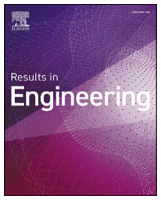
Results in Engineering 26 (2025) 105165

Contents lists available at ScienceDirect

Results in Engineering

journal homepage: www.sciencedirect.com/journal/results-in-engineering



Research paper

Engineering a multi model fallback system for edge devices

Gaurav Kadve [a], Abishi Chowdhury [a], Vishal Krishna Singh [b], Amrit Pal [a,*]

[a] School of Computer Science and Engineering, Vellore Institute of Technology, Chennai, 600127, Tamil Nadu, India
[b] School of Computer Science and Electronics Engineering, University of Essex, Colchester, UK

ARTICLE INFO

Keywords:
Resource constrained devices
Machine learning
IoT
Edge computing
TinyML

ABSTRACT

Machine learning (ML) is an effective way to extract information from data and perform decision making on it. The growing deployment of machine learning applications in edge environments, such as IoT devices and embedded systems, has highlighted the need for efficient, resource-aware systems for edge environments. This paper presents a novel multi-model fallback system designed for deployment on resource-constrained edge devices, leveraging the advancements of TinyML. The proposed system utilizes a model pool of optimized models and a confidence-based switching mechanism to dynamically select the most reliable model for inference. Through the integration of optimization techniques such as quantization, pruning, and clustering, the system achieves significant reductions in model size and inference time without compromising accuracy. The system's efficacy is validated through extensive experiments using the benchmark MNIST and CIFAR10 datasets, demonstrating its ability to balance trade-offs among accuracy, speed, and resource usage. The experimental analysis shows that the proposed optimization techniques reduce the size of the model ranging from 57% to 90% while ensuring its effectiveness. The proposed approach highlights the selection of a threshold value to make models adaptive to the considered application. The results show the system's adaptability, positioning it as a compelling option for real-time, low-power tasks in different engineering areas such as IoT, autonomous technologies, and continuous monitoring.

1. Introduction

With an exponential growth in technology and the need for real-time computing, the traditional centralized cloud computing face a lot of challenges. This is where the decentralization of computing comes in picture, where the computations happen closer to the source, in the other words, at the edge [1]. Edge computing is a computing paradigm that brings data processing and generation closer to the source of data generation, reducing the need for data to travel to distant cloud servers, thereby minimizing latency and bandwidth usage [2]. Fig. 1 shows architecture of an IoT system which has employed an edge layer. Physical layer collects sensor data and the actuators perform actions, edge layer processes data locally for real-time decisions, cloud layer provides advanced analysis, updates, and long-term storage. The comparison between these technologies is briefly highlighted by the work of Keyan Cao et al. [3]. Machine learning (ML) is an effective way to extract information from data and perform decision making on it. In an edge computing environment, ML model deployment can bring this power closer to the data source [4], thus making real time decision making possible along with numerous benefits such as enhanced privacy and security and better energy efficiency since lower data transmission is required [5].

The growing deployment of machine learning applications in edge environments, such as IoT devices and embedded systems, has highlighted the need for efficient, resource-aware solutions. But there are limitations for deployment of ML models on the edge [6]. Edge devices are constrained by limited computational power, memory, and energy and therefore present unique challenges in the implementation of sophisticated machine learning models [7]. Moreover, real-time applications require ultra-low latency, which challenges the execution of heavy computations within a limited timeframe. Large scale ML models which have a large number of parameters and high computational requirements are not deployable at the edge because of these resource constraints, therefore lightweight models that have smaller size and lower computational requirements are the most ideal choices as they are designed for environments with limited computational resources. These models are optimized to maintain a balance between computational efficiency and accuracy through architecture simplifications, and optimization techniques [8] [9]. Size reduction of models often introduce

* Corresponding author.
E-mail address: amrit.pal@vit.ac.in (A. Pal).

https://doi.org/10.1016/j.rineng.2025.105165
Received 25 January 2025; Received in revised form 22 April 2025; Accepted 30 April 2025
Available online 7 May 2025

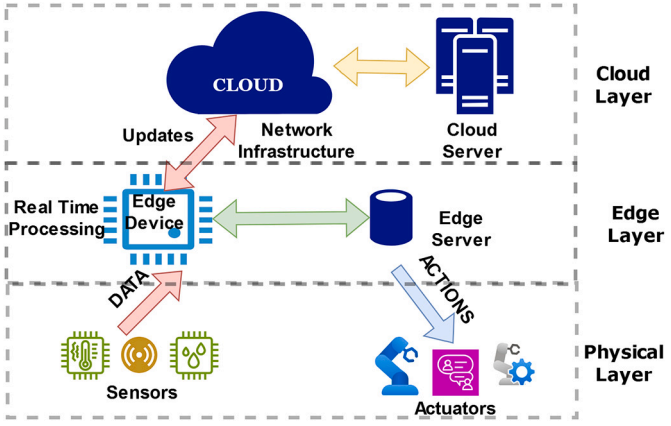


Fig. 1. IoT System.

errors and indirectly degrade performance of the model and minimizing it can be challenging [10] in many cases.

TinyML is a field dedicated to enabling machine learning on such resource-constrained devices, offers innovative approaches to optimize models for real-time, low-power operation [11][12][13][14].

Optimization techniques like quantization, pruning and clustering are often used to reduce model size. TinyML frameworks like tensorflowlite provide tools to create and deploy these models efficiently. TinyML has applications in diverse areas like predictive maintenance, environmental monitoring, speech and wake-word recognition, gesture detection, and smart agriculture without require cloud connectivity. Most systems in TinyML use single model to provide insights or perform specific task. In few cases, the efficiency and accuracy of a single model is questionable. Additionally, edge devices are deployed in dynamic environments where inputs might highly differ and being dependent on only a single model is not optimal.

Despite the rapid growth of TinyML and edge computing, current inference frameworks tend to be single-model based, which can collapse under dynamic input distribution drifts or abrupt noise. This paper addresses the limitations of traditional single-model inference systems, particularly their struggles with varying input distributions, noise, and edge cases. Additionally, certain adaptive schemes suggest overhead in the form of online model retraining or switching without confidence checks, resulting in suboptimal performance and resource utilization. This work addresses two main gaps: (i) absence of robust adaptation as data distributions dynamically shift at the edge, and (ii) the absence of lightweight, deterministic switching procedures that exchange accuracy, latency, and memory without costly retraining. Our confidence thresholded multi-model fallback method integrates resource-aware optimization algorithms quantization, pruning, and clustering—into an efficient pipeline. Most TinyML edge deployments rely on a single and statically optimized model which do not adapt to runtime shifts and sometimes face accuracy degradation under unexpected inputs. Alternatively, other approaches where heuristic switching systems are utilized, such as CPU-utilization-based model switching or early-exit multi-branch networks without using confidence thresholds also avoid retraining but have continuous monitoring overhead or ambiguous decision criteria [15]. In contrast, our method uses a deterministic threshold criterion across a pool of quantized, pruned, and clustered models requiring only two simple comparisons per model and guarantees robust accuracy with good inference speed on devices like Raspberry Pi 5. The research question this paper addresses is: Can a multi-model fallback system improve inference reliability and efficiency on resource-constrained edge devices while balancing accuracy, speed, and model size?

The system employs advanced optimization techniques such as quantization, pruning, and clustering, reducing model size and inference time while maintaining high accuracy. To validate the proposed framework, extensive experiments are conducted using the MNIST [28] dataset, a standard benchmark in image classification. The results demonstrate the system's capability to balance trade-offs among model size, inference speed, and accuracy, making it highly suitable for real-time edge applications.

The remainder of this paper is organized as follows: Section 2 mentions the related work in this field. Section 3 discusses the problem statement. Section 4 elaborates the optimization techniques and introduces multi-model fallback system and its confidence threshold mechanism. Section 5 explains the results and analysis. Section 6 concludes with a discussion of findings and possible future works.

2. Related work

TinyML is a fast-growing field at the intersection of machine learning and embedded systems. Early works in TinyML focused on techniques for adapting traditional ML models to resource-constrained devices to enable local inference with very low latency and energy consumption. Techniques such as model compression, quantization, pruning, and efficient neural architecture designs have been some of the most foundational techniques in achieving these goals. Also the availability of dedicated hardware and software frameworks specifically designed for TinyML applications has accelerated the practical adoption of TinyML in applications ranging from real-time object detection to predictive maintenance.

The paper by G. Meghani [16] provides a detailed survey on efficient deep learning models, highlighting the need to optimize for smaller model sizes, faster inference, and better computational efficiency. The paper is structured around several challenges, such as the cost of deploying large models, enabling real-time on-device inference, and privacy concerns because of data sensitivity. He identified areas where efficiency can be improved: compression techniques, advanced learning strategies, automation tools, efficient model architectures, and infrastructure. He also points out the importance of Pareto optimality, putting an emphasis on models that achieve the best trade-offs in accuracy and efficiency.

Another paper by T. Hoefler et al. [17] provides a detailed analysis on sparsity for efficient deep learning models. They draw parallels between sparsity in artificial neural networks and human brain, which naturally optimize connectivity. They then introduce a taxonomy of sparsification techniques that include techniques for structured pruning, ephemeral sparsification, and hardware acceleration. They also suggest metrics such as pruned parameter efficiency to standardize the evaluation and benchmark methodologies for comparison of sparsification techniques. Hoefler et al. claim that the computational benefits that could be obtained by sparsity could be the impetus for a new generation of highly efficient and scalable models.

Wu et al. [18] introduces integer quantization to make deep learning inference much more efficient. Authors provide the basic math behind uniform quantization and assess how it works with different types of neural networks, like CNNs and transformers. This allows us to use fast integer math pipelines and therefore reduces both memory usage as well as calculations. Wu et al. note that Quantization-aware training (QAT) is necessary to ensure that the model is accurate. The authors describe methods in order to obtain accuracy up to 1% with the floating-point standard, demonstrating real-life applicability of integer quantization.

Huffman coding for compression of deep learning neural networks along with pruning and trained quantization by Song Han et al. [19] achieved up to 49x reduction in storage without much effect on accuracy.

Other papers by T. Liang et al. [20], L. Capogrosso et al. [21], B. Rokh et al. [22] and V. Tsoukas et al. [25] describe and analyze multiple optimization techniques and have further stated their effectiveness in multiple real world scenarios. Quantization outperformed pruning in the work of A. Kuzmin et al. [23], S. Ameen et al. [24] performed analysis of various models and their performance metrics on Raspberry Pi 5, thus providing detailed insights about model deployment on it. Table 1 summarizes the various methods discussed above.

Table 1
Comparative Analysis of Optimization Techniques in TinyML.

Paper	Technique	Key Benefits	Limitations
Meghani [16]	Model Optimization Strategies	Reduces model size, faster inference, improves computational efficiency	Limited focus on hardware-specific optimizations
Hoefler et al. [17]	Sparsification Techniques	Enhances computational efficiency and scalability through structured pruning	Sparse models may still require specialized hardware for best performance
Wu et al. [18]	Integer Quantization	Lowers memory usage and computational cost via efficient integer operations	High accuracy demands quantization-aware training (QAT)
Han et al. [19]	Huffman Coding with Pruning & Quantization	Achieves up to 49× storage reduction with minimal accuracy loss	May require retraining or fine-tuning to fully regain accuracy
Liang et al. [20]	Various Optimization Techniques	Demonstrates effectiveness in real-world scenarios with combined methods	Further validation on diverse hardware setups is needed
Capogrosso et al. [21]	Model Compression Techniques	Improves storage efficiency for resource-constrained devices	Potential trade-offs in accuracy; empirical support is limited
Rokh et al. [22]	Efficient Deep Learning Architectures	Enhances computational performance, especially in image classification	Limited applicability outside image classification domains
Kuzmin et al. [23]	Quantization vs. Pruning	Shows that quantization often outperforms pruning in preserving accuracy	Pruning alone may not achieve optimal efficiency
Ameen et al. [24]	Model Performance on Raspberry Pi 5	Provides practical insights into real-world deployment performance	Findings are limited to the tested models and Raspberry Pi 5 platform
Tsoukas et al. [25]	Analysis of Optimization Techniques	Examines trade-offs between accuracy and efficiency in TinyML	Lacks a unified benchmark for evaluation

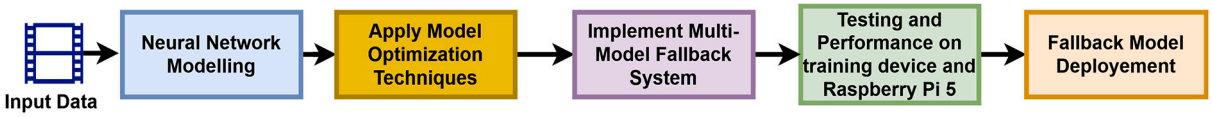


Fig. 2. Key stages of the proposed approach starting from input data to model deployment.

3. Problem statement

In complex real-world cases, many single-model systems fail to deliver reliability and accuracy when input conditions are diverse. Such a limitation may well constrain the performance of significant applications where there is a strong requirement for precise predictions and decisions. A Multi-Model Fallback System is a powerful alternative wherein several models with complementary strengths are used that can dynamically switch from one model to another based on a pre-set confidence threshold. This paper addresses challenges like the time-space-accuracy trade-off which aims to optimize computational resources while maintaining or improving accuracy in a multi-model setup, developing a reliable confidence threshold mechanism to enable seamless transitions between models, minimizing incorrect predictions specially for exceptional cases and ensuring robustness. The main objective is to design and evaluate a novel multi-model framework that integrates fallback mechanisms with confidence threshold-based switching.

This work considers the development and optimization of a multi-model fallback system S, consisting of n models $M_1, M_2, \ldots, M_n$, where each model M_i has a known confidence function $C_i(x)$ and prediction function $P_i(x)$, to maximize the overall prediction reliability and accuracy of the system in a diverse input space. The input domain $\mathcal{X}$ represents the various conditions of the system. The main non-optimized model M_0 and n optimized models $\{M_1, M_2, \ldots, M_n\}$ have a prediction function $P_i(x) : \mathcal{X} \to \mathcal{Y}$, where $\mathcal{Y}$ is the prediction space. The confidence function $C_i(x) : \mathcal{X} \to [0, 1]$, represents the confidence of M_i for input $x \in \mathcal{X}$. Here, $C_i(x)$ is the difference between prediction with highest probability and the second highest probability. The overall goal is the selection of the model M_i that maximizes the function in equation (1).

$$\arg\max_i (max(C_i(x)) + \frac{1}{T_i(x)}) \tag{1}$$

The objective is to design a reliable fallback mechanism in which several optimized models are evaluated in sequence, and the system dynamically selects the most suitable model for inference. The mechanism should prioritize low inference time and high accuracy, while ensuring robustness in cases where individual models may fail or underperform. A fallback to a more comprehensive model is triggered when earlier models are insufficient for confident predictions. The goal is to maximize system performance across a diverse input domain by balancing speed, model compactness, and prediction reliability.

4. Proposed model

This section outlines the fundamental edge computing models applied in the suggested approach, subsequently introducing the model utilized for the edge computing nodes. The section presents the compression approaches available for reducing the size of the model and generating a pool of models which can be used for decision making. Building on the basic model and the necessary components, the section details the threshold-based strategy. Fig. 2 provides an overview of the development process presented in this paper, highlighting the key stages. Fig. 3 visualizes the data flow of the proposed system.

4.1. Initial model

The initial model is designed using sequential layers of various filtering functions with the objective to achieve maximum accuracy on training. The model consists of multiple layers to extract features and train the weights of the neural network, as listed in Table 2.

The first layer is the input layer that inputs images of size 28 x 28 pixels. The input pixels are then normalized to a range between 0 and 1 which is achieved by dividing each pixel intensity by 255, as seen in equation (2). This scaling reduces computational complexity and helps the network learn effectively.

$$Normalized\,PixelValue = Original\,PixelValue/255 \tag{2}$$

The second layer is a convolution layer with 32 filters/kernels of size 3x3. Each pixel value in the input image is convolved with a filter (kernel), generating feature maps. If the input matrix is denoted by X, the k-th filter by F_k, and the convolution operation by *, the convolution output (feature map) at position (i,j) is equation (3).

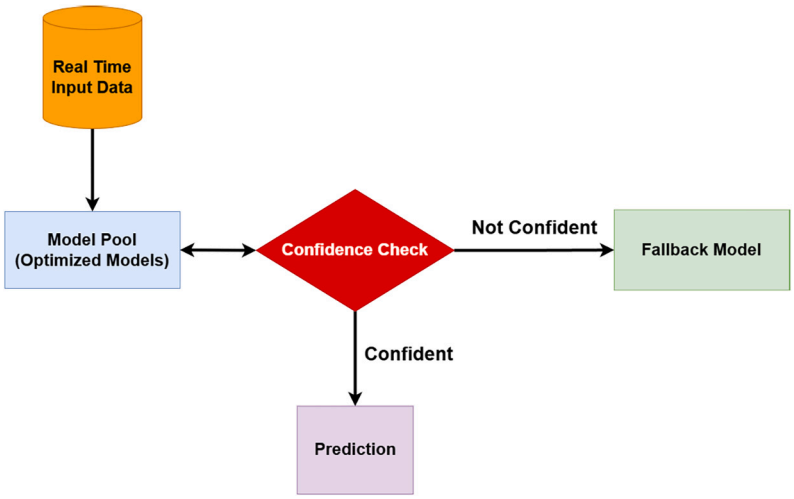


Fig. 3. Data flow for the proposed approach.

Table 2
Summary of Neural Network Architecture.

Layer Name	Type	Input Shape	Output Shape
conv2d	Conv2D	(None, 28, 28, 1)	(None, 26, 26, 32)
max_pooling2d	MaxPooling2D	(None, 26, 26, 32)	(None, 13, 13, 32)
conv2d_1	Conv2D	(None, 13, 13, 32)	(None, 11, 11, 64)
max_pooling2d_1	MaxPooling2D	(None, 11, 11, 64)	(None, 5, 5, 64)
flatten	Flatten	(None, 5, 5, 64)	(None, 1600)
dense	Dense	(None, 1600)	(None, 128)
dropout	Dropout	(None, 128)	(None, 128)
dense_1	Dense	(None, 128)	(None, 10)

$$\left(X * F_k\right)_{i,j} = \sum_{m=1}^{3}\sum_{n=1}^{3} X_{i+m-1,j+n-1} \cdot F_k[m,n] \tag{3}$$

The feature map Z is passed through the ReLU activation function i.e. equation (4). This sets all negative values to zero, introducing non-linearity into the network.

$$ReLU(z) = \max(0, z) \tag{4}$$

The third layer is a maximum pooling layer with pool size of 2x2. The max-pooling layer reduces the spatial dimensions of the feature map by taking the maximum value in each 2×2 region, which helps reduce computation and achieve translation invariance. For each 2×2 subregion (i, j) in the feature map Z max-pooling can be defined as in equation (5).

$$MaxPool(Z)_{i,j} = \max\{Z[i,j], Z[i+1,j], Z[i,j+1], Z[i+1,j+1]\} \tag{5}$$

The fourth layer is again a convolution layer but with 64 filters of size 3x3. It extracts more complex features by using an increased number of filters. For each input feature map, each filter F_k is applied to calculate the feature maps as described in the first convolutional layer and ReLU activation is applied again. This layer is again followed by Maxpooling layer with pool size of 2x2. The sixth layer is a flatten layer that transforms the 2D matrices of each feature map into a single 1D vector. If the input to this layer has shape (h, w, d), where h is the height, w is the width, and d is the depth, the output will have shape $(h \times w \times d, 1)$. The seventh layer is a fully connected layer with 128 units, here. The flattened input vector is multiplied by a weight matrix W and added to a bias vector b, as shown in equation (6).

$$z = W \cdot x + b \tag{6}$$

where x is the flattened vector from the previous layer.

The eighth layer is a dropout layer set to 50% which sets half of the units to zero randomly, this is done to prevent overfitting, as shown in equation (7).

$$Dropout(x) = x \cdot m \tag{7}$$

where m is the binary mark with probability $= 0.5$.

The final layer is a fully connected layer with 10 units. The output from the previous layer is multiplied by a weight matrix W_o and added to a bias vector b_o, as in equation (8).

$$z_o = W_o \cdot x + b_o \tag{8}$$

Softmax function is applied to produce class probabilities, shown in equation (9).

$$Softmax(z_i) = \frac{e_i^z}{\sum_j e_j^z} \tag{9}$$

z_i is the output of the dense layer and also the output of the model.

4.2. System flow

Fig. 4 shows how the CNN model is trained and optimized to be deployed on resource constrained devices and Fig. 5 shows how the model file is used to get inference from the input on a resource constrained device. Edge device has a interpreter to interpret the optimized model information.

4.3. Model compression process

4.3.1. Quantization

For optimization of the ML models, one of the most core techniques involves the reduction of size and computational requirement using minimal compromise in accuracy. One such technique is strongly required

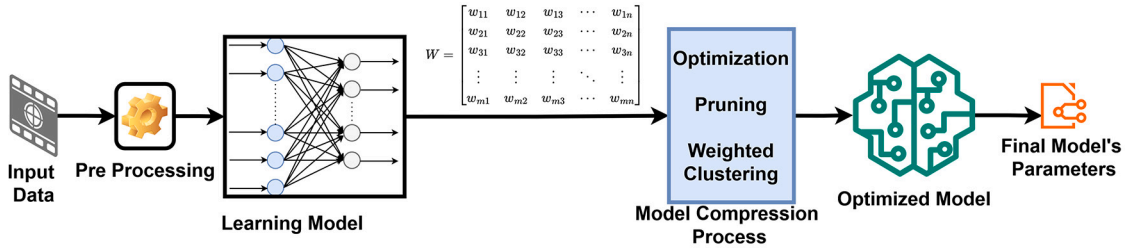


Fig. 4. Optimization Flow.

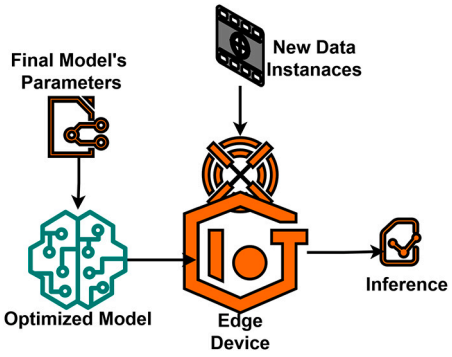


Fig. 5. Inference Flow.

for the deployment of deep learning models on resource-constrained edge hardware like memory, storage, and processing power.

Basic quantization essentially entails the reduction of the precision provided to the numerical values, both weights and activations, of a network [16]. The network is scaled down and gains faster computations by representing the values using fewer bits. Thus, this makes it suitable for low-resource environments like micro-controllers, mobile devices, and IoT-based applications. There are mainly two types of quantization suitable for optimizing ML models, post-training quantization and quantization aware training.

In post-training quantization, quantization is done once the training of the model is completed. It can be further classified into two types: Dynamic range and Full integer quantization. Dynamic range quantization converts a model's weights from a 32-bit floating point to an 8-bit integer representation while keeping activations in a floating point. Only weights are quantized here; activations are converted to integers at runtime, adjusting the scaling factor dynamically, as in equation (10).

$$\text{Quantized Value} = \text{Round}\left(\frac{\text{Float Value}}{\text{Scale}}\right) \tag{10}$$

where Scale is the dynamic range of the weights, i.e., the min and max.

Full Integer Quantization approach [18] converts both weights and activations to integers (e.g., 8-bit integers), as in equation (11). To improve accuracy, we compute scales for each layer so that floating-point operations approximate integer operations closely.

$$\text{Quantized Output} = \text{Scale}_w \times \text{Quantized Input} + \text{Zero Offset} \tag{11}$$

where Scale_w is the scale of weights.

For the activation function, equation (12) is computed.

$$Q_a = \text{Round}\left(\frac{a}{S_a}\right) + Z_a \tag{12}$$

where S_a is the scale and Z_a is the zero-point of the activations.

For matrix multiplication of weights W and activations A, the result Z is represented in equation (13).

$$Z = S_w \cdot S_a \cdot (W - Z_a) \cdot (A - Z_a) \tag{13}$$

The entire operation is performed in integer arithmetic, and the result is scaled back to a floating point if needed using the scales and zero points.

In quantization aware training, the parameters of the neural network are quantized during training so that the model adjusts them to minimize the loss of accuracy that might occur due to quantization. This method gives better accuracy than post training quantization as the model learns to adapt to the reduced parameters [16].

During QAT, fake quantized nodes i.e. nodes which are round off weights and activations to lower precision during the forward pass and then they are replaced with the original values as shown in equation (14). Therefore, gradients are also calculated with these fake quantized values during backpropagation. The quantization error is introduced in each forward pass, allowing the model to learn weights that are robust to quantization effects.

$$F_{fq} = \text{Round}\left(\frac{f}{S}\right) \cdot S \tag{14}$$

where F_{fq} is the fake quantized value, f is the original value, and S is the scale.

The benefits of quantization are reduction in model size, since we are converting higher precision value to a lower precision value; faster inference time, since lower bit arithmetic is computationally faster, leading to reduced latency and improved throughput and lower power consumption.

Drawbacks are that quantization reduces the precision of weights and activations, leading to rounding errors and a less precise representation of the model. This can degrade the model's ability to make accurate predictions, especially in tasks requiring fine-grained details, but this can be minimized by adapting quantization aware training.

4.3.2. *Pruning*

Pruning is removing or zeroing the weights, or even full neurons, from a neural network based on algorithms to optimize it [17]. Once these components are removed, the model gets to be compact with fewer parameters. This shrinking in size may even mean that inference times are faster and computation requirements lower. Pruning contributes towards how well the model performs without significantly losing its robustness. It is mainly applied in cases when there is limited computational power, such as in edge devices or in real-time applications.

Pruning can be applied to the entire network at once (Global pruning) or it can be applied to each layer of the network independently (Layer-wise pruning). Many different types of pruning techniques can be applied to a model such as structured pruning, magnitude-based pruning, random pruning, etc [17].

In structured pruning, entire groups of weights like entire units, kernels or filters, blocks of layer, etc. are pruned from the network. In this approach, a mask M_i is applied to the weight matrix W_i. The mask M_i zeroes out entire filters or blocks as per the pruning criteria as in equation (15).

$$W_i' = W_i * M_i \tag{15}$$

In unstructured pruning, individual weights are pruned irrespective of their position in the network based on a certain criterion. In this ap-

proach, a binary mask m_{ij} is applied to the weights w_{ij} of a weight matrix W_i. When a certain criterion is met, the binary mask is set to zero and the weight w_{ij} is zeroed out as in equation (16).

$$w'_{ij} = w_{ij} \cdot m_{ij} \tag{16}$$

where $m_{ij} \in \{0, 1\}$.

In magnitude-based pruning, weights with smaller magnitudes or magnitudes below a certain threshold are pruned as they contribute less to the model's output. Pruning based on magnitude involves removing weights below a threshold τ, as in equation (17).

$$m_{ij} = \begin{cases} 0 & \text{if } |w_{ij}| < \tau \\ 1 & \text{otherwise} \end{cases} \tag{17}$$

In random pruning, weights are pruned randomly irrespective of their magnitude or importance. Random pruning selects weights at random for removal. This type of pruning is not very effective in most cases. For each weight w_{ij}, the mask m_{ij} is defined by a probability p, as in equation (18):

$$m_{ij} = \begin{cases} 0 & \text{with probability } p \\ 1 & \text{with probability } 1 - p \end{cases} \tag{18}$$

In polynomial decay based pruning, weights are pruned according to the pruning rate $r(t)$ computed by the polynomial function. The pruning rate varies over time as shown in equation (19).

$$r(t) = r_{ij} + (r_{initial} - r_{final}) \left(1 - \frac{t}{T}\right)^d \tag{19}$$

where $r_{initial}$ and r_{final} are the initial and final pruning rates, T is the total training time, and d is the polynomial degree.

In sensitivity-based pruning, weights are pruned based on their ability to affect the model's performance. The sensitivity of each weight is evaluated as its effect on the loss L when adjusted. For a small change Δw_{ij}, S_{ij} is evaluated as in equation (20).

$$S_{ij} = \left| \frac{\partial L}{\partial w_{ij}} \right| \tag{20}$$

where the loss function L is calculated using categorical cross-entropy i.e. equation (21).

$$L = -\frac{1}{N} \sum_{i=1}^{N} \sum_{c=1}^{C} y_{i,c} \log(p_{i,c}) \tag{21}$$

In hybrid pruning, weights are pruned by a combination of different pruning techniques to get better results and in dynamic pruning, pruning is performed during training therefore minimizing the loss of accuracy, [20].

Pruning offers several benefits for deep learning models. This method reduces model size, leading to improved efficiency in storage and computation. By requiring fewer computations, pruned models enable faster inference times, which is especially advantageous for deployment on resource-constrained devices. Additionally, pruning lowers memory usage by reducing the number of parameters in a model, making it more manageable during both training and inference. Another significant advantage is improved generalization, as pruning removes redundant or less important weights, potentially reducing overfitting. Despite its benefits, pruning also has limitations. One major drawback is the potential for accuracy loss, as excessive or poorly executed pruning can degrade model performance. Some techniques can be complex to implement, requiring careful tuning of thresholds or additional computations to determine which weights to prune. Furthermore, sparse models generated through pruning may not always translate into performance gains on hardware that is not optimized for sparse computations. Lastly, certain pruning methods, such as dynamic or sensitivity-based pruning, can increase training time and complexity, posing challenges in practical applications.

4.3.3. Clustering

Clustering is a vital optimization technique in TinyML, enabling efficient model deployment on resource-constrained devices [26]. By grouping similar weights and replacing them with shared representative values, clustering significantly reduces memory usage and computational complexity. This technique minimizes energy consumption, which is crucial for battery-powered devices, without substantially compromising model performance. It is widely used in conjunction with quantization and pruning to achieve lightweight yet effective edge models. The clustering techniques we use for our approach are kmeans clustering and uniform clustering [26] [19].

In k-means clustering, a centroid-based method that partitions data into k clusters by minimizing the sum of squared distances from each point to the cluster's centroid. For weights W in layer L_i and where C_k is the kth cluster and μ_k is the centroid of C_k, cost function J is calculated according to equation (22).

$$J = \sum_{k=1}^{K} \sum_{w \in C_k} \|w - \mu_k\|^2 \tag{22}$$

where J is the cost function.

Uniform clustering divides the range of data into equally spaced intervals called bins and assigns each data point to the nearest interval. Let the bins $B = \{b_1, b_2, b_3, \ldots, b_k\}$ and find b_k using equation (23).

$$b_k = \min(W) + k \cdot \frac{\max(W) - \min(W)}{K}, \quad k \in \{1, 2, 3, \ldots, K\} \tag{23}$$

Clustering techniques help in reducing the size of models, making them more manageable for deployment on devices with limited computational power, memory, and storage. By grouping similar data points together, clustering can help create compact representations of data, leading to reduced memory usage and faster inference times. This is especially useful for applications in edge computing, where processing must happen locally on devices such as microcontrollers or embedded systems. Furthermore, clustering can assist in simplifying complex data, improving generalization, and enabling more efficient data processing.

However, clustering also presents challenges. The limited computational resources on edge devices make it difficult to execute computationally expensive clustering algorithms, especially when handling large or high-dimensional datasets. Additionally, determining the optimal number of clusters can be challenging in resource constrained settings, where experimentation and iterative methods may not always be feasible. Many clustering algorithms are also sensitive to initialization or outliers, which can negatively impact the performance of models on low-power devices.

4.3.4. Flatbuffers format

FlatBuffers is an efficient serialization library [27] designed for low overhead in both memory and time, and it is widely used for storing and optimizing models for deployment on edge devices. The process of converting a Keras model into the FlatBuffer format involves several mathematical and computational steps. First, the Keras model is transformed into a computational graph through topological sorting, ensuring that operations are processed in the order of their dependencies. Each Keras layer is then represented as a node in the graph, along with its associated weights, biases, and operations. Next, the weights and biases from the Keras layers are packed into the FlatBuffers format. This involves storing weights in a contiguous array for efficient access, while sparse matrices are compressed by storing only non-zero elements and their indices.

The operations within the model, such as Conv2D or Dense layers, are mapped to their FlatBuffers counterparts, using highly optimized mathematical kernels for computation. These kernels often leverage hardware acceleration, such as General Matrix Multiplication (GEMM) for matrix operations. Parameters of each operation, including strides, padding, and activation functions, are aligned to the FlatBuffers schema representation. Once the graph is optimized, it is serialized into the

Algorithm 1: Multi-Model Fallback with Confidence Threshold.

Input: I: input data
$\mathcal{O} = \{M_1, M_2, \ldots, M_k\}$: set of optimized models
M_0: main (fallback) model
$\tau \geq 0$: confidence threshold
Output: y^*: final predicted class–probability vector

foreach $M \in \mathcal{O}$ **do**
  $y \leftarrow M$.predict(I) // Sort probabilities descending order
  $[p_{(1)}, p_{(2)}, \ldots] \leftarrow$ SortDesc(y) // Compute margin between highest and second highest scores
  $\Delta \leftarrow (p_{(1)} - p_{(2)})$ **if** $\Delta \geq \tau$ **then**
    return y // "Confident" result from optimized model
  end
end
return M_0.predict(I) // Fallback to main (more accurate) model

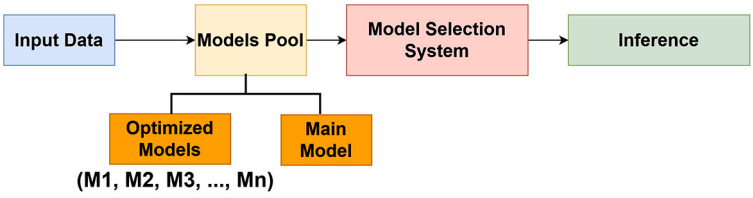


Fig. 6. Multi Model Fallback System.

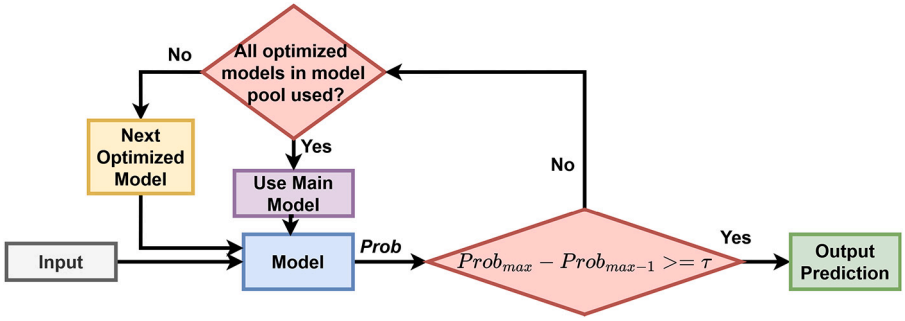


Fig. 7. Model Selection System.

FlatBuffers format. This step encodes all operations, tensors, and parameters in a platform-independent manner, ensuring efficient decoding while considering factors like endianness and memory alignment. Finally, model metadata, such as input shapes, output shapes, and data types, is embedded within the FlatBuffers file to support efficient inference and seamless integration into edge applications. This Flatbuffers format is also referred to as tflite file format.

At the end of this process, we get the optimized models which are used as input to the multi-model system. The multi-model fallback system further performs confidence threshold-based switching as described in the next section.

4.4. Multi-model fallback system with confidence threshold based switching

Traditional single machine learning models, while powerful, often fail to generalize across diverse input scenarios due to limitations such as overfitting, underfitting, or inadequate handling of outliers and edge cases. These models may struggle to maintain consistent performance when faced with varying data distributions, noise, or unseen inputs, leading to unreliable predictions. A multi-model fallback system with confidence-based switching addresses these challenges by dynamically selecting the most appropriate model for inference based on prediction confidence. This system combines a sequential model pool, consisting of optimized models, and a fallback main model to ensure high accuracy and reliability. The core mechanism revolves around evaluating model outputs against a confidence-based threshold, allowing the system to switch between models until an optimal prediction is achieved. By leveraging multiple models and enforcing a confidence criterion, this approach ensures adaptability to diverse scenarios, minimizes prediction errors, and enhances system robustness. The working of this system can be observed from the Fig. 6. The input data passes through a model pool. The model pool is divided into two sections: optimized models and main model. The main model is the parent model or the best non-optimized model. The optimized models consist of various models that are optimized by different optimization techniques which are discussed in further sections. The model selection system uses these models from the model pool to get inference about the input data. The models are selected sequentially, which means that the next model is selected only if the previous one is used. Let $M_1, M_2, M_3, \ldots, M_n$ be the models in the model pool, and M_0 be the main model.

4.4.1. Model selection

The flow diagram in Fig. 7 describes the model selection process which employs a sequential approach to dynamically select the most reliable model for prediction. The system iteratively evaluates candidate models' predictions and switches between them based on a confidence threshold.

The process begins with input data being passed to a model, which produces predictions in the form of probabilities. These probabilities are sorted, and the difference between the highest confidence score ($Prob_{max}$) and the next highest confidence score($Prob_{max\text{-}1}$) is calculated. The system checks validity using equation (24).

$$\text{Is } \left(\text{Prob}_{\max} - \text{Prob}_{\max\text{-}1}\right) \geq \tau? \tag{24}$$

Here, τ is the predefined confidence threshold. Mathematically, consider that Y is the output vector of probabilistic prediction of the model M_i. Then the vector Y is sorted in descending order. The first item in sorted vector Y is subtracted with second item of sorted vector Y to get a confidence score, as in equation (25).

$$Y[0] - Y[1] \geq \tau \tag{25}$$

If the condition is satisfied, the prediction is accepted as reliable. If not, the system verifies whether all models in the model pool have been evaluated. If additional models are available, the system switches to the next optimized model and repeats the evaluation. If all models are exhausted without meeting the threshold, the main model is used to generate predictions.

Algorithm 1 presents the algorithm for the multi-model fallback system that uses confidence threshold-based switching. The algorithm begins by taking an input vector I, a sequence of lightweight optimized classifiers $M_1, M_2, \ldots M_k$, a heavier but more accurate fallback model M_0, and a confidence margin τ. It then evaluates each M_i in turn, obtaining a class-probability vector p. For each p, the algorithm computes the difference between the highest and second-highest probabilities; if this margin meets or exceeds τ, it accepts p and gets terminated. If none of the M_i models is sufficiently confident, the algorithm invokes M_0 on I and returns its output. By tuning τ, one can balance average inference speed against overall accuracy.

This mechanism ensures that the system dynamically selects the most confident model, enhancing prediction robustness and reliability, especially for special or exceptional cases.

Our proposed fallback mechanism is lightweight and computationally efficient and is thus well-suited for resource-constrained edge devices. Unlike reinforcement learning frameworks with repeated interaction, adaptation, and re-weighting of model parameters or Bayesian decision paradigms that are costly at the probabilistic modeling level, our switching based on fixed thresholds is deterministic. This approach avoids the additional computational overhead associated with training loops or probability computations. Additionally, the models deployed using the FlatBuffers format are optimized strictly for inference The models are inference-optimized for fast and efficient deployment on edge devices and are not dynamically updateable or retrainable for reinforcement learning. This characteristic also defines the requirement for a deterministic, fixed model switching since it provides low power consumption and latency with fast decision-making.

5. Result and analysis

5.1. About dataset

For testing and training of the models, we have used a handwritten digit (0-9) classification dataset from MNIST (Modified National Institute of Standards and Technology) [28]. The MNIST dataset is a widely used dataset in machine learning and computer vision for evaluating image classification models. The MNIST dataset contains grayscale images of size 28 x 28 pixels of digits ranging from 0 to 9. The dataset contains 60,000 images for training purpose and 10,000 images for testing purpose. This dataset allows supervised learning of ML models since each image is labeled with the corresponding digit it represents. Fig. 8 visualizes a sample from the dataset. The pixels of images in the dataset range from 0 to 255. The pixels are normalized to a range of 0 to 1 by dividing each pixel by 255 as in equation (26). This normalization aids in stabilizing and accelerating the learning process by keeping pixel values within a smaller standardized range.

$$Normalized\,Pixel\,Value = Original\,Pixel\,Value/255 \tag{26}$$

Fig. 8. Sample Data.

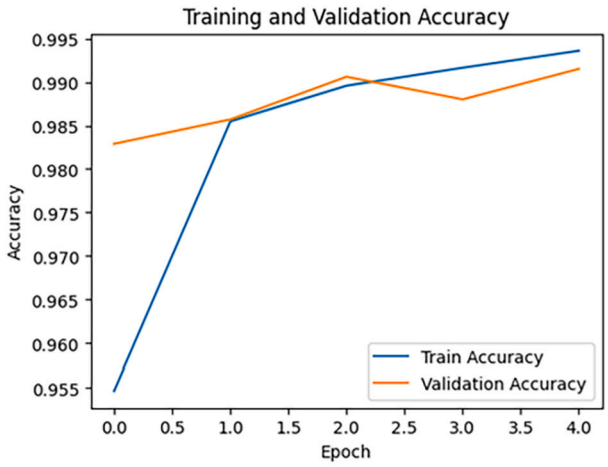


Fig. 9. Model Accuracy.

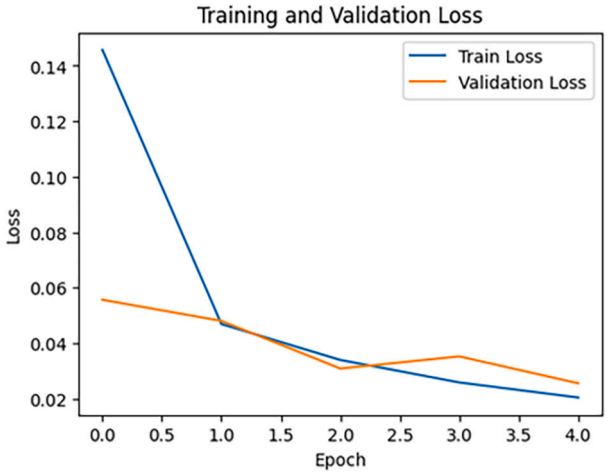


Fig. 10. Model Loss.

5.2. Analysis of models

The main model discussed in initial model subsection is tested using MNIST [28] testing dataset. Upon training we get high accuracy rate of 0.9918. The Training accuracy and loss over the epochs are shown in Fig. 9 and 10 respectively.

Performance metrics of all the optimized models trained using the optimization techniques discussed in previous section is analyzed using testing data. The performance metrics for the models are listed in Table 3.

From these model size comparison graphs in Fig. 11 and Fig. 12, we can observe that the keras models have the larger model size than the flatbuffer ones. The main model and the Sensitivity-based pruned keras model have the highest model size. The dynamic range quantized and full integer quantized models have the least model size.

In Fig. 13- inference time vs model line graph, we can observe that kmeans clustered keras model take the most time to provide inference. Hybrid pruned model take the second highest time followed by the main model and sensitivity based pruned model. The other models are optimized models converted in flatbuffer format that takes comparatively less time than the keras models. Among the flatbuffer models, dynamic

Table 3
Comparison of Model Sizes, Accuracy, and Inference Times.

Model Name	Model Size (MB)	Accuracy	Inference Time (seconds/sample)
Main Model	2.61	0.9918	0.000307
Dynamic Range Quantized	0.22	0.9908	0.000137
Full Int Quantized	0.22	0.1135	0.000215
Non Quantized	0.86	0.9918	0.000169
Hybrid Pruned Keras	0.88	0.9918	0.000585
Sensitivity based pruned Keras	2.61	0.9876	0.000279
Sensitivity based pruned	0.86	0.9876	0.000161
Uniform Clustered Keras	1.1	0.9896	0.000286
Uniform Clustered	0.36	0.9896	0.000144
Kmeans Clustered Keras	1.1	0.9905	0.000439
Kmeans Clustered	0.36	0.9905	0.000145

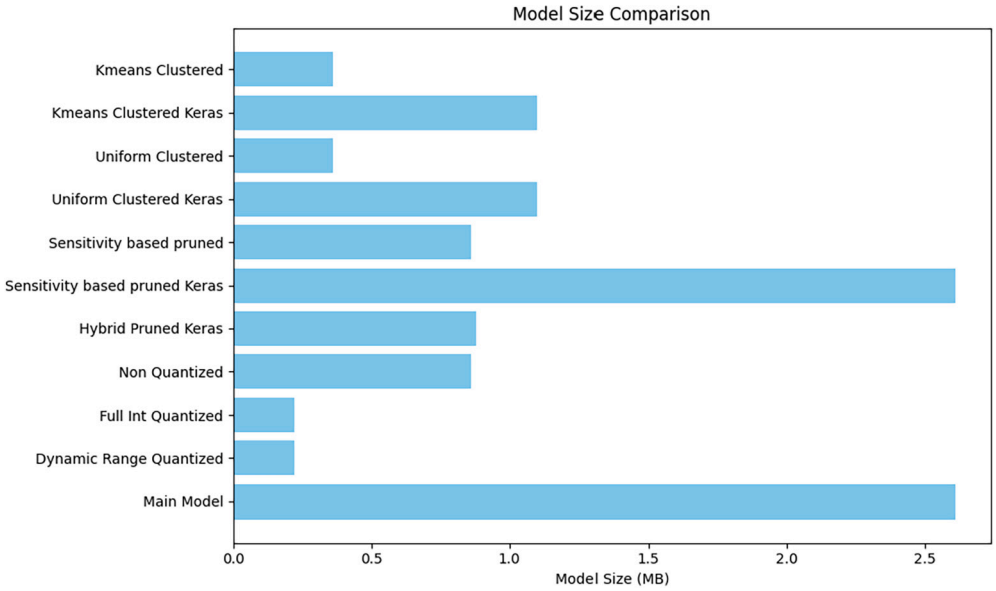


Fig. 11. Model Size Comparison: Bar Graph.

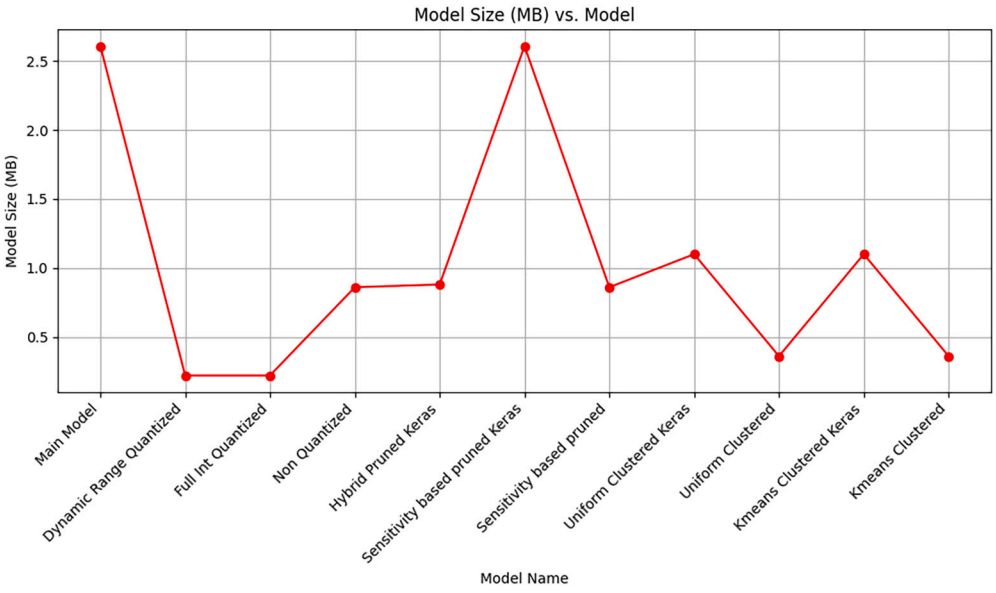


Fig. 12. Model Size Comparison: Line Graph.

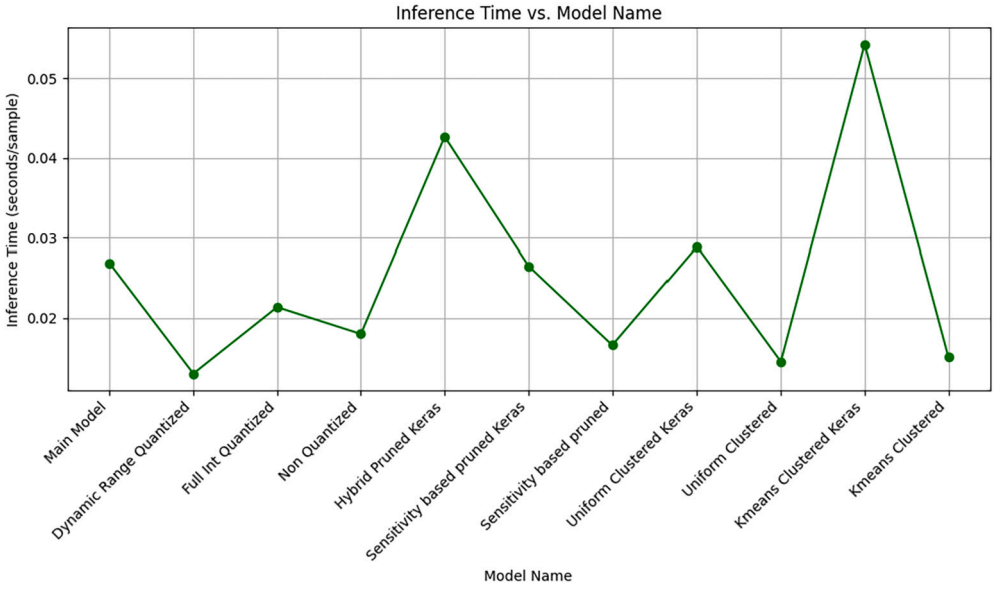


Fig. 13. Inference time comparison.

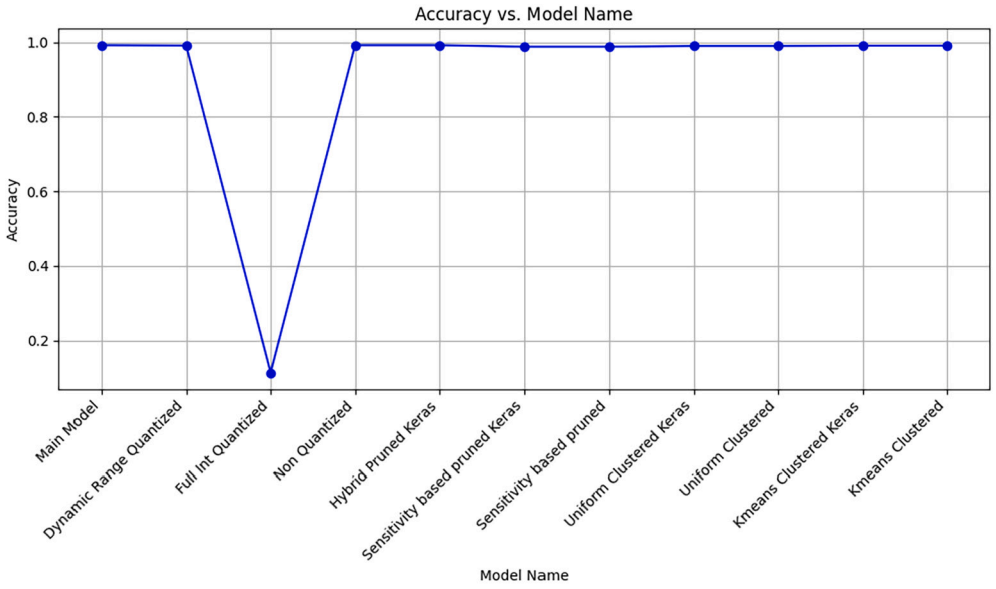


Fig. 14. Accuracy comparison.

range quantized model takes the least time for inference, followed by the other models and full integer quantized model takes the highest time.

The line graph in Fig. 14 compares accuracy of models. We can conclude that full integer quantization has the least accuracy and is not suitable for inference purpose. Other models have good accuracy percentage.

The graph in Fig. 15 visualizes all the comparative parameters of the models. We can observe that dynamic range quantized model and full integer quantized model has the least model size but full integer quantized model has very low accuracy which makes dynamic range quantized flatbuffers model as the best model for resource constraint devices.

Among the evaluated models, the Dynamic Range Quantized model demonstrates the most efficient performance, achieving a significantly reduced model size of 0.22 MB, high accuracy of 99.08%, and the fastest inference time of 0.000137 seconds per sample. These results make it the most suitable candidate for deployment in resource-constrained and latency-sensitive environments.

The Non Quantized model maintains a high accuracy of 99.18% with a moderate model size of 0.86 MB and an inference time of 0.000169 seconds per sample. This balance between accuracy and computational efficiency makes it a strong contender. Similarly, the Sensitivity Based Pruned tflite model achieves a comparable model size of 0.86 MB with slightly lower accuracy of 98.76% and an improved inference time of 0.000161 seconds per sample, making it suitable for scenarios where size and speed are prioritized over peak accuracy.

The Uniform Clustered tflite model offers a compact size of 0.36 MB, accuracy of 98.96%, and a fast inference time of 0.000144 seconds per sample. Meanwhile, the Kmeans Clustered tflite model achieves a slightly higher accuracy of 99.05% with the same size of 0.36 MB but at a marginally slower inference time of 0.000145 seconds per sample. Both models are efficient solutions for applications that demand compression with minimal accuracy loss.

In contrast, the Hybrid Pruned Keras model, while achieving high accuracy of 99.18%, suffers from a significantly slower inference time of 0.000585 seconds per sample, limiting its practical use despite a mod-

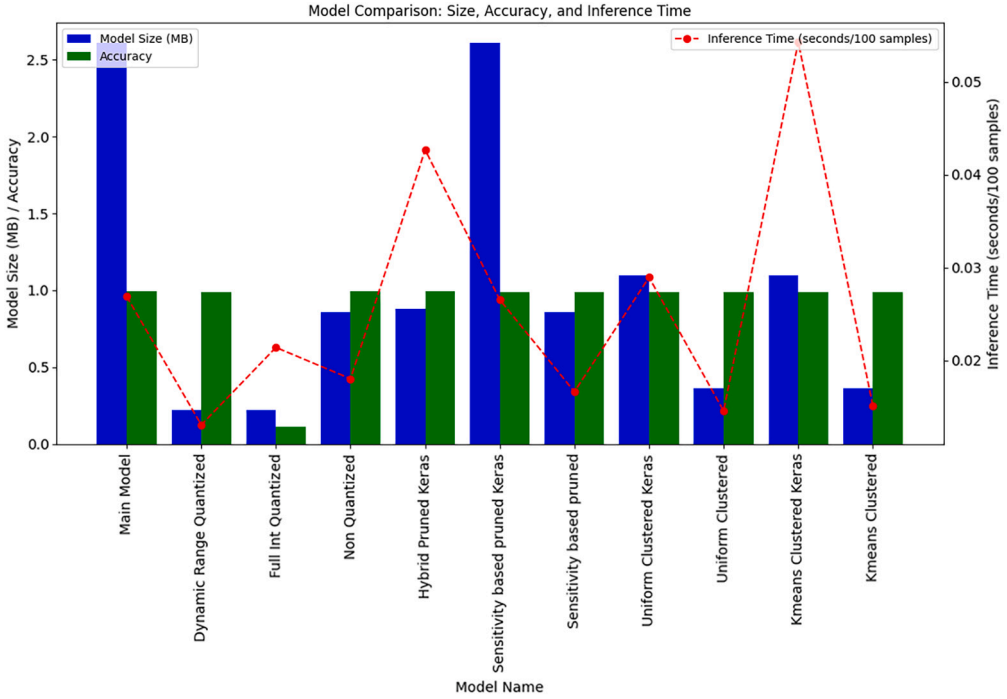


Fig. 15. Comparison of Model Size, Accuracy, and Inference Time.

Table 4
Comparative Summary of Optimization Techniques.

Technique	Size Reduction	Accuracy Change	Inference-Time Improvement	Key Observation
Dynamic Range Quantization	91.6%	−0.10% (→99.08%)	55% faster	Excellent balance of speed, size, and accuracy.
Full Integer Quantization	91.6%	−88.83% (→11.35%)	30% faster	Severe accuracy loss; unsuitable for classification.
Sensitivity-Based Pruning (TFLite)	67.0%	−0.42% (→98.76%)	48% faster	Good compression and speed-up with minimal accuracy drop.
Hybrid Pruning (Keras)	66.3%	−0.00% (→99.18%)	91% slower	No accuracy loss but inference significantly slowed.
Uniform Weight Clustering (TFLite)	86.2%	−0.22% (→98.96%)	53% faster	Strong compression; slight errors on curved digits.
K-Means Weight Clustering (TFLite)	86.2%	−0.13% (→99.05%)	53% faster	Marginal lookup overhead; accuracy near uniform clustering.
Non-Quantized TFLite Conversion	67.0%	−0.00% (→99.18%)	45% faster	Easy conversion yields speed-up without size or accuracy loss.

est size of 0.88 MB. Finally, the Full Int Quantized model achieves an extremely small size of 0.22 MB but exhibits a severe drop in accuracy to 11.35%, rendering it unsuitable for application.

The Dynamic Range Quantized model emerges as the best overall choice due to its optimal trade-off between size, accuracy, and speed. However, alternative models such as Non Quantized, Sensitivity Based Pruned, Uniform Clustered tflite, and Kmeans Clustered tflite provide viable solutions depending on specific application requirements. Models such as Hybrid Pruned Keras and Full Int Quantized, while demonstrating unique characteristics, may not be optimal for practical deployment due to limitations in speed and accuracy, respectively. To directly compare each optimization's impact, Table 4 summarizes relative reductions in model size, changes in accuracy, and inference-time improvements versus the baseline model.

Similarly, multiple models were trained and optimized on CIFAR10 dataset [29]. The performance of the system on the CIFAR10 dataset is discussed in further sections.

5.3. Confidence threshold analysis of fallback system

We tested the multi-model fallback system at different confidence thresholds ranging from 0.05 to 0.9 using MNIST [28] and CIFAR10 [29] testing datasets and obtained performance results as listed in Table 5 and Table 6 respectively. In Table 5, threshold represents the confidence threshold τ, avg time is the average time taken for the system to get inference on a single image, model switches shows the number of times model switching occurred and the columns $M_1.M_2, M_3$ and M_o shows the number of times each model was used. Since M_1 was the first model in the sequential model pool, therefore it was used every single time. Other models where used only when confidence criteria was not met. From the line graphs Fig. 16 and Fig. 17, we can observe that confidence threshold is correlated to inference time and accuracy. The system switches models if the confidence threshold criteria is not met. When confidence threshold criteria is met, the system takes least inference time and provides it comparatively faster. From Table 5, we can observe that as the threshold τ increases, accuracy and inference time also increases. Similarly, Table 6 shows the performance metrics of system performance when two optimized models and a main model are utilized. The line graphs in Fig. 18 and Fig. 19 visualizes the comparison between confidence threshold and accuracy and inference time. Therefore, in both cases setting the confidence threshold between range of 0.25 to 0.5 is optimal because setting it lower or higher will result in higher inference time or accuracy trade-off.

5.4. Qualitative error analysis and threshold impact

This section visualizes the effectiveness of the proposed system using confusion matrices. Fig. 20 shows the baseline confusion matrix for the dynamic range quantized TFLite model on the MNIST test set, with the most prominent misclassifications occurring between 4 and 9, 7 and 1, 3 and 5 indicating that stroke similarity drives these errors.

Table 5
Performance Metrics for Different Thresholds for MNIST.

Threshold	Accuracy (%)	Avg Time (s)	Model Switches	Optimized Model 1 (M_1)	Optimized Model 2 (M_2)	Optimized Model 3 (M_3)	Main Model (M_o)
0.05	98.75	0.0025	17	10000	16	1	0
0.1	98.75	0.0024	32	10000	29	3	0
0.15	98.75	0.0024	49	10000	42	6	1
0.2	98.82	0.0025	69	10000	59	9	1
0.25	98.9	0.0026	85	10000	73	10	2
0.3	98.92	0.0026	109	10000	88	16	5
0.35	98.97	0.0026	127	10000	101	19	7
0.4	98.98	0.0026	146	10000	116	22	8
0.45	98.99	0.0027	168	10000	128	29	11
0.5	99.03	0.0027	196	10000	144	36	16
0.55	99.07	0.0028	225	10000	156	46	23
0.6	99.12	0.0029	266	10000	178	59	29
0.65	99.14	0.003	301	10000	197	69	35
0.7	99.16	0.0031	348	10000	223	82	43
0.75	99.21	0.0035	409	10000	250	98	61
0.8	99.21	0.0036	486	10000	285	119	82
0.85	99.25	0.004	615	10000	349	157	109
0.9	99.22	0.0046	780	10000	427	202	151

Table 6
Performance Metrics for Different Thresholds for CIFAR10.

Threshold	Accuracy (%)	Avg Time (s)	Model Switches	Optimized Model 1 (M_1)	Optimized Model 2 (M_2)	Main Model (M_o)
0.05	85.02	0.0018	168	10000	163	5
0.10	85.30	0.0017	330	10000	316	14
0.15	85.57	0.0022	500	10000	458	42
0.20	85.74	0.0026	689	10000	607	82
0.25	85.82	0.0031	855	10000	735	120
0.30	86.02	0.0035	1028	10000	870	158
0.35	86.32	0.0024	1213	10000	1002	211
0.40	86.54	0.0050	1419	10000	1147	272
0.45	86.72	0.0060	1647	10000	1289	358
0.50	86.86	0.0069	1866	10000	1443	423
0.55	87.02	0.0075	2125	10000	1601	524
0.60	87.21	0.0099	2385	10000	1741	644
0.65	87.24	0.0102	2644	10000	1901	743
0.70	87.47	0.0140	2944	10000	2072	872
0.75	87.46	0.0173	3297	10000	2252	1045
0.80	87.49	0.0201	3698	10000	2477	1221
0.85	87.43	0.0241	4224	10000	2734	1490

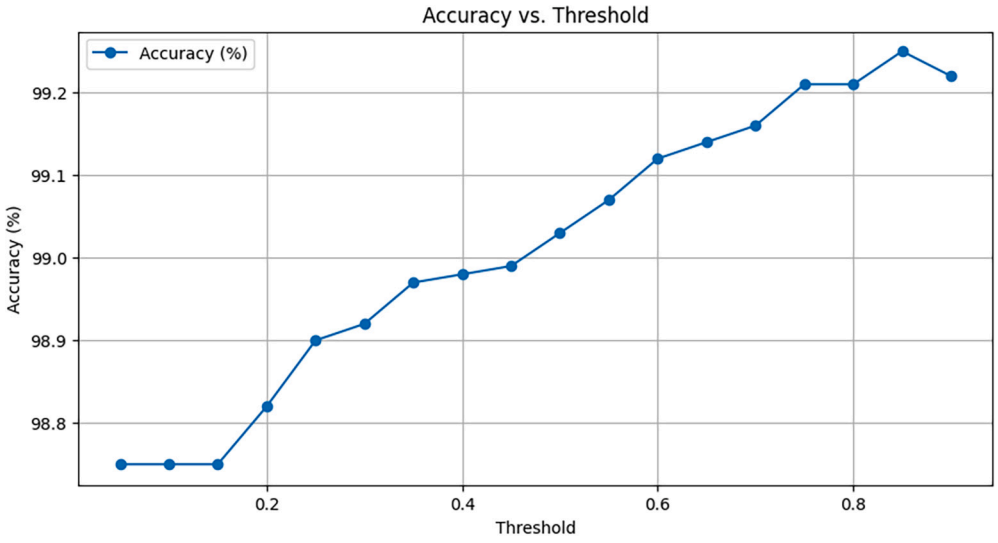


Fig. 16. Accuracy vs Confidence Threshold for MNIST.

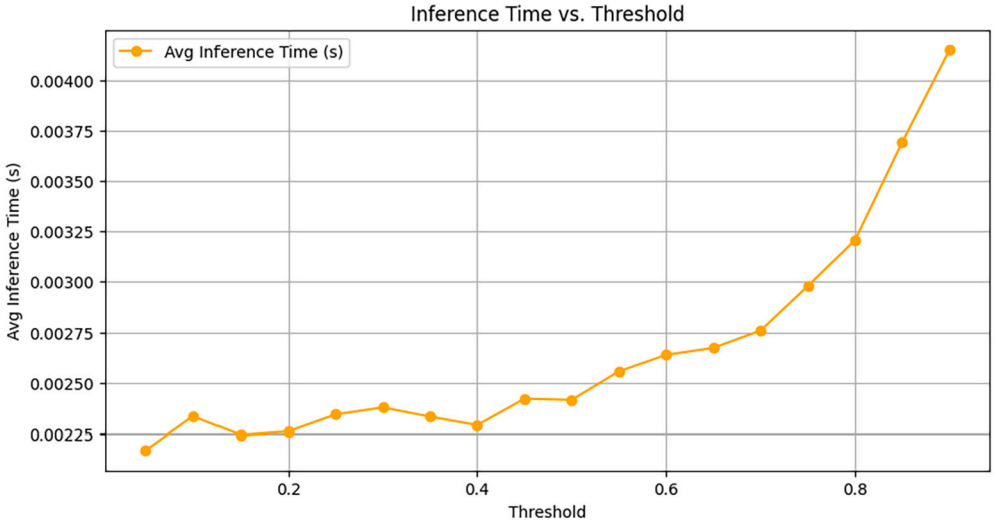


Fig. 17. Inference time vs Confidence Threshold for MNIST.

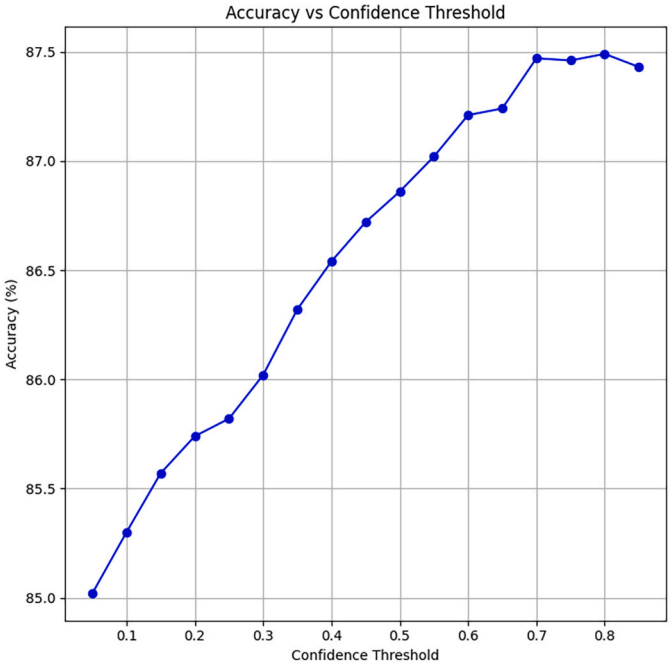


Fig. 18. Accuracy vs Confidence Threshold for CIFAR10.

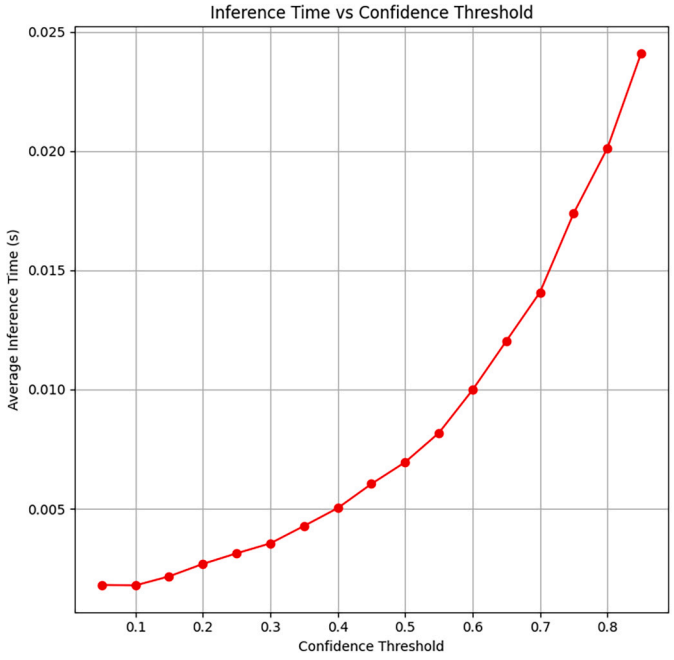


Fig. 19. Inference time vs Confidence Threshold for CIFAR10.

To demonstrate how our multi-model fallback system addresses these errors, we present confusion matrices at three confidence thresholds $\tau = 0.05$, 0.50, and 0.85. Raising τ gradually reroutes ambiguous predictions to the main model, visibly fading the off-diagonal error cells and confirming the fallback mechanism's corrective effect. Fig. 21 Low threshold ($\tau = 0.05$): At this setting, the optimized model handles nearly all inputs, so fallback is rarely invoked. The three primary confusions remain at their baseline levels: 4 to 9 appears 6 times, 7 to 1 appears 3 times, and 3 to 5 appears 6 times. Fig. 22 Moderate threshold ($\tau = 0.50$): Raising τ to 0.50 routes borderline cases to the main model, visibly shrinking the confusion cells. Key confusions decrease for 4 and 9, 7 and 1, 3 and 5 illustrating effective targeted correction with a modest fallback rate. Fig. 23 High threshold ($\tau = 0.85$): With this strict cutoff, only highly confident predictions are handled by the optimized model; the rest default to the main model. The most frequent confusions are now almost eliminated—4 to 9 down to 1, 7 to 1 to 0, and 3 to 5 to 1, thus demonstrating near-complete error correction.

5.5. Hardware testing

The multi-model fallback system with a confidence threshold-based switching algorithm was also tested on an edge device: Raspberry Pi 5 using MNIST [28] testing images, as shown in Fig. 24 and performance of the system was evaluated as listed in Table 7. Threshold represents confidence threshold τ and avg time is the average time taken to get inference of a single input image.

From Fig. 25 and Fig. 26, we can conclude that the multi-model system works efficiently when the threshold is set around 0.3 to 0.6, higher than this range will affect the inference time in applications.

Fig. 27 presents a possible edge implementation of the multi-model fallback system. Here, data is fed into the edge device which consists of the model pool along with the fallback system, the system then provides inference in real time and the inference is exchanged with the IoT network for any further processes.

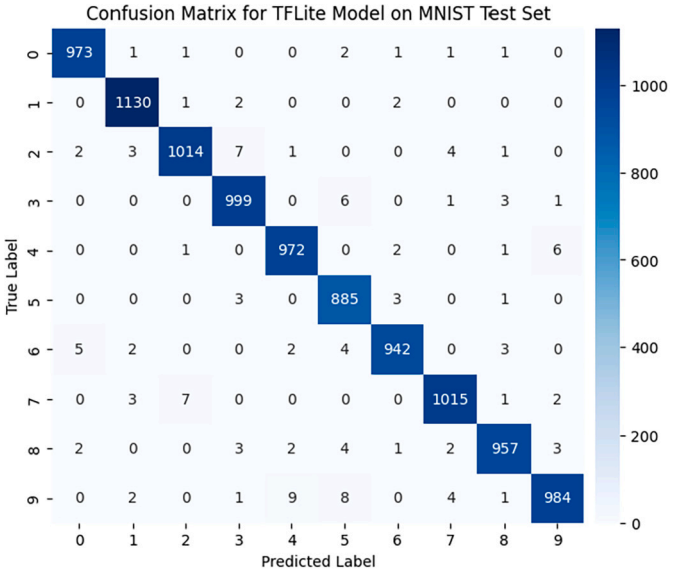


Fig. 20. Confusion matrix for DRQ model on MNIST.

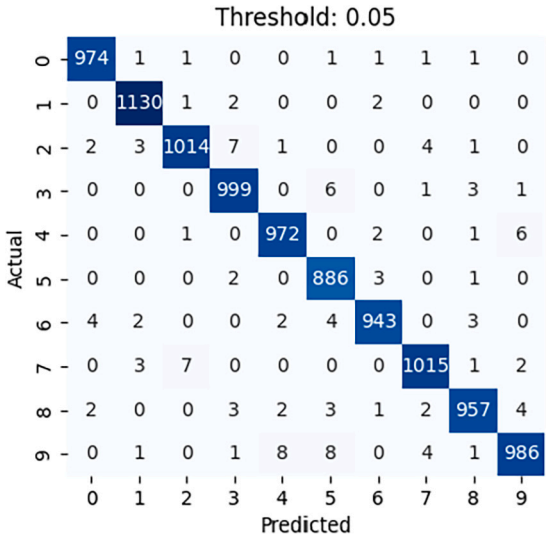


Fig. 21. Confusion matrix for $\tau = 0.05$.

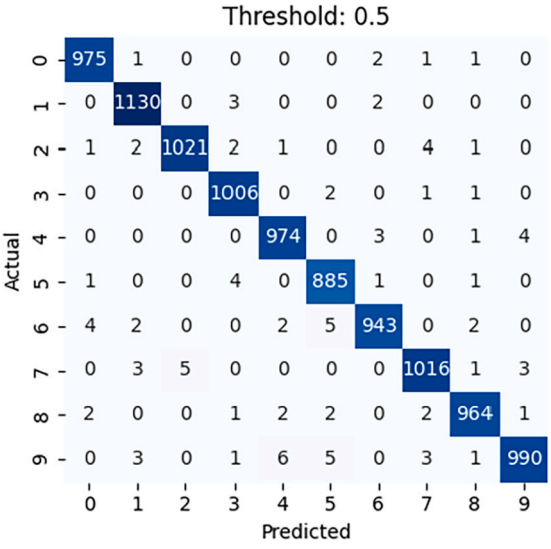


Fig. 22. Confusion matrix for $\tau = 0.5$.

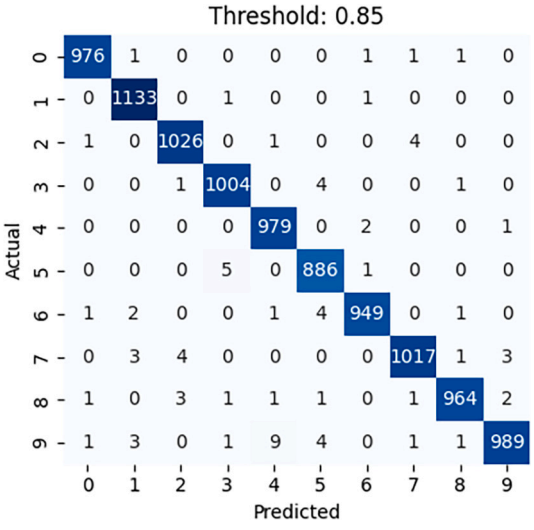


Fig. 23. Confusion matrix for $\tau = 0.85$.

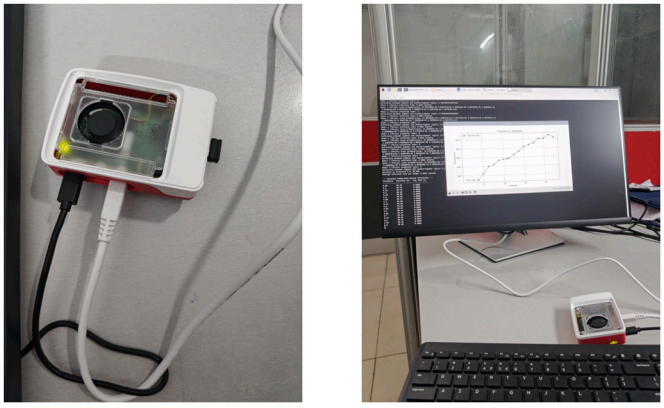


Fig. 24. Experimental Setup.

Table 7
Accuracy Comparison Across Thresholds on Raspberry Pi 5.

Threshold	Accuracy (%)	Avg Time (s)
0.05	98.75	0.0005
0.1	98.75	0.0005
0.15	98.75	0.0005
0.2	98.82	0.0005
0.25	98.90	0.0006
0.3	98.92	0.0006
0.35	98.97	0.0006
0.4	98.98	0.0006
0.45	98.99	0.0006
0.5	99.03	0.0007
0.55	99.07	0.0007
0.6	99.12	0.0009
0.65	99.14	0.0009
0.7	99.16	0.0010
0.75	99.21	0.0012
0.8	99.21	0.0014
0.85	99.25	0.0016
0.9	99.22	0.0021

5.6. Discussion

The experimental analysis shows that a confidence thresholded multi-model fallback pipeline can significantly enhance the reliability and efficiency of the device inference, however bringing this to real world deployment reveals some practical factors. First and foremost

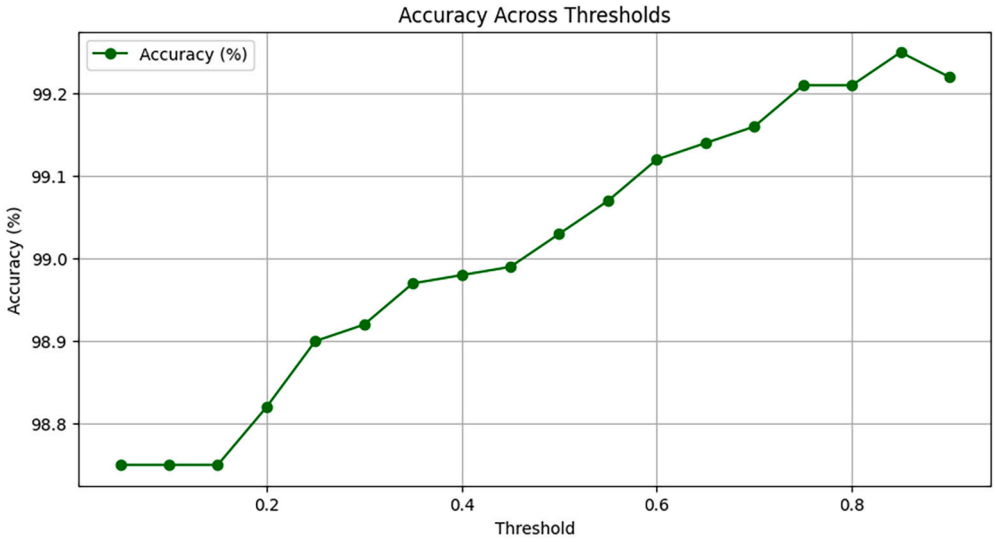


Fig. 25. Accuracy vs Threshold.

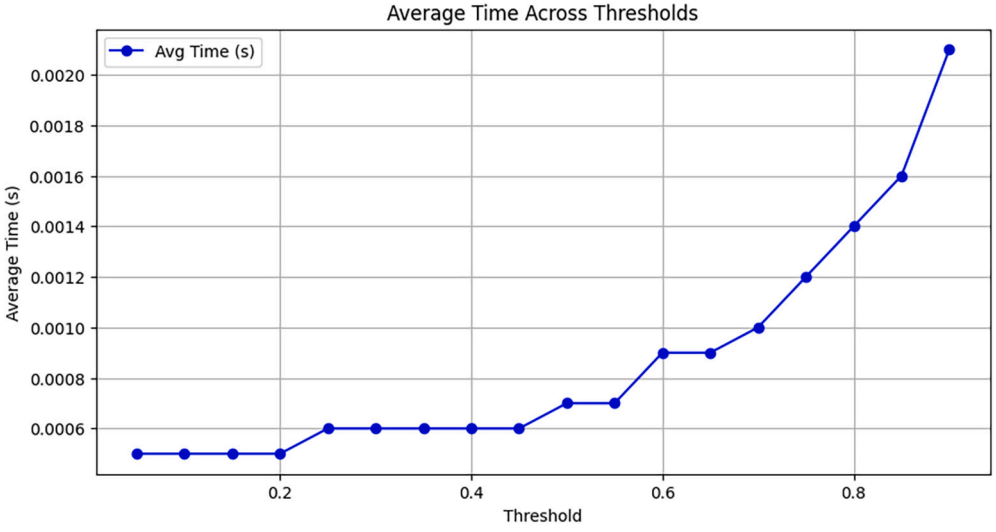


Fig. 26. Average Time vs Threshold.

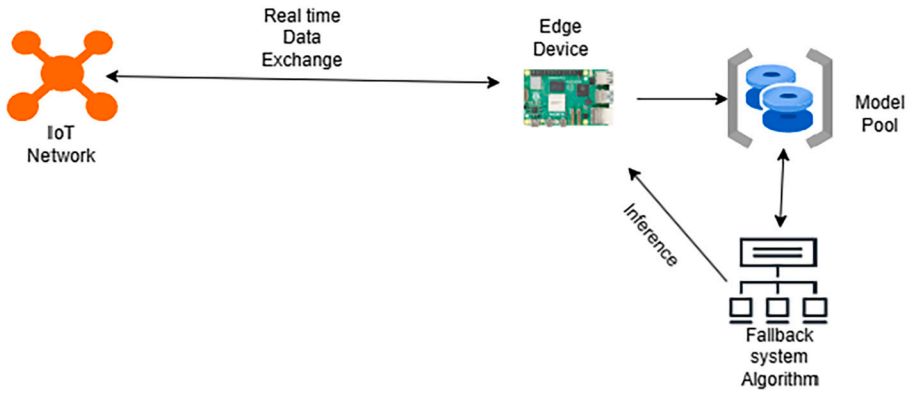


Fig. 27. System Implementation.

are hardware heterogeneity and performance variability: whereas our pruned and quantized models provided 30–55% inference-time savings on a Raspberry Pi 5, such optimizations could lead only 10 to 20% benefits on low-power devices. Furthermore, memory limitations can start to be challenging when caching several model variants. Each taking 0.2 to 0.9 MB on devices with less than 1 MB of flash. Thus there is a need to trade off between the number of backup models against storage capacity and potentially offload second-order variants to external flash or dynamically swap them, albeit with the added cost of latency. Implementation and development costs also add to these difficulties. Handling several optimized models in addition to the primary model increases engineering effort by orders of magnitude, with ongoing integration pipelines that automate quantization, pruning, and clustering processes for every target architecture and dataset. Threshold tuning and calibration introduce another layer of overhead, since the ideal confidence threshold ($\tau \in [0.25, 0.5]$) was dataset-specific and needs to be readjusted on typical field data for each new task. High model switching rates also have overhead on context loading and cache thrashing. With respect of these challenges, with careful profiling of target hardware, automation of multi-model pipelines, and construction of lightweight threshold-tuning tools, practitioners can release adaptive inference systems that make tradeoffs between accuracy, speed, and resource consumption.

6. Conclusion and future work

The proposed multi-model fallback system with confidence threshold-based switching addressed the limitations of traditional single-model architectures by dynamically selecting the most reliable model for inference. This adaptability ensures robust and accurate predictions, even in scenarios with diverse input distributions and noise. In this paper, we have presented and extensively tested a novel TinyML-powered multi-model fallback system based on quantization, pruning and clustering under a single confidence-threshold switching system. Through large-scale experiments on MNIST and CIFAR-10, The proposed approach is able to compress the model sizes by as much as 90% and achieve sub-millisecond inference without compromising accuracy more than 98%. Additionally, we have demonstrated the method scales to actual hardware by running on a Raspberry Pi 5, verifying real-world end-to-end improvements in both latency and resource consumption. It has been observed that setting a threshold between the range of 0.25 to 0.5 is the most optimal choice in our case. Optimal thresholds might vary for each application as it is purely based on the input model pool and the use-case. Therefore, further testing is necessary for usage of the proposed system architecture in other use-cases.

The multi-model fallback system is designed for efficiency in resource-constrained edge environments, but some limitations still exist. The performance of the system depends on the hardware capabilities, which means that the inference time may vary between different edge devices. It also boosts adaptability but still has a chance of adding some minor computational overhead in few cases. Another limitation is utilization of flatbuffer format for models, which reduces the inference time but they cannot be retrained or fine-tuned on the device by itself.

Despite these advances, several constraints temper the generality and performance of the proposed pipeline. Inference-time improvements proved highly platform-dependent—speed-ups of 30%–55% on the Raspberry Pi 5 may shrink to 10%–20% on microcontrollers. The requirement to cache multiple optimized variants can exhaust flash memory on devices with under 1 MB capacity. Engineering complexity is also elevated by the need to integrate quantization, pruning, clustering and threshold-tuning workflows across diverse hardware targets. Finally, frequent model switches may induce cache-thrashing, increased power draw and thermal spikes in tightly constrained embedded systems.

Future research could focus on on-device continual learning or federated update schemes could be introduced to adapt both the model pool and confidence threshold to non-stationary field data. Energy-aware scheduling frameworks may be used to optimize inference accuracy and power consumption in battery-powered deployments. The fallback pipeline could be extended to multi-modal inputs—such as audio, time-series or sensor fusion—to assess its applicability beyond vision benchmarks. Finally, embedding AutoML-driven routines to automate model selection, compression and confidence-margin calibration would reduce manual tuning and accelerate end-to-end deployment for new edge applications.

CRediT authorship contribution statement

Gaurav Kadve: Writing – original draft, Visualization, Software, Methodology, Data curation. **Abishi Chowdhury:** Writing – review & editing, Visualization, Validation, Resources, Formal analysis. **Vishal Krishna Singh:** Writing – review & editing, Visualization, Validation, Investigation, Formal analysis. **Amrit Pal:** Writing – review & editing, Supervision, Resources, Methodology, Formal analysis, Conceptualization.

Declaration of competing interest

All authors declare that they don't have any conflict of interest.

Data availability

Data will be made available on request.

References

[1] L. Kong, J. Tan, J. Huang, G. Chen, S. Wang, X. Jin, S.K. Das, Edge-computing-driven internet of things: a survey, ACM Comput. Surv. 55 (8) (2022) 1–41.
[2] X. Kong, Y. Wu, H. Wang, F. Xia, Edge computing for internet of everything: A survey, IEEE Internet Things J. 9 (23) (2022) 23472–23485.
[3] K. Cao, Y. Liu, G. Meng, Q. Sun, An overview on edge computing research, IEEE Access 8 (2020) 85714–85728.
[4] H. Hua, Y. Li, T. Wang, N. Dong, W. Li, J. Cao, Edge computing with artificial intelligence: a machine learning perspective, ACM Comput. Surv. 55 (9) (2023) 1–35.
[5] S. Iftikhar, S.S. Gill, C. Song, M. Xu, M.S. Aslanpour, A.N. Toosi, S. Uhlig, AI-based fog and edge computing: a systematic review, taxonomy and future directions, Internet of Things 21 (2023) 100674.
[6] M.S. Murshed, C. Murphy, D. Hou, N. Khan, G. Ananthanarayanan, F. Hussain, Machine learning at the network edge: a survey, ACM Comput. Surv. 54 (8) (2021) 1–37.
[7] G. Carvalho, B. Cabral, V. Pereira, J. Bernardino, Edge computing: current trends, research challenges and future directions, Computing 103 (5) (2021) 993–1023.
[8] S. Soro, TinyML for ubiquitous edge AI, arXiv preprint arXiv:2102.01255, 2021.
[9] A.M. Hayajneh, M. Hafeez, S.A. Zaidi, D. McLernon, TinyML empowered transfer learning on the edge, IEEE Open J. Commun. Soc. (2024).
[10] R. Sanchez-Iborra, A.F. Skarmeta, Tinyml-enabled frugal smart objects: challenges and opportunities, IEEE Circuits Syst. Mag. 20 (3) (2020) 4–18.
[11] M.N. Ramadan, M.A. Ali, S.Y. Khoo, M. Alkhedher, AI-powered IoT and UAV systems for real-time detection and prevention of illegal logging, Results Eng. 24 (2024) 103277.
[12] S. Nilnoree, T. Mizutani, An innovative framework for incorporating iPhone LiDAR point cloud in digitized documentation of road operations, Results Eng. (2025) 103953.
[13] A. Morchid, Z. Oughannou, R. El Alami, H. Qjidaa, M.O. Jamil, H.M. Khalid, Integrated internet of things (IoT) solutions for early fire detection in smart agriculture, Results Eng. 24 (2024) 103392.
[14] H. Ramzey, M. Badawy, A.A. Elbaset, Crude oil industry remote monitoring and management based on industrial Internet of things and edge computing integration: a comprehensive survey, Results Eng. 103034 (2024).
[15] S. Teerapittayanon, B. McDanel, H.T. Kung, Branchynet: fast inference via early exiting from deep neural networks, in: 2016 23rd International Conference on Pattern Recognition (ICPR), IEEE, December 2016, pp. 2464–2469.
[16] G. Menghani, Efficient deep learning: a survey on making deep learning models smaller, faster, and better, ACM Comput. Surv. 55 (12) (2023) 1–37.
[17] T. Hoefler, D. Alistarh, T. Ben-Nun, N. Dryden, A. Peste, Sparsity in deep learning: pruning and growth for efficient inference and training in neural networks, J. Mach. Learn. Res. 22 (241) (2021) 1–124.
[18] H. Wu, P. Judd, X. Zhang, M. Isaev, P. Micikevicius, Integer quantization for deep learning inference: principles and empirical evaluation, arXiv preprint arXiv:2004.09602, 2020.

[19] S. Han, H. Mao, W.J. Dally, Deep compression: compressing deep neural networks with pruning, trained quantization and Huffman coding, arXiv preprint arXiv:1510.00149, 2015.
[20] T. Liang, J. Glossner, L. Wang, S. Shi, X. Zhang, Pruning and quantization for deep neural network acceleration: a survey, Neurocomputing 461 (2021) 370–403.
[21] L. Capogrosso, F. Cunico, D.S. Cheng, F. Fummi, M. Cristani, A machine learning-oriented survey on tiny machine learning, IEEE Access (2024).
[22] B. Rokh, A. Azarpeyvand, A. Khanteymoori, A comprehensive survey on model quantization for deep neural networks in image classification, ACM Trans. Intell. Syst. Technol. 14 (6) (2023) 1–50.
[23] A. Kuzmin, M. Nagel, M. Van Baalen, A. Behboodi, T. Blankevoort, Pruning vs quantization: which is better?, Adv. Neural Inf. Process. Syst. 36 (2023) 62414–62427.
[24] S. Ameen, K. Siriwardana, T. Theodoridis, Optimizing deep learning models for raspberry Pi, arXiv preprint arXiv:2304.13039, 2023.
[25] V. Tsoukas, A. Gkogkidis, E. Boumpa, A. Kakarountas, A review on the emerging technology of TinyML, ACM Comput. Surv. (2024).
[26] M. Cho, K.A. Vahid, S. Adya, M. Rastegari, Dkm: differentiable k-means clustering layer for neural network compression, arXiv preprint arXiv:2108.12659, 2021.
[27] W. van Oortmerssen, Flatbuffers: a memory efficient serialization library, Web Page https://android-developers.googleblog.com/2014/06/flatbuffers-memory-efficient.html, 2014.
[28] Y. LeCun, C. Cortes, C.J. Burges, MNIST Handwritten Digit Database, AT&T Labs, 2010.
[29] A. Krizhevsky, V. Nair, G. Hinton, The CIFAR-10 dataset, online: http://www.cs.toronto.edu/kriz/cifar.html, 55(5), 2.