1:.6f}).")
            configs_3b = configs_at_best_loss_phase1

        loss_3b_current = evaluate_quantized_model_loss(
            build_quantized_model_from_config(fp32_model_ref, model_arch_parameters,
configs_3b, default_rounding_scheme_class),
            test_dataset
        )
        print(f"  Step 3B: After Phase 1 (Uniform Weights), Loss is
{loss_3b_current:.6f}")

        if found_uniform_weights_nfb_met_target:
```

```
            print(f"\n  Step 3B (Phase 2): Optimizing weights layer-wise from uniform
NFB={uniform_nfb_for_weights_target_met} (Current Loss: {loss_3b_current:.6f})")
            for layer_keys_group in conceptual_layer_weight_keys:
                for weight_key in layer_keys_group:
                    if weight_key not in configs_3b:
                        continue
                    original_nfb_weight = configs_3b[weight_key]['num_fractional_bits']
                    print(f"    Step 3B (Phase 2): Tuning weight '{weight_key}', current
NFB = {original_nfb_weight}")

                    for candidate_nfb in range(original_nfb_weight - 1,
min_search_frac_bits - 1, -1):
                        temp_configs_3b_iter = copy.deepcopy(configs_3b)
                        temp_configs_3b_iter[weight_key]['num_fractional_bits'] =
candidate_nfb
                        temp_model_3b_iter = build_quantized_model_from_config(
                            fp32_model_ref, model_arch_parameters, temp_configs_3b_iter,
default_rounding_scheme_class)
                        loss_temp_3b_iter =
evaluate_quantized_model_loss(temp_model_3b_iter, test_dataset) if temp_model_3b_iter else
float('inf')

                        if loss_temp_3b_iter <= target_loss_max:
                            configs_3b = temp_configs_3b_iter
                            loss_3b_current = loss_temp_3b_iter
                            print(f"      Step 3B (Phase 2): Accepted NFB={candidate_nfb}
for weight '{weight_key}'. New Loss: {loss_3b_current:.6f}")
                        else:
                            print(f"      Step 3B (Phase 2): Rejected NFB={candidate_nfb}
for weight '{weight_key}' (Loss {loss_temp_3b_iter:.6f} > {target_loss_max:.6f}). Keeping
NFB={configs_3b[weight_key]['num_fractional_bits']}.")
                            break
        else:
            print(f"  Step 3B (Phase 2): Skipped layer-wise weight optimization because
Phase 1 did not meet target_loss_max ({target_loss_max:.6f}).")

        model_loss_final_path_b = build_quantized_model_from_config(
            fp32_model_ref, model_arch_parameters, configs_3b,
default_rounding_scheme_class)
        configs_loss_final_path_b = copy.deepcopy(configs_3b)
        loss_model_loss_final_path_b =
evaluate_quantized_model_loss(model_loss_final_path_b, test_dataset) if model_loss_final_path_b
else float('inf')

        final_mem_model_loss_path_b =
calculate_memory_footprint(configs_loss_final_path_b, all_q_points_info['weights'],
num_params_per_weight_map)
        print(f"--- STEP 3B Finished --- model_loss_final (Path B):
Loss={loss_model_loss_final_path_b:.6f}, Mem={final_mem_model_loss_path_b:.4f}MB")

        print("Path B Results: ")
        print(f"  model_memory (met budget from Step 2, but not loss target):
Loss={loss_model_memory:.6f}, Mem={final_memory_mb:.4f}MB")
        print(f"  model_loss_final (aimed for loss target in Step 3B):
Loss={loss_model_loss_final_path_b:.6f}, Mem={final_mem_model_loss_path_b:.4f}MB")


    print("\n--- Q-CapsNets Framework Algorithm Finished ---")

    if model_satisfied:
        print("Returning model_satisfied (from Path A)")
        return model_satisfied, configs_satisfied, None, None
    else:
        print("Returning model_memory and model_loss_final_path_b (from Path B)")
        return model_memory, configs_for_model_memory, model_loss_final_path_b,
configs_loss_final_path_b


# --- Main Execution Block ---
if __name__ == "__main__":
```

```
    print("--- Q-CapsNets Framework Script (Loss-based, Unscaled Data) ---")
    print(f"TensorFlow Version: {tf.__version__}")

    fp32_model_instance, all_q_points_info, num_params_map = None, None, None
    calibration_ds, test_ds = None, None
    calibration_data_loaded = {}

    try:
        print(f"\nAttempting to load FP32 model from: {FP32_MODEL_PATH}")
        fp32_model_instance = load_fp32_model(FP32_MODEL_PATH)
        fp32_model_instance.summary(line_length=120)
        all_q_points_info = get_tensor_identifiers(fp32_model_instance)
        if not all_q_points_info['weights']:
            print("CRITICAL WARNING: No weight keys identified.")
        else:
            num_params_map = get_num_parameters_for_weights(fp32_model_instance,
all_q_points_info['weights'])

        print("\nLoading and preparing data (UNSCALED)...")
        val_full_raw_data = load_csv_data_from_path(PARENT_DIR_CSV, FILE_VALIDATION_CSV)
        X_val_raw, Y_val_raw = val_full_raw_data[:, :MODEL_ARCH_PARAMS['input_shape'][1]],
val_full_raw_data[:, MODEL_ARCH_PARAMS['input_shape'][1]:]
        test_full_raw_data = load_csv_data_from_path(PARENT_DIR_CSV, FILE_TESTING_CSV)
        X_test_raw, Y_test_raw = test_full_raw_data[:,
:MODEL_ARCH_PARAMS['input_shape'][1]], test_full_raw_data[:,
MODEL_ARCH_PARAMS['input_shape'][1]:]
        train_full_raw_data = load_csv_data_from_path(PARENT_DIR_CSV, FILE_TRAINING_CSV)
        X_train_raw, Y_train_raw = train_full_raw_data[:,
:MODEL_ARCH_PARAMS['input_shape'][1]], train_full_raw_data[:,
MODEL_ARCH_PARAMS['input_shape'][1]:]

        calibration_ds = create_tf_dataset_from_array(X_val_raw, Y_val_raw,
INPUT_SEQUENCE_LENGTH, OUTPUT_SEQUENCE_LENGTH, CALIBRATION_BATCH_SIZE, shuffle=False)
        test_ds = create_tf_dataset_from_array(X_test_raw, Y_test_raw,
INPUT_SEQUENCE_LENGTH, OUTPUT_SEQUENCE_LENGTH, TEST_BATCH_SIZE, shuffle=False)
        train_ds = create_tf_dataset_from_array(X_train_raw, Y_train_raw,
INPUT_SEQUENCE_LENGTH, OUTPUT_SEQUENCE_LENGTH, TEST_BATCH_SIZE, shuffle=False)
        print("Calibration and Test datasets created from raw data.")
        print(f"  Calibration dataset size:
{tf.data.experimental.cardinality(calibration_ds).numpy()} batches")
        print(f"  Test dataset size: {tf.data.experimental.cardinality(test_ds).numpy()}
batches")

    except Exception as e:
        print(f"ERROR during initial setup (loading model, data): {e}")
        import traceback
        traceback.print_exc()
        fp32_model_instance = None

    if fp32_model_instance and calibration_ds and all_q_points_info and num_params_map and
test_ds and tf.data.experimental.cardinality(calibration_ds).numpy() > 0:
        if os.path.exists(CALIBRATION_RESULTS_PATH):
            print(f"\nAttempting to load existing calibration results from
{CALIBRATION_RESULTS_PATH}")
            try:
                with open(CALIBRATION_RESULTS_PATH, 'r') as f:
                    calibration_data_loaded = json.load(f)
                print(f"Successfully loaded calibration data. Number of keys:
{len(calibration_data_loaded)}.")
            except Exception as e:
                print(f"Error loading calibration file: {e}. Will run new calibration.")
                calibration_data_loaded = {}

        if not calibration_data_loaded:
            print("\nPerforming new calibration as no valid data was loaded or re-
calibration forced...")
            if CALIBRATION_NUM_BATCHES > 0:
                calibration_data_loaded = perform_calibration(fp32_model_instance,
calibration_ds, all_q_points_info, CALIBRATION_NUM_BATCHES)
            else:
```

```
                print("Skipping new calibration as CALIBRATION_NUM_BATCHES is 0. Creating
dummy calib data.")
                calibration_data_loaded = {}
                for key_type, key_list in all_q_points_info.items():
                    if key_type == 'caps_tuple_conceptual_keys':
                        continue
                    for k_item in key_list:
                        calibration_data_loaded[k_item] = {'clip_min': -1., 'clip_max':
1., 'num_integer_bits': 1, 'is_weight': (key_type == 'weights')}

        if calibration_data_loaded and tf.data.experimental.cardinality(test_ds).numpy() >
0:
            print("\n--- Calling Q-CapsNets Framework (Loss-based, Step 1 Focus) ---")

            # #-------------------------------------------------
            # ####### ******* IMORTANT NOTE ******* ########
            # #### **** REASON ::: Because calibration function just run prob model for
small number of bathces
            # #### So usually internal digitcaps range is between -0.1 and 0.1 or 0 and 1
            # ###### But it also may have spikes until 2 -2 or 4 and -4 so it can't
capture them
            # ##### for all dataset because we run it for small number of batches
            #
            # ################ This part just for testing the manually wider clipping ranges
for sensitive tensors *********#######
            # ######## To see if everything related to the digitcaps internal tensors is
working as expected. #############
            # ######## And check if the MSE drop is still there with wider ranges for
internals. ########
            # ######### This could make us sure that qmodel,layers and fp32 model and
layer and finally primitives
            # ######### are working as expected. #########

            print("\n!!! EXPERIMENT: Applying WIDER clipping ranges for specific sensitive
tensors !!!")

            sensitive_keys_to_widen = [
                "digitcaps_internal_inputs_hat",
                "digitcaps_internal_routing_logits_b",
                "digitcaps_internal_routing_coeffs_c",
                "digitcaps_internal_weighted_sum_s",
                "digitcaps_internal_squashed_v_loop",
                "digitcaps_layer_output",
            ]

            new_clip_min = -3.6  # Example generous range
            new_clip_max = 3.6   # Example generous range

            for key_to_modify in sensitive_keys_to_widen:
                if key_to_modify in calibration_data_loaded:
                    print(f"  Overriding clipping for '{key_to_modify}': From
[{calibration_data_loaded[key_to_modify]['clip_min']:.4f},
{calibration_data_loaded[key_to_modify]['clip_max']:.4f}] to [{new_clip_min}, {new_clip_max}]")
                    calibration_data_loaded[key_to_modify]['clip_min'] = new_clip_min
                    calibration_data_loaded[key_to_modify]['clip_max'] = new_clip_max
                    calibration_data_loaded[key_to_modify]['num_integer_bits'] = 3
                else:
                    print(f"  Warning: Sensitive key '{key_to_modify}' not found in
calibration_data_loaded for override.")
            print("!!! EXPERIMENT: Finished applying wider clipping ranges. !!!\n")

            # #-----------------------------------

            fp32_actual_loss = evaluate_fp32_model_loss(fp32_model_instance, test_ds)
            fp32_actual_loss_train = evaluate_fp32_model_loss(fp32_model_instance,
train_ds)
            fp32_actual_loss_val = evaluate_fp32_model_loss(fp32_model_instance,
calibration_ds)

            current_fp32_loss_for_framework = fp32_actual_loss_val
```

```
1166.
1167.             if current_fp32_loss_for_framework == float('inf') or
np.isnan(current_fp32_loss_for_framework):
1168.                 print(f"ERROR: FP32 model evaluation returned invalid loss
({current_fp32_loss_for_framework}). Cannot proceed.")
1169.             else:
1170.                 print(f"Using FP32 model loss: {current_fp32_loss_for_framework:.6f} as
baseline.")
1171.
1172.                 chosen_scheme_name = "round_to_nearest"
1173.                 chosen_scheme_class = ROUNDING_SCHEMES_TO_TRY[chosen_scheme_name]
1174.                 print(f"Using rounding scheme: {chosen_scheme_name}")
1175.
1176.                 # ============================================================
1177.                 # ===== NEW: GENERATE AND DISPLAY SENSITIVITY HEATMAP =====
1178.                 # ============================================================
1179.                 # You can control whether to run this with a simple flag
1180.                 RUN_SENSITIVITY_ANALYSIS = True
1181.                 if RUN_SENSITIVITY_ANALYSIS:
1182.                     # Define the range of NFB values you want to test for the heatmap
1183.                     nfb_heatmap_range = range(MIN_SEARCH_FRACTIONAL_BITS,
INITIAL_SEARCH_FRACTIONAL_BITS + 1)
1184.
1185.                     generate_nfb_sensitivity_heatmap(
1186.                         fp32_model_ref=fp32_model_instance,
1187.                         model_arch_params=MODEL_ARCH_PARAMS,
1188.                         all_q_points_info=all_q_points_info,
1189.                         calibration_data_loaded=calibration_data_loaded,
1190.                         test_dataset=calibration_ds,  # Use validation set for consistency
with framework
1191.                         baseline_loss=current_fp32_loss_for_framework,
1192.                         nfb_range=nfb_heatmap_range,
1193.                         default_rounding_scheme_class=chosen_scheme_class
1194.                     )
1195.                 # ============================================================
1196.                 # ============================================================
1197.
1198.                 current_initial_search_frac_bits = INITIAL_SEARCH_FRACTIONAL_BITS
1199.                 current_min_search_frac_bits = MIN_SEARCH_FRACTIONAL_BITS
1200.                 print(f"Step 1 NFB Search Range:
Initial={current_initial_search_frac_bits}, Min={current_min_search_frac_bits}")
1201.
1202.                 # Call framework with loss parameters
1203.                 model1, configs1, model2, configs2 = run_q_capsnets_framework(
1204.                     fp32_model_ref=fp32_model_instance,
1205.                     model_arch_parameters=MODEL_ARCH_PARAMS,
1206.                     all_q_points_info=all_q_points_info,
1207.                     calibration_data_loaded=calibration_data_loaded,
1208.                     num_params_per_weight_map=num_params_map,
1209.                     test_dataset=calibration_ds,
1210.                     fp32_model_loss_metric=current_fp32_loss_for_framework,
1211.                     loss_tolerance_percentage=LOSS_TOLERANCE_PERCENT,
1212.                     memory_budget_mb=MEMORY_BUDGET_MB,
1213.                     default_rounding_scheme_class=chosen_scheme_class,
1214.                     initial_search_frac_bits=current_initial_search_frac_bits,
1215.                     min_search_frac_bits=current_min_search_frac_bits
1216.                 )
1217.
1218.                 print("\n--- Framework Execution Summary ---")
1219.                 final_model_to_print_details = None
1220.                 final_configs_to_print_details = None
1221.                 final_model_label = ""
1222.                 path_taken_message = ""
1223.
1224.                 if model2 and configs2:  # Path B was taken, model2 is
model_loss_final_path_b
1225.                     path_taken_message = "Path B was taken."
1226.                     print(path_taken_message)
1227.
1228.                     final_model_to_print_details = model2
```

```
                    final_configs_to_print_details = configs2
                    final_model_label = "model_loss_final (Path B - Aimed for Loss
Target)"

                    if model1 and configs1:  # model1 is model_memory from Path B
                        loss_model_memory = evaluate_quantized_model_loss(model1, test_ds)
if model1 else float('inf')
                        loss_model_memory_train = evaluate_quantized_model_loss(model1,
train_ds)
                        loss_model_memory_val = evaluate_quantized_model_loss(model1,
calibration_ds)
                        mem_model_memory = calculate_memory_footprint(configs1,
all_q_points_info['weights'], num_params_map) if configs1 else 0.0
                        print("  Summary for model_memory (Path B - Met Memory Budget):")
                        print(f"    Test Loss: {loss_model_memory:.6f}, Train Loss:
{loss_model_memory_train} , Memory: {mem_model_memory:.4f} MB")
                        print(f" validation loss: {loss_model_memory_val:.6f}")

                        ##### SAVE PATH B RETURNED MODEL #####
                        print(f"\n--- Saving final Path B quantized model to: {save_path}
---")
                        try:
                            final_model_to_print_details.save(save_path)
                            print("Model saved successfully.")
                        except Exception as e:
                            print(f"Error saving model: {e}")
                    else:
                        print("  model_memory (Path B) data not available.")

                elif model1 and configs1:  # Path A was taken, model1 is model_satisfied
                    path_taken_message = "Path A was taken."
                    print(path_taken_message)
                    final_model_to_print_details = model1
                    final_configs_to_print_details = configs1
                    final_model_label = "model_satisfied (Path A)"

                if final_model_to_print_details and final_configs_to_print_details:
                    print(f"\n--- Detailed Configuration for Final Output Model:
{final_model_label} ---")
                    loss_final_model =
evaluate_quantized_model_loss(final_model_to_print_details, test_ds)
                    loss_final_model_train_A =
evaluate_quantized_model_loss(final_model_to_print_details, train_ds)
                    loss_final_model_val =
evaluate_quantized_model_loss(final_model_to_print_details, calibration_ds)
                    mem_final_model =
calculate_memory_footprint(final_configs_to_print_details, all_q_points_info['weights'],
num_params_map)
                    print(f"  Overall Performance of '{final_model_label}':")
                    print(f"    Test  Loss (MSE): {loss_final_model:.6f}")
                    print(f"    Train Loss (MSE): {loss_final_model_train_A:.6f}")
                    print(f"  Validation loss (MSE): {loss_final_model_val:.6f}")
                    print(f"     Memory:      {mem_final_model:.4f} MB")
                    print(f"  Reference FP32 Loss (MSE):
{current_fp32_loss_for_framework:.6f}")
                    print(f"  Reference FP32 Train Loss (MSE):
{fp32_actual_loss_train:.6f}")
                    print(f"  Reference FP32 Validation loss (MSE): {fp32_actual_loss_val}
")
                    print(f"  Reference FP32 Test Loss (MSE): {fp32_actual_loss:.6f}")

                    ##### SAVE PATH A RETURNED MODEL #####
                    print(f"\n--- Saving final Path A quantized model to: {save_path} ---
")
                    try:
                        final_model_to_print_details.save(save_path)
                        print("Model saved successfully.")
                    except Exception as e:
                        print(f"Error saving model: {e}")

```

```
                    # ================================================================ #
                    # ===== NEW: SAVING THE FINAL MODEL AND WEIGHTS IN ALL FORMATS ==== #
                    # ================================================================ #

                    # Define paths for the new weight files
                    weights_keras_path = os.path.join(SCRIPT_DIRECTORY,
"final_quantized_model.weights.h5")
                    weights_csv_path = os.path.join(SCRIPT_DIRECTORY,
"final_quantized_weights_for_hls.csv")

                    print("\n--- Saving final model and weights ---")
                    try:
                        # 1. Save the full model (architecture + weights)
                        print(f"Saving full model to: {save_path}")
                        final_model_to_print_details.save(save_path)
                        print("Full model saved successfully.")

                        # 2. Save just the weights in Keras format
                        print(f"Saving weights to: {weights_keras_path}")
                        final_model_to_print_details.save_weights(weights_keras_path)
                        print("Keras weights (.weights.h5) saved successfully.")

                        # 3. Save weights to CSV using our new helper function
                        save_weights_to_csv(final_model_to_print_details,
weights_csv_path)

                    except Exception as e:
                        print(f"An error occurred during final model/weight saving: {e}")
                        import traceback
                        traceback.print_exc()

                    print("\n--- Saving final quantization configurations to JSON ---")
                    configs_for_json = {}
                    if not isinstance(final_configs_to_print_details, dict):
                        print(f"  ERROR: final_configs_to_print_details is not a
dictionary! Type: {type(final_configs_to_print_details)}")
                    else:
                        for key, config_dict_value in
final_configs_to_print_details.items():
                            if key == 'digit_caps_internal':
                                configs_for_json[key] = {}
                                if not isinstance(config_dict_value, dict):
                                    print(f"  WARNING: Value for 'digit_caps_internal'
(key: {key}) is not a dict, it's a {type(config_dict_value)}. Storing as string.")
                                    configs_for_json[key] = str(config_dict_value)
                                    continue
                                for inner_key, specific_tensor_config_object in
config_dict_value.items():
                                    if isinstance(specific_tensor_config_object, dict):
                                        serializable_inner_config = {}
                                        for k_param, v_param in
specific_tensor_config_object.items():
                                            if k_param ==
'quantization_function_tf_class':
                                                serializable_inner_config[k_param] =
v_param.__name__ if hasattr(v_param, '__name__') and v_param is not None else str(v_param)
                                            else:
                                                serializable_inner_config[k_param] =
v_param
                                        configs_for_json[key][inner_key] =
serializable_inner_config
                                    else:
                                        print(f"  ERROR IN CONFIG STRUCTURE: Value for
'digit_caps_internal/{inner_key}' was expected to be a dict, but found type:
{type(specific_tensor_config_object)}.")
                                        configs_for_json[key][inner_key] = f"ERROR:
Expected dict, got {type(specific_tensor_config_object).__name__} with value
{str(specific_tensor_config_object)}"
                            else:
                                if isinstance(config_dict_value, dict):
```

```
                                    serializable_config = {}
                                    for k_param, v_param in config_dict_value.items():
                                        if k_param == 'quantization_function_tf_class':
                                            serializable_config[k_param] =
v_param.__name__ if hasattr(v_param, '__name__') and v_param is not None else str(v_param)
                                        else:
                                            serializable_config[k_param] = v_param
                                    configs_for_json[key] = serializable_config
                                else:
                                    print(f"  WARNING: Value for top-level key '{key}' is
not a dict, it's a {type(config_dict_value)}. Storing as string.")
                                    configs_for_json[key] = str(config_dict_value)

                        print("\n  Weight Configurations:")
                        if 'weights' in all_q_points_info:
                            for weight_key in sorted(all_q_points_info['weights']):
                                if weight_key in configs_for_json:
                                    cfg = configs_for_json[weight_key]
                                    print(f"    - {weight_key}:")
                                    print(f"        NFB={cfg.get('num_fractional_bits',
'N/A')}, NIB={cfg.get('num_integer_bits', 'N/A')}, ClipMin={cfg.get('clip_min', 'N/A'):.4f},
ClipMax={cfg.get('clip_max', 'N/A'):.4f},
Scheme={cfg.get('quantization_function_tf_class','N/A')}")
                                else:
                                    print(f"    - {weight_key}: Not found in final
configurations.")
                        else:
                            print("    No weight keys found in all_q_points_info.")

                        print("\n  Activation Configurations (includes DR activations):")
                        if 'activations' in all_q_points_info:
                            for act_key in sorted(all_q_points_info['activations']):
                                if act_key in configs_for_json:
                                    cfg = configs_for_json[act_key]
                                    print(f"    - {act_key}:")
                                    print(f"        NFB={cfg.get('num_fractional_bits',
'N/A')}, NIB={cfg.get('num_integer_bits', 'N/A')}, ClipMin={cfg.get('clip_min', 'N/A'):.4f},
ClipMax={cfg.get('clip_max', 'N/A'):.4f},
Scheme={cfg.get('quantization_function_tf_class','N/A')}")
                                else:
                                    print(f"    - {act_key}: Not found in final
configurations.")
                        else:
                            print("    No activation keys found in all_q_points_info.")

                        output_config_filename =
"final_quantization_config_for_fpga_loss_based.json"
                        output_config_path = os.path.join(SCRIPT_DIRECTORY,
output_config_filename)
                        try:
                            with open(output_config_path, 'w') as f_json:
                                json.dump(configs_for_json, f_json, indent=4,
sort_keys=True)
                            print(f"\n  Successfully saved detailed quantization
configurations to: {output_config_path}")
                        except Exception as e:
                            print(f"\n  Error saving quantization configurations to JSON:
{e}")
                else:
                    if not path_taken_message:
                        print("Framework did not successfully produce a final model to
detail from Path A or Path B.")
                    elif not (final_model_to_print_details and
final_configs_to_print_details):
                        print(f"Model or configuration data missing for the determined
path ({path_taken_message}). Cannot print details.")
            else:
                print("Skipping main framework run: missing calibration data or test dataset
is empty.")
        else:
```

```
        print("FP32 model/data setup failed, or calibration dataset is empty. Cannot proceed with quantization framework.")

    print("\n--- Main script execution finished ---")
```

10.2. High Level Synthesis (HLS) Codes:

10.2.1. Configuration Header:

```
#ifndef MY_CAPSNET_CONFIG_H
#define MY_CAPSNET_CONFIG_H

#include "ap_fixed.h"

// ============================================================================
// == 1. Architectural Parameters
// ============================================================================
const int INPUT_SEQ_LEN    = 15;
const int INPUT_FEATURES   = 4;
const int N_CLASS            = 2;
const int ROUTINGS           = 3;
const int NB_FILTERS_CONV1_CONV2 = 16;
const int KERNEL_SIZE_ALL_CONV   = 2;
const int DIM_CAPSULE_CAPS1      = 7;
const int NUM_CAPSULE_CAPS1_CHANNELS = 10;
const int DIM_CAPSULE_CAPS2      = 7;
const int DENSE_UNIT           = 16;

// ============================================================================
// == 2. Calculated Dimensions
// ============================================================================
const int CONV1_OUT_SEQ_LEN   = INPUT_SEQ_LEN - KERNEL_SIZE_ALL_CONV + 1;       // 14
const int CONV2_OUT_SEQ_LEN   = CONV1_OUT_SEQ_LEN - KERNEL_SIZE_ALL_CONV + 1;   // 13
const int PC_CONV_FILTERS     = DIM_CAPSULE_CAPS1 * NUM_CAPSULE_CAPS1_CHANNELS; // 70
const int PC_CONV_OUT_SEQ_LEN = CONV2_OUT_SEQ_LEN - KERNEL_SIZE_ALL_CONV + 1;   // 12
const int NUM_PRIMARY_CAPSULES= PC_CONV_OUT_SEQ_LEN * NUM_CAPSULE_CAPS1_CHANNELS; // 120
const int FLATTEN_CAPS_UNITS  = N_CLASS * DIM_CAPSULE_CAPS2;                    // 14

// ============================================================================
// == 3. Fixed-Point Type Definitions
// ============================================================================
// NEW: A high-precision type for all data passed between layers to minimize error accumulation
typedef ap_fixed<32, 16, AP_RND, AP_SAT> internal_t;  // for forcing some layers to full precision
// --- NEW: Granular types for accumulators to enable sensitivity analysis ---

// To test a lower precision, you will only need to modify one of these lines.
////// FOR CONTROLLING AP-FIXED FOR ACCUMULATORS OF INTERNALS OF DIGICAPS, PRIMARYCAP and OTHERS(conv and dense) SEPARATELY
typedef ap_fixed<32, 16, AP_RND, AP_SAT> internal_primary_acc_t; // For PrimaryCaps accumulators
typedef ap_fixed<32, 16, AP_RND, AP_SAT> internal_digit_acc_t;   // For DigitCaps accumulators

// Input data type
typedef ap_fixed<1 + 16, 1, AP_RND, AP_SAT> input_t;

// Conv1 Layer Types
typedef ap_fixed<1 + 16, 1, AP_RND, AP_SAT> conv1_act_t;
typedef ap_fixed<1 + 7,  1, AP_RND, AP_SAT> conv1_weight_t;
typedef ap_fixed<1 + 7,  1, AP_RND, AP_SAT> conv1_bias_t;

// Conv2 Layer Types
```

```
51. typedef ap_fixed<1 + 16, 1, AP_RND, AP_SAT> conv2_act_t;
52. typedef ap_fixed<1 + 6,  1, AP_RND, AP_SAT> conv2_weight_t;
53. typedef ap_fixed<1 + 6,  1, AP_RND, AP_SAT> conv2_bias_t;
54.
55. // PrimaryCaps Layer Types
56. typedef ap_fixed<1 + 16, 1, AP_RND, AP_SAT> pc_conv_act_t;      // Output of its internal
conv1d
57. typedef ap_fixed<2 + 6,  2, AP_RND, AP_SAT> pc_conv_weight_t;
58. typedef ap_fixed<1 + 6,  1, AP_RND, AP_SAT> pc_conv_bias_t;
59. typedef ap_fixed<1 + 16, 1, AP_RND, AP_SAT> pc_squash_act_t;    // Final output of primary
caps
60.
61. // DigitCaps Layer Types
62. typedef ap_fixed<1 + 6,  1, AP_RND, AP_SAT> dc_w_t;                  // Transformation
Matrix W
63. typedef ap_fixed<3 + 16, 3, AP_RND, AP_SAT> dc_inputs_hat_t;
64. typedef ap_fixed<3 + 16, 3, AP_RND, AP_SAT> dc_routing_logits_b_t;
65. typedef ap_fixed<3 + 16, 3, AP_RND, AP_SAT> dc_routing_coeffs_c_t;
66. typedef ap_fixed<3 + 16, 3, AP_RND, AP_SAT> dc_weighted_sum_s_t;
67. typedef ap_fixed<3 + 16, 3, AP_RND, AP_SAT> dc_squash_v_loop_t;     // For intermediate
routing loop outputs
68. typedef ap_fixed<3 + 16, 3, AP_RND, AP_SAT> dc_final_act_t;         // Final output of
digitcaps
69.
70. typedef ap_fixed<3 + 16, 3, AP_RND, AP_SAT> flattened_caps_act_t;         // flatten layer
71.
72. // Decoder Part Types
73. typedef ap_fixed<1 + 16, 1, AP_RND, AP_SAT> dense1_act_t;
74. typedef ap_fixed<1 + 7,  1, AP_RND, AP_SAT> dense1_weight_t;
75. typedef ap_fixed<1 + 6,  1, AP_RND, AP_SAT> dense1_bias_t;
76. typedef ap_fixed<2 + 16, 2, AP_RND, AP_SAT> dense2_act_t;
77. typedef ap_fixed<2 + 6,  2, AP_RND, AP_SAT> dense2_weight_t;
78. typedef ap_fixed<1 + 7,  1, AP_RND, AP_SAT> dense2_bias_t;
79.
80. // Final Output Types
81. typedef ap_fixed<3 + 16, 3, AP_RND, AP_SAT> final_output_t;         // Final X,Y coordinates
82. typedef ap_fixed<1 + 6,  1, AP_RND, AP_SAT> output_dense_weight_t;
83. typedef ap_fixed<1 + 6,  1, AP_RND, AP_SAT> output_dense_bias_t;
84.
85. #endif // MY_CAPSNET_CONFIG_H
86.
```

10.2.2. Layers Header:

```
 1. #ifndef LAYERS_H_
 2. #define LAYERS_H_
 3.
 4. #include "config.h"
 5. #include "hls_weights.h"
 6.
 7. // For this experiment, we quantize the data between conv1 and conv2.
 8. // All other interfaces remain high-precision.
 9.
10. void conv1d_layer_1(
11.     const input_t   input_feature_map[INPUT_SEQ_LEN][INPUT_FEATURES],
12.     conv1_act_t     output_feature_map[CONV1_OUT_SEQ_LEN][NB_FILTERS_CONV1_CONV2]
13. );
14.
15. void conv1d_layer_2(
16.     const conv1_act_t input_feature_map[CONV1_OUT_SEQ_LEN][NB_FILTERS_CONV1_CONV2],
17.     conv2_act_t       output_feature_map[CONV2_OUT_SEQ_LEN][NB_FILTERS_CONV1_CONV2]
18. );
19.
20. void primary_caps_layer(
21.     const conv2_act_t input_feature_map[CONV2_OUT_SEQ_LEN][NB_FILTERS_CONV1_CONV2],
22.     pc_squash_act_t   output_capsules[NUM_PRIMARY_CAPSULES][DIM_CAPSULE_CAPS1] // MODIFIED:
Output is now low-precision
23. );
```

```
24.
25. void digit_caps_layer(
26.     const pc_squash_act_t input_capsules[NUM_PRIMARY_CAPSULES][DIM_CAPSULE_CAPS1],
27.     dc_final_act_t        output_capsules[N_CLASS][DIM_CAPSULE_CAPS2] // MODIFIED: Output is
now low-precision
28. );
29.
30. void dense_layer_1(
31.     const flattened_caps_act_t input_vector[FLATTEN_CAPS_UNITS],
32.     dense1_act_t               output_vector[DENSE_UNIT] // MODIFIED: Output is now low-
precision
33. );
34.
35. void dense_layer_2(
36.     const dense1_act_t input_vector[DENSE_UNIT],
37.     dense2_act_t       output_vector[DENSE_UNIT] // MODIFIED: Output is now low-precision
38. );
39.
40. void output_layer(
41.     const dense2_act_t input_vector[DENSE_UNIT], // MODIFIED: Input is now low-precision
42.     final_output_t     output_vector[N_CLASS]
43. );
44.
45. #endif // LAYERS_H_
46.
```

10.2.3. Layers Definition:

```
  1. #include "layers.h"
  2. #include "hls_math.h"
  3.
  4. ///// Conv1d With Manual Memory Duplication Or Manual Memory Banking ///////
  5. void conv1d_layer_1(const input_t i_fm[INPUT_SEQ_LEN][INPUT_FEATURES],
  6.                     conv1_act_t o_fm[CONV1_OUT_SEQ_LEN][NB_FILTERS_CONV1_CONV2]) {
  7.
  8.     // --- PRAGMAS for constant arrays ---
  9.     // Partitioning weights and biases completely turns them into registers for fast,
parallel access.
 10.     #pragma HLS ARRAY_PARTITION variable=conv1_kernel complete dim=0
 11.     #pragma HLS ARRAY_PARTITION variable=conv1_bias complete
 12.
 13.     // --- LOCAL COPIES for DATA DUPLICATION ---
 14.     // Create two local arrays to act as separate memory banks for the input feature map.
 15.     // This allows us to perform more than two reads per clock cycle.
 16.     static input_t i_fm_copy1[INPUT_SEQ_LEN][INPUT_FEATURES];
 17.     static input_t i_fm_copy2[INPUT_SEQ_LEN][INPUT_FEATURES];
 18.
 19.     // Partition both copies completely on the second dimension (the features).
 20.     #pragma HLS ARRAY_PARTITION variable=i_fm_copy1 complete dim=2
 21.     #pragma HLS ARRAY_PARTITION variable=i_fm_copy2 complete dim=2
 22.
 23.     // --- INITIAL DATA READ ---
 24.     // A pipelined loop to efficiently load data from the primary input into our two local
copies.
 25.     READ_INPUT_LOOP: for (int i = 0; i < INPUT_SEQ_LEN; ++i) {
 26.         for (int j = 0; j < INPUT_FEATURES; ++j) {
 27.             #pragma HLS PIPELINE II=1
 28.             input_t temp = i_fm[i][j];
 29.             i_fm_copy1[i][j] = temp;
 30.             i_fm_copy2[i][j] = temp;
 31.         }
 32.     }
 33.
 34.     // --- MAIN COMPUTE LOOP (Flattened Pipeline) ---
 35.     // By nesting the loops this way and placing the pipeline pragma on the inner loop,
 36.     // we create a single, deep pipeline that processes one output value per iteration.
 37.     FLATTENED_LOOP_OUTER: for (int olen = 0; olen < CONV1_OUT_SEQ_LEN; ++olen) {
 38.         FLATTENED_LOOP_INNER: for (int och = 0; och < NB_FILTERS_CONV1_CONV2; ++och) {
```

```
            #pragma HLS PIPELINE II=1

            // Initialize the accumulator with the bias for the current output channel.
            internal_t accumulator = (internal_t)conv1_bias[och];

            // This loop will be processed sequentially by the pipeline stages.
            KERNEL_LOOP: for (int k = 0; k < KERNEL_SIZE_ALL_CONV; ++k) {
                // This innermost loop is unrolled, allowing all features to be processed in parallel.
                #pragma HLS UNROLL
                for (int ich = 0; ich < INPUT_FEATURES; ++ich) {

                    // The key to solving the port conflict: read from different copies for different
                    // parts of the kernel. This assumes a KERNEL_SIZE of 2.
                    // If KERNEL_SIZE > 2, a more complex banking scheme would be needed.
                    input_t data_term;
                    if (k == 0) {
                        data_term = i_fm_copy1[olen + k][ich];
                    } else {
                        data_term = i_fm_copy2[olen + k][ich];
                    }

                    accumulator += data_term * (internal_t)conv1_kernel[k][ich][och];
                }
            }

            // Apply activation function (ReLU) and write the final result.
            if (accumulator < 0) {
                o_fm[olen][och] = (conv1_act_t)0;
            } else {
                o_fm[olen][och] = (conv1_act_t)accumulator;
            }
        }
    }
}

void conv1d_layer_2(const conv1_act_t i_fm[CONV1_OUT_SEQ_LEN][NB_FILTERS_CONV1_CONV2], conv2_act_t o_fm[CONV2_OUT_SEQ_LEN][NB_FILTERS_CONV1_CONV2]) {
    #pragma HLS ARRAY_PARTITION variable=conv2_kernel complete
    #pragma HLS ARRAY_PARTITION variable=conv2_bias complete
    OUTPUT_CH_LOOP: for (int och = 0; och < NB_FILTERS_CONV1_CONV2; ++och) {
        OUTPUT_LEN_LOOP: for (int olen = 0; olen < CONV2_OUT_SEQ_LEN; ++olen) {
            #pragma HLS PIPELINE II=1
            // Using the standard accumulator type for Conv/Dense layers
            internal_t accumulator = (internal_t)conv2_bias[och];
            KERNEL_LOOP: for (int k = 0; k < KERNEL_SIZE_ALL_CONV; ++k) {
                for (int ich = 0; ich < NB_FILTERS_CONV1_CONV2; ++ich) {
                    accumulator += (internal_t)i_fm[olen + k][ich] * (internal_t)conv2_kernel[k][ich][och];
                }
            }
            o_fm[olen][och] = (accumulator < 0) ? (conv2_act_t)0 : (conv2_act_t)accumulator;
        }
    }
}

void primary_caps_layer(const conv2_act_t i_fm[CONV2_OUT_SEQ_LEN][NB_FILTERS_CONV1_CONV2], pc_squash_act_t o_caps[NUM_PRIMARY_CAPSULES][DIM_CAPSULE_CAPS1]) {
    pc_conv_act_t conv_output[PC_CONV_OUT_SEQ_LEN][PC_CONV_FILTERS];
    pc_squash_act_t reshaped_output[NUM_PRIMARY_CAPSULES][DIM_CAPSULE_CAPS1];
    #pragma HLS ARRAY_PARTITION variable=conv_output complete dim=2
    #pragma HLS ARRAY_PARTITION variable=reshaped_output complete dim=2
    #pragma HLS ARRAY_PARTITION variable=primarycap_conv1d_kernel complete
    #pragma HLS ARRAY_PARTITION variable=primarycap_conv1d_bias complete

    for (int och = 0; och < PC_CONV_FILTERS; ++och) {
        for (int olen = 0; olen < PC_CONV_OUT_SEQ_LEN; ++olen) {
            #pragma HLS PIPELINE II=1
```

```
103.             // Using the specific accumulator type for this layer
104.             internal_primary_acc_t accumulator =
(internal_primary_acc_t)primarycap_conv1d_bias[och];
105.             for (int k = 0; k < KERNEL_SIZE_ALL_CONV; ++k) {
106.                 for (int ich = 0; ich < NB_FILTERS_CONV1_CONV2; ++ich) {
107.                     accumulator += (internal_primary_acc_t)i_fm[olen + k][ich] *
(internal_primary_acc_t)primarycap_conv1d_kernel[k][ich][och];
108.                 }
109.             }
110.             conv_output[olen][och] = (accumulator < 0) ? (pc_conv_act_t)0 :
(pc_conv_act_t)accumulator;
111.         }
112.     }
113.
114.     for (int i = 0; i < PC_CONV_OUT_SEQ_LEN; ++i) {
115.         for (int j = 0; j < NUM_CAPSULE_CAPS1_CHANNELS; ++j) {
116.             #pragma HLS PIPELINE II=1
117.             for (int k = 0; k < DIM_CAPSULE_CAPS1; ++k) {
118.                 reshaped_output[i * NUM_CAPSULE_CAPS1_CHANNELS + j][k] = conv_output[i][j *
DIM_CAPSULE_CAPS1 + k];
119.             }
120.         }
121.     }
122.
123.     SQUASH_LOOP: for (int i = 0; i < NUM_PRIMARY_CAPSULES; ++i) {
124.         #pragma HLS PIPELINE II=1
125.         internal_primary_acc_t s_squared_norm = 0;
126.         for (int j = 0; j < DIM_CAPSULE_CAPS1; ++j) {
127.             internal_primary_acc_t val = (internal_primary_acc_t)reshaped_output[i][j];
128.             s_squared_norm += val * val;
129.         }
130.         const float epsilon = 1e-9;
131.         internal_primary_acc_t scale_factor = 0;
132.         if ((float)s_squared_norm > epsilon) {
133.              float s_sq_double = (float)s_squared_norm;
134.              float sqrt_val = hls::sqrt(s_sq_double + epsilon);
135.              if (sqrt_val > epsilon) {
136.                 scale_factor = (internal_primary_acc_t)(s_sq_double / ((1.0 + s_sq_double)
* sqrt_val));
137.              }
138.         }
139.         for (int j = 0; j < DIM_CAPSULE_CAPS1; ++j) {
140.             o_caps[i][j] = (pc_squash_act_t)((internal_primary_acc_t)reshaped_output[i][j]
* scale_factor);
141.         }
142.     }
143. }
144.
145. void digit_caps_layer(const pc_squash_act_t
i_caps[NUM_PRIMARY_CAPSULES][DIM_CAPSULE_CAPS1], dc_final_act_t
o_caps[N_CLASS][DIM_CAPSULE_CAPS2]) {
146.     dc_inputs_hat_t inputs_hat[NUM_PRIMARY_CAPSULES][N_CLASS][DIM_CAPSULE_CAPS2];
147.     dc_routing_logits_b_t b[NUM_PRIMARY_CAPSULES][N_CLASS] = {0};
148.     dc_routing_coeffs_c_t c[NUM_PRIMARY_CAPSULES][N_CLASS];
149.     dc_weighted_sum_s_t s[N_CLASS][DIM_CAPSULE_CAPS2];
150.     dc_squash_v_loop_t v[N_CLASS][DIM_CAPSULE_CAPS2];
151.     #pragma HLS ARRAY_PARTITION variable=digitcaps_W complete
152.     #pragma HLS ARRAY_PARTITION variable=inputs_hat complete dim=0
153.     #pragma HLS ARRAY_PARTITION variable=b complete dim=0
154.     #pragma HLS ARRAY_PARTITION variable=c complete dim=0
155.     #pragma HLS ARRAY_PARTITION variable=s complete dim=0
156.     #pragma HLS ARRAY_PARTITION variable=v complete dim=0
157.
158.     for (int i = 0; i < NUM_PRIMARY_CAPSULES; ++i) {
159.         for (int j = 0; j < N_CLASS; ++j) {
160.             #pragma HLS PIPELINE II=1
161.             for (int k = 0; k < DIM_CAPSULE_CAPS2; ++k) {
162.                 internal_digit_acc_t accumulator = 0;
163.                 for (int l = 0; l < DIM_CAPSULE_CAPS1; ++l) {
```

```
164.                     accumulator += (internal_digit_acc_t)i_caps[i][l] *
(internal_digit_acc_t)digitcaps_W[0][i][j][k][l];
165.                 }
166.                 inputs_hat[i][j][k] = (dc_inputs_hat_t)accumulator;
167.             }
168.         }
169.     }
170.
171.     ROUTING_LOOP: for (int r = 0; r < ROUTINGS; ++r) {
172.         COEFF_LOOP_I: for (int i = 0; i < NUM_PRIMARY_CAPSULES; ++i) {
173.             #pragma HLS PIPELINE
174.             internal_digit_acc_t exp_sum = 0;
175.             for (int j_sum = 0; j_sum < N_CLASS; ++j_sum) {
176.                 exp_sum += (internal_digit_acc_t)hls::exp((float)b[i][j_sum]);
177.             }
178.             if ((float)exp_sum > 1e-9) {
179.                 internal_digit_acc_t inv_exp_sum = (internal_digit_acc_t)(1.0 /
(float)exp_sum);
180.                 for (int j_apply = 0; j_apply < N_CLASS; ++j_apply) {
181.                     c[i][j_apply] =
(dc_routing_coeffs_c_t)((internal_digit_acc_t)hls::exp((float)b[i][j_apply]) * inv_exp_sum);
182.                 }
183.             } else {
184.                  for (int j_apply = 0; j_apply < N_CLASS; ++j_apply) c[i][j_apply] =
(dc_routing_coeffs_c_t)(1.0 / N_CLASS);
185.             }
186.         }
187.
188.         WEIGHTED_SUM_J: for (int j = 0; j < N_CLASS; ++j) {
189.             for (int k = 0; k < DIM_CAPSULE_CAPS2; ++k) {
190.                 #pragma HLS PIPELINE
191.                 internal_digit_acc_t accumulator = 0;
192.                 for (int i = 0; i < NUM_PRIMARY_CAPSULES; ++i) {
193.                     accumulator += (internal_digit_acc_t)c[i][j] *
(internal_digit_acc_t)inputs_hat[i][j][k];
194.                 }
195.                 s[j][k] = (dc_weighted_sum_s_t)accumulator;
196.             }
197.         }
198.
199.         SQUASH_V_LOOP: for (int j = 0; j < N_CLASS; ++j) {
200.             #pragma HLS PIPELINE
201.             internal_digit_acc_t s_squared_norm = 0;
202.             for (int k = 0; k < DIM_CAPSULE_CAPS2; ++k) {
203.                 internal_digit_acc_t val = (internal_digit_acc_t)s[j][k];
204.                 s_squared_norm += val * val;
205.             }
206.             const float epsilon = 1e-9;
207.             internal_digit_acc_t scale_factor = 0;
208.             if ((float)s_squared_norm > epsilon) {
209.                  float s_sq_double = (float)s_squared_norm;
210.                  float sqrt_val = hls::sqrt(s_sq_double + epsilon);
211.                  if (sqrt_val > epsilon) {
212.                     scale_factor = (internal_digit_acc_t)(s_sq_double / ((1.0 +
s_sq_double) * sqrt_val));
213.                  }
214.             }
215.             for (int k = 0; k < DIM_CAPSULE_CAPS2; ++k) {
216.                 v[j][k] = (dc_squash_v_loop_t)((internal_digit_acc_t)s[j][k] *
scale_factor);
217.             }
218.         }
219.
220.         if (r < ROUTINGS - 1) {
221.             UPDATE_B_I: for (int i = 0; i < NUM_PRIMARY_CAPSULES; ++i) {
222.                 for (int j = 0; j < N_CLASS; ++j) {
223.                     #pragma HLS PIPELINE
224.                     internal_digit_acc_t agreement = 0;
225.                     for (int k = 0; k < DIM_CAPSULE_CAPS2; ++k) {
```

```
                        agreement += (internal_digit_acc_t)inputs_hat[i][j][k] *
(internal_digit_acc_t)v[j][k];
                    }
                    b[i][j] = (dc_routing_logits_b_t)((internal_t)b[i][j] +
(internal_t)agreement);
                }
            }
        }
    }

    for (int j = 0; j < N_CLASS; ++j) {
        for (int k = 0; k < DIM_CAPSULE_CAPS2; ++k) {
            #pragma HLS UNROLL
            o_caps[j][k] = (dc_final_act_t)v[j][k];
        }
    }
}

void dense_layer_1(const flattened_caps_act_t i_vec[FLATTEN_CAPS_UNITS], dense1_act_t
o_vec[DENSE_UNIT]) {
    #pragma HLS PIPELINE II=1
    #pragma HLS ARRAY_PARTITION variable=dense_1_kernel complete
    #pragma HLS ARRAY_PARTITION variable=dense_1_bias complete
    for (int j = 0; j < DENSE_UNIT; ++j) {
        internal_t accumulator = (internal_t)dense_1_bias[j];
        for (int i = 0; i < FLATTEN_CAPS_UNITS; ++i) accumulator += (internal_t)i_vec[i] *
(internal_t)dense_1_kernel[i][j];
        o_vec[j] = (accumulator < 0) ? (dense1_act_t)0 : (dense1_act_t)accumulator;
    }
}

void dense_layer_2(const dense1_act_t i_vec[DENSE_UNIT], dense2_act_t o_vec[DENSE_UNIT]) {
    #pragma HLS PIPELINE II=1
    #pragma HLS ARRAY_PARTITION variable=dense_2_kernel complete
    #pragma HLS ARRAY_PARTITION variable=dense_2_bias complete
    for (int j = 0; j < DENSE_UNIT; ++j) {
        internal_t accumulator = (internal_t)dense_2_bias[j];
        for (int i = 0; i < DENSE_UNIT; ++i) accumulator += (internal_t)i_vec[i] *
(internal_t)dense_2_kernel[i][j];
        o_vec[j] = (accumulator < 0) ? (dense2_act_t)0 : (dense2_act_t)accumulator;
    }
}

void output_layer(const dense2_act_t i_vec[DENSE_UNIT], final_output_t o_vec[N_CLASS]) {
    #pragma HLS PIPELINE II=1
    #pragma HLS ARRAY_PARTITION variable=output_kernel complete
    #pragma HLS ARRAY_PARTITION variable=output_bias complete
    for (int j = 0; j < N_CLASS; ++j) {
        internal_t accumulator = (internal_t)output_bias[j];
        for (int i = 0; i < DENSE_UNIT; ++i) accumulator += (internal_t)i_vec[i] *
(internal_t)output_kernel[i][j];
        o_vec[j] = (final_output_t)accumulator;
    }
}

```

10.2.4. Top Function Header:

```
#ifndef CAPSNET_HLS_TOP_H_
#define CAPSNET_HLS_TOP_H_

#include "config.h"

void capsnet_hls_top(
    const input_t           input_feature_map[INPUT_SEQ_LEN][INPUT_FEATURES],
    final_output_t          final_output_vector[N_CLASS]
);

```

```
#endif // CAPSNET_HLS_TOP_H_

```

10.2.5. Top Function Definition:

```
#include "capsnet_hls_top.h"
#include "layers.h"

void capsnet_hls_top(
    const input_t           input_feature_map[INPUT_SEQ_LEN][INPUT_FEATURES],
    final_output_t          final_output_vector[N_CLASS]
) {
    #pragma HLS INTERFACE m_axi port=input_feature_map bundle=gmem0
    #pragma HLS INTERFACE m_axi port=final_output_vector bundle=gmem1
    #pragma HLS INTERFACE s_axilite port=return

    #pragma HLS DATAFLOW

    // MODIFIED: Buffers for conv1 and conv2 outputs are now low-precision
    conv1_act_t conv1_output[CONV1_OUT_SEQ_LEN][NB_FILTERS_CONV1_CONV2];
    conv2_act_t conv2_output[CONV2_OUT_SEQ_LEN][NB_FILTERS_CONV1_CONV2];

    pc_squash_act_t p_caps_output[NUM_PRIMARY_CAPSULES][DIM_CAPSULE_CAPS1];
    dc_final_act_t d_caps_output[N_CLASS][DIM_CAPSULE_CAPS2];
    flattened_caps_act_t flattened_output[FLATTEN_CAPS_UNITS];
    // MODIFIED: The buffer for dense1's output is now also low-precision
    dense1_act_t dense1_output[DENSE_UNIT];
    // MODIFIED: The buffer for dense2's output is now low-precision
    dense2_act_t dense2_output[DENSE_UNIT];

    // The sequential pipeline of function calls
    conv1d_layer_1(input_feature_map, conv1_output);
    conv1d_layer_2(conv1_output, conv2_output);
    primary_caps_layer(conv2_output, p_caps_output);
    digit_caps_layer(p_caps_output, d_caps_output);

    for(int i=0; i < N_CLASS; ++i) {
        for(int j=0; j < DIM_CAPSULE_CAPS2; ++j) {
            #pragma HLS PIPELINE
            flattened_output[i * DIM_CAPSULE_CAPS2 + j] = d_caps_output[i][j];
        }
    }

    dense_layer_1(flattened_output, dense1_output);
    dense_layer_2(dense1_output, dense2_output);
    output_layer(dense2_output, final_output_vector);
}

```

10.2.6. TestBench:

```
#pragma clang diagnostic push /// Supress the warning due to using ap-fixed and math function of standard library in vitis 2024.2

#pragma clang diagnostic ignored "-Wmacro-redefined"

#include "capsnet_hls_top.h" //HLS header goes FIRST
//you must always include your project-specific HLS headers before any standard system or C++ libraries
#pragma clang diagnostic pop

#include <iostream>
#include <fstream>
#include <cmath>
#include <iomanip>


// --- Configuration for the Final Test ---
```

```
// TESTSET NUMBER
const int NUM_TEST_SAMPLES = 310;

// VALIDATION SET NUMBER
// const int NUM_TEST_SAMPLES = 311;

// TRAINSET NUMBER
// const int NUM_TEST_SAMPLES = 961;

// NEW: The test now passes if the final MSE is below this threshold.
// Based on your last result of ~0.000042, a tolerance of 0.0001 is a reasonable goal.
const double MSE_TOLERANCE = 0.15;

// ---- PATH CONFIGURATION

// #define INPUT_DAT_FILE "C:/Users/RN/CapsNet_Project_v2/capsnet_hls2/X_train_ordered.dat"
// #define GOLDEN_OUTPUT_DAT_FILE
"C:/Users/RN/CapsNet_Project_v2/capsnet_hls2/Y_train_golden_ordered.dat"

#define INPUT_DAT_FILE "C:/Users/RN/CapsNet_Project_v2/capsnet_hls2/X_test_ordered.dat"
#define GOLDEN_OUTPUT_DAT_FILE
"C:/Users/RN/CapsNet_Project_v2/capsnet_hls2/Y_test_golden_ordered.dat"

// #define GOLDEN_OUTPUT_DAT_FILE
"C:/Users/RN/CapsNet_Project_v2/capsnet_hls2/hls_quantized_model_outputs.dat"

// #define INPUT_DAT_FILE "C:/Users/RN/CapsNet_Project_v2/capsnet_hls2/X_val_ordered.dat"
// #define GOLDEN_OUTPUT_DAT_FILE
"C:/Users/RN/CapsNet_Project_v2/capsnet_hls2/Y_val_golden_ordered.dat"

int main() {
    // 1. Declare arrays
    input_t input_sequence[INPUT_SEQ_LEN][INPUT_FEATURES];
    final_output_t dut_output[N_CLASS];
    final_output_t golden_output[N_CLASS];

    // 2. Open data files
    std::ifstream f_input(INPUT_DAT_FILE);
    std::ifstream f_golden(GOLDEN_OUTPUT_DAT_FILE);
    if (!f_input.is_open() || !f_golden.is_open()) {
        std::cerr << "ERROR: Could not open test vector files." << std::endl;
        return 1;
    }

    std::cout << "--- Starting Full-Dataset MSE-based Verification ---" << std::endl;
    std::cout << "Processing " << NUM_TEST_SAMPLES << " samples..." << std::endl;

    // 3. Initialize variables for analysis
    double max_overall_difference = 0.0;
    double total_squared_error = 0.0;

    // 4. Loop through the entire dataset
    for (int s = 0; s < NUM_TEST_SAMPLES; ++s) {
        for (int i = 0; i < INPUT_SEQ_LEN; ++i) for (int j = 0; j < INPUT_FEATURES; ++j)
f_input >> input_sequence[i][j];
        for (int i = 0; i < N_CLASS; ++i) f_golden >> golden_output[i];

        capsnet_hls_top(input_sequence, dut_output);

        for (int i = 0; i < N_CLASS; ++i) {
            double diff = (double)dut_output[i] - (double)golden_output[i];
            total_squared_error += diff * diff;

            double abs_diff = fabs(diff);
            if (abs_diff > max_overall_difference) {
                max_overall_difference = abs_diff;
            }
        }
    }

```

```
    f_input.close();
    f_golden.close();

    // 5. Calculate the final MSE
    double final_mse = total_squared_error / (NUM_TEST_SAMPLES * N_CLASS);

    // 6. Print the Final, Detailed Report
    std::cout << "\n************************************************************" <<
std::endl;
    std::cout << "--- HLS Model Final Analysis Report ---" << std::endl;
    std::cout << std::fixed << std::setprecision(8);
    std::cout << "Total Samples Processed:      " << NUM_TEST_SAMPLES << std::endl;
    std::cout << "Bit-Accuracy MSE vs. Python:  " << final_mse << std::endl;
    std::cout << "Overall Maximum Difference:   " << max_overall_difference << std::endl;
    std::cout << "Verification MSE Tolerance:   " << MSE_TOLERANCE << std::endl;
    std::cout << "************************************************************" <<
std::endl;

    // 7. NEW: Determine final pass/fail based on MSE
    if (final_mse <= MSE_TOLERANCE) {
        std::cout << "\nSUCCESS: The HLS design passed the MSE tolerance check." <<
std::endl;
        return 0; // Return 0 for success
    } else {
        std::cout << "\nFAILURE: The HLS design failed the MSE tolerance check." <<
std::endl;
        return 1; // Return 1 for failure
    }
}

```