INSTANT: Inference-Aware Fast Feedforward Networks

RENAN BERAN KILIC, Department of Information Engineering and Computer Science, University of Trento, Trento, Italy
KASIM SINAN YILDIRIM, Department of Information Engineering and Computer Science, University of Trento, Trento, Italy
GIOVANNI IACCA, Department of Information Engineering and Computer Science, University of Trento, Trento, Italy

Many embedded applications have strict energy, memory, and time constraints, making neural network (NN) inference particularly challenging. Recently, a novel NN architecture, called Fast Feedforward Networks (FFFs), has been proposed to achieve inference with extremely lightweight computational demands and minimal latency. Yet, compared to feedforward networks with similar sizes, FFFs still lag behind in terms of performance, indicating that they do not utilize all of their parameters effectively. In this article, we explore a possible reason for this performance gap: the uncertainty in how samples are assigned to the network's leaves. We attempt to overcome this challenge by making FFFs' training inference-aware, hence introducing Inference-Aware Fast Feedforward Networks (IAFFFs). We imitate FFFs' inference during training by using a step activation function alongside the traditional sigmoid activation function. We test different aware scheduling methods, which we dub "awareness scheduler", to adjust the balance between the two activation functions during training, and examine how different schedules impact the model's performance. Additionally, we employ leaf-weight virtualization with inference-aware retraining to compress our models so they can fit onto edge devices. We further employ an iterative compression approach to find an optimal awareness scheduler for compression to minimize performance drop due to compression. We experiment with different model sizes on various microcontrollers (MCUs) with different memory constraints to observe the latency and energy consumption introduced by the compression algorithm.

CCS Concepts: • **Computer systems organization** → *Embedded software*; • **Computing methodologies** → *Neural networks;*

Additional Key Words and Phrases: Fast feedforward networks, inference-aware models, efficient machine learning

ACM Reference Format:
Renan Beran Kilic, Kasim Sinan Yildirim, and Giovanni Iacca. 2026. INSTANT: Inference-Aware Fast Feedforward Networks. *ACM Trans. Embedd. Comput. Syst.* 25, 4, Article 62 (June 2026), 38 pages. https://doi.org/10.1145/3815117

This work was funded by the European Union (Project No. 101071179). The views and opinions expressed are those of the authors only and do not necessarily reflect those of the European Union or EISMEA. Neither the European Union nor the granting authority can be held responsible for them.
Authors' Contact Information: Renan Beran Kilic, Department of Information Engineering and Computer Science, University of Trento, Trento, Italy; e-mail: renanberan.kilic@unitn.it; Kasim Sinan Yildirim (corresponding author), Department of Information Engineering and Computer Science, University of Trento, Trento, Italy; e-mail: kasimsinan.yildirim@unitn.it; Giovanni Iacca, Department of Information Engineering and Computer Science, University of Trento, Trento, Italy; e-mail: giovanni.iacca@unitn.it.

ACM 1539-9087/2026/06-ART62
https://doi.org/10.1145/3815117

1 Introduction

The rise of **Internet of Things** (**IoTs**) devices, such as smartphones, wearables, drones, and smart speakers [48], has significantly changed the way we interact with the world. These devices utilize their sensing, computing, networking, and communication capabilities to collect, process, and transmit various types of data, including images, audio, wireless signals, and text, from individuals and their environments. In recent years, advancements in **Artificial Intelligence** (**AI**), particularly in **Neural Networks** (**NNs**), have driven the integration of AI with IoT, leading to the emergence of the concept known as the **AI of Things** (**AIoTs**) [46]. However, while AI models are typically resource-hungry, most IoT devices are resource-constrained [5], which creates an urgent need for AI models that are energy-, memory-, and computation-efficient [57].

Several approaches have been proposed to address this challenge. Among these, recent works presented [2] and then expanded [13] ***Fast Feedforward Networks*** (**FFFs**), a fast and computationally-efficient inference algorithm specifically designed for resource-constrained IoT devices. FFFs split the input space into regions using a differentiable binary tree, with shallow **Feedforward Networks** (**FFs**) at the leaves. This neural architecture achieves logarithmic time complexity, enabling substantially faster inference compared to traditional FFs with the same number of parameters. However, this lower computational complexity also has some drawbacks: (1) it comes at the cost of *performance degradation*, which is particularly noticeable in tiny models that are especially beneficial for devices with strict memory constraints; and (2) the *model size increases exponentially* as the binary tree grows, due to the increase in the number of leaves. This rapid growth makes larger FFFs difficult to deploy on memory-constrained embedded IoT platforms. To the best of our knowledge, only a few studies focused on FFFs for IoT tasks and edge deployment [9, 13, 26]. Therefore, improving the performance of FFFs and their memory efficiency is still an open research area.

In this article, we introduce INSTANT, an end-to-end framework for the efficient deployment of tiny FFFs into various embedded platforms with different resource constraints.[1] Figure 1 provides an overview of the INSTANT deployment pipeline, which comprises inference-aware training and model compression to generate an accurate and optimized model, as well as platform synthesis to produce the code that executes the model on the selected target platform. To realize INSTANT, we first explore the reasons behind the performance degradation of FFFs by assessing their accuracy on three common inference tasks, namely image recognition, human activity recognition, and audio processing, respectively using the MNIST [15], **Human Activity Recognition** (**HAR**) [24, 42], and Google **Speech Commands** (**SC**) v2 [53] datasets.

As FFF models have only recently been introduced, there are only a few works that have addressed their main limitations. To the best of our knowledge, the few existing FFF-based approaches aim at hardening the node decisions at inference time [2], or to balance the sample distribution of the leaves [8]. Yet, their performance still lags behind that of standard FFs. In this work, we first try to understand the reason for such a performance gap. Our preliminary evaluations (Section 3.3) indicate that this performance drop is primarily due to the standard training procedure for FFFs, which employs sigmoid activation, while, in contrast, the inference process uses step activation, creating a mismatch that results in prediction errors at the leaves. To address this, we propose a novel inference-aware training approach, which uses what we call an *awareness scheduler* during training to balance training and inference activations, thereby reducing the mismatch between them. We refer to models trained with our method as **Inference-Aware Fast Feedforward Networks** (**IAFFFs**).

To tackle the memory overhead of IAFFFs, we employ leaf-weight virtualization [26], a compression method specifically designed for FFFs. Leaf-weight virtualization divides model leaves into

[1] https://github.com/DIOL-UniTN/INSTANT

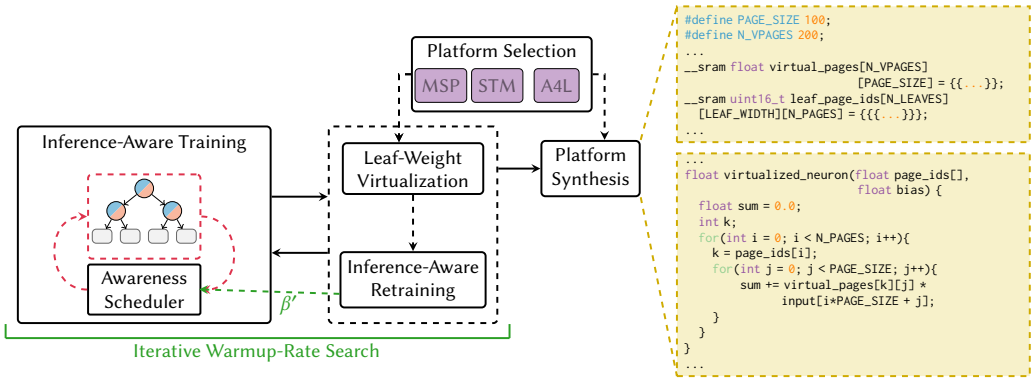


Fig. 1. Deployment pipeline for running Inference-Aware Fast Feedforward Networks (IAFFFs) on resource-constrained platforms. MSP, STM, and A4L refer to MSP432, STM32, and APOLLO4L, respectively. The selected device's configurations, such as memory size and memory section names, are passed to the relevant components. Compression is done by iterative leaf-weight virtualization via a warmup rate search, which iteratively searches for an optimal warmup rate β'. β' is updated as in Algorithm 1.

pages and groups similar leaves within the same virtual page, so that during inference, only the virtual pages need to be stored and loaded to their corresponding leaf locations. Specifically, we combine leaf-weight virtualization with the introduced inference-aware retraining to compress IAFFF models with minimal performance drop, making them suitable for devices with limited resources. To minimize the performance drop after virtualization, we employ an iterative procedure to search for an optimal awareness scheduler that minimizes the performance drop of the virtualized IAFFF model. This approach limits the accuracy drop after compression, having up to 1.5% less accuracy drop after compression compared to not using the approach.

We also implement a runtime library to run virtualized models on different edge platforms, such as MSP432P401R, STM32L476RG, and APOLLO4 Blue Lite. To evaluate our framework, we perform a comprehensive comparison on the MNIST, HAR, and SC datasets focusing mainly on the runtime overhead, accuracy, and memory footprint. Our results show that for models of comparable sizes, IAFFFs deliver notable accuracy improvements across all datasets while introducing negligible energy and latency overhead due to leaf-weight virtualization. In particular, (1) IAFFFs consistently outperform FFFs in terms of accuracy, with gains of up to 6% on MNIST, and around 2% on SC and HAR. (2) Leaf-weight virtualization introduces some latency overhead, around 8.5 msec on HAR, and around 2 msec on SC and MNIST, but achieves an 8× compression ratio and delivers substantially higher performance, up to around 20% improvement on SC compared to equally sized FFFs and FFs.

The rest of the article is structured as follows: In the next section (Section 2), we introduce background concepts on FFFs, such as their elements, the training and inference processes, and the loss function used for training. In Section 3, we first introduce two entropy-related metrics used in our experiments and then define the capacity of FFFs, which helps explain the performance gap between FFFs and FFs. We then provide details on the datasets and models employed,[2] and analyze the limitations of FFFs through preliminary experiments, focusing on their model capacity and the uncertainty of their samples. Section 4 presents our proposed method, IAFFFs, which aims at improving FFF performance by making training inference-aware and increasing certainty

[2]These metrics, datasets, and models are presented in Section 3 because they are used in our preliminary experiments, presented in the same section.

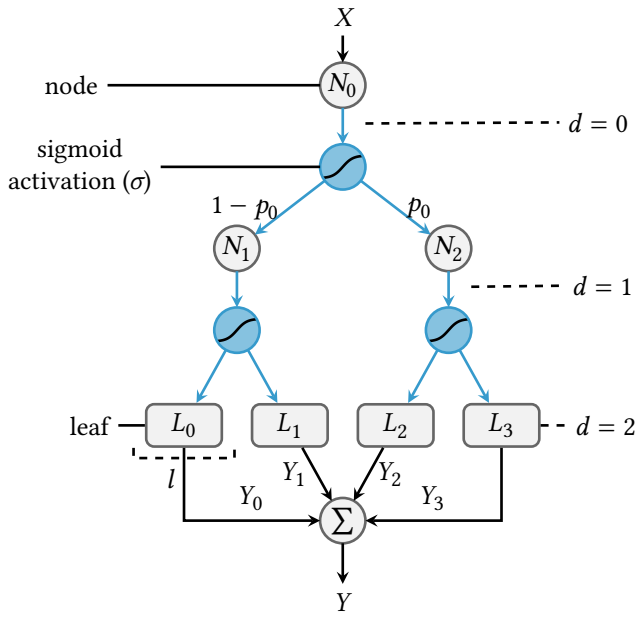


Fig. 2. An FFF with $f_{d,l}$ configuration, where depth (d) equals 2, and the leaf width is l. p_0 denotes the probability of X being routed to the branch on the right (N_2), while $1 - p_0$ denotes the probability of it being routed to the branch on the left (N_1).

when selecting leaves. We then present our proposed scheduling method to optimize the training process. Section 5 describes how we use leaf-weight virtualization with inference-aware retraining by iteratively optimizing the awareness scheduler, and how we automate the deployment of virtualized models on resource-constrained devices. Then, Section 6 presents the methods and experimental results, respectively. In Section 7, we discuss related work, including previous studies on model compression, runtime of tiny models, and the use of conditional models. In Section 8, we outline potential directions for improving IAFFFs and INSTANT, as well as their contributions to expand the literature. Finally, Section 9 concludes this work.

2 Fast Feedforward Networks: Training, Inference, and Loss

As mentioned earlier, our pipeline leverages FFFs as the core machine learning model In this section, we present the key background concepts underlying our FFFs.

2.1 Fast Feedforward Networks

In essence, an FFF is a balanced binary-tree structured NN, designed to benefit from the fact that different neurons are activated by various regions of the input space [2, 13]. As shown in Figure 2, an FFF with a depth of d has two main types of computing blocks: $2^d - 1$ nodes (whose set is indicated with N), and 2^d leaves (whose set is indicated with L). Each node can be seen as a single neuron computing a weighted sum of its inputs and bias. The leaves are instead tiny FFs with a single hidden layer of size l (*leaf width*), with each leaf specifically trained to handle different regions of the input space.

2.1.1 Training. Having an FFF $f_{d,l}$ with depth d and leaf width l, its training width is defined as $2^d l$. During training, each node i with weights W_i takes (in batch) an input X to produce p_i, i.e., the

probability of an input being routed to the branch on the right:

$$p_i = \sigma(X \cdot W_i + b_i)$$

where σ is the sigmoid activation function. Accordingly, $1 - p_i$ is the probability of X being routed to the branch on the left. Indicating with I_L and I_R the set of branches on the left and on the right, respectively, we can calculate the path/leaf probability P_i, i.e., the probability that an input is routed to a leaf f_i, as follows:

$$P_i = \prod_{i \in I_R} p_i \prod_{i \in I_L} (1 - p_i). \tag{1}$$

To obtain the network output given an input X, we then sum, across all leaves, the output of each leaf, $f_i(X)$, multiplied by the corresponding path/leaf's probability:

$$Y = \sum_{i=0}^{2^d} P_i \cdot f_i(X).$$

2.1.2 Inference. During inference, an FFF $f_{d,l}$ uses only a single leaf, following a path according to each node's probability. This is because, for the purpose of inference, the sigmoid activation is cast to a step activation [2]:

$$p_i = S(X \cdot W_i + b_i),$$

where the step function S is defined as

$$S(x) = \begin{cases} x & \text{if } x \geq 0 \\ 0 & \text{if } x < 0 \end{cases}$$

From Equation (1), since $p_i \in \{0, 1\}$, it results that $P_{i=l'} = 1$, and $P_{i \neq l'} = 0$, i.e., only a single leaf l' will classify the input such that the output in the inference is

$$Y = f_{l'}(X).$$

2.1.3 Hardening Loss. Let $\mathcal{L}$ be the loss due to the outputs of an FFF. Belcak and Wattenhofer introduced a regularized loss [2] as $\mathcal{L}' := \mathcal{L} + h\mathcal{L}_h$, where

$$\mathcal{L}_h := \sum_{S \in B} \sum_{N \in \mathcal{N}} H(N(S)), \tag{2}$$

is the so-called *hardening loss* [2], $\mathcal{N}$ is the set of nodes, $B \in D_S$ is a batch of samples, $H(\cdot)$ is the entropy of a Bernoulli random variable, and h is a training hyperparameter controlling the effect of the hardening loss.

3 Experimental Analysis of Performance Degradation in FFFs

In this section, we first present the metrics (Section 3.1) along with the datasets and models (Section 3.2) used in our preliminary experiments. Then, we discuss the primary performance weaknesses of FFFs and explore their possible causes through in-depth analyses of overfitting, model capacity, and input sample certainty (Sections 3.3 to 3.5). Then, we discuss how these analyses motivate our proposed improvements to the FFF training process. In our analysis, we primarily compare FFFs with FFs, as both perform the same operations but differ in key performance metrics, namely accuracy (training, validation, and test), computational cost (training and inference), and parameter count. Note that all metrics calculated for validation and test sets are based on inference (as formulated in Section 2.1.2), where the model makes hard decisions (based on a step activation) in the routers.

3.1 Metrics of Interest in FFFS

3.1.1 Number of Parameters. Even though only a single leaf is executed at inference time, FFFs have an exponential number of parameters. In fact, the number of total parameters in an FFF with depth d and leaf width l is

$$\gamma = 2^{d-1}\gamma_N + 2^d l\gamma_L,$$

where γ_N and γ_L are the numbers of parameters in each node and leaf, respectively. Hence, increasing the depth increases the model size exponentially, and just as in FFs, performance may scale with model size.

3.1.2 Leaf Oscillation Rate. Given an input sample X, we define its *leaf change vector* as

$$\mathbf{c} := \{c_1, c_2, \dots, c_{T-1}\} := \{\,|\operatorname{sgn}(a_i - a_{i-1})| \;\mid\; i = 2, 3, \dots, T\}, \tag{3}$$

where $0 \leq a_i \leq L$ is the leaf the sample is routed to at epoch i, T is the number of training epochs, and $\operatorname{sgn}(\cdot)$ indicates the sign function. Using Equation (3), we measure the stability of leaf selection (i.e., of nodes' routing) by defining the leaf oscillation rate vector for any FFF-based model $f_{d,l}$ as

$$= \frac{1}{T-1}\sum_{i=1}^{T-1} c_i.$$

3.1.3 Entropy as a Measure of Certainty. Indicating with P_i the path probability of an input sample being routed to the ith leaf, as in Equation (1), we define the entropy of that sample as

$$H = -\sum_{i=0}^{2^d} P_i \log_{2^d} P_i. \tag{4}$$

This metric essentially measures how "stable" the choice of the leaf is for the given input sample.

3.1.4 Entropy as a Measure of Leaves' Sample Balance. We can apply a similar approach to assess the "balance" of the sample distribution for each leaf, providing insight into how balanced the overall tree is. Let the sample ratio of the ith leaf be defined as

$$r_i = \frac{|X_i|}{|X|}, \tag{5}$$

where $|X|$ is total number of samples and $|X_i|$ is the number of samples that are routed to leaf i. Using r_i, we can define the leaves' sample balance H' as

$$H' = -\sum_{i=0}^{2^d} r_i \log_{2^d} r_i. \tag{6}$$

3.1.5 FFFs' Capacity. If we define model capacity as the number of parameters for both FF and FFF models, this assumption aligns well with FFs, since their computational complexity scales sublinearly with the number of parameters. For FFFs, if we assume that the routing mechanism assigns samples to leaves perfectly—that is, each sample is directed to the most "appropriate" leaf (i.e., the one that correctly classifies it), and all leaves are effectively utilized—then the capacity is also used efficiently. Under this assumption, no part of the model is wasted, and it is reasonable to treat the number of parameters as a fair measure of capacity for FFFs as well, a perspective we adopt throughout this article.

3.2 Datasets and Models

We evaluate the proposed method by testing it on three datasets that have been selected to represent three different application domains. The three considered datasets are MNIST [15], HAR [24, 42], and Google SC v2 [53]. We briefly describe the three datasets below.

- **MNIST** can be seen as an example of a simple, yet realistic image-based application. It includes 10 gray-scale hand-written digits, with images of size 28×28, such that each sample has 784 features.
- **HAR** is an example of a wearable device application where the goal is to classify human activities based on sensor data. This dataset contains six activity classes (e.g., walking, standing). Data are collected by a gyroscope and an accelerometer, which capture 3-axis angular velocity and 3-axis acceleration, respectively, at a sampling rate of 50 Hz. This results in 3 data points for each axis per sensor. We use 1-second frames as input to our model for both training and inference, where each frame contains 300 features, i.e., two 3-axis feature sets (one for each sensor) sampled over 50 time steps.
- **SC** is an example of an audio-based application. The data consist of 1-second speech samples, each sampled at 16 kHz. The dataset includes 35 distinct word classes (e.g., yes, no, up, down). Ten of them are used as commands by convention. The other words are considered to be auxiliary and labeled as "unknown". A subset of the "unknown" class with a similar number of samples with respect to the other classes is included to avoid data imbalance. The set also contains noise samples of different lengths, and a 1-second random frame is taken from them during training. In the end, 12 classes are used during training, including 10 command classes, one unknown class, and one noise class. We preprocess each audio sample by extracting 13 Mel-Frequency Cepstral Coefficients, obtained by applying a Short-Time Fourier Transform with a window size of 25 ms and a hop size of 16 ms, resulting in 793 (61×13) input features for each 1-second sample. For each set, we take 10% of the training set as the validation set.

In our experiments, we use FFs, FFFs, and IAFFFs with different architectures. It should be noted that all FFF models are trained using the above-mentioned hardening loss [2] setting h to 1 (except where noted), as part of standard training. In contrast, the hardening loss is not applied to IAFFFs: based on our preliminary experiments, using the hardening loss on IAFFFs does not yield satisfactory performance, as this hardening loss tends to route many input samples into only one or two leaves, limiting the model's performance. Moreover, we use FFFs with higher leaf width on the HAR dataset to achieve similar model complexity as on the other datasets, since HAR has fewer input features than MNIST and SC.

3.3 The Gap Between Inference and Training

Given FF and FFF models of the same size, we observe in Figure 3 that (1) in FFs, validation and test accuracies are similar, indicating good generalization; and (2) FFFs' validation accuracy is notably higher than test accuracy. This is due to the fact that, by design, FFFs tend to overfit difficult samples by routing them to many (or even all) leaves during training, reducing loss but making the decision-making mechanism in the nodes less meaningful, and not leveraging their efficient binary-tree structure. As a result, the model's performance drops when evaluated during inference. Moreover, this gap is further exacerbated by the inherent structural difference between training and inference, as the model uses a sigmoid activation during training but a step activation during inference.

3.4 Uncertainty Lowers Accuracy

One of the main bottlenecks of FFF models is their nodes' inability to send some input samples to a specific leaf. As a result, these samples oscillate between different leaves during training (we

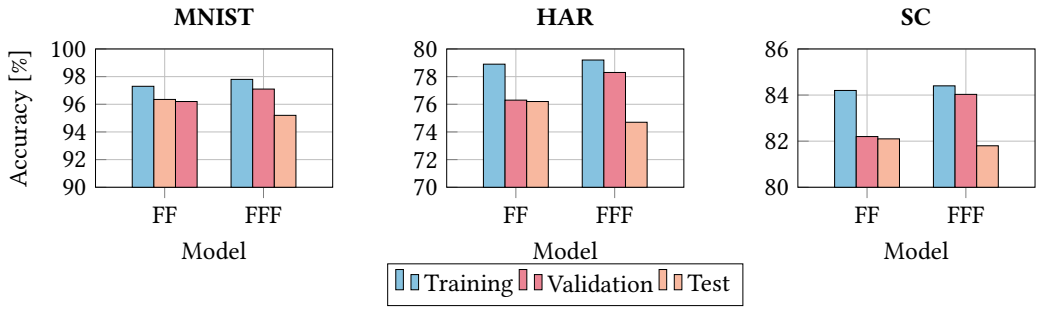


Fig. 3. Training, validation, and test accuracies of FF and FFF models with a similar size (~100kB) on MNIST, HAR, and SC. The FFF models have a depth of 2, with a leaf width of 8 for MNIST and SC, and a leaf width of 16 for HAR. The FF models are single-layer with a width of 32 for MNIST and SC, and a leaf width of 64 for HAR.

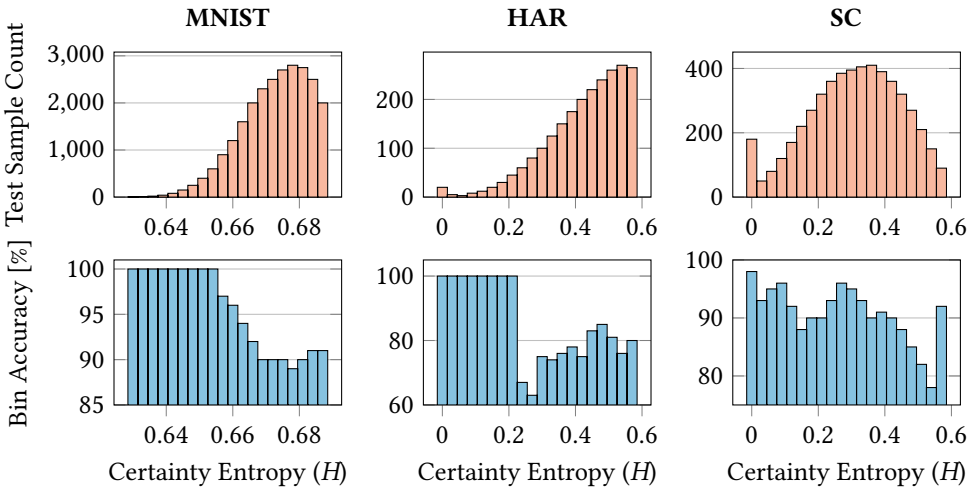


Fig. 4. The top histogram shows the number of samples vs. certainty entropy, while the bottom histogram shows the bin accuracy vs. certainty entropy. The bin accuracy corresponds to each bin's accuracy in the top histogram. Both histograms are based on results from a pre-trained FFF model, using $f_{2,8}$ for MNIST and SC, and $f_{2,16}$ for HAR.

will discuss this effect later in Section 6.1.3). Instead of a single leaf specializing in those samples, multiple leaves overfit them.

We examine this concept in Figure 4 by showing the number of test samples and their accuracies with respect to certainty entropy calculated as in Equation (4). In the figure, sample count and bin accuracy represent the number of samples and the accuracy in the corresponding entropy bin. Knowing that certainty entropy is high if a sample is "uncertain" (i.e., the choice of the leaf it is routed to is more stable) and low otherwise, it can be seen that in all datasets a large number of samples are characterized by high uncertainties. Accordingly, the accuracy of samples tends to decrease with their certainty entropy. This trend is especially visible for MNIST and HAR. On SC, even though the uncertain sample count does not increase with the entropy, again, uncertain samples tend to be incorrect often. Each case contains numerous uncertain samples, which, if misclassified, can significantly impact the model's accuracy. These samples are, in fact, the primary

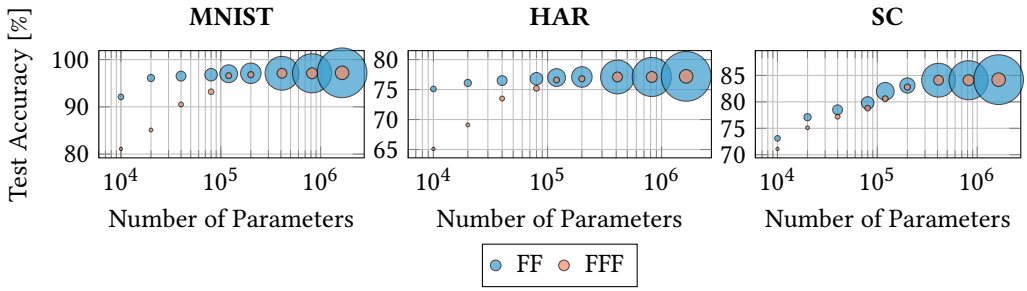


Fig. 5. Test accuracy, number of parameters, and inference cost (represented by circle size) for different FF and FFF models. All FF models are single-layer, while FFF models have a depth of 2.

contributors to the observed performance drop between validation and test. We believe this happens because of inadequate training of the nodes, which are, instead, supposed to make more confident decisions. To improve FFF performance, it is therefore crucial to reduce this uncertainty—something we aim at achieving by refining the training process at the node level.

3.5 Model Capacity: The Gap Between FFs and FFFs

For linear models like FFs and FFFs, we expect them to have similar performance if their capacity is similar. To verify this, we compare the performance of varying FF and FFF models with different architectures to determine if this is indeed the case.

In Figure 5, we see the number of parameters and their corresponding accuracy of varying FF and FFF models. It can be seen that FFF models offer much faster inference compared to FFs with the same size. Yet, even for smaller FFF models, we would expect similar performances to FF models, given that they have the same capacity. In practice, this is not the case. Across all three datasets, we observe a consistent trend: as the model size increases, all FFF models eventually converge to similar accuracy levels as FFs in all the datasets. Yet, FF models perform remarkably better than the FFFs when the model size is small, especially on MNIST and HAR; SC already benefits from the high capacity of FFF models with a smaller drop in accuracy. Furthermore, FFF models tend to perform worse at small sizes. This suggests there is a major bottleneck in extremely small FFF models, which is particularly relevant for use cases like resource-constrained devices where memory and computation are highly constrained. We believe one reason for this performance drop is that the leaves, having smaller widths and fewer neurons than FFs of the same model size, are less capable of solving the classification problem at hand, especially when there is high uncertainty on the routing of many samples; in these cases, the resulting loss is higher.

The main findings of these preliminary analyses, which motivate our next steps, can be summarized as follows:

> While FFFs demonstrate sufficient capacity, their nodes' predictions sometimes show uncertainty. This could potentially be improved by refining the model architecture or optimizing the training process at the node level to produce more confident outputs.

4 Inference-Aware Fast Feedforward Networks

In this section, we outline the key steps we took to improve the training process by addressing the generalization and performance bottlenecks of FFFs discussed in the previous section. As illustrated in the block on the left of Figure 1, our solution includes two main components: an inference-aware training procedure and an awareness scheduler, which are described in the next.

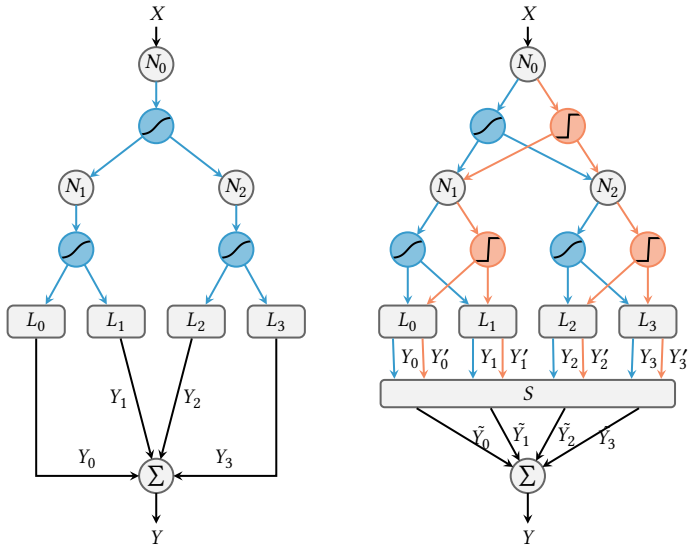


Fig. 6. Comparison of the training processes for FFFs (left) and IAFFFs (right), both using a $f_{2,l}$ configuration. Blue circles indicate sigmoid activation functions, while orange circles indicate step activation functions. S denotes the awareness scheduler.

4.1 Inference-Aware Training

Following the insights from our experimental analysis in Section 3.3, we modify the FFF training process to incorporate the step activation that is normally used at inference time already during training—a strategy we call *inference-awareness*, since it aligns the training dynamics with inference behavior. We refer to the resulting models as IAFFFs.

In Figure 6, we see the training processes of FFF and IAFFF models side-by-side. For IAFFFs, the steps are similar to FFFs. The two main differences are (1) the activation functions and (2) the awareness scheduler (S). As seen earlier, FFFs use sigmoid activation during training, but switch to step activation (where nodes make hard decisions) during inference. To make FFFs aware of this change, in IAFFFs, we introduce step activations alongside sigmoid activations already during training. This reduces the dissimilarity between training and inference. Furthermore, we use an awareness scheduler (see the next subsection) to control the degree of inference-awareness during IAFFF training. We use this scheduler to improve performance by balancing sigmoid-activated and step-activated outputs when calculating the loss. The sigmoid-activated outputs represent the training outputs, while the step-activated outputs represent inference outputs, reflecting their roles in vanilla FFFs.

We now present a mathematical description of the IAFFF training. Following similar steps to those described in Section 2.1.1, we can derive the step-activated node outputs Y'_i, as

$$p'_i = \Theta\left(X \cdot W_i + b_i\right)$$

$$P'_i = \prod_{i \in I_R} p'_i \prod_{i \in I_L} (1 - p'_i)$$

$$Y'_i = P'_i \cdot f_i(X),$$

where $\Theta(x)$ indicates the step activation function defined as

$$\Theta(x) = \begin{cases} 1 & \text{if } x \geq 0 \\ 0 & \text{if } x < 0. \end{cases}$$

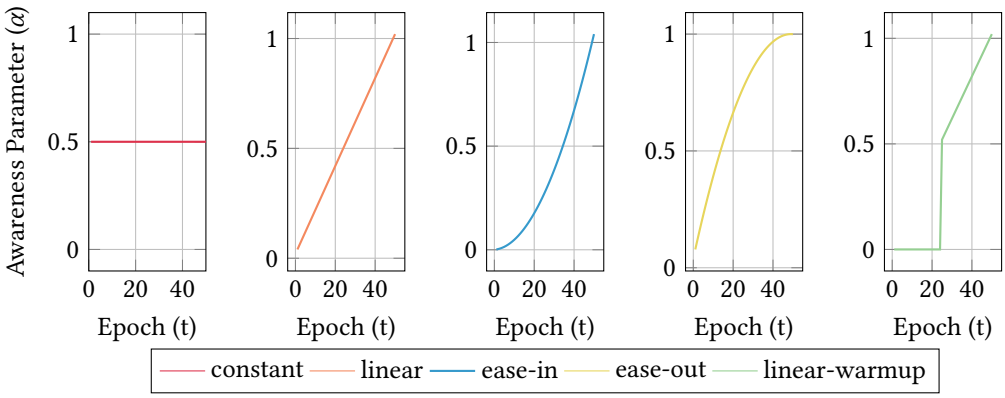


Fig. 7. Different awareness schedules experimented for inference-aware training.

By combining these step-activated node outputs Y_i' with the sigmoid-activated node outputs Y_i (calculated as the original FFFs described in Section 2.1.1), we can then obtain the output of the network, which is given by

$$Y = S(Y_i, Y_i') = \sum_{i=0}^{2^d} ((1-\alpha)Y_i + \alpha Y_i'), \tag{7}$$

where α, determined by the awareness scheduler, controls the balance between the outputs of the two activations.

4.2 Awareness Scheduler

During training, we experiment with different awareness schedulers to optimize the balance between sigmoid-activated (training) and step-activated (inference) outputs. As shown in Equation (7), the scheduler sets α, which determines the weighting between the training-output Y_i (sigmoid-activated) and inference-output Y_i' (step-activated) in the combined network output Y, directly influencing the loss computation and guiding the model toward the desired balance between training and inference behaviors. Higher values of α favor inference-like behavior, while lower values favor training-like behavior.

To evaluate how quickly (or gradually) α should increase, i.e., how training should become closer to inference over the epochs, we test several schedulers,[3] shown in Figure 7, namely constant, linear, ease-in, ease-out, and linear-warmup, respectively defined as

$$\begin{aligned}
\text{Constant:} \quad & \alpha = 0.5 \\
\text{Linear:} \quad & \alpha = \frac{t}{T} \\
\text{Ease-in:} \quad & \alpha = \left(\frac{t}{T}\right)^2 \\
\text{Ease-out:} \quad & \alpha = \left(\frac{t}{T} - 1\right)^2 \\
\text{Linear-warmup} \quad & \alpha = \begin{cases} \frac{t}{T} & \text{if } t \geq \beta T \\ 0 & \text{if } t < \beta T \end{cases}
\end{aligned} \tag{8}$$

[3]See Appendix A for additional experiments.

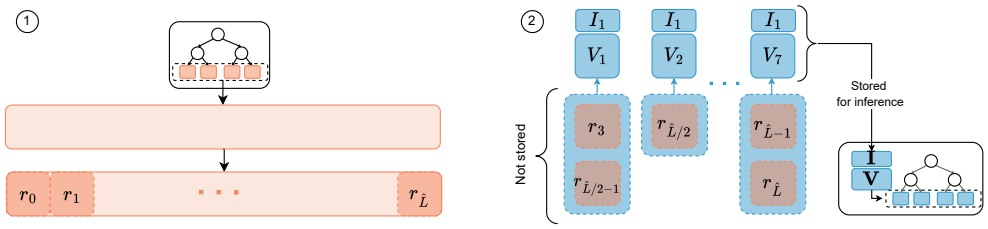


Fig. 8. Leaf-weight virtualization illustration. (1) Leaf weights are flattened and divided into pages, called *leaf-weight pages*. (2) Then, the leaf-weight pages (r_i) are clustered into *virtual pages* (**V**). Only virtual pages and *leaf-page indices* (**I**) are stored for inference.

Here, T represents the total number of training epochs, and β denotes the warmup rate, which determines the number of epochs at the start of training in which $\alpha = 0$. Apart from the constant scheduler, each scheduler increases the awareness parameter α over the epochs at different rates. In all cases, training starts in a *training-heavy* regime ($\alpha < 0.5$) to expose leaves to a broader range of data, and then gradually shifts to an *inference-heavy* regime ($\alpha > 0.5$) to prevent overfitting and align training with inference.

5 Compression and Implementation for Edge Devices

This section explains the compression method used to deploy IAFFFs on our devices, and gives details on both the algorithmic and practical implementation aspects.

5.1 Inference-Aware Virtualization via Iterative Warmup-Rate Update

For large models, fitting them on memory-constrained devices without major performance loss is challenging, and pruning and quantization are widely used techniques to address this. In our approach, we use leaf-weight virtualization to compress the IAFFFs. Recently, findings have shown the benefits of this compression technique on FFFs due to their leaves having strong similarity with each other, allowing redundant information to be clustered and compressed [26].

As illustrated in Figure 8, leaf-weight virtualization divides the weights of individual leaves into $\hat{L}$ pages called *leaf-weight pages*, clustering similar weights together. These pages are then mapped to a smaller set of *virtual pages* **V**, each representing a cluster of similar leaf-weight pages, and their original locations in the leaves—i.e., *leaf-page indices* stored in an *index matrix* **I**—allow the network to reconstruct the full leaf-weights during inference, so that only the virtual pages and index matrix need to be stored, reducing memory usage. To minimize accuracy loss after virtualization, we perform retraining with $\alpha = 1$, ensuring that the leaves are trained under the same inference-like conditions they will face during deployment, which is consistent with the inference-aware training approach.

Since the scheduler plays a role in compression both directly (pretraining) and indirectly (retraining), it might have different effects on the performance of the compressed model. We experiment with this hypothesis by training and virtualizing IAFFFs with a linear-warmup scheduler[4] while varying β, and Figure 9 presents the results. The results show that model performance depends on β, and the optimal value varies for each dataset. Furthermore, the figure illustrates that the optimal β values are different between virtualized (V-IAFFF) and unvirtualized (IAFFF) models. Because deviations from this optimal β reduce performance, which is especially evident in virtualized IAFFFs, it is then necessary to determine the optimal warmup rate for each dataset.

[4]Since this scheduler achieved the best performance in our experiments, see Section 6.1.

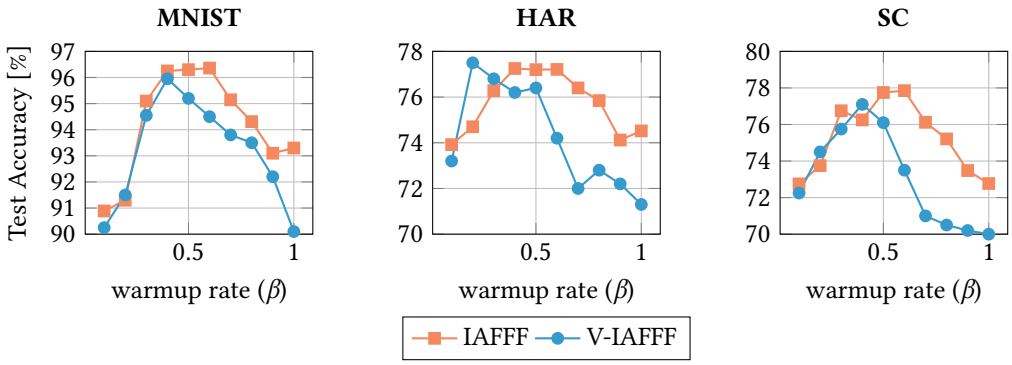


Fig. 9. V-IAFFF (virtualized IAFFF) models accuracy with varying warmup rate (β) with $f_{2,8}$, $f_{2,16}$, and $f_{2,8}$ architectures for MNIST, HAR, and SC, respectively, yielding 2x compression.

ALGORITHM 1: The algorithm for iterative leaf-weight virtualization via binary warmup-rate search. The trainVirtualize() method trains the model f using warmup-rate β', then virtualizes and retrains it again using the same warmup-rate. The validate() method gets the inference accuracy of the model using the validation data.

Data: Step size $n = 0.05$, array $B = [0.0, 0.05, \ldots, 1.0]$, model f, virtualized model f'
Result: Optimal warmup-rate β'
bestAccuracy $\leftarrow$ 0.0;
left $\leftarrow$ 1;
right $\leftarrow \frac{1}{n} + 1$; `/* Index of last element in B */`
while *left* $\leq$ *right* **do**
  mid $\leftarrow \lfloor (left + right)/2 \rfloor$; `/* Index of current β' */`
  $\beta' \leftarrow B$[mid];
  $f' \leftarrow$ trainVirtualize(f, β');
  accuracy $\leftarrow$ validate(f');
  if *accuracy* $\geq$ *bestAccuracy* **then**
    bestAccuracy $\leftarrow$ accuracy;
    left $\leftarrow$ mid+1 ; `/* Try a larger β' */`
  end
  else
    right $\leftarrow$ mid−1 ; `/* Try a smaller β' */`
  end
end
return β'

To efficiently find the optimal warmup rate, we employ binary search (see Algorithm 1). Starting from $\beta = 0.5$, we iteratively halve the search range, retraining the model with different warmup rates and evaluating validation performance after compression, until the best β is identified. This value is then used for final model compression.

Decompression often requires additional operations, such as index mapping, whose computational cost depends on the compression method and its configuration. In virtualized models, this index mapping is performed during inference. To support virtualized IAFFFs on diverse edge platforms,

we developed a library that generates both the runtime environment and the necessary code. An example simplified code of the procedure can be seen on the right side of Figure 1. The library is lightweight and user-friendly, enabling straightforward deployment of virtualized IAFFF models on edge devices. It automatically adapts to device specifications, such as memory size and section names, to fit model parameters and generate the necessary code for efficient inference.

5.2 Implementation Details

In the following sections, we describe the implementation details of our deployment pipeline, including the details of the training using the awareness scheduler and the model compression with leaf-weight virtualization via inference-aware retraining. We also discuss how we implemented our platform synthesis to deploy IAFFFs on resource-constrained devices.

We implemented the inference-aware training of our models and the virtualization process in Python using the PyTorch framework. Moreover, we implemented our platform synthesis (i.e., code generation and memory tiling) in pure C code using our simple Python synthesis library. The generated C code handles model deployment and runtime execution, ensuring efficient deployment on memory- and compute-constrained hardware.

We selected three low-power devices, namely MSP432P401R (MSP432), STM32L476RG (STM32), and APOLLO4 Blue Lite (APOLLO4L), for our runtime experiments to represent a range of commonly used low-power devices in embedded systems. We selected these devices to cover a range of memory limitations and system capabilities. Specifically:

- **MSP432P401R** has 64 kB of SRAM, with the MCU clock speed set to 32 MHz.
- **STM32L476RG** has 128 kB of SRAM, with the MCU clock speed set to 64 MHz.
- **APOLLO4 Blue Lite** has 385kB of TCM (tightly coupled memory), with the MCU clock speed set to 96MHz (low power mode) or 128 MHz (high performance mode).

Model inputs, activations, and weights are stored in the **Tightly Coupled Memory** (**TCM**) on the APOLLO4L, while on the other two devices, they are stored in SRAM. Storing inputs, activations, and weights in these memories, which are closest to the processor, minimizes latency and ensures the fastest possible inference.

5.2.1 Time and Energy Measurement. In all our on-device experiments, we have run the models 100 times to ensure the reliability of the results, and the measurements are averaged across the runs. In the latency measurement experiments, the APOLLO4L device was set to low power mode (when set to high performance mode, approximately half the latency is expected). Furthermore, we have not activated any compiler optimizations to compare and evaluate the overhead introduced by virtualization accurately. This allows us to avoid any optimizations that might interfere with the virtualization process and ensures that the generated C code is as close as possible to the original C code. While measuring the energy, we measured the current dissipation from VDD through devices using a high-precision **Source Measure Unit** (**SMU**), the Tektronix Keithley 2450 SMU. We then calculated the energy by multiplying the current by the voltage and the latency. We set the devices into sleeping modes to see the power consumption when they were idle, and then we woke them up to run the inference and measured the energy consumption during inference. The energy consumption was averaged across the 100 available runs to ensure the measurements are representative of typical device behavior. For memory transfer measurements, we transferred a 1,000-sized vector again 100 times.

5.2.2 Inference-Aware Training and Awareness Scheduler. As discussed in Section 4, the training process of IAFFFs is similar to that of FFFs, but with a change in the activations used during training. By adding the step activation alongside the sigmoid activation we aim at balancing the effect of those activations on training (see Section 4.1). Specifically, this balance is achieved by weighting these

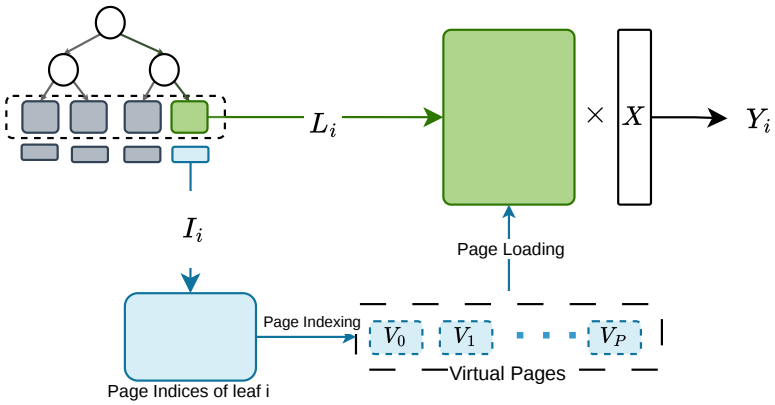


Fig. 10. Page loading of a virtualized IAFFF during inference.

activation outputs as in Equation (7). Simply setting $\alpha = 0.5$ throughout training does not yield satisfactory accuracy (see Section 6.1), so we employ an awareness scheduler to adjust α dynamically. The chosen scheduler starts with $\alpha = 0$ and gradually increases it at varying rates, optionally including a warmup phase, to transition effectively from training-heavy to inference-heavy behavior.

5.2.3 Virtualization and Inference-Aware Retraining. After training the model, we apply leaf-weight virtualization to compress it, as described in Section 5.1. Specifically, we carry out the virtualization as in [26], by first computing the Fisher information matrix for each leaf, which is then combined with the magnitude of each leaf page to determine the cost of matching pages. To preserve accuracy, the virtualized model is retrained in the inference-heavy state ($\alpha = 1$), ensuring that the leaves learn under the same conditions they will encounter during inference.

5.2.4 Platform Synthesis. After training and virtualization, our Python synthesis library generates the C code required to run the models on target devices (see the Platform Synthesis block in Figure 1). This generated code includes model parameters stored as C arrays in a header file, device memory sections, and the functions necessary for inference. For virtualized models, the header file also includes virtual pages $\mathbf{V}$ and leaf-weight page indices $\mathbf{I}$, enabling reconstruction during inference. The code also provides the necessary functions to run both the virtualized and uncompressed models, including the inference function that selects between standard neuron computations or virtualized neuron computations to produce the model's output.

IAFFFs use two separate sets of parameter matrices: one for internal leaf-decision nodes, and one for leaf computations (i.e., for classification). During inference, the model's binary tree structure directs the input sample toward one of the leaves based on the decisions made after each node. After virtualization, the decision part (i.e., the node computations) of the network remains unchanged, while the leaf computations use the shared virtual pages and index matrix, reducing the total number of parameters while preserving functionality.

During inference, we load the virtual pages to their corresponding location in the leaves, as illustrated in Figure 10. The virtual pages matrix $\mathbf{V}$ has dimensions $P \times N' \times S$, where P is the number of virtual pages, S is the page size, and N' is the number of virtual pages. The index matrix of a leaf i, $\mathbf{I_i}$, has dimensions $l \times \hat{N}$, where l is the leaf width, and $\hat{N}$ is the number of pages in a leaf. If a sample is classified by the ith leaf, we retrieve the page indices for the calculation of neuron j from the vector $\mathbf{I'} := \mathbf{I}_{i,j}$. The kth virtual page is then loaded from $\mathbf{V}_{\mathbf{I'}_k}$. With this virtualization implementation, we can deploy the model and evaluate the latency overhead introduced by loading virtual pages at inference time, and explore performance, latency, and memory tradeoffs.

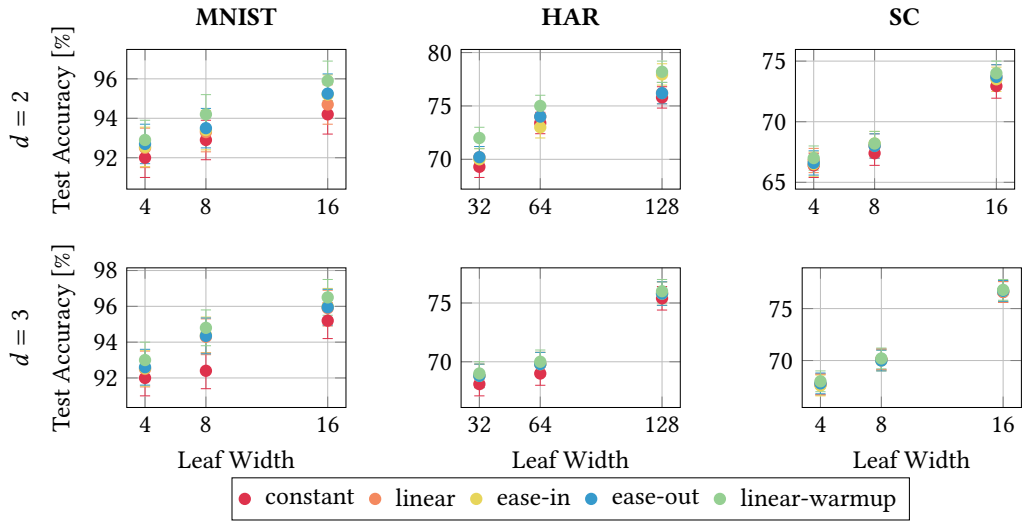


Fig. 11. Accuracy comparison of IAFFF models with different architectures and awareness schedulers across varying datasets.

6 Evaluation

In the following sections, we present our experimental results. Section 6.1.1 compares IAFFF models trained using different awareness schedulers. Sections 6.1.2 to 6.1.5 highlight the advantages of IAFFFs over alternative models and explore the underlying factors behind their performance. Section 6.3 covers runtime experiments and evaluates the effectiveness of virtualization compared to other compression techniques for deploying IAFFFs on resource-constrained devices.

All models in all experiments have been trained for 50 epochs using the Adam optimizer with a learning rate of 0.002 and a mini-batch size of 512. We repeated each experiment on each dataset with 10 different seeds to ensure repeatability. A linear-warmup scheduler with a 0.5 warmup rate was used for all the IAFFF experiments, unless stated otherwise. No additional compression methods, such as pruning or quantization, were applied to the models, and model parameters were kept in full precision unless stated otherwise.

6.1 Advantages of Inference-Aware Models, and Awareness Scheduling

In this section, we explore the accuracy of IAFFFs on varying architecture configurations (i.e., depth and leaf width). We also examine the distribution of samples across leaves (leaf sample balance) and investigate how different awareness schedulers influence accuracy for various IAFFF architectures.

6.1.1 Choosing the Right Scheduler. We use an awareness scheduler when training IAFFFs, as scheduling the awareness parameter α in Equation (7) helps reduce overfitting. As discussed in Section 4.2, to identify the most effective scheduler, we experiment with five different schedulers, namely constant, linear, ease-in, ease-out, and linear-warmup.

Figure 11 shows the accuracy of IAFFF models trained with different schedulers across various datasets and architectures. In our experiments, using a constant α consistently achieved the poorest accuracy across all configurations and datasets. For this reason, we focused on exploring different schedulers to improve performance. We observe that the linear-warmup scheduler achieves the best performance on all datasets, particularly on HAR and on MNIST models with low depth. The ease-in and ease-out schedulers yield similar results, while ease-out is dominant in many cases. Yet,

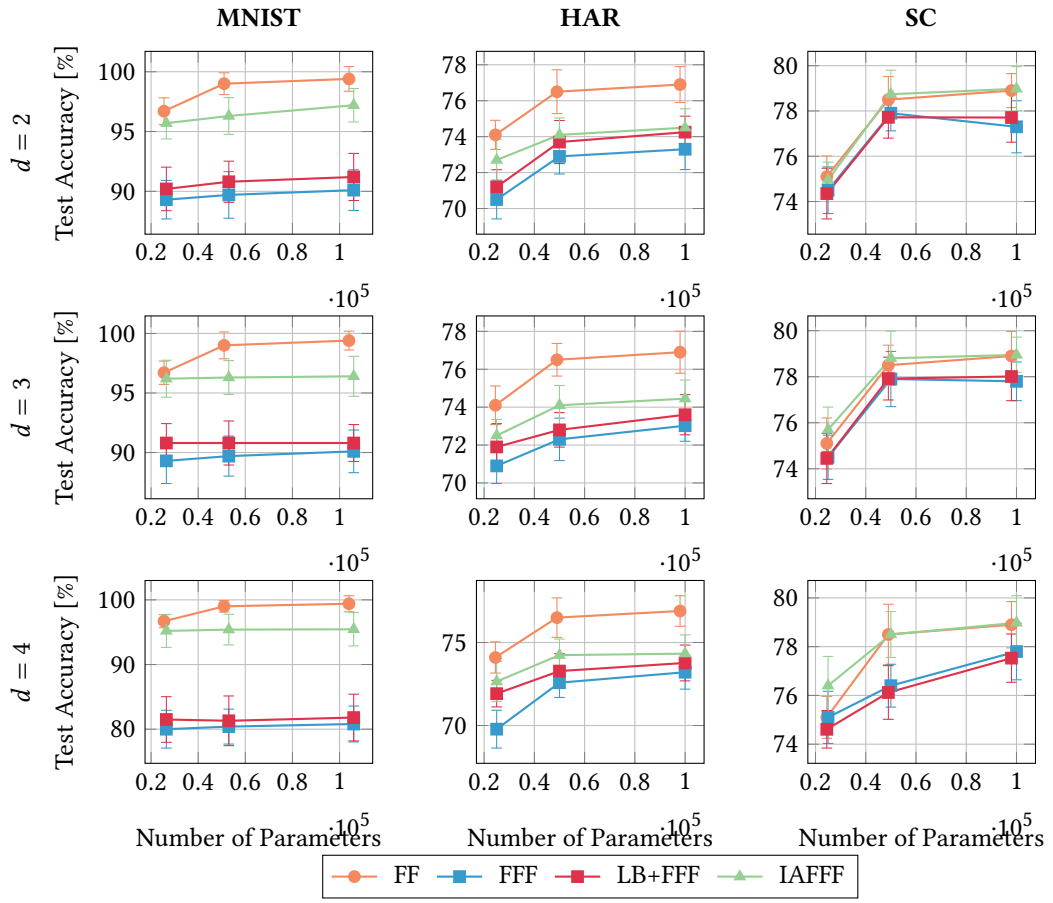


Fig. 12. Accuracy comparison of FF, FFF [2], LB+FFF [8], and IAFFF (ours) models across different architectures.

both ease-in and ease-out perform better than the linear scheduler, which consistently shows the worst performance after a constant α. It can also be noted that different schedulers have little effect on the SC dataset. One possible reason is that, on this dataset, the performance of FFFs is already comparable to that of FFs, even at smaller sizes, suggesting the room for improvement without added complexity is limited.

All together, these results confirm that IAFFFs benefit from an initial training-heavy phase, allowing leaves to see more data and improve generalization. Based on these findings, we adopt the linear-warmup scheduler for all subsequent experiments.

6.1.2 Performance Comparisons. In Section 6.1.1, we observed that all models employing the linear-warmup scheduler have the best performance. In this section, we present the results of IAFFF models with different architectures trained using this scheduler. To compare with a state-of-the-art FFF approach, we included **load-balanced FFFs** (**LB+FFF**) [8] in our experiments. In LB+FFF, a load-balancing regularization mechanism is used to balance the sample distribution of the leaves evenly. No additional regularization methods were applied to the load-balanced FFF models.

In Figure 12, we compare FFs, FFFs, LB+FFFs, and IAFFF architectures at three different depths and leaf widths to match similar model sizes across MNIST, HAR, and SC datasets. For every

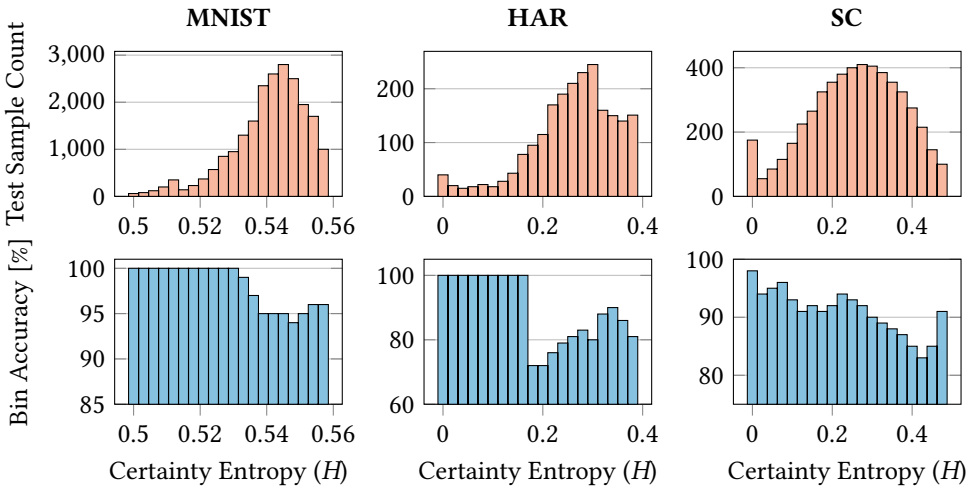


Fig. 13. The top histogram shows the number of samples vs. certainty entropy, while the bottom histogram shows the bin accuracy vs. certainty entropy. The bin accuracy corresponds to each bin's accuracy in the top histogram. Both histograms are based on results from a pre-trained IAFFF model, using $f_{2,8}$ for MNIST and SC, and $f_{2,16}$ for HAR.

dataset and model configuration, IAFFFs consistently outperform both FFFs and LB+FFFs, with notable improvements on all datasets. On MNIST, depth-2 IAFFFs achieve test accuracies comparable to those of FFs, with up to a 6% improvement over FFFs in smaller models. On SC, IAFFFs also improve performance, though gains are smaller due to already high FFFs' accuracy in small models. For HAR, IAFFFs and FFs achieve similar results for smaller models, but the performance advantage of IAFFFs grows with model size, while it decreases for deeper models with narrower leaves. Of note, the load-balanced FFFs perform only marginally better than standard FFFs in our setup.

Overall, these trends suggest that the optimal IAFFF architecture depends on the dataset and task. In any case, IAFFFs tend to outperform FFFs and LB+FFFs while achieving performance comparable to FFs in most scenarios.

6.1.3 Certain Samples, Better Performance. In Section 3.4, our preliminary experiments have shown that "uncertain" samples degrade accuracy across all datasets, as illustrated in Figure 4, where we observed that uncertain samples are frequently misclassified, with lower accuracy achieved in the bins with higher certainty entropy. We conducted experiments to determine whether inference-aware training affects the number of uncertain samples. Figure 13 presents the certainty histograms and corresponding bin accuracies after inference-aware training on each dataset. By comparing this figure with Figure 4, we observe that inference-aware training increases the certainty of many samples, evidenced by fewer samples in high-entropy bins and lower overall certainty entropy. The resulting accuracy improvements are substantial on all tasks, and particularly pronounced for MNIST. This indicates that a key factor underlying the accuracy gains is the increased certainty of many samples.

Moreover, Figure 14 illustrates the mean histogram of sample leaf oscillations across classes for different datasets. While each dataset contains a varying number of samples with high leaf oscillation rates, in every case, inference-aware training reduces the number of samples with ($\delta > 0.5$) and decreases their oscillation, thereby improving training stability.

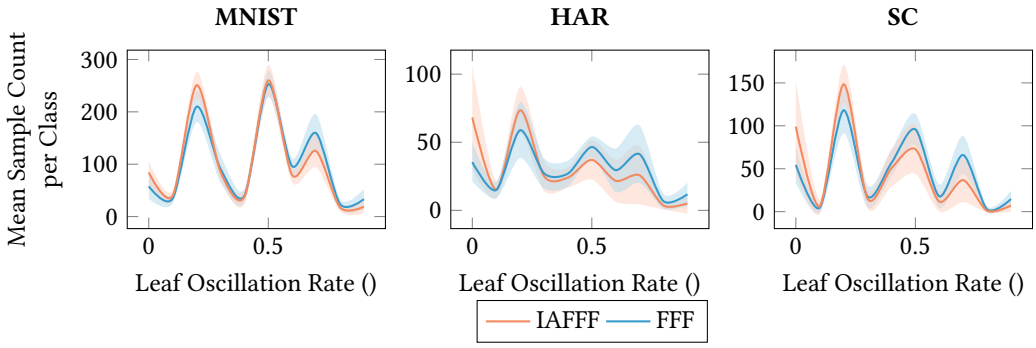


Fig. 14. Leaf oscillation plot for FFF and IAFFF models with $f_{2,8}$ configuration on MNIST and SC, and with $f_{2,16}$ configuration on HAR.

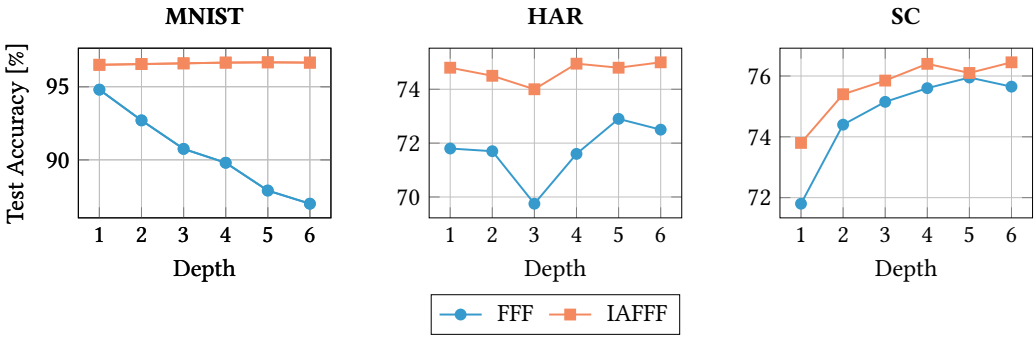


Fig. 15. FFF and IAFFF model performance vs. model depth with fixed leaf width of 8 on MNIST and SC, and leaf width of 32 on HAR.

6.1.4 Can We Go Deeper? We also experimented to see how the depth of FFF and IAFFF models affects their performance. Figure 15 shows that, for MNIST and HAR, FFF performance decreases as depth increases, while on SC, it improves. Inference-aware training within IAFFFs mitigates this performance drop: on MNIST, accuracy slightly improves with depth, while on HAR, it initially decreases up to depth 3 but then rises thereafter. On SC, the positive trend continues, with gains of up to 2%.

6.1.5 Leaf Sample Balance. In this section, to understand the factors behind the performance improvement, we analyze how the leaf sample distribution changes in IAFFFs compared to FFFs without hardening regularization (referred to as FFFs-NoReg). This allows us to identify the effects of inference-aware training on leaf utilization and how this change contributes to improved model performance. This analysis is also beneficial when employing compression methods such as pruning, quantization, or weight sharing: leaves with fewer samples are less critical and can be targeted for compression without significantly affecting performance.

As can be seen in Figure 16, in FFFs-NoReg, leaves are distributed quite evenly compared to both FFFs and IAFFFs across all datasets. In the case of FFFs, many samples are gathered in a single leaf due to the hardening loss, which improves model performance [2], and also results in other leaves having far fewer samples. IAFFF models show an intermediate distribution: 20–40% of the leaves are more heavily utilized, while the rest are less so. This suggests that inference-aware training helps selectively focus on a subset of leaves, rather than using an even distribution or concentrating

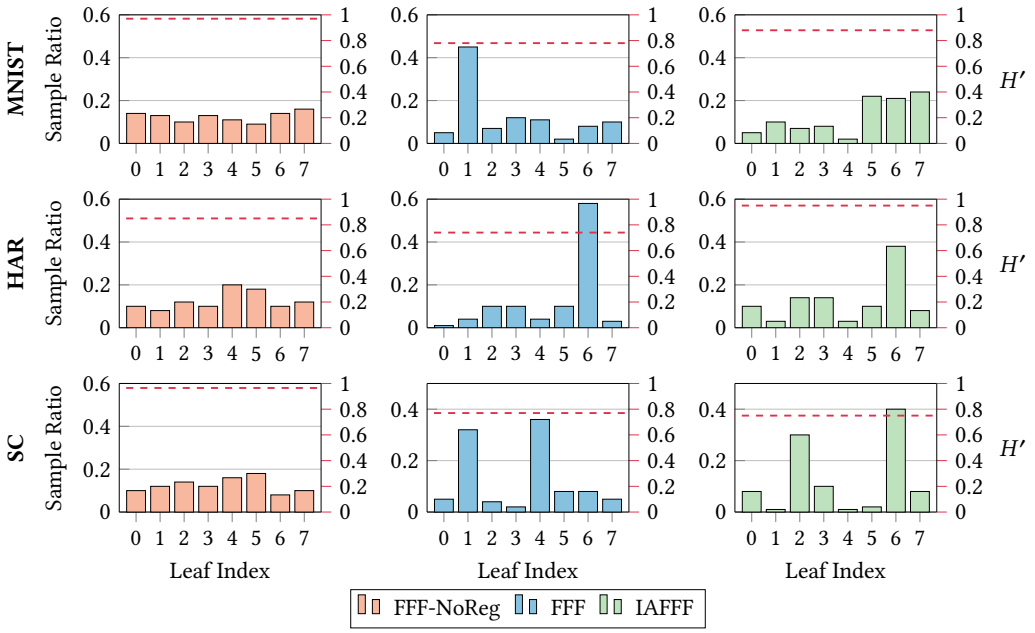


Fig. 16. Leaves' sample balance of FFFs-NoReg, FFFs, and IAFFFs on varying datasets, expressed as sample ratio vs. leaf index. FFFs-NoReg refers to FFFs without hardening regularization.

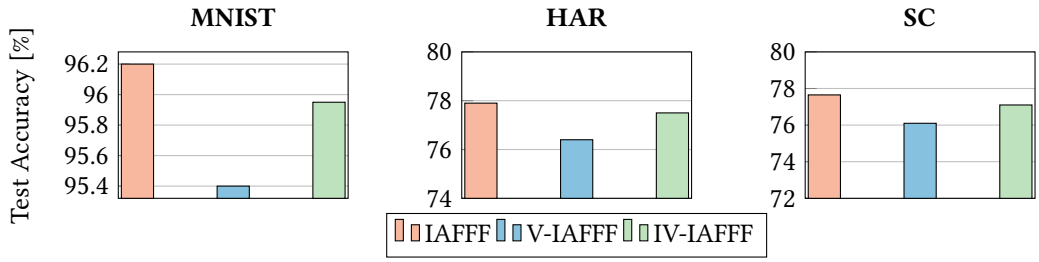


Fig. 17. IAFFF, V-IAFFF, and IV-IAFFF (Iteratively Virtualized IAFFF) models accuracy comparisons. Model configurations are $f_{2,8}$, $f_{2,16}$, $f_{2,8}$, with β' equal to 0.375, 0.25, 0.4375 (in the case of IV-IAFFF), respectively for MNIST, HAR, and SC.

on a single leaf, and contributes to improved model accuracy by efficiently allocating learning to the most informative leaves.

6.1.6 Iterative Virtualization via Warmup Rate Update. We also evaluated how virtualization, combined with warmup rate search, affects IAFFFs' performance before and after compression. In Figure 17, we compare the performance of IAFFF, V-IAFFF, and IV-IAFFF (Iteratively Virtualized IAFFF) models using optimal and suboptimal warmup rates. Although uncompressed IAFFFs with the optimal β sometimes perform slightly worse than those with suboptimal β, after virtualization, models using the optimal β consistently outperform those with suboptimal values. This demonstrates the importance of iteratively updating the warmup rate during compression. On each dataset, especially in the case of MNIST, we see a notable drop in the accuracy when β is suboptimal, while in the iterative case, the drop is negligible.

Table 1. Time (s) and Energy (kJ) Consumption of Varying Models Executed on CPU

Model	MNIST		HAR		SC	
	Time	Energy	Time	Energy	Time	Energy
FF	129.81	32.9	57.11	11.22	132.23	34.11
FFF	196.94	41.4	70.11	14.76	205.30	41.76
IAFFF	199.68	42.12	74.24	15.84	209.87	42.84
V-FFF	422.24	98.24	177.82	44.88	401.98	88.14
V-IAFFF	431.83	102.24	184.94	45.72	432.19	91.44
IV-IAFFF	1,425.33	291.96	420.84	83.16	1,828.95	342.36

6.1.7 Time and Energy Consumption during Training. In this section, we report the time and energy consumption (using CodeCarbon [12]) of training and compression for IAFFFs, FFFs, and FFs. Models were trained on a machine with an Intel i5-8365U CPU; GPU times are not reported because our IoT-focused experiments did not leverage GPUs. Results are averaged over 10 runs for each configuration.

As shown in Table 1, both IAFFFs and FFFs require more time and energy than FFs due to their more complex training procedures (e.g., rooting and iterating through leaves). IAFFFs incur slightly higher costs than FFFs because of additional activation steps and awareness scheduling, but this increase is relatively small compared to the performance gains. V-FFFs and V-IAFFFs exhibit similar tradeoffs, while iterative virtualization (IV-IAFFF) introduces significant overhead due to the multiple rounds of training and virtualization; this overhead can be justified by improved final-model performance and varies across datasets, depending on the number of rounds needed to find an optimal warmup rate.

6.2 Scalability of Inference-Aware Training

In this section, we explore more complex tasks and additional models using FFFs and inference-aware training to evaluate the scalability of our approach. First, we evaluate the performance of IAFFFs on the Speech Commands dataset, while incrementing the number of classes in Section 6.2.1, to assess scalability with respect to the number of classes. Secondly, in Section 6.2.2, we combine IAFFFs with the state-of-the-art ConvNeXt [34] convolutional image-classification model, to study the scalability of the proposed approach on more complex image-classification tasks such as CIFAR-10, CIFAR-100, Tiny-ImageNet, and ImageNet-1K.

6.2.1 Experiments on SC with Increasing Number of Classes. For these experiments, the SC dataset was split into three subsets, defined by the number of classes, to evaluate IAFFF performance as class count increases:

- **12 classes**: The commonly used version for IoT applications, including: yes, no, up, down, left, right, on, off, stop, go, unknown, background.
- **24 classes**: The 12-class set, plus the number words and the word "tree" (which can be challenging due to its similarity to "three"): zero, one, two, three, four, five, six, seven, eight, nine, ten, tree.
- **36 classes**: The full set, including additional proper nouns and animal labels present in the dataset.

Figure 18 shows the performance of FF, FFF, and IAFFF models on these subsets. IAFFF consistently outperforms both FF and FFF across all subsets, and the performance gap between IAFFF and FFF increases with the number of classes. Not only that: IAFFFs also reached up to 2.3% improvement over FFs. This suggests that inference-aware training can be particularly beneficial for more complex classification tasks where models must learn finer-grained decision boundaries.

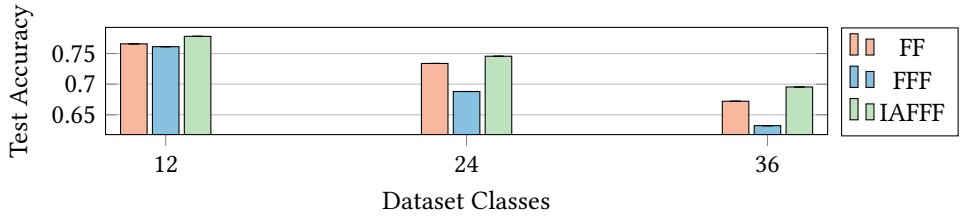
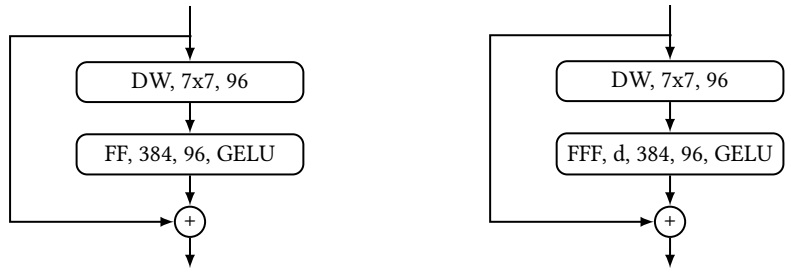


Fig. 18. Model accuracies for different numbers of classes in the Speech Commands (SC) dataset. All models have a training width of 128; FFF and IAFFF have a depth of 2.



(a) ConvNeXt block (b) F-ConvNeXt block

Fig. 19. ConvNeXt-based model blocks. DW refers to the depth-wise convolution layer, and GELU refers to the GELU activation function used after the first layer.

6.2.2 Experiments with ConvNeXt Models. To see how IAFFFs can be scaled to larger models, we integrated them into ConvNeXt [34] blocks, which have been shown to achieve state-of-the-art performance on image classification tasks. A ConvNeXt block, shown in Figure 19(a), consists of a depthwise convolution followed by two linear (i.e., feedforward) layers. Depthwise convolution is a special grouped convolution in which the number of channels is equal to the number of groups. Between the two linear layers, there is a **Gaussian Error Linear Unit** (**GELU**) as the activation function, which is a smoother variant of the **Rectified Linear Unit** (**ReLU**) [21, 34, 37]. The ConvNeXt block also has a residual connection between the input and output. To integrate this kind of block with FFF-based structures, we replace the two linear layers with an FFF, yielding a **FastConvNeXt** (**F-ConvNeXt**) block, shown in Figure 19(b). Applying inference-aware training produces instead an inference-aware variant, which we dub as IAF-ConvNeXt. This allows us to evaluate the benefits of inference-aware training in a more complex architecture and on larger datasets. Furthermore, this choice allows us to evaluate our approach while preserving convolutional feature extraction, which is essential for spatiotemporal tasks.

We used ConvNeXt-Tiny [34] as the model configuration, with FFF layers having a depth of 2 for the F-ConvNeXt and IAF-ConvNeXt models. We trained the models on CIFAR-10, CIFAR-100, Tiny-ImageNet, and ImageNet-1K datasets, and compared their performance in terms of accuracy, number of parameters, and MACs.

The results in Figure 20 show that F-ConvNeXt has consistently lower accuracy than ConvNeXt across all datasets; in contrast, inference-aware training recovers the accuracy drop and even outperforms ConvNeXt on CIFAR-100 and ImageNet-1K. The figure also shows that using FFF layers reduces model size by up to a factor of 2, which does not necessarily translate to a comparable reduction in MACs since the convolutional layers still dominate computational cost. However,

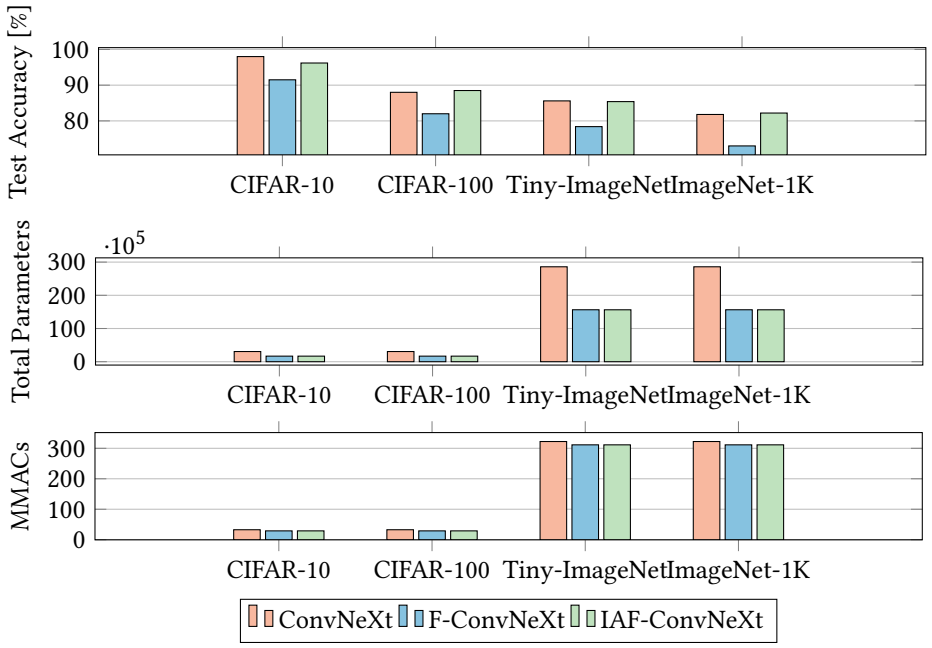


Fig. 20. Performance of varying ConvNeXt [34] based models trained on CIFAR-10, CIFAR-100, Tiny-ImageNet, and ImageNet-1K.

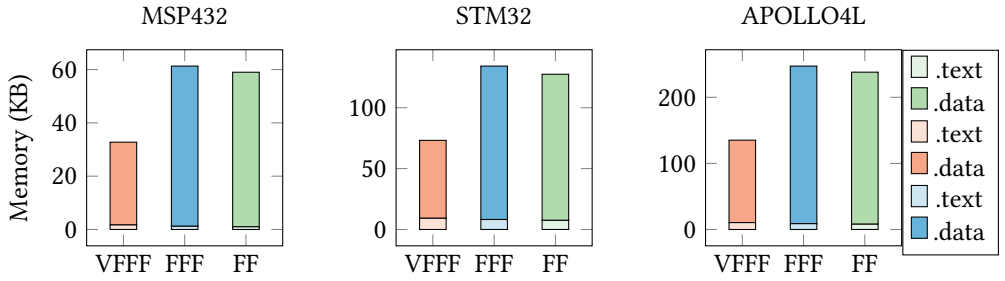


Fig. 21. Memory usage (`.text` and `.data` sections) for each device on MNIST. Each model is sized to fit its target device: models on MSP432, STM32, and APOLLO4L have a training width of 32, 64, and 128, respectively. Virtualization compresses the models by a factor of 2.

IAF-ConvNeXt does not add any parameters or MACs, making it an efficient way to improve performance without increasing model complexity.

6.3 Runtime Experiments

In this section, we discuss the models run on the three resource-constrained platforms mentioned in Section 5, focusing on accuracy, inference latency, memory usage, and overall efficiency.

6.3.1 Memory Footprint and Runtime Memory Requirements. We also analyzed the memory usage of FF, FFF, and VFFF models on the three target selected devices, to assess their suitability for deployment on resource-constrained hardware. Figure 21 presents the memory breakdown of the three models of the case of MNIST. It can be seen that FFFs require more memory than

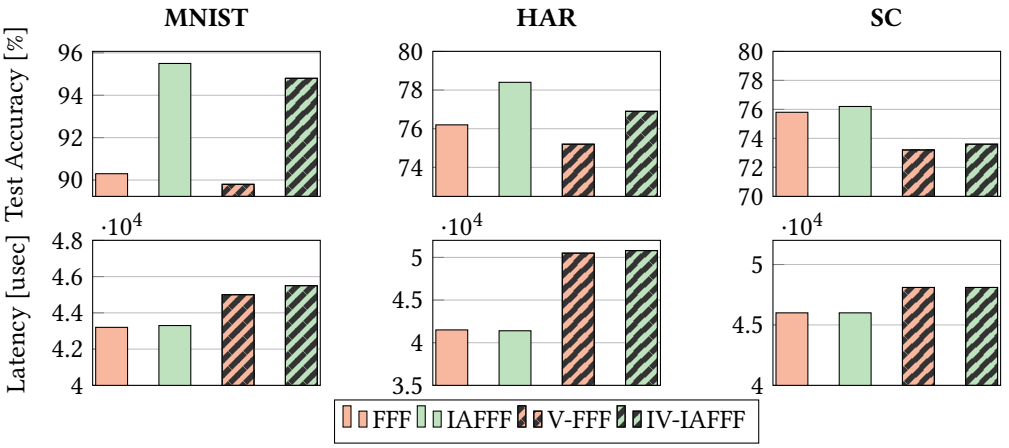


Fig. 22. FFF and IAFFF models accuracy (above) and latency (below) comparisons with $f_{2,8}$, $f_{2,16}$, $f_{2,8}$ architectures on MNIST, HAR, and SC, respectively. V-FFFs represent virtualized FFFs, IV-IAFFFs represent iteratively virtualized IAFFFs, and FFFs and IAFFFs represent uncompressed models. All model inference latencies are measured on MSP432.

FFs, because of the additional parameters introduced by duplicated nodes; in contrast, virtualized models compress these parameters and substantially reduce the overall memory footprint. Even though virtualization incurs a small overhead in `.text` and `.data` for storing indices and the virtualized neuron implementation (see `virtualized_neuron` in Figure 1), the net effect is a significant reduction in total memory usage, improving deployability on smaller devices.

6.3.2 Virtualization of Inference-Aware FFFs. In Figure 22, it can be seen that, in all three datasets, the addition of the virtualization leads to a decrease in accuracy for both the FFF and IAFFF models across all datasets: approximately 1% for MNIST and HAR, and around 3.5% for SC.

Although the accuracy drop of **virtualized IAFFFs (V-IAFFF)** is slightly larger than that of **virtualized FFFs (V-FFF)**, V-IAFFF models consistently outperform V-FFF models across all datasets. The slightly larger drop for V-IAFFFs is likely due to their more balanced leaf sample distribution, as more leaves contribute to the model's decisions. In contrast, FFFs focus on one or two dominant leaves, making the remaining leaves less effective and easily compressible. This result supports our observations from Section 6.1.5 and highlights that IAFFFs offer a robust accuracy advantage in virtualized deployments.

6.3.3 On Resource-Constrained Devices. We compress IAFFFs using leaf-weight virtualization with inference-aware retraining to fit them on memory-constrained platforms. Yet such compression algorithms often have inference overhead during runtime. In this section, we present experiments conducted on several edge devices described earlier to measure latencies and accuracies between varying models. To fairly evaluate performance, we maximized model sizes within the memory limits of MSP432 (64 kB), STM32 (128 kB), and APOLLO4L (256 kB).

Figure 23 illustrates the tradeoffs between model width, inference latency, and accuracy. Uncompressed FF models achieve high accuracy (e.g., 95–97% on MNIST) but suffer from the highest latency, often exceeding 0.5 seconds on embedded devices at large widths. FFF models dramatically reduce latency—often by more than half—but at the cost of a notable accuracy drop, sometimes 8–10% (e.g., from 95% to 85% on MNIST). Compressed and virtualized IAFFF models provide a strong compromise: virtualization introduces only a modest accuracy drop (typically 2–4%), yet

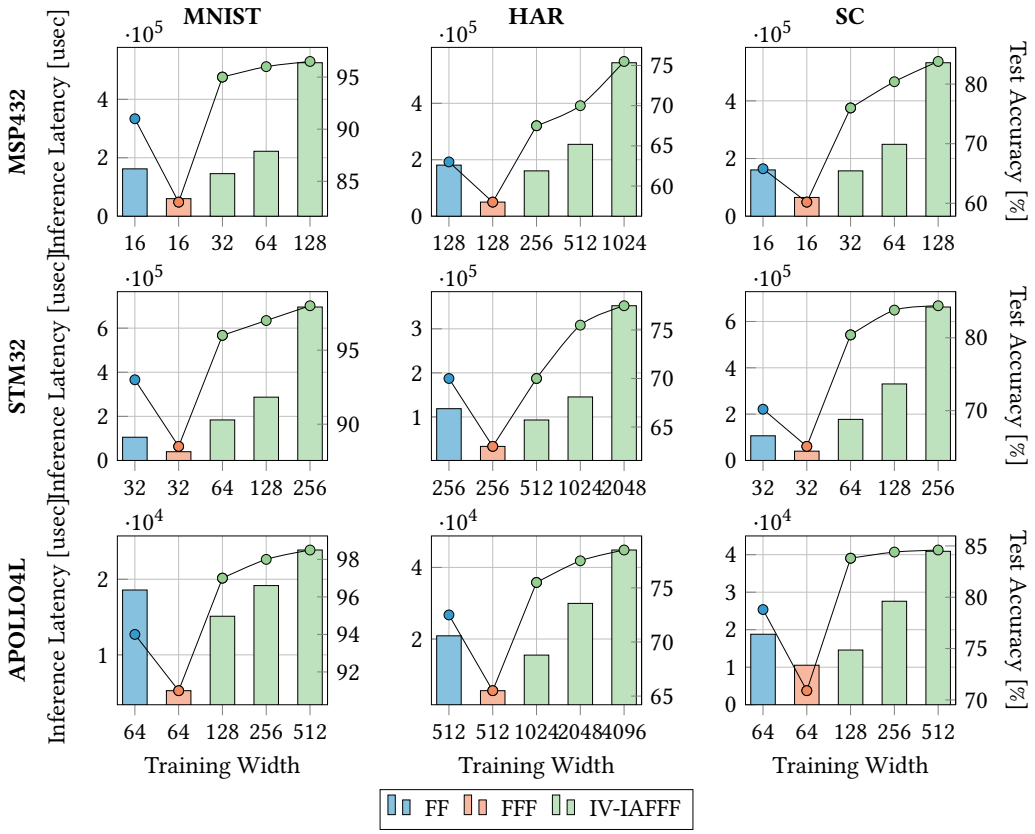


Fig. 23. Latency (bars) and accuracy (lines) comparisons of FF, FFF [2], and iteratively virtualized IAFFF models across different training widths with fixed FFF architectures of $f_{2,l}$. Each model architecture is designed to fit into the specified device.

these models still outperform FFFs (e.g., 93–96% on MNIST) while maintaining much lower latency than FFs. This pattern holds across all datasets and devices, demonstrating that IAFFF models can be effectively virtualized to deliver both high accuracy and fast inference, making them particularly suitable for deployment on resource-constrained hardware.

6.3.4 Inference Energy Consumption. Figure 24 shows the energy consumption of the models, with the virtualized model configurations reported in Table 2, so as to make the tradeoffs between inference overhead and compression clear.

The figure shows that FF models consume the most inference energy across datasets and devices, often by a large margin (for example, up to a factor of 5 more than FFFs on MNIST) because of their much higher latency. Moreover, FFF models reduce energy by roughly an order of magnitude due to lower compute, while virtualized FFFs (VFFF and IV-IAFFF) use slightly more energy than FFFs—typically under 10%—because of virtualization overhead (more memory accesses, loops; see the code block in Figure 1). Some overhead might also be caused by more instruction fetches, as observed in Appendix B. Despite this overhead, virtualized models remain far more efficient than FFs and offer higher accuracy (see Figure 23). The virtualization cost is small on MSP devices, and larger on STM32. Furthermore, the APOLLO4L's high-performance mode increases energy for all models and magnifies the virtualization overhead.

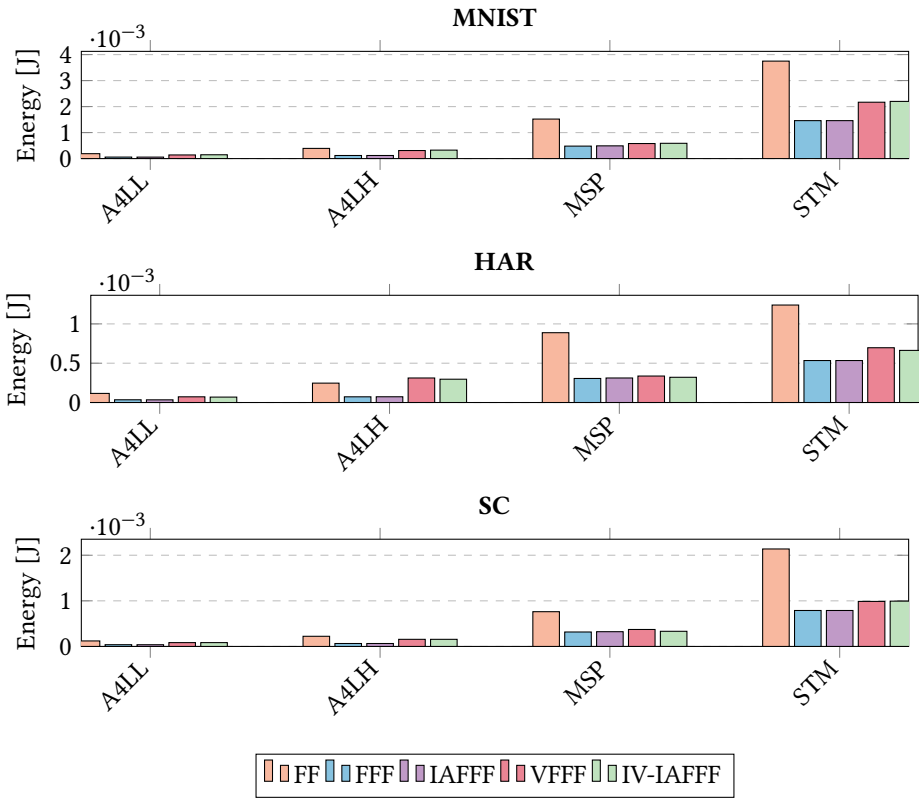


Fig. 24. Inference energy consumption comparisons across our boards for various models and configurations displayed in Table 2. Each model is sized to fit its target device: models on the MSP432 use a training width of 32, while those on the STM32 and APOLLO4L use widths of 64 and 128, respectively. The comparison also includes FF, and FFF [2] models have depths of 2.

Table 2. Architectures of the Virtualized Models Tested in Figure 24

Dataset	APOLLO4L				STM32				MSP432			
	VFFF		V-IAFFF		VFFF		V-IAFFF		VFFF		V-IAFFF	
	S	$\hat{\mathbf{N}}/\mathbf{l}$	**S**	$\hat{\mathbf{N}}/\mathbf{l}$	**S**	$\hat{\mathbf{N}}/\mathbf{l}$	**S**	$\hat{\mathbf{N}}/\mathbf{l}$	**S**	$\hat{\mathbf{N}}/\mathbf{l}$	**S**	$\hat{\mathbf{N}}/\mathbf{l}$
MNIST	200	250	208	241	200	125	122	210	100	125	64	196
HAR	50	500	80	313	50	250	80	313	100	250	124	202
SC	500	100	328	152	300	84	220	114	200	64	128	98

S: page size, $\hat{N}$: number of pages per leaf, *l*: leaf width.

Finally, it is worth noticing that smaller page sizes increase page indexing due to an increased number of pages per leaf, which causes more memory accesses and loop iterations and thus higher overhead compared to larger pages for models of similar complexity. Some of the observed differences between VFFF and V-IAFFF arise from different virtual configurations (see Table 2). For example, on HAR, the V-IAFFF model uses smaller pages and more pages per leaf than V-FFF, leading to more memory accesses and higher energy; on MNIST, the V-IAFFF model uses larger pages and fewer pages per leaf, resulting in fewer accesses and lower energy.

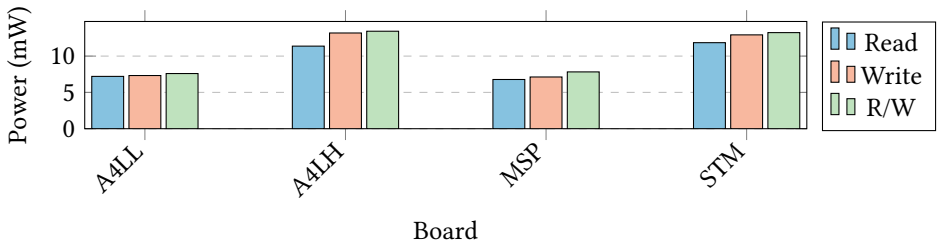


Fig. 25. Power consumption of different kinds of memory accesses across different boards. The memories used are SRAM for MSP432 and STM32, and TCM for APOLLO4L. R and W refer to read and write operations, respectively.

Table 3. Number of Memory Operations

	Apollo							
	FFF		IAFFF		VFFF		V-IAFFF	
Datasets	R	W	R	W	R	W	R	W
MNIST	50,496	26	50,496	26	50,520	26	50,520	26
HAR	39,040	42	39,040	42	39,840	42	39,620	42
SC	49,024	26	49,024	26	49,424	26	49,500	26
	STM32							
	FFF		IAFFF		VFFF		V-IAFFF	
Datasets	R	W	R	W	R	W	R	W
MNIST	25,448	18	25,448	18	25,320	18	25,402	18
HAR	19,520	26	19,520	26	19,680	26	19,492	26
SC	24,512	18	24,512	18	24,602	18	24,680	18
	MSP							
	FFF		IAFFF		VFFF		V-IAFFF	
Datasets	R	W	R	W	R	W	R	W
MNIST	12,624	14	12,624	14	14,357	14	14,357	14
HAR	9,760	18	9,760	18	13,160	18	13,160	18
SC	12,256	14	12,256	14	14,188	14	14,188	14

R and W refer to read and write operations, respectively.

6.3.5 Memory Characteristics. The number of memory accesses (reads and writes) is one of the main measures that help explain the energy and latency overhead of virtualization. We report memory access power in Figure 25 and list access counts in Table 3; virtualized models show more accesses—especially reads—because of page loads during inference. Some energy increase is also caused by higher latency and more instruction fetches (see Appendix B).

Given a board b, we calculate its memory energy consumption E_b using the memory read and write power consumption (P_b^r and P_b^w, respectively), the number of read and write operations (c_r and c_w, respectively), and the clock speed of the board (f_b), as reported in Section 5.2:

$$E_b = (P_b^r c_r + P_b^w c_w)/f_b). \tag{9}$$

These energy values are reported in Figure 26.

Due to different memory access power consumptions of the boards (Figure 25) and the numbers of memory accesses of varying models (Table 3), the energy consumption of virtualized models varies as shown in Figure 26, yet virtualized models always have higher energy consumption than

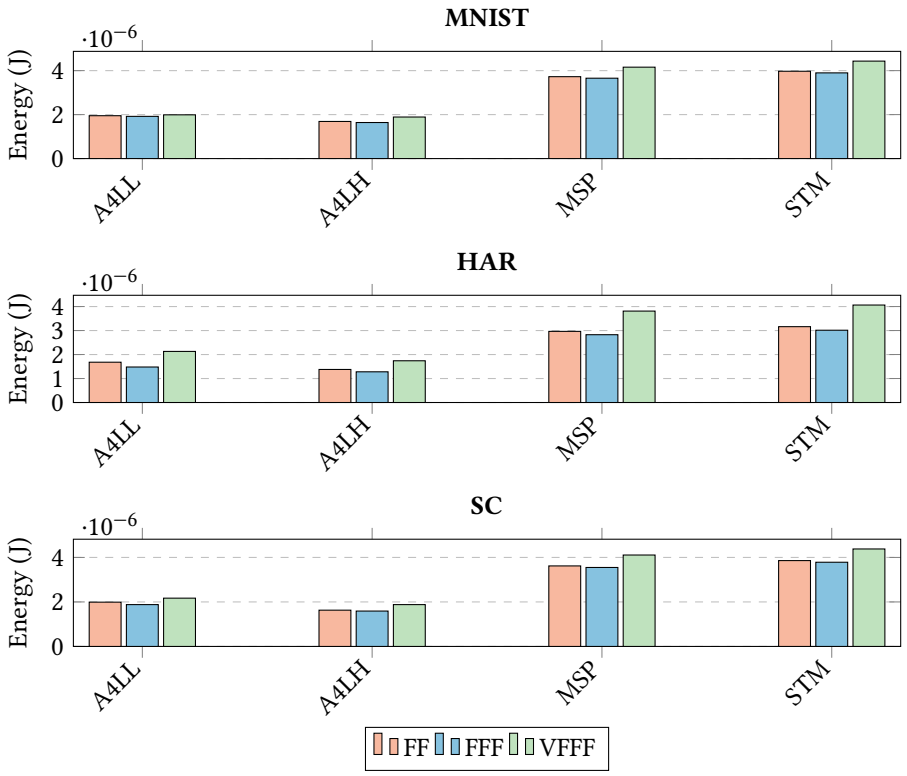


Fig. 26. Approximate energy consumption of memory operations per board for FF, FFF [2], and VFFF per inference.

non-virtualized ones, and the increase is quite negligible on APOLLO4L in low power mode and MSP432 devices, while it is more pronounced on STM32. This also gives insights into the increase in the model energy consumption compared to FFFs on these boards and the models reported in Figure 24, such as STM32, which have higher energy consumption due to a more significant memory overhead.

6.3.6 Other Compression Methods. To provide baselines for our proposed **Iterative Virtualization (IV)** approach, we consider INT8 and INT16 quantization, along with **Gradual Magnitude Pruning (GMP)** [58], and various comparisons thereof. GMP has been specifically selected for comparison since recent studies have investigated its use on low-complexity architectures such as MobileNet-V1 [22].

Table 4 presents a comparison of various compression methods applied to IAFFFs across MNIST, HAR, and SC. The results show that quantization alone can drastically reduce the model size (from more than 100 kB to 28 or 56 kB), while maintaining nearly identical accuracy to the uncompressed model. This is often expected when applying INT8 or INT16 quantization, while lower bit widths often reduce the accuracy significantly.

When either GMP or our IV algorithm is applied individually, the model accuracy decreases; especially, GMP introduces a noticeable drop for extreme compression cases (e.g., $r \geq 4\times$), where the accuracy drop is more than 2% on MNIST, and around 2% and 3% on HAR and SC, respectively. In contrast, IV results in a much smaller accuracy reduction, with slightly lower accuracy

Table 4. Accuracy and Model Size with Different Compression Methods on IAFFFs, on Varying Datasets

Compression Ratio (r)	Compression Method(s)	MNIST | $\mathbf{f_{2,8}}$		HAR | $\mathbf{f_{2,64}}$		SC | $\mathbf{f_{2,8}}$	
		Accuracy [%]	Model Size [kB]	Accuracy [%]	Model Size [kB]	Accuracy [%]	Model Size [kB]
1×	**None**	96.21	111	78.06	107	78.28	112
2×	**INT16**	96.43	56	78.08	28	78.85	28
2×	**GMP**	95.11	56	73.91	28	75.25	28
2×	**IV**	95.83	57	76.68	28	78.05	28
4×	**GMP + INT16**	95.01	28	72.31	14	76.43	14
4×	**IV + INT16**	**95.68**	29	**73.41**	14	**83.69**	13
8×	**IV + GMP + INT16**	91.59	15	70.71	7	73.71	7
4×	**INT8**	95.83	28	73.68	28	77.95	28
4×	**GMP**	93.78	28	70.74	28	69.95	28
4×	**IV**	95.03	29	72.18	28	75.35	28
16×	**GMP + INT8**	93.02	7	68.81	7	73.33	7
16×	**IV + INT8**	**94.88**	7	**71.91**	7	**75.19**	7
64×	**IV + GMP + INT8**	86.09	2	56.21	2	45.21	2

INT16 or INT8 formats are used for quantization. Gradual magnitude pruning (GMP) and Iterative Virtualization (IV) are used to introduce sparsity.

than uncompressed models at 2× compression. 4× compressed models using IV outperform those compressed with GMP, showing over 5% higher accuracy on SC.

When GMP or IV is combined with quantization, model accuracy decreases further, especially under extreme compression (e.g., $r \geq 4\times$, ~14 kB models). Still, the accuracy advantage of IV over GMP remains consistent across all datasets. Moreover, combining all compression methods leads to a substantial accuracy drop—at least 9%—with models reaching their smallest size (~2 kB). Even 8× compression cases with all methods combined degrade the performance more than the ones with 16× compression. This shows that, while combining all compression methods achieves the highest compression ratio, it also results in the largest accuracy drop, so such combinations need to be used carefully.

7 Related Work

Many techniques have been proposed for designing an efficient AI deployment framework on resource-limited devices. In this section, we discuss the related work and where our study fits in the literature.

7.1 Efficient Models with Conditional Execution

Deep Neural Networks (DNNs) achieve state-of-the-art accuracy but often require billions of operations, making them impractical for memory- and energy-constrained IoT devices [6]. To address this challenge, research has focused on two strategies: compressing large models (e.g., pruning, quantization, weight sharing) and designing lightweight architectures, such as MobileNets [44], ShuffleNets [55], and SqueezeNets [23]. However, these approaches often require extensive manual tuning and may still execute redundant computations.

Conditional execution provides a complementary solution by dynamically activating only a subset of model components (layers, channels, or blocks) depending on the input. This reduces unnecessary computation and allows models to adapt resource usage to the task at hand. Techniques such as early exiting, dynamic routing, and input-dependent gating embody this principle and have shown particular promise for mobile and IoT platforms, where strict latency and energy constraints demand efficient execution. In contrast, in [33], rather than using dynamic conditional models, the authors propose an automated kernel-generating approach that dynamically masks inputs to

introduce sparsity and improve inference efficiency, though their focus is on high-performance devices such as GPUs.

Despite this potential, research on conditional execution remains limited and mostly focuses on tasks that are less relevant for resource-limited devices. Most existing work has focused on convolutional layers [4, 10, 25, 27, 38, 47, 50] or on computationally demanding tasks such as large-scale image classification (e.g., ImageNet and CIFAR-10) [10, 50] and image recognition [38]. More recently, conditional execution has been applied to accelerate **Large Language Models** (**LLMs**) through techniques such as **Mixture-of-Experts** (**MoEs**) and early-exit networks [45].

In contrast, the application of conditional execution to shallow, low-complexity models remains underexplored, even though such models are highly relevant for TinyML and IoT scenarios. Exploring conditional execution in these settings could unlock efficient, adaptive models tailored to the constraints of embedded and resource-limited platforms.

7.2 Model Compression and Deployment

Model compression plays a central role in reducing neural network size and memory usage, enabling deployment on resource-constrained devices. Three widely used approaches are quantization, pruning, and knowledge distillation. Quantization reduces model precision to save storage and accelerate computation, either during training (QAT) [11, 32] or after training (PTQ) [41]. Although PTQ is faster, it usually sacrifices accuracy compared to QAT [31], and many methods have been proposed to reduce this gap [7, 18]. Knowledge distillation, in contrast, trains a smaller student network under the guidance of a larger pre-trained teacher to retain accuracy while reducing complexity [31]. Pruning, another common strategy, removes redundant structures such as neurons, channels, or entire layers [36, 52, 56], often relying on criteria such as the sum of absolute weights [29, 30] and following a train–compress–retrain cycle to minimize accuracy loss [20, 31]. Such pruning-based methods have also been adopted in conditional execution models [38, 47].

Beyond pruning, weight sharing provides an alternative by clustering and reusing parameters across different parts of the network [43, 49]. This approach is particularly relevant for conditional models such as FFFs, which employ a binary-tree architecture where many leaves perform similar tasks. Building on this idea, weight virtualization was introduced for multi-task and multi-model learning [28]. Inspired by this technique, leaf-weight virtualization has been proposed specifically for FFFs [26]. Since each leaf resembles a shallow neural network, this method clusters similar leaves and maps them to shared parameters, exploiting structural redundancy more effectively than conventional compression. As a result, leaf-weight virtualization achieves stronger compression and greater efficiency, making FFFs more practical for deployment in resource-constrained environments.

7.3 Efficient AI Systems

Cloud platforms have traditionally served as the primary environment for training and inference of AI models. However, with the rise of IoT, advances in edge computing, and the increasing computational capacity of mobile devices, AI models are now often deployed directly on the edge. This shift brings notable benefits, including lower latency, reduced bandwidth usage, enhanced data privacy, and improved real-time responsiveness [51]. Yet, deploying AI on edge devices remains challenging due to their strict memory, energy, and computational limitations, making the design of efficient systems essential.

A major challenge lies in the limited compute capacity of edge hardware, which demands lightweight models and careful use of optimization techniques (see Section 7.2). Much of the research still emphasizes large, complex neural network architectures [54], while conditional execution—despite its strong potential for reducing inference cost—has received less attention. Some prior work explores early-exit mechanisms to reduce average inference time [19], and

conditional networks such as **Mixtures-of-Experts** (**MoEs**) have been applied to large-scale models like LLMs, often in combination with compression techniques to make them deployable at the edge [35]. However, shallow conditional models that exploit divide-and-conquer principles remain underexplored, despite their suitability for small-scale IoT applications. Other works employ low-complexity, efficient models without conditional execution to develop edge inference frameworks. For example, [14] uses stochastic differential equation networks for time-series data, achieving significantly improved throughput. This approach could be extended to other widely used datasets, such as HAR and KWs, or even image datasets for broader benchmarking. In [1], the authors explore the use of Tsetling machines to address IoT tasks, demonstrating an alternative strategy for achieving efficient inference on resource-constrained edge devices, highlighting the importance of designing models that balance accuracy and efficiency in IoT applications.

Storage capacity further compounds these challenges, as AI models must be compact enough to fit within device memory. Compression methods are therefore essential, but compressed models are still rarely deployed on real edge hardware. This makes it particularly important to evaluate the latency and resource overheads that compression introduces at inference time. While AI inference frameworks such as NCNN [40], OpenVINO [16], and ONNX Runtime [17] provide limited support for model optimization on specific hardware, they lack features such as weight virtualization and do not yet fully support conditional models like FFFs.

7.4 Fast Feedforward Networks

In [2], FFFs were proposed as a conditional architecture that speeds up standard feedforward networks by using a binary-tree structure to route inputs to specific leaves, thus reducing computational cost. The authors also introduced a hardening regularization term to encourage more confident routing decisions, which improves accuracy. In [8], FFFs are trained with a load-balancing regularization approach to improve the performance by balancing the sample distribution of the leaves. Aside from these regularization approaches, there has been no prior work on enhancing the accuracy of FFFs through additional regularization. To the best of our knowledge, our work is the first to improve FFF accuracy by modifying its architecture during training.

FFFs have been studied primarily in the context of accelerating LLMs. For example, [3] applied FFFs to sparse language models to further improve efficiency, while [9] explored unifying MoEs and FFFs to speed up LLM inference. Despite these advances, the performance bottlenecks of FFFs in small-scale IoT tasks have not been systematically analyzed, and the causes of their inefficiency in such settings remain poorly understood. Establishing standardized evaluation metrics, studying input distributions across leaves, and analyzing node-level routing decisions could provide valuable insights into these bottlenecks; yet, such investigations are currently lacking in the literature.

Structurally, FFFs offer faster inference than traditional FFs. However, their binary-tree organization causes the model size to grow exponentially as the tree deepens, making them difficult to deploy on embedded and memory-constrained platforms. Recent work introduced leaf-weight virtualization, a specialized weight-sharing technique to compress FFFs by mapping similar leaves into shared parameter sets [26]. While effective in reducing storage requirements, compression methods such as pruning, quantization, and virtualization introduce additional operations at inference time, including index mapping for pruning and virtualization or dequantization for quantization. In FFFs, leaf-weight virtualization requires uncompressing the model during inference, which may introduce non-negligible latency.

Despite the importance of these factors, there is no comprehensive study evaluating compressed FFFs on real resource-constrained devices, nor one that quantifies the tradeoffs between compression, inference latency, and deployment feasibility. This gap highlights the need for systematic analysis of FFFs beyond LLMs, particularly in the context of IoT and TinyML applications.

8 Discussion and Future Works

Future work could explore optimizing inference-aware training for larger or more complex datasets, applying it to other conditional models, designing specialized models for specific real-time applications, and automatically constructing more efficient unbalanced decision trees. These directions aim at extending IAFFF benefits to broader contexts while improving model efficiency and applicability.

Inference-aware training on other conditional neural networks. Other conditional networks, such as MoE models, could also benefit from inference-aware training. These networks also contain many experts, but during inference, often one or two are selected based on the output of their routing mechanism. Nevertheless, the selection process can itself be optimized during training using inference-aware training. By integrating inference-aware training, the model can optimize this selection process during training, potentially improving both generalization and inference performance.

Task-specific node decisions. With the demonstrated improvements of IAFFFs on IoT tasks, future work could focus on task-specific node decisions to further optimize inference. For instance, in audio applications, nodes could base their routing decisions on input features like frequency and amplitude, allowing the network to adaptively focus on the most informative signals and improve inference performance.

On real-time applications. Another promising direction is adapting IAFFF architectures for real-time applications by dynamically loading only the leaves relevant to the current system state. This selective allocation of resources can reduce memory usage and accelerate inference, enabling the model to respond efficiently to changing inputs and operational conditions.

Self-building differential tree-based networks. IAFFFs do not utilize all leaves efficiently—some are rarely activated during inference—likely because they are trained with a fixed architecture. To address this, future work could explore self-building differential tree-based networks that adaptively construct paths during training. These adaptive networks would tailor the architecture to the data distribution, improving both accuracy and resource efficiency. Additionally, they could be optimized for metrics such as energy consumption and latency, making them especially suitable for deployment on resource-constrained platforms.

9 Conclusion

In this work, we focused on overcoming the performance limitations of FFFs on resource-constrained devices. We made a comprehensive analysis using multiple entropy-related metrics for input certainty and leaf sample distribution, which provided insight into the inference performance bottlenecks of FFFs. To enhance performance, we proposed IAFFFs to improve small FFF models' performance, which are especially related to resource-constrained devices with extreme memory constraints. For model compression, we adopted a leaf-weight virtualization strategy. Our results have shown that an awareness scheduler with an optimal warm-up rate remarkably improves virtualization performance. Based on that, we implemented an iterative training approach with compression to achieve an optimal awareness scheduler.

Our experiments across multiple datasets and embedded platforms demonstrate clear tradeoffs between accuracy and latency. Uncompressed FF models achieve the highest accuracy, but they have high inference latency, limiting their practicality for real-time or low-power applications. FFF models, though much faster, suffer from a notable drop in accuracy, especially on smaller models. IAFFF models, especially when compressed and virtualized, offer a compelling balance: they maintain accuracy levels close to FFs while achieving inference speeds better than FFFs. This balance is achieved with only a modest accuracy drop due to virtualization, and IAFFFs consistently outperform FFFs even after compression. These results highlight the effectiveness of

inference-aware training and our new pipeline that improves leaf-weight virtualization in enabling efficient, accurate, and deployable neural networks for embedded AI scenarios.

References

[1] Abu Bakar, Tousif Rahman, Rishad Shafik, Fahim Kawsar, and Alessandro Montanari. 2023. Adaptive intelligence for batteryless sensors using software-accelerated tsetlin machines. In *Proceedings of the 20th ACM Conference on Embedded Networked Sensor Systems (SenSys'22)*. Association for Computing Machinery, Boston, Massachusetts, 236–249. DOI: https://doi.org/10.1145/3560905.3568512

[2] Peter Belcak and Roger Wattenhofer. 2023. Fast Feedforward Networks. arXiv:2308.14711. Retrieved from https://arxiv.org/abs/2308.14711

[3] Peter Belcak and Roger Wattenhofer. 2024. UltraSparseBERT: 99% conditionally sparse language modelling. In *Proceedings of the Annual Meeting of the Association for Computational Linguistics*. Lun-Wei Ku, Andre Martins, and Vivek Srikumar (Eds.), Association for Computational Linguistics, Bangkok, Thailand, 104–108. DOI: https://doi.org/10.18653/v1/2024.acl-short.10

[4] Ufuk Can Bicici, Tuna Han Salih Meral, and Lale Akarun. 2024. Conditional information gain trellis. *Pattern Recognition Letters* 184, C (2024), 212–218. DOI: https://doi.org/10.1016/j.patrec.2024.06.018

[5] Parimala Boobalan, Swarna Priya Ramu, Quoc-Viet Pham, Kapal Dev, Sharnil Pandya, Praveen Kumar Reddy Maddikunta, Thippa Reddy Gadekallu, and Thien Huynh-The. 2022. Fusion of federated learning and industrial Internet of Things: A survey. *Computer Networks* 212, C (2022), 109048.

[6] Han Cai, Ji Lin, Yujun Lin, Zhijian Liu, Haotian Tang, Hanrui Wang, Ligeng Zhu, and Song Han. 2022. Enable deep learning on mobile devices: Methods, systems, and applications. *ACM Transactions on Design Automation of Electronic Systems* 27, 3 (2022), 50 pages. DOI: https://doi.org/10.1145/3486618

[7] Yaohui Cai, Zhewei Yao, Zhen Dong, Amir Gholami, Michael W. Mahoney, and Kurt Keutzer. 2020. ZeroQ: A novel zero shot quantization framework. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 13169–13178.

[8] Andreas Charalampopoulos, Nikolas Chatzis, Foivos Ntoulas-Panagiotopoulos, Charilaos Papaioannou, and Alexandros Potamianos. 2024. Enhancing fast feed forward networks with load balancing and a master leaf node. Retrieved from https://arxiv.org/abs/2405.16836

[9] Vyacheslav Chaunin and Nikolay Mikhaylovskiy. 2025. Matrix mixture of experts is the best fast feed-forward. In *First Conference of Mathematics of AI*. Retrieved from https://openreview.net/forum?id=DtLHemXw3w

[10] Ying Chen, Feng Mao, Jie Song, Xinchao Wang, Huiqiong Wang, and Mingli Song. 2021. Self-born wiring for neural trees. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*. IEEE Computer Society, Los Alamitos, CA, USA, 5027–5036. DOI: https://doi.org/10.1109/iccv48922.2021.00500

[11] Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. 2015. BinaryConnect: Training deep neural networks with binary weights during propagations. In *Proceedings of the 29th International Conference on Neural Information Processing Systems - Volume 2 (NIPS'15)*. MIT Press, Montreal, Canada, 3123–3131.

[12] Benoit Courty, Victor Schmidt, Sasha Luccioni, Goyal-Kamal, MarionCoutarel, Boris Feld, Jérémy Lecourt, LiamConnell, Amine Saboni, Inimaz, supatomic, Mathilde Léval, Luis Blanche, Alexis Cruveiller, ouminasara, Franklin Zhao, Aditya Joshi, Alexis Bogroff, Hugues de Lavoreille, Niko Laskaris, Edoardo Abati, Douglas Blank, Ziyao Wang, Armin Catovic, Marc Alencon, Michał Stęchły, Christian Bauer, Lucas Otávio N. de Araújo, JPW, and MinervaBooks. 2024. mlco2/codecarbon: v2.4.1. Zenodo. DOI: https://doi.org/10.5281/zenodo.11171501

[13] Leonardo Lucio Custode, Pietro Farina, Eren Yildiz, Renan Beran Kilic, Kasim Sinan Yildirim, and Giovanni Iacca. 2024. Fast-Inf: Ultra-fast embedded intelligence on the batteryless edge. In *Proceedings of the Conference on Embedded Networked Sensor Systems*. ACM, New York, NY, USA, 239–252.

[14] Yimin Dai, Li-Lian Wang, and Rui Tan. 2025. *Stochastic Differential Equation Networks for Time Series at Edge*. ACM, New York, NY, USA, 345–356. DOI: https://doi.org/10.1145/3715014.3722051

[15] Li Deng. 2012. The MNIST database of handwritten digit images for machine learning research. *IEEE Signal Processing Magazine* 29, 6 (2012), 141–142.

[16] OpenVINO developers. 2018. OpenVINO. Retrieved May 12, 2026 from https://github.com/openvinotoolkit/openvino

[17] ONNX Runtime developers. 2018. ONNX Runtime. Retrieved May 12, 2026 from https://github.com/microsoft/onnxruntime

[18] Jun Fang, Ali Shafiee, Hamzah Abdel-Aziz, David Thorsley, Georgios Georgiadis, and Joseph Hassoun. 2020. Near-lossless post-training quantization of deep neural networks via a piecewise linear approximation. *CoRR* abs/2002.00104 (2020). Retrieved from https://arxiv.org/abs/2002.00104

[19] Jiaqi Guan, Yang Liu, Qiang Liu, and Jian Peng. 2017. Energy-efficient Amortized Inference with Cascaded Deep Classifiers. arXiv:1710.03368. Retrieved from https://arxiv.org/abs/1710.03368

[20] Song Han, Jeff Pool, John Tran, and William J. Dally. 2015. Learning both Weights and Connections for Efficient Neural Networks. arXiv:1506.02626. Retrieved from https://arxiv.org/abs/1506.02626

[21] Dan Hendrycks and Kevin Gimpel. 2023. Gaussian Error Linear Units (GELUs). arXiv:1606.08415. Retrieved from https://arxiv.org/abs/1606.08415

[22] Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. 2017. MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications. arXiv:1704.04861. Retrieved from https://arxiv.org/abs/1704.04861

[23] Forrest N. Iandola, Song Han, Matthew W. Moskewicz, Khalid Ashraf, William J. Dally, and Kurt Keutzer. 2016. SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and <0.5MB model size. arXiv:1602.07360. Retrieved from https://arxiv.org/abs/1602.07360

[24] Andrey Ignatov. 2017. Real-time human activity recognition from accelerometer data using Convolutional Neural Networks. Retrieved May 12, 2026 from https://github.com/aiff22/HAR

[25] Yani Ioannou, Duncan Robertson, Darko Zikic, Peter Kontschieder, Jamie Shotton, Matthew Brown, and Antonio Criminisi. 2016. Decision Forests, Convolutional Networks and the Models in-Between. arXiv:1409.0473. Retrieved from https://arxiv.org/abs/1701.00133

[26] Renan Beran Kilic, Kasim Sinan Yildirim, and Giovanni Iacca. 2025. Multi-objective evolutionary optimization of virtualized fast feedforward networks. In *Proceedings of the Applications of Evolutionary Computation*. Pablo García-Sánchez, Emma Hart, and Sarah L. Thomson (Eds.), Springer Nature, Cham, Switzerland, 270–286.

[27] Kontschieder, Peter and Fiterau, Madalina and Criminisi, Antonio and Bulò, Samuel Rota. 2016. Deep neural decision forests. In *Proceedings of the International Joint Conference on Artificial Intelligence*. AAAI Press, New York, NY, USA, 4190–4194.

[28] Seulki Lee and Shahriar Nirjon. 2020. Fast and scalable in-memory deep multitask learning via neural weight virtualization. In *Proceedings of the International Conference on Mobile Systems, Applications, and Services*. ACM, New York, NY, USA, 175–190.

[29] Hao Li, Asim Kadav, Igor Durdanovic, Hanan Samet, and Hans Peter Graf. 2016. Pruning filters for efficient convNets. *CoRR* abs/1608.08710 (2016). Retrieved from http://arxiv.org/abs/1608.08710

[30] Qi Li, Hengyi Li, and Lin Meng. 2022. Feature map analysis-based dynamic CNN pruning and the acceleration on FPGAs. *Electronics* 11, 18 (2022), 15 pages.

[31] Zhuo Li, Hengyi Li, and Lin Meng. 2023. Model compression for deep neural networks: A survey. *Computers* 12, 3 (2023), 22 pages.

[32] Xiaofan Lin, Cong Zhao, and Wei Pan. 2017. Towards accurate binary convolutional neural network. In *Proceedings of the 31st International Conference on Neural Information Processing Systems (NIPS'17)*. Curran Associates Inc., Long Beach, California, USA, 344–352.

[33] Renyuan Liu, Yuyang Leng, Shilei Tian, Shaohan Hu, Chun-Fu (Richard) Chen, and Shuochao Yao. 2024. DynaSpa: Exploiting spatial sparsity for efficient dynamic DNN inference on devices. In *Proceedings of the Conference on Embedded Networked Sensor Systems*. ACM, New York, NY, USA, 422–435. DOI: https://doi.org/10.1145/3666025.3699348

[34] Zhuang Liu, Hanzi Mao, Chao-Yuan Wu, Christoph Feichtenhofer, Trevor Darrell, and Saining Xie. 2022. A ConvNet for the 2020s. In *2022 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. 11966–11976. DOI: https://doi.org/10.1109/CVPR52688.2022.01167

[35] Xudong Lu, Qi Liu, Yuhui Xu, Aojun Zhou, Siyuan Huang, Bo Zhang, Junchi Yan, and Hongsheng Li. 2024. Not all experts are equal: Efficient expert pruning and skipping for mixture-of-experts large language models. In *Proceedings of the Annual Meeting of the Association for Computational Linguistics*. Lun-Wei Ku, Andre Martins, and Vivek Srikumar (Eds.), Association for Computational Linguistics, Bangkok, Thailand, 6159–6172. DOI: https://doi.org/10.18653/v1/2024.acl-long.334

[36] Jian-Hao Luo, Hao Zhang, Hong-Yu Zhou, Chen-Wei Xie, Jianxin Wu, and Weiyao Lin. 2019. ThiNet: Pruning CNN filters for a thinner net. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 41, 10 (2019), 2525–2538.

[37] Vinod Nair and Geoffrey E. Hinton. 2010. Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th International Conference on International Conference on Machine Learning*. Omnipress, Madison, WI, USA, 807–814.

[38] Meike Nauta, Ron van Bree, and Christin Seifert. 2021. Neural prototype trees for interpretable fine-grained image recognition. In *2021 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. 14928–14938. DOI: https://doi.org/10.1109/CVPR46437.2021.01469

[39] Nicholas Nethercote and Julian Seward. 2007. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*. Association for Computing Machinery, New York, NY, USA, 89–100. DOI: https://doi.org/10.1145/1250734.1250746

[40] Hui Ni and The ncnn contributors. 2017. ncnn. Retrieved May 12, 2026 from https://github.com/Tencent/ncnn

[41] Renkun Ni, Hong-Min Chu, Oscar Castañeda, Ping yeh Chiang, Christoph Studer, and Tom Goldstein. 2020. WrapNet: Neural Net Inference with Ultra-Low-Resolution Arithmetic. arXiv:2007.13242. Retrieved May 12, 2026 from https://arxiv.org/abs/2007.13242

[42] Andrey Ignatov. 2018. Real-time human activity recognition from accelerometer data using convolutional neural networks. *Applied Soft Computing* 62 (2018), 915–922. DOI: https://doi.org/10.1016/j.asoc.2017.09.027

[43] Wolfgang Roth and Franz Pernkopf. 2020. Bayesian neural networks with weight sharing using dirichlet processes. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 42, 1 (2020), 246–252.

[44] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. 2018. MobileNetV2: Inverted residuals and linear bottlenecks. In *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. IEEE Computer Society, Los Alamitos, CA, USA, 4510–4520. DOI: https://doi.org/10.1109/CVPR.2018.00474

[45] Simone Scardapane, Alessandro Baiocchi, Alessio Devoto, Valerio Marsocci, Pasquale Minervini, and Jary Pomponi. 2024. Conditional computation in neural networks: Principles and research trends. *Intelligenza Artificiale* 18, 1 (2024), 175–190. DOI: https://doi.org/10.3233/ia-240035

[46] Shakhrul Iman Siam, Hyunho Ahn, Li Liu, Samiul Alam, Hui Shen, Zhichao Cao, Ness Shroff, Bhaskar Krishnamachari, Mani Srivastava, and Mi Zhang. 2025. Artificial intelligence of things: A survey. *ACM Transactions on Sensor Networks* 21, 1 (2025), 1–75.

[47] Ryutaro Tanno, Kai Arulkumaran, Daniel Alexander, Antonio Criminisi, and Aditya Nori. 2019. Adaptive neural trees. In *Proceedings of the 36th International Conference on Machine Learning (Proceedings of Machine Learning Research)*. PMLR, 6166–6175. Retrieved from https://proceedings.mlr.press/v97/tanno19a.html

[48] Vivek Menon U, Vinoth Babu Kumaravelu, Vinoth Kumar C, Rammohan A, Sunil Chinnadurai, Rajeshkumar Venkatesan, Han Hai, and Poongundran Selvaprabhu. 2025. AI-powered IoT: A survey on integrating artificial intelligence with IoT for enhanced security, efficiency, and smart applications. *IEEE Access* 13 (2025), 50296–50339. DOI: https://doi.org/10.1109/access.2025.3551750

[49] Karen Ullrich, Edward Meeds, and Max Welling. 2017. Soft weight-sharing for neural network compression. In *International Conference on Learning Representations*. Retrieved from https://openreview.net/forum?id=HJGwcKclx

[50] Alvin Wan, Lisa Dunlap, Daniel Ho, Jihan Yin, Scott Lee, Henry Jin, Suzanne Petryk, Sarah Adel Bargal, and Joseph E. Gonzalez. 2020. NBDT: Neural-backed decision trees. *CoRR* abs/2004.00221 (2020). Retrieved from https://arxiv.org/abs/2004.00221

[51] Xubin Wang, Zhiqing Tang, Jianxiong Guo, Tianhui Meng, Chenhao Wang, Tian Wang, and Weijia Jia. 2025. Empowering edge intelligence: A comprehensive survey on on-device AI models. *Computing Surveys* 57, 9 (2025), 139. DOI: https://doi.org/10.1145/3724420

[52] Xin Wang, Fisher Yu, Zi-Yi Dou, Trevor Darrell, and Joseph E. Gonzalez. 2018. SkipNet: Learning dynamic routing in convolutional networks. In *Proceedings of the European Conference on Computer Vision*. Springer-Verlag, Berlin, 420–436.

[53] Pete Warden. 2018. Speech commands: A dataset for limited-vocabulary speech recognition. *CoRR* abs/1804.03209 (2018). Retrieved from http://arxiv.org/abs/1804.03209

[54] Dongqing Zhang, Jiaolong Yang, Dongqiangzi Ye, and Gang Hua. 2018. LQ-Nets: Learned quantization for highly accurate and compact deep neural networks. *CoRR* abs/1807.10029. (2018). Retrieved from http://arxiv.org/abs/1807.10029

[55] Xiangyu Zhang, Xinyu Zhou, Mengxiao Lin, and Jian Sun. 2018. ShuffleNet: An extremely efficient convolutional neural network for mobile devices. In *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 6848–6856. DOI: https://doi.org/10.1109/CVPR.2018.00716

[56] Hao Zhou, Jose M. Alvarez, and Fatih Porikli. 2016. Less is more: Towards compact CNNs. In *Proceedings of the European Conference on Computer Vision*. Bastian Leibe, Jiri Matas, Nicu Sebe, and Max Welling (Eds.), Springer International Publishing, Cham, Switzerland, 662–677.

[57] Zhi Zhou, Xu Chen, En Li, Liekang Zeng, Ke Luo, and Junshan Zhang. 2019. Edge intelligence: Paving the last mile of artificial intelligence with edge computing. *Proc. IEEE* 107, 8 (2019), 1738–1762.

[58] Michael Zhu and Suyog Gupta. 2017. To prune, or not to prune: Exploring the efficacy of pruning for model compression. Retrieved from https://arxiv.org/abs/1710.01878

Appendix

A Other Scheduler Experiments

We evaluated four additional schedulers beyond those described in Section 4.2: constant-warmup, ease-in-warmup, ease-out-warmup, and sigmoid. This additional set of experiments allowed us to assess the impact of warmup rates across different scheduler types and determine whether a sigmoid scheduler could further improve performance.

The four additional awareness scheduling strategies are visualized in Figure 27 and are defined as

$$\begin{aligned}
\text{Constant-warmup:} \quad & \alpha = \begin{cases} 0 & \text{if } t < \beta T \\ 0.5 & \text{if } t \geq \beta T \end{cases} \\
\text{Ease-in-warmup:} \quad & \alpha = \begin{cases} 0 & \text{if } t < \beta T \\ \left(\frac{t}{T}\right)^2 & \text{if } t \geq \beta T \end{cases} \\
\text{Ease-out-warmup:} \quad & \alpha = \begin{cases} 0 & \text{if } t < \beta T \\ 1 - \left(1 - \frac{t}{T}\right)^2 & \text{if } t \geq \beta T \end{cases} \\
\text{Sigmoid:} \quad & \alpha = \frac{1}{1 + \exp(-(t - T))}
\end{aligned} \tag{10}$$

In these equations, T denotes the total number of training epochs, and β sets the warmup duration during which the awareness parameter α remains at zero. After the warmup, each schedule increases α according to its rule, allowing us to study how different growth patterns of α affect model performance.

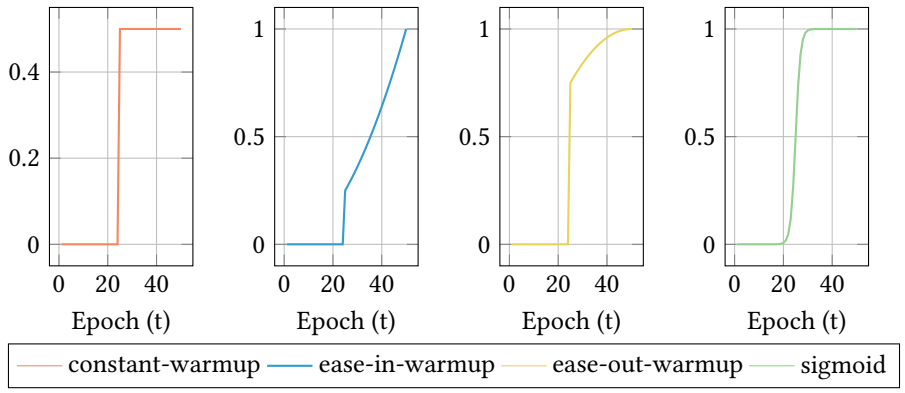


Fig. 27. Different awareness schedules experimented for inference-aware training.

Figure 28 presents the accuracy of IAFFF models trained with these four additional schedulers on various datasets and architectures. It can be seen that the additional schedulers introduced do not lead to substantial improvements in accuracy with respect to those reported in the main text. Moreover, their results are generally close across different datasets and architectures. While minor variations exist, none of these schedulers consistently outperform the linear-warmup scheduler, which remains the most effective overall (see Section 6.1.1). This suggests that, beyond the schedulers discussed in the main text, further modifications to the warmup strategy yield limited gains.

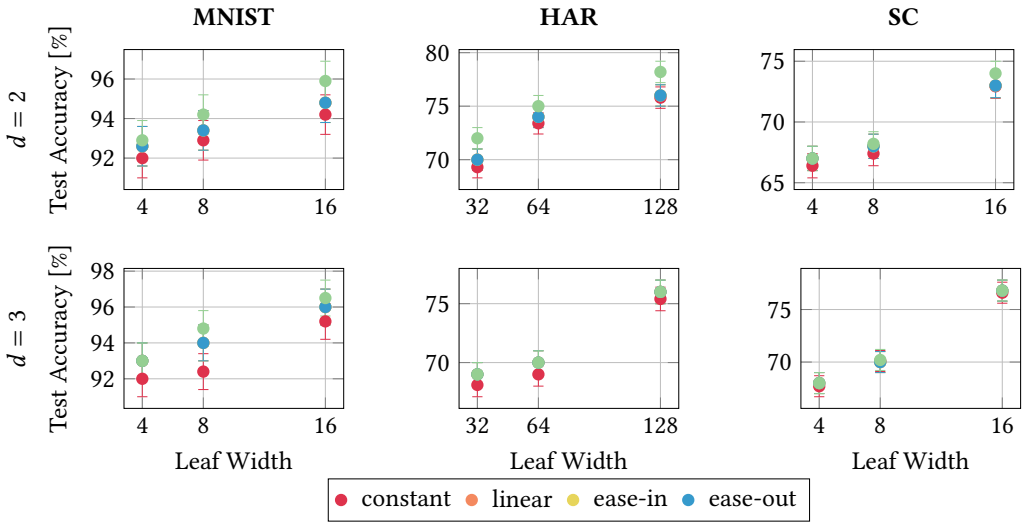


Fig. 28. Accuracy comparison with other awareness schedules experimented for inference-aware training.

Overall, these findings indicate that the choice among these alternative schedulers has little impact on performance, and for this reason, we used the linear-warmup scheduler in all the experiments reported in the main text.

B Cache Characteristics

In this section, we analyze the memory characteristics of different models on the MNIST dataset using Valgrind's Cachegrind [39] feature on both Intel and ARM (Raspberry Pi 1) CPUs. Cachegrind provides detailed information about cache misses and hits, allowing us to study the cache behavior of models. The choice of Raspberry Pi aims at showing how the cache behavior of the models changes in a more resource-constrained platform, and whether the virtualization overhead is more significant in such environments. Further cache information about the CPUs used in the experiments is provided in Table 5. Models are run 1,000 times in each case to provide a stable evaluation of the cache behavior during inference.

In Tables 6 and 7, the results show that virtualized models have more **instruction fetches (Ir)** and **data reads (Dr)** than non-virtualized models, which is expected due to the additional instructions and memory accesses introduced by virtualization. Moreover, virtualized models also have more cache misses (I1mr, ILmr, D1mr, DLmr, D1mw, DLmw) compared to non-virtualized models, which further contributes to the inference overhead of virtualization. The increase in cache misses is more significant in the Raspberry Pi environment, likely due to its more limited cache size and memory bandwidth compared to the Intel CPU. Even though the increase in cache misses is more significant in the Raspberry Pi environment, the overall inference overhead of virtualization is still relatively small compared to the accuracy benefits it provides, as shown in Figure 23.

Table 5. Cache Characteristics of Intel i5-8365U and Raspberry Pi 1

Cache Level	Raspberry Pi 1 (single core)	Intel CPU (4 cores)	Notes
L1d	16 KiB	128 KiB (4 instances)	Data cache
L1i	16 KiB	128 KiB (4 instances)	Instruction cache
L2	128 KiB	1 MiB (4 instances)	L2 per core (Intel);
L3	None	6 MiB (1 instance)	Shared across all Intel cores

Table 6. Cachegrind Results on Intel i5-8365 for Different Models on the MNIST Dataset

Function/File	Ir	I1mr	ILmr	Dr	D1mr	DLmr	Dw	D1mw	DLmw
First Dataset (fff)									
neuron	2,066,380,000	1	1	1,032,840,000	6,457,049	6,499	104,180,000	0	0
fff	5,204,000	10	10	1,251,000	10,001	11	288,000	9,001	10
TOTAL (neuron+fff)	2,071,584,000	11	11	1,034,091,000	6,467,050	6,510	104,468,000	9,001	10
PROGRAM TOTAL	2,071,729,473	1,099	1,083	1,034,124,637	6,469,272	7,497	104,478,926	9,342	315
Second Dataset (vfff)									
vneuron	2,278,912,000	3	3	1,154,560,000	7,290,037	9,970	129,536,000	0	0
neuron	57,164,000	2	2	28,552,000	130,050	180	2,932,000	0	0
vfff	5,076,000	8	8	1,123,000	12,000	12	416,000	9,000	2
TOTAL (vneuron+neuron+vfff)	2,330,792,000	13	13	1,184,525,200	7,422,087	8,162	132,884,000	10,000	2
PROGRAM TOTAL	2,442,082,145	1,155	1,135	1,184,529,688	7,434,539	9,283	132,896,811	10,384	342

Instruction Fetches (Ir), L1 Instr Misses (I1mr), LL Instr Misses (ILmr), Data Reads (Dr), L1 Data Misses (D1mr), LL Data Misses (DLmr), Data Writes (Dw), L1 Data Write Misses (D1mw), LL Data Write Misses (DLmw). FFF and VFFF models with $f_{2,128}$ architecture are used for this experiment.

Table 7. Cachegrind Results on Raspberry Pi 1 with ARMv6 CPU and 512MB RAM for Different Models on the MNIST Dataset

Function/File	Ir	I1mr	ILmr	Dr	D1mr	DLmr	Dw	D1mw	DLmw
First Dataset (fff)									
neuron	2,170,420,000	2	2	1,032,700,000	6,460,049	35,049	207,380,000	1	1
fff	6,482,000	10	10	1,809,000	11,001	11,000	289,000	8	3,006
TOTAL (neuron+fff)	2,176,902,000	12	12	1,034,509,000	6,471,050	46,049	207,669,000	8,002	3,007
PROGRAM TOTAL	2,177,076,270	1,101	799	1,034,555,941	6,473,237	46,235	207,688,449	8,209	3,194
Second Dataset (vfff)									
vneuron	2,744,832,000	4	4	1,410,432,000	7,927,007	37,938	257,792,000	0	0
neuron	40,084,000	2	2	28,540,000	141,047	10,170	5,780,000	1	1
vfff	2,946,000	19	18	1,681,000	12,003	13	289,000	11,000	3,058
TOTAL (vneuron+neuron+vfff)	2,809,862,000	25	14	1,440,653,000	8,080,057	48,11	263,861,000	11,001	3,109
PROGRAM TOTAL	2,791,010,780	1,117	1192	1,440,828,448	8,082,295	49,224	263,880,631	11,213	3,337

Instruction Fetches (Ir), L1 Instr Misses (I1mr), LL Instr Misses (ILmr), Data Reads (Dr), L1 Data Misses (D1mr), LL Data Misses (DLmr), Data Writes (Dw), L1 Data Write Misses (D1mw), LL Data Write Misses (DLmw). FFF and VFFF models with $f_{2,128}$ architecture are used for this experiment.

Received 8 October 2025; revised 3 March 2026; accepted 15 April 2026