MiniLearn: On-Device Learning for Low-Power IoT Devices

Christos Profentzas
Chalmers University of Technology
Gothenburg, Sweden
chrpro@chalmers.se

Magnus Almgren
Chalmers University of Technology
Gothenburg, Sweden
magnus.almgren@chalmers.se

Olaf Landsiedel
Kiel University & Chalmers University of Technology
Kiel, Germany
ol@informatik.uni-kiel.de

Abstract

Recent advances in machine learning enable new, intelligent applications in the Internet of Things. For example, today's smartwatches use Deep Neural Networks (DNNs) to detect and classify human activities. The training of DNNs, however, is done offline with previously collected and labeled datasets using extensive computational resources such as GPUs on cloud services. Once being quantized and deployed on an IoT device, a DNN commonly remains unchanged.

We argue that this static nature of trained DNNs strongly limits their flexibility to adapt to requirements that change dynamically. For example, the device may need to adjust on the fly to the limited memory and energy resources, but only the retraining or pruning of the DNN in the cloud can address these issues. Moreover, the user may need to add new classes or refine existing ones, due to different problem domains materializing dynamically. Retraining DNNs requires a high volume of data collected from IoT devices and transmitted to the cloud. However, IoT devices depend on energy-efficient communication with limited reliability and network bandwidth. In addition, cloud storage of extensive IoT data raises significant privacy concerns.

This paper introduces MiniLearn that enables re-training of DNNs on resource-constrained IoT devices. MiniLearn allows IoT devices to re-train and optimize pre-trained, quantized neural networks using IoT data collected during deployment of an IoT device. We show that MiniLearn speeds up inference by a factor of up to 2 and requires up to 50% less memory compared to original DNN. In addition, MiniLearn increases classification accuracy for a sub-set by 3% to 9% of the original DNN.

Categories and Subject Descriptors

I.2.6 [**Artificial Intelligence**]: Learning

General Terms

Deep Learning, Measurement, Performance

Keywords

TinyML, On-Device Learning, Low-Power

1 Introduction

Compression methods for Deep Neural Networks (DNNs) like quantization and pruning [35, 16] enable intelligent applications on resource-constrained devices [16, 21, 28]. Today, smartwatches and fitness-tracker employ DNNs to detect and classify the activities of their users (Human Activity Recognition [18]) and voice-controlled virtual assistants such as Amazon's Alexa utilize DNNs on IoT devices to spot keywords [7]. However, training, optimization, and compression of DNNs are commonly conducted before deployment. The training process uses previously collected and labeled datasets and often employs vast computational resources like GPUs or even cloud services. Once deployed, DNNs commonly remain unchanged.

Should the underlying tasks change during deployment due to, for example, problem domain shifts or just the need for additional classes in a classifier, DNNs are commonly re-trained in the cloud. In addition, the device may need to adjust on the fly to the limited memory and energy resources, but only the re-training or pruning of the DNN in the cloud can address such issues. For example, when a user uses her smartwatch only for one or two activities, it would be better to act proactive and optimize the DNN for a subset of activities (and have low demand for resources) than instead disable the DNN when the battery is low, or memory is almost full.

The re-training of DNNs follows the traditional offline training approach and commonly requires uploading new datasets to cloud devices and training a new model, which is deployed on the IoT device. This is due to the following two reasons: (A) Training and refinement are both expensive in terms of memory and computation, and thus, today, avoided on resource-constrained IoT devices. Deep neural networks have millions of trainable parameters, while resource-constrained IoT devices have constrained memory, such as 64-256 KBs of RAM, run at a speed of 32-64 MHz CPU, and often operate on batteries, which rules out the training of an entire neural network from scratch. (B) Many optimization methods such as pruning [20] and quantization [14] consider deep neural network compression as the final step and make re-training of the network prohibitively

hard: typically deployed low-bit, integer networks on IoT devices (i.e., 8-bit quantization) work very well for inference but face challenges such as vanishing gradient problems on gradient descent learning algorithms [3, 14]. Offline training leads to significant communication overhead and often also privacy concerns, when personal data is uploaded as part of the training data to, for example, GPUs in the cloud.

In this paper, we argue that in many application settings, the privacy-preserving way of re-training on the constrained IoT device is beneficial for the device and user, outweighing the common approach of uploading data to the cloud for training. We introduce MiniLearn, an open-source architecture training DNNs on constrained microcontrollers, filter pruning, and fine-tuning. It addresses the above challenges as follows: First, MiniLearn stores intermediate compressed outputs of quantized layer(s) as training samples to reduce memory requirements. Second, MiniLearn uses pre-trained quantized neural networks to initialize floating-point hidden layer(s) and reduces their size with static pruning. Third, after training, we quantize and fine-tune the layer(s) back to integers. A (re)trained neural network in MiniLearn consists of re-trained, quantized, and pruned layer(s).

In summary, this paper makes the following key contributions:

- We show that low-power IoT devices can re-train and optimize pre-trained networks using data locally without the need for privacy-sensitive and communication intense data-upload to, i.e, cloud services for training. Moreover, we reduce memory consumption and inference latency with increased accuracy when re-training for a subset of classes.
- We design and implement MiniLearn, an open-source[1] on-device learning system for low-power IoT devices, and present the challenges and trade-offs regarding on-device learning on resource-constrained devices.
- We quantify the performance of MiniLearn in terms of accuracy, computation, memory, and energy consumption. Notably, we find that after MiniLearn, we can reduce the neural network's inference time up to 48% and memory up to 50%, with increased accuracy by 3% to 9% for a subset of the original network.

Paper outline. The rest of the paper is organized as follows. Section 2 provides the necessary background and related work. Section 3 overviews the main challenges for on-device learning. Section 4 explains the architecture of MiniLearn and shows how we address the challenges of on-device learning. Section 5 describes the methodology of our experiments. Section 6, presents our experimental results Section 7, presents the related work, and Section 8 concludes the paper.

2 Background

A typical deep neural network consists of the input and several hidden layers, with a final linear output providing the class prediction. While many types of neural networks exist (i.e., recurrent and convolutional), convolutional layers have prevailed since they can also solve traditional tasks of re-

[1] available after publication

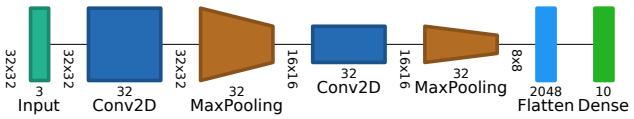


Figure 1. A representative architecture of a Convolution Neural Network (CNN) typically found on low-power IoT devices. The CNN consists of several layers stacking together: convolution kernels, max pooling, and fully connected (flatten & dense). The final layer is the output for the classification task.

current networks. For example, a problem of audio recognition can be classified using a 2D-convolutional hidden layer(s) [11], and Human Activity Recognition (HAR) can be classified using 1D-convolutional hidden layers [18]. An example of a typical Convolutional Neural Network (CNN) for IoT devices is illustrated in Figures 1.

Optimizing and compressing deep neural networks before embedded system deployment is common due to otherwise high memory, computational, and energy requirements. A common compression method is int-8 quantization [35] by using an 8-bit fixed-point representation of original floating-point values. It reduces the size without changing the original architecture of the network. Quantization typically rescales the min-max range of the neural network's weights and introduces a shifting offset for the zero-point as it may be different from real numbers after quantization. The pre-quantized network still needs to be trained in high-level libraries on high-end computers. This has led to two main variations of quantization: post-training and quantization-aware training.

Post-training quantization. With this method [16, 21, 8], the deep neural network is trained as usual in floating-point using a standard available library (e.g., Tensorflow, PyTorch). After the training period, we can use quantization statically on the network by finding a new min-max range for integer weights based on already-trained layers. This method may need a representative data set to calibrate and fine-tune the weights based on the outputs of the activation layers. This fine-tuning is a one-time process before the deployment on the IoT device.

Quantization-aware training. Quantization aware training method [14] simulates the quantization by making it part of the training process to learn the quantization values automatically. The method requires the duplication of weights for each layer in two formats. One format is in the regular 32-bit floating-point, and the other is the equivalent quantized 8-bit values. During the forward pass, it uses the integer precision layers to calculate the output of the network to simulate the deployment on the IoT device. However, during the backward pass, it uses the floating-point values to calculate the gradients and updates of the network. Then, it quantizes the floating-point weights and replaces the previous integer values. As the quantization error becomes part of the learning process, the method can lead to increased accuracy of the final quantized network. However, with this method, the quantization parameters become part of the training hyper-

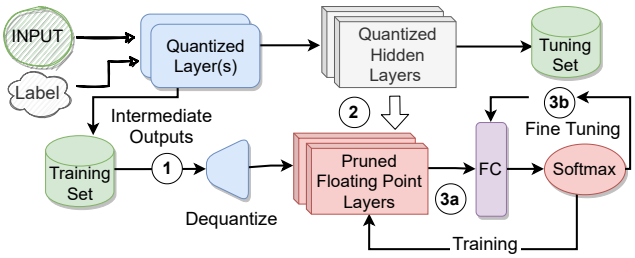


Figure 2. The main structure of MiniLearn and how to address the three main challenges. First, during the forward pass, it stores the output of the first layer in int-8 format, while during training, it dequantizes one batch at a time to floating point. Second, it dequantizes the filters to floating point and applies static pruning to reduce the number of trainable parameters. Third, it trains the convolutional layers and fine-tunes the Fully Connected (FC) layers.

parameter that we need to manually tune and expect different results depending on the training process.

Pruning. Another compression method is the pruning of either the weights or filters [20, 22] of the network. The main idea is to identify and remove redundant parts of the networks, categorized in two different methods: weight pruning and filter pruning. Weight pruning [22] targets redundant zeroes and model sparsity by creating more dense layers. In contrast, filter pruning [20] considers convolutional filters as blocks and removes filters that do not contribute enough to each layer's output.

Inference. Inference with IoT devices has become widely available, for example, ARM CMSIS-NN [16] and Tensorflow Micro [8] offer libraries for quantized neural network inference on low-power devices. The libraries can optimize the neural networks for low-power inference hardware (e.g., Cortex-M, ESP), and may increase performance by utilizing available on-board DSP accelerator. However, the neural networks are trained and quantized in the cloud (typically with post-training quantization as described above).

3 MiniLearn Overview

This section describes the motivation behind MiniLearn. First, we introduce a motivational application scenario where we need on-device learning. Second, we present the data flow, training steps, and system challenges to enable training and fine-tuning on low-power IoT devices.

3.1 Application Scenario

As a motivating scenario, we consider a smartwatch application tracking users' health and fitness progress from exercise. In the past, most of the data analysis happened in the cloud, but with recent advances, we see parts of inference happen locally on the device. The smartwatch has a pre-trained network based on generic data engaging in various activities and sports. With MiniLearn, we envision a scenario where the local user shifts her activities to a smaller group of activities after using the device for a while. While the user engages in activities, the device creates a training set using the hidden layer outputs and labeling either by using another sensor or asking the user through user interaction. When the training set is filled with enough data (typically 100-600), it initiates the retraining and fine-tuning of the neural networks with appended Fully Connected (FC) layer(s). The local training set never leaves the device, and the cloud provider does not acquire any personal data, preserving the privacy of the user.

3.2 Data Flow

In Figure 2, we illustrate the flow of data collection and training process. First, a device sensor or user provides the label during the forward pass, and MiniLearn stores the pair of the label/output of the last hidden layer in a local data set. The process continues as long the device has enough storage for the training samples. In the second step, MiniLearn initiates the training process after collecting enough training data.

3.3 Training Steps

MiniLearn achieves on-device learning by filter pruning and learning to low-power IoT devices in three steps, illustrated in Figure 2. First, it utilizes efficient memory storage by collecting and storing the quantized hidden layer outputs (suitable for de-quantization) in integer format. Second, it uses floating-point hidden layers to allow training with gradient descent algorithms. Third, it de-quantizes the training data only during training, which combines a low memory footprint and adequate gradients for learning.

3.4 System Challenges

We identify three main system challenges when designing MiniLearn. First, common deep learning algorithms depend on a significant number of training samples that require resources unavailable on common IoT devices. Second, neural networks are typically quantized on low-power IoT for efficiency, and reversing the quantization to enable additional learning on existing neural networks is not well-studied [9]. Third, even quantized neural networks have millions of parameters, and additional training requires significant resources.

(1) Training samples. First, on-device learning needs to optimize the data storage of training samples to address memory limitations, together with computational and energy trade-offs. For example, a typical batch of 32 images of the popular CIFAR-10 data set will occupy 128 KB, a significant amount for a low-power device. Typically most data sets are collected with high-dimensional features on the cloud, where data pre-processing and reduction may happen. Cloud services have an abundance of resources, and they can utilize pipelines and distributed nodes for data processing. In contrast, the local data of IoT has limited capabilities for pre-processing high-dimensional data.

(2) Learning based on quantized layers. Second, deep neural networks on low-power IoT are typically quantized to address memory and computational constraints. Quantization is straightforward and is adequate for inference-only tasks, but utilizing or reversing already quantized layers for additional learning is an open challenge [9, 3, 14]. The main reason is that after quantization, most hidden layers are restrained by the low dynamic range and precision. Since the training process of neural networks commonly uses gradient descent algorithms [3], without applying a technique to

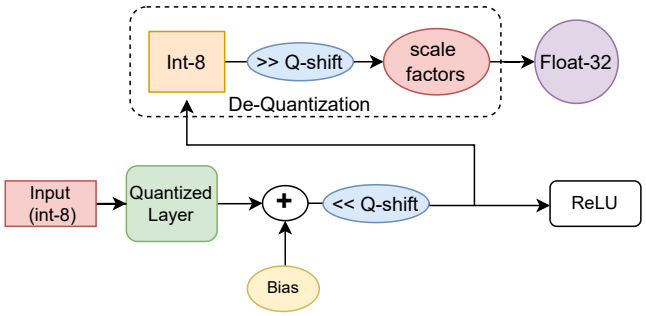


Figure 3. MiniLearn reverses the quantization for training purposes. The figure shows the forward pass on a quantized layer and the dequantization operation over one sample. In paractice MiniLearn applies the operation on a collection of samples during training. The shift operator makes sure the zero-point offset and quantized domains in the same when we reverted back to real numbers.

reverse or extend parts of the quantization, it leads to inadequate results [14].

(3) Number of training parameters. Third, typically deep neural networks have millions of parameters to be trained. On-device learning needs to incorporate methods to reduce the number of parameters without compromising the model's accuracy. However, standard methods (e.g., neural architecture search, pruning) for removing parameters are iterative approaches [23, 20] that find the optimal configuration through several possible architectures by storing checkpoints and then choosing the one performing the best. This is unfeasible using resource-constrained devices that depend on KBs of memory and batteries to operate.

4 MiniLearn Design

MiniLearn's main objective is to provide on-device learning using personalized IoT data. MiniLearn retrains neural networks towards the end-user preferences without cloud interaction by keeping the data on the device and preserving the user's privacy.

Our design is based on two key observations. First, we can reduce the training memory requirements, by dynamically collecting intermittent outputs of hidden layers in integer precision and dequantize (see bellow) to floating-point precision only during training. This approach provides a sufficient dynamic range for calculating gradients during back-propagation, avoiding duplicated weights. Second, training hidden layers in floating precision while keeping the pre-trained network in integer format reduces the demand for computational and energy resources without affecting the overall learning process.

Dequantization. To address the resource-constrained devices, we enforce the optimization and the proper placement of the dequantization process. In Figure 3, we present in detail where we collect the intermittent outputs and which operations are involved. We collect the output before applying ReLU to avoid consecutive zeros. Due to quantization, the pre-trained networks apply Q-shift, an offset of the difference between the integer domain's zero-point and real numbers (as explained in Section 2). We need to reverse both the min-max scale factors between integer and real numbers and shift back and reverse the zero-point offset. A key observation is that the division is a power of 2 which can also be done using bit-shifting, and most cross-compilers can highly optimize and combine these operations.

4.1 MiniLearn Architecture

In this section, we explain in detail the main parts of the MiniLearn architecture that address the three challenges stated above.

(1) Sample Storage First, MiniLearn addresses memory constraints by collecting intermediate outputs of the first layer(s) in their compressed quantized (int-8). We divide the samples into a training set for the hidden layer(s) and a tuning set for the fully connected layer(s) (see part 3a-b). The samples are restored to floating-point only during training using the scale factors from quantization.

(2) Network architecture. MiniLearn avoids dynamic range and precision problems by trading-off memory for floating-point precision for both the hidden layer(s) and fully connected layers during training. The first step is to dequantize the filters of hidden layers and prune them in case of memory constraints. We apply proper static pruning to avoid iterative approaches using standard filter ranking methods (e.g., based on the L1 - L2 norm). We let the fully connected layers compensate for the filter reduction by a final fine-tuning for a few extra epochs. Finally, we modify the fully-connected layers from the initial network to have the number of neurons corresponding to the subset we want to optimize.

(3a) Filter training. MiniLearn is able to use standard training algorithms and reduce the number of parameters with a two-step approach. We represent convolutional filters and fully connected layers with 32-bit floating-point, but we apply pruning on the filters. This way, we can train with back-propagation and min-batch stochastic gradient descent, and at the same time reduce the number of trainable parameters. For calculating the loss of the network, we use cross-entropy. We use small batches to dequantize and temporarily store the input samples (collected from the first layer) in a 32-floating point. After training, we quantize the convolutional filters back to integer so as to address the device's memory and computational limitation during inference.

(3b) Fine-tuning. MiniLearn compensates for filter pruning quantization with a final fine-tuning on the fully connected layer(s). This step is necessary as prior pruning and quantization introduce some loss of information. The previous layers are frozen during this step, and the only trainable parameters are from the fully connected layers. The complete algorithm is presented in Algorithm 1, explained in detail in the next section. The final network consists of unchanged pre-trained layers(s), the re-trained and quantized convolutional layer(s), and floating-point fully connected layers.

4.2 MiniLearn Stages

In this section, we explain in detail the MiniLearn learning algorithm stage by stage. There are three main components involved: a) Pruning and filter selection, b) Re-training

Algorithm 1: MiniLearn Algorithm

Data: Training Set, Fine-tuning Set
Stage-I: Pruning
for *Convolutional-layer in Network* **do**
  for *Filter in Convolutional-layer* **do**
    FilterScores[i] ¡- Norm L2 (Filter) ; //select filters
  end
  Pruning (Percentage, FilterScores)
end
Stage-II: Training Filters
for *epoch e = 1, ... , E* **do**
  for *Sample in Training Set* **do**
    // 32-bit floating point format
    $Y^{(i)}, X^{(i)}$:= de-quantize (Sample, Q-factors);
    $Y^{(j)}$:= Forward (Hidden-layers, $X^{(i)}$);
    delta := $-\sum_{i=1}^{m} y_i \log(y_j)$ // cross entropy loss
    Back propagation (Hidden-layers, delta);
  end
end
for *Filter in Convolutional-layer* **do**
  Quantize (Filter, Q-factors) ; //quantize filters back to int
end
Stage-III: Fine-Tuning
FC := Randomized Fully Connected Layer(s)
Freeze (Hidden-Layers) // no updates prior to FC
for *epoch e = 1, ... , E* **do**
  for *batch b in Fine-Tuning Set* **do**
    // 32-bit floating point format
    $Y^{(i)}, X^{(i)}$:= de-quantize (b, Q-factors);
    $Y^{(j)}$:= Forward (FC, $X^{(i)}$);
    delta := $-\sum_{i=1}^{m} y_i \log(y_j)$ // cross entropy loss
    Back propagation (FC, delta);
  end
end

and adjusting the filters, and c) Final fine-tuning.

4.2.1 Dequantizing and pruning of filters (Stage-I)

The primary purpose of pruning is to address the memory constraints of the device. We use the L1 norm of each filter-matrix as a static method for selecting the convolution filters. Other options include L2 and Max norm. In the case of dequantized weights, filters with smaller L1 norm result in relatively small activations implying that they are less significant for the networks [20].

We keep intact the first convolution layer(s) in integer format to retain compressed inputs for the network. The choice of which layer to freeze and re-train is a hyperparameter of the algorithm. Next, we define a floating-point hidden convolutional layer(s) and initialize their weights by converting the quantized values back to float-point using their scale factors. Similarly, we define floating-point fully connected layer(s) based on the pre-trained network, but we reduced the output neurons to the number of sub-classes we want to optimize.

Filter pruning has a cascade effect, where pruning a prior layer reduces the input dimension of the next layer. This creates two implicit memory reductions. First, we reduce memory space from the fully connected layer(s), as they directly multiply their weights with the input. Second, the size of intermittent hidden layer(s) results is reduced due to filter dimension reduction.

4.2.2 Training the filters (Stage-II)

The second step is to train the floating-point convolutional filters using the training set collected by the IoT device. The samples consist of pre-computed outputs of the first convolution filter(s) in int-8 format. We convert each sample to a floating-point format during training per batch size. This way, we do not have to store the complete training data in floating-point. We train using the standard backpropagation algorithm with mini-batch stochastic gradient descent. We keep a small learning rate, and we need to use a small training set in this stage. The filters are initialized based on pre-trained weights, and we only need to tune the filters for the subset of classes we want to optimize.

4.2.3 Fine-tuning (Stage-III)

The final step is to quantize back the pruned convolutional filters from floating-point to int-8 to utilize optimized libraries during inference. We use the scale factors used in Stage-I in reverse order to convert them back to int-8 values for each filter. Together with the prior pruning, this step introduces some loss of information. To compensate, we perform a final fine-tuning using only the fully connected layer that we keep in floating-point for the final deployment. However, we randomly initialize the weights to find a new minimum as the network weights have shifted. The training algorithm is the same as Stage-II, with the difference that we train only the fully connected layer with the fine-tuning set, and we forward-pass through the new weights of the convolutional filters. The final network consists of the pruned and quantized convolutional filters with the newly trained fully connected layer(s).

5 Experimental Methodology

In this section, we describe the methodology of our experiments. Connecting with the application scenario in Section 3, the goal of the evaluation is to show the trade-offs of optimizing a pre-trained neural network (e.g., activity recognition with a smartwatch) to a sub-set of classes (two or three activities) and reducing the demand for resources of the IoT device. We start with software and hardware implementation, and then we present the datasets, baselines, and pre-trained network architectures.

Software & hardware setup. We use deep neural networks pre-trained in PyTorch and Python 3.8. We use the PyTorch MinMax Quantization to extract the 8-bit integer weights. We implement MiniLearn in C, and we compile our code using the Arm-GCC 9.2.1. for Cortex-M hardware. Finally, we utilize the CMSIS-NN [16] library for inference on IoT devices. We evaluate MiniLearn on nRF-52840 SoC featuring: a 32-bit ARM Cortex-M4 with FPU at 64 MHz, 256 KB of RAM, and 1 MB of Flash. We use the Nordic-Semiconductors Power Profiler Kit (PPK) shield [27] to accurately measure the power consumption.

Data sets. We use three representative datasets, one for audio recognition, one for color-image recognition, and one for human activity recognition.

Table 1. Networks used in the experiments. The table shows the shape size, computations in terms of multiply-accumulate (MAC), and the layer output in Bytes for each layer.

Pre-trained Network on KWS

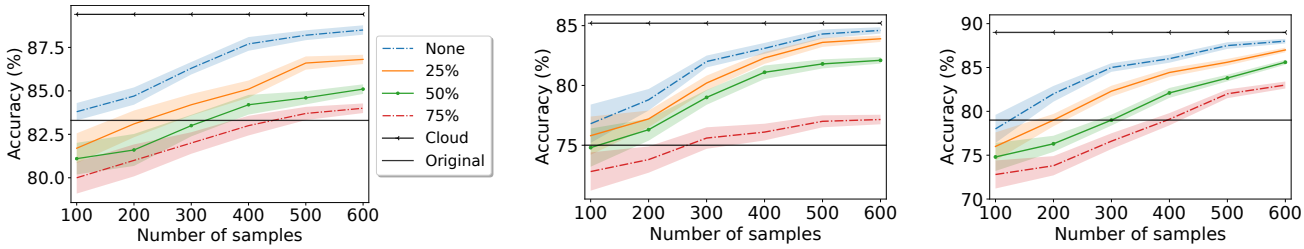
Layer	Shape	MAC	Output
Input	(62, 12, 1)	-	744
Conv1D	(58, 8, 16)	0.19M	7,424
MaxPool	(29, 8, 16)	0.06M	3,712
Conv1D	(27, 6, 32)	0.75M	5,184
MaxPool	(13, 6, 32)	0.03M	2,496
Conv1D	(11, 4, 64)	0.81M	2,816
Conv1D	(9, 2, 32)	0.33M	576
FC	(576, 35)	0.03M	35

Pre-trained Network on CIFAR-10

Layer	Shape	MAC	Output
Input	(32, 32, 3)	-	3,072
Conv2D	(30, 30, 32)	1.55M	28,800
MaxPool	(15, 15, 32)	0.07M	7,200
Conv2D	(13, 13, 32)	1.11M	5,408
MaxPool	(6, 6, 32)	0.02M	1,152
Conv2D	(4, 4, 32)	0.73M	512
FC	(512 , 10)	0.03M	10

Pre-trained Network on WISDM

Layer	Shape	MAC	Output
Input	(90, 3, 1)	-	270
Conv1D	(90, 32 ,1)	0.03	2,880
Conv1D	(90, 32 , 1)	0.03M	2,880
Conv1D	(90, 32 ,1)	0.03M	2,880
FC	(96, 128)	0.03M	12,288
FC	(128, 6)	0.03M	6



(a) Accuracy on KWS-subset. (b) Accuracy on CIFAR-subset. (c) Accuracy on WISDM-subset.

Figure 4. Average accuracy using MiniLearn with different pruning percentages on KWS 4a, CIFAR 4b, and WISDM 4c. The original baseline is the pre-trained network test on the sub-set, and *cloud* represents sending data and retraining in the cloud. The shaded area represents the standard deviation.

CIFAR-10: A widely used image classification dataset. The dataset consists of 60,000 32x32 color images, where 50,000 are for training and 10,000 for testing. There are ten classes: airplane, automobile, bird, cat, deer, dog, frog, horse, ship, and truck.

Google Keyword Spotting (KWS) [33] (v2): The dataset contains 110,000 audio samples with a total of 35 classes. Each audio sample is a keyword recorded as a speech command of a small duration. We make the following common pre-processing to create images from audio files [11]. We generate 2D images by taking the time window of the audio (timestamp) as the width and Mel-frequency cepstral coefficients (MFCC [11]) as height. The window size is 31.25 ms with a moving step at 16.125 ms. The data set is divided into 85,000 training samples and 11,000 test samples in total.

WISDM-HAR. Wireless Sensor Data Mining (WISDM) [15] is a human activity recognition data set. The data are collected and labeled by using accelerometers of Android-based cell phones. It has six classes (activities) stored as time series: 1) walking, 2) jogging, 3) walking upstairs, 4) walking downstairs, 5) sitting, 6) standing. The data type is a time series of signals needing pre-processing before use. Our experiments parse the data per 128 time-step and apply min-max normalization over the nine-axis (final dimension 128x9), leading to 7,300 training samples and 3,000 testing samples.

Class subsets. We generate subsets from the initial classification domains for our experiments as follows. For CIFAR-10, we reduce from 10 to 3 classes. For KWS, we reduce from 35 to 5 classes. For WISDM-HAR, we reduce from 6 to 3 classes. The number of classes reflects the memory constraints of the device; for more classes, we will need to extend the device's hardware capabilities. We kept data samples (100 - 600) data separate while training the baseline network and we use the separate data for retraining and fine-tuning with MiniLearn. We repeat our experiments over 30 times, randomly drawing a subset of unique (with no replacement) 3-classes for CIFAR, 3-classes for WISDM-HAR, and 5-classes for KWS subsets, respectively. The training sets on the device (MiniLearn) are used by 10% for filter training and 90% for fine-tuning across all experiments. We report the average accuracy as percent (%) over the test-set and standard deviation as $\pm$.

Baselines. We compare the accuracy of MiniLearn on IoT devices with two baselines. a) Accuracy we would have obtained if we used the **Original** pre-trained network tested on the subset classes. This baseline represents the current approach of pre-deployed neural networks on IoT devices without learning capabilities. b) Accuracy we would have obtained if we retrain and download a new model from the cloud services optimized for the 3-class and 5-class subset, respectively. This *cloud* baseline represents the ideal case where the cloud has access to previous models and as well all-new personal IoT data sent by the device, and it acts as an upper bound, as arguably, the cloud service has access to a wide range of training and optimization methods.

Pre-trained neural networks Our experiments use three pre-trained and quantized neural networks [11] reported in

Table 1. We report the shape size for each layer: Convolutional 1D & 2D, MaxPooling, and Fully Connected. We also report the Multiply-Accumulate operations (MAC) per cycle (in Millions) and the size of the layer output in Bytes.

Data & code availability. We provide the source code and evaluation data of MiniLearn in a public repository.[2]

6 Results & Discussion

This section presents the results of our MiniLearn evaluation on low-power IoT devices. We answer the following questions. (a) Is it feasible to re-train and improve existing deployed deep learning networks with on-device learning? (b) What is the performance of on-device learning applied to low-power devices? (c) What is the overhead of MiniLearn in terms of computation, memory, and energy consumption?

6.1 MiniLearn Accuracy Results

In Figure 4, we compare the average accuracy of the baselines **Cloud** and **Original** to **MiniLearn** using different percentages of filter pruning with the different number of training samples. We prune by using a 25% step for each experiment, while the architecture of **Cloud** and **Original** are unchanged. None refers to no pruning at all.

In Figure 4a, we present the accuracy of the 5-subclasses on the KWS with different pruning percentages. In Figure 4b, we present the accuracy of the 3-subclasses from the CIFAR-10, and in Figure 4c, we present the accuracy of the 3-subclasses on WISDM-HAR. We observe that without pruning, we can immediately increase the accuracy on the device and reach equivalent accuracy with the **Cloud**, by using 600 samples locally instead of retraining and downloading a new model from the Cloud. With aggressive pruning (e.g., 75%), even though we can save significant memory and energy for the device (see Subsection 6.2), the capacity of the network decreases, and with 100 samples, the accuracy is less than the original one. However, increasing the training samples to 400 and 500, respectively, even the pruned neural networks have better accuracy than the original. We observe a similar trend for all data sets in Figure 4. However, we need more training samples on the KWS date set to compensate for the loss of accuracy due to the pruning compared to the CIFAR-subset and WISDM-subset. The main reason is that KWS-subset has more classes than the other data sets.

Communication Cost. For the approach of the cloud-based baseline, we calculate the communication cost for sending training samples from the device to the cloud by utilizing short-range Low-Power Bluetooth (BLE) commonly available on low-power IoT devices. We report the transmission time using the standard throughput of 700 kbps (payload) using the default 1 Mbps mode [24]. For the energy communication cost, we report the energy required by BLE [24] using the PPKv1. by applying 3 V, which is 7.5 mA.

With the cloud baseline, in order to train a new neural network optimized for the sub-set classes, we need to upload training samples from the device to the cloud. After training and optimizing a new neural network, we need to download the model from the cloud to the device. With 600 samples, the device needs to send to the cloud a total of 446 KB personal data and download a network update of 45 KB, for the KWS sub-set. For the CIFAR sub-set, it is 1,842 KB for personal data and 96 KB for the network. For the WISDM sub-set, it is 162 KB for personal data and 18 KB for the network. Using BLE communication, it will take 2 seconds and consume 45 mJ for the WISDM sub-set, 22 seconds and consume 462 mJ for CIFAR-3, 5,6 seconds and consumes 210 mJ for KWS. Even though sending the data to the cloud and downloading a new model will consume less energy than retraining with MiniLearn (See Table 2), the difference is not significant to justify the sacrifice of user's privacy with the cloud approach.

6.2 MiniLearn Performance on IoT Devices

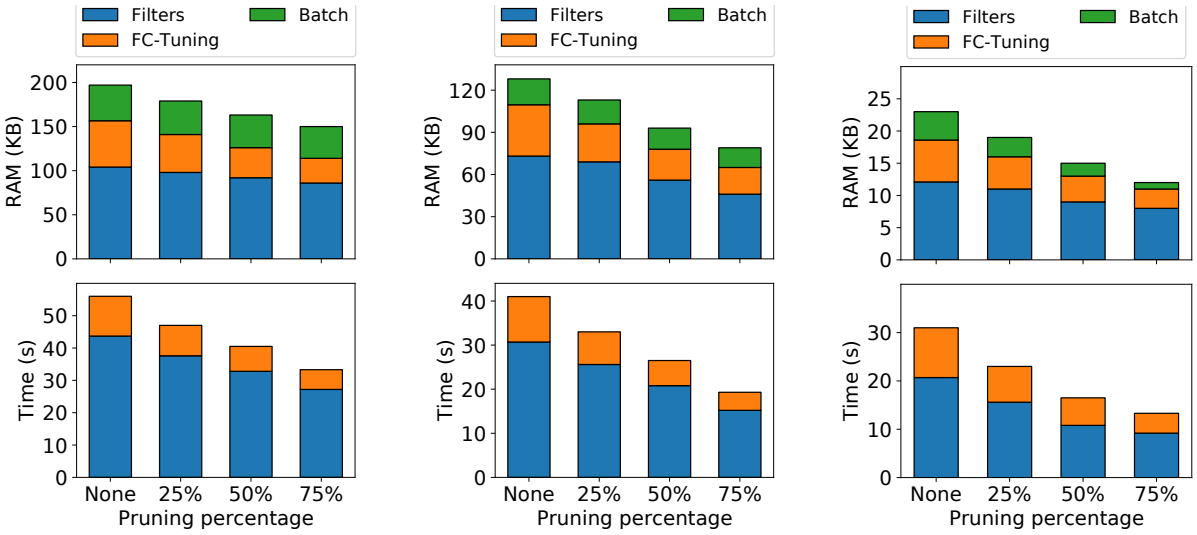
In this part, we present the performance in terms of memory, computational, and energy consumption. We repeat each experiment 30 times, and we report the following results: The memory allocation for RAM and Flash on nRF-52840 SoC. The average inference time is based on the cycle clock register. The average energy consumption, based on the average electric current draw in mA, by applying 3 V, and we report the standard variation as $\pm$, when significant.

Memory Footprint. We present two aspects of the memory consumption on the device using MiniLearn. First, we report the consumption that is needed to retrain and optimize the neural network. We use flash memory only during training to store and read the training sets. In Table 2, we see the flash memory consumption. MiniLearn needs only a small batch of data, and each epoch reads them from the flash memory. For a sample size of 100, flash consumption is 42% with KWS, 23% with CIFAR, and 6% with WISDM-HAR However, for a sample size of 600, the consumption reaches 92% with KWS, 83% with CIFAR, and 20% with WISDM-HAR.

Figure 5 illustrates the parts of the RAM being used: a) The constant-size training batch (de-quantized each epoch) is read from the flash. b) Memory for the training of the convolutional filters. For example, on KWS-subset, the memory reaches 196 KB of RAM without pruning, while on CIFAR-subset reaches 128 KB of RAM, and on WISDM-subset reaches 92 KB. Each percentage of pruning reduces the RAM consumption accordingly. c) Memory for fine-tuning the fully connected layer(s). Fine-tuning takes significantly smaller memory than the overall training. For example, without pruning, on KWS-subset, fine-tuning takes 12 KB , while on CIFAR-subset takes 10 KB, and 6 KB on WISDM-HAR-subset.

Second, we report the memory consumption we save after retraining with MiniLearn during inference. In Figure 7, we illustrate the memory consumption during inference of neural networks with different pruning optimization of the network. We illustrate the RAM consumption with the corresponding accuracy after MiniLearn using 600 samples. The original memory consumption of the KWS network is 61 KB, the CIFAR network is 23 KB, and WISDM-HAR is 12 KB. With progressive pruning (75%), the KWS network consumption is reduced to 31 KB, the CIFAR network is reduced to 11.6 KB, and the WISDM-HAR network is reduced to 6 KB, which is almost 50% less memory than the origi-

[2] available after publication



(a) MiniLearn train on KWS-subset. (b) MiniLearn train on CIFAR-subset. (c) MiniLearn train on WISDM-HAR.

Figure 5. MiniLearn performance during training in terms of time and memory consumption (using 600 samples). We apply different pruning percentages with a 25% step.

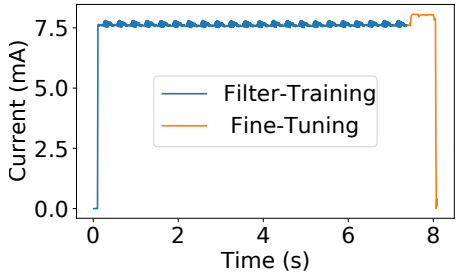


Figure 6. Electric current drawn during training on nRF-52840-DK, by applying at 3 V, during a complete training with 100 samples on CIFAR-subset.

nal, while the accuracy has slightly increased compared to the original network (see Figure 4).

Performance. We present two aspects of the device's performance. First, we report the time to retrain and optimize the neural network using MiniLearn. In Table 2, we report the performance during complete training (including fine-tuning) with different data sizes and no pruning, while in Figure 5, with different percentages of pruning using 600 samples. The training time is significantly reduced as pruning removes training parameters. The time for fine-tuning is reduced with higher degrees of pruning due to the cascade effect: the number of neurons for the fully connected layers is reduced with smaller filters. For example, on CIFAR-subset, applying 75% pruning MiniLearn takes 45% less time to retrain the network.

Second, we report the time we save after retraining the network, measuring the time to perform inference. In Figure 7, we illustrate the inference time after applying retraining with MiniLearn. The initial inference time of the KWS network (with no pruning) takes 1890 ms, the CIFAR network takes 1110 ms, and the WISDM-HAR takes 992 ms. However, with 75% pruning, the KWS network takes 945 ms, the CIFAR network takes 577 ms, and the WISDM-HAR takes 477 ms, respectively, which is almost 48% less time compared to the original.

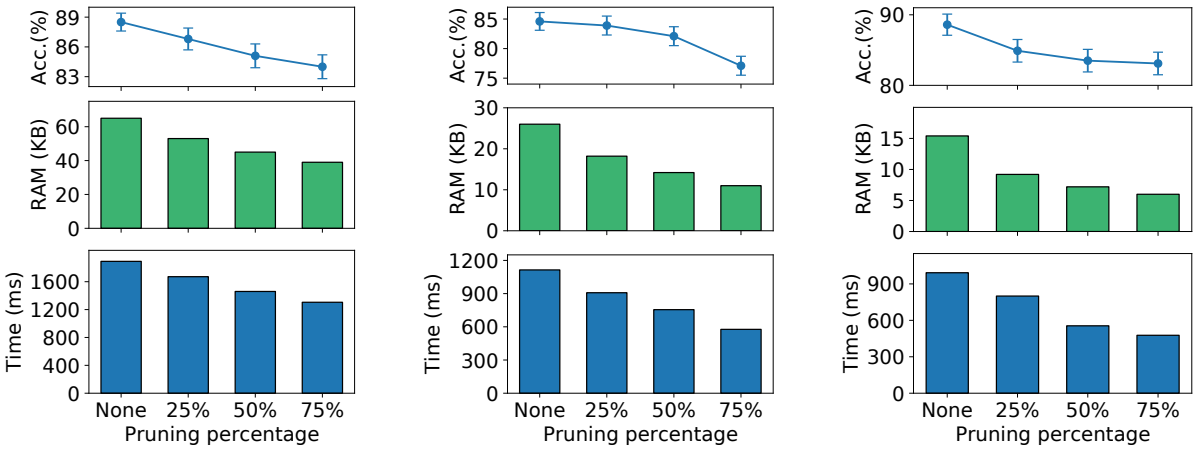
Energy Consumption. In Figure 6, we illustrate the electric current draw during training with 100 samples on CIFAR-subset. The average current is 7.6 mA, drawn mainly by the FPU unit. Compared to inference using CMSIS-NN, the average current is 8 mA drawn mainly by the DSP unit. In Table 2, we report the energy consumption of MiniLearn without pruning and data sizes from 100 to 600. Energy consumption is related to the training time. As we increase the data size from 100 to 200, the algorithm takes more time which consumes more energy. For example, to achieve an accuracy of 88.5% close to **Cloud** baseline on the KWS-subset, we need to train with 600 samples and spend 1486 mJ.

6.3 Discussion and Limitations

In this section, we provide an overview of the remaining challenges of on-device learning for low-power devices and discuss the main limitations of our approach.

Number of classes. With MiniLearn, we can reduce the number of classes on the fly for resource efficiency and personalization of the neural network on the end-user. However, due to resource constraints, we can only apply MiniLearn for a subset of the original classes. If we want to increase the number of classes or personalize the network to more classes for the end user, we need to increase the memory and energy capabilities of the IoT device.

Convolutions filters. A major trade-off we face is the number of convolutions filters we want to prune. The neural network needs to have an adequate number of filters to parse the input data and extract significant features. However, the number of filters we can train with MiniLearn is limited by



(a) Performance of inference after applying MiniLearn on KWS subset.

(b) Performance of inference after applying MiniLearn on CIFAR subset.

(c) Performance of inference after applying MiniLearn on WISDM-HAR subset.

Figure 7. Inference time and memory consumption of the neural network after applying MiniLearn with corresponding accuracy (trained on 600 samples). We apply different pruning percentages with a 25% step.

the memory of the device.

Energy. It is not necessary to apply pruning with MiniLearn if we want to increase the accuracy for a particular subset of classes. However, without pruning, MiniLearn requires an increase in energy consumption for retraining the filters. We can reduce the energy consumption for training by applying pruning, but this will reduce the accuracy, indicating a significant trade-off between energy and accuracy.

Privacy. Besides reducing the dependency on cloud services, which usually have a subscription, one of the main benefits of MiniLearn is that we keep the user's personal data on the device. If privacy and communication reliability are not such issues, the cloud will be preferable to optimize the neural network.

7 Related Work

This section lists the related work starting with previous work on quantization and pruning, which are the primary methods for compressing DNNs. Next, we present other work on-device for edge devices compared to our method for low-power devices. The related work includes edge learning and federating learning. Finally, we present other methods that do not consider learning on the device.

Quantization Previous works focused on quantization[35] assume the process takes place in the cloud, and disregard any further optimization after deployment on the device. Contrary, MiniLearn utilizes on-device learning using data available after deployment. On the other hand, training with quantized networks [3] or using so-called quantize-aware training [14], are not applicable for low-power devices as they assume powerful devices, for example, storing both the floating-point and integer representation of the network.

Pruning Pruning on convolutional filters [20] has been shown to be computationally more efficient than targeting the overall network. One of the reasons is that in large neural networks, the fully connected layer occupies most of the space, while the convolutional filters [22, 13] require repetitive computations. However, current pruning methods are part of cloud training without targeting learning opportunities with on-device learning. In contrast, MiniLearn targets low-power IoT devices for learning opportunities on existing deployed DNN without cloud interaction.

On-device learning. The majority of Machine Learning (ML) methods primarily occur in cloud services, but recently we see a shift from the classic cloud-based training and inference in the cloud towards collecting data and training on edge devices. One example is TensorFlow Lite [32] which utilizes smartphones and other edge devices for machine learning training close to the end-user. However, other proposals exist for on-device and online learning in the literature. To start with, incremental learning [1], transfer learning [25], and few-shot learning [31] are some alternatives. With these methods, the existing models may retrain and add new classes by using a small number of training samples. TinyOL [29] utilizes online learning using encoder-decoders. However, TinyOL never reduces the size of the network and uses mean square error (MSE) as a loss function, commonly used for regression problems. MiniLearn proposes an architecture for on-device training of convolutional networks using the cross-entropy loss function suitable for classification problems, also reducing the size of the network.

Federated learning. A distributed version of on-device learning is Federated Learning (FL). With FL, multiple devices participate in training a global model. Each device uses its own datasets and needs to exchange only the training parameters with each other, for example, in FedHome [34] and FedHealth [6]. In order to ensure the privacy of the users, federated learning needs to make heavy use of cryptographic protocols such as Secure Multiparty Computation (SMC) [4] and Homomorphic Encryption (HE) [12]. Currently, federated learning is available only on edge devices. MiniLearn complements federating learning by bringing parts of training on low-power devices. Federated learning can utilize

Table 2. MiniLearn training evaluation, without pruning, on CIFAR and KWS subsets. We report the RAM and Flash memory footprints, the average training time, and average energy consumption during the complete training (including fine-tuning).

MiniLearn on KWS-subset.

Data-Size	RAM (KB)	Flash (KB)	Time (s)	Energy (mJ)	Accuracy (%)
100	196	439	10±0.2	254±0.2	83.8±0.3
200	196	580	16±0.1	406±0.1	84.7±0.2
300	196	620	25±0.1	635±0.1	86.3±0.2
400	196	761	36±0.1	914±0.1	87.7±0.2
500	196	840	47±0.1	1194±0.1	88.2±0.1
600	196	950	56±0.1	1486±0.1	88.5±0.1

MiniLearn on CIFAR-subset.

Data-Size	RAM (KB)	Flash (KB)	Time (s)	Energy (mJ)	Accuracy (%)
100	128	240	8±0.2	203±0.2	76.8±0.5
200	128	370	16±0.2	406±0.2	78.8±0.3
300	128	510	23±0.2	584±0.2	82.1±0.2
400	128	660	30±0.2	762±0.2	83.1±0.2
500	128	790	35±0.2	890±0.2	84.3±0.1
600	128	857	40±0.2	1016±0.2	84.6±0.1

MiniLearn on WISDM-HAR subset.

Data-Size	RAM (KB)	Flash (KB)	Time (s)	Energy (mJ)	Accuracy (%)
100	92	64	6±0.1	138±0.2	78.1±0.3
200	92	91	10±0.1	228±0.1	82.1±0.2
300	92	118	14±0.1	319±0.1	85.3±0.2
400	92	145	21±0.1	479±0.1	87.7±0.2
500	92	172	25±0.2	670±0.1	88.5±0.1
600	92	201	30±0.3	786±0.1	89.5±0.1

MiniLearn after creating a global model to personalize the model for the end-user using her device only.

Multi-task Learning. Another method to alter classes on the fly without retraining the complete neural network is multi-task learning [19]. Another concept to optimize multi-task learning is the virtualization of weight parameters to share weights of a neural network across multiple networks in the device[17]. However, low-power IoT devices have extreme memory constraints, and in order to benefit, they need to be equipped with a RAM capacity of MBs, compared to KBs of RAM typically found on low-power devices. In contrast, MiniLearn has an immediate optimization on a low-power device with memory and computational constraints.

Network architecture search. Another method typically used by cloud services in network architecture search NAS [26, 2]. With NAS, the designing and training of optimized neural networks are automated for the specific hardware and application domain. NAS tries to search the space of possible DNN architecture with a search strategy and performance goal for the target hardware or application. NAS provides a static optimization on an estimation of the device's resource utilization, while MiniLearn compliments NAS in that it can dynamically address changes in computational, memory, and energy requirements.

Offloading. Methods that do not utilizing re-training of DNNs focus on partitions of networks between a IoT and edge minimize execution latency [5], but increase the communication cost. Similarly, distributed inference hierarchies can offload inputs between cloud, edge, and IoT devices [30]. Adapting these methods on the fly for the IoT device will need several updates with high communication costs. Other frameworks [10] use offloading for forward propagation in DNNs with a focus on battery energy optimization for mobile devices. MiniLearn differs by focusing on learning opportunities on the device and avoiding cloud interaction.

8 Conclusion

As deep neural networks become available on-body sensors and smartwatches, on-device learning opportunities become relevant. Currently, conventional deep learning algorithms rely on IoT devices to send an enormous amount of data to the cloud leading to three main issues. First, IoT devices depend on energy-efficient communication with limited reliability and network bandwidth, and they may skip transmitting essential training samples to the cloud. Second, cloud storage of extensive IoT data raises significant privacy concerns. Third, IoT devices operate in open environments, and the initial requirements for utilizing deep neural networks may change. This paper proposes MiniLearn to retrain and improve pre-trained neural networks on resource-constrained IoT devices. We show that MiniLearn can improve the accuracy of a subset of classes using local data, and it can reduce the memory and inference latency of the initial network using filter pruning and fine-tuning. In detail, it can take up to 48% less inference time and up to 50% less memory, increasing at the same time the accuracy for a classification sub-set by 3% to 9%.

9 Acknowledgments

This work was supported by the Swedish Research Council (VR) through the project "AgreeOnIT", the Swedish Civil Contingencies Agency (MSB) through the projects "RICS2" and "RIOT", and the Vinnova-funded project "KIDSAM".

10 References

[1] J. Bai, A. Yuan, Z. Xiao, H. Zhou, D. Wang, H. Jiang, and L. Jiao. Class incremental learning with few-shots based on linear programming for hyperspectral image classification. *IEEE Transactions on Cybernetics*, 2020.

[2] H. Cai, T. Chen, W. Zhang, Y. Yu, and J. Wang. Efficient architecture search by network transformation. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 32, 2018.

[3] Z. Cai, X. He, J. Sun, and N. Vasconcelos. Deep learning with low precision by half-wave gaussian quantization. In *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2017.

[4] N. Chandran, D. Gupta, A. Rastogi, R. Sharma, and S. Tripathi. Ezpc: Programmable and efficient secure two-party computation for machine learning. In *2019 IEEE European Symposium on Security and Privacy (EuroS P)*, pages 496–511, 2019.

[5] X. Chen, M. Li, H. Zhong, Y. Ma, and C.-H. Hsu. Dnnoff: Offloading dnn-based intelligent iot applications in mobile edge computing. *IEEE Transactions on Industrial Informatics*, 18(4):2820–2829, 2022.

[6] Y. Chen, X. Qin, J. Wang, C. Yu, and W. Gao. Fedhealth: A federated transfer learning framework for wearable healthcare. *IEEE Intelligent Systems*, 35(4):83–93, 2020.

[7] H. Chung, M. Iorga, J. Voas, and S. Lee. "alexa, can i trust you?". *IEEE Computer Journal*, 50(9):100–104, 2017.

[8] R. David, J. Duke, A. Jain, V. J. Reddi, N. Jeffries, J. Li, N. Kreeger, I. Nappier, M. Natraj, S. Regev, R. Rhodes, T. Wang, and P. Warden. Tensorflow lite micro: Embedded machine learning on tinyml systems. *arXiv:2010.08678*, 2021.

[9] Z. Dong, Z. Yao, A. Gholami, M. W. Mahoney, and K. Keutzer. Hawq: Hessian aware quantization of neural networks with mixed-precision. In *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*, October 2019.

[10] A. E. Eshratifar and M. Pedram. Energy and performance efficient computation offloading for deep neural networks in a mobile cloud computing environment. In *Proceedings of the 2018 on Great Lakes Symposium on VLSI*, GLSVLSI '18, page 111–116, New York, NY, USA, 2018. Association for Computing Machinery.

[11] S. K. Gouda, S. Kanetkar, D. Harrison, and M. K. Warmuth. Speech recognition: Keyword spotting through image recognition. *arXiv:1803.03759*, 2020.

[12] T. Graepel, K. Lauter, and M. Naehrig. Ml confidential: Machine learning on encrypted data. In T. Kwon, M.-K. Lee, and D. Kwon, editors, *Information Security and Cryptology – ICISC 2012*, pages 1–21, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[13] S. Han, H. Mao, and W. J. Dally. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *International Conference on Learning Representations (ICLR)*, 2016.

[14] B. Jacob, S. Kligys, B. Chen, M. Zhu, M. Tang, A. Howard, H. Adam, and D. Kalenichenko. Quantization and training of neural networks for efficient integer-arithmetic-only inference. In *IEEE Conference on Computer Vision and Pattern Recognition*, CVPR '18, 2018.

[15] J. R. Kwapisz, G. M. Weiss, and S. A. Moore. Activity recognition using cell phone accelerometers. In *Proceedings of the Fourth International Workshop on Knowledge Discovery from Sensor Data*, pages 10–18, 2010.

[16] L. Lai, N. Suda, and V. Chandra. Cmsis-nn: Efficient neural network kernels for arm cortex-m cpus. *arXiv:1801.06601*, 2018.

[17] S. Lee and S. Nirjon. Fast and scalable in-memory deep multitask learning via neural weight virtualization. In *ACM International Conference on Mobile Systems, Applications, and Services*, MobiSys '20. Association for Computing Machinery, 2020.

[18] S.-M. Lee, S. M. Yoon, and H. Cho. Human activity recognition from accelerometer data using convolutional neural network. In *IEEE International Conference on Big Data and Smart Computing*, BigComp '17, 2017.

[19] S.-G. Leem, I.-C. Yoo, and D. Yook. Multitask learning of deep neural network-based keyword spotting for iot devices. *IEEE Transactions on Consumer Electronics*, 65(2):188–194, 2019.

[20] H. Li, A. Kadav, I. Durdanovic, H. Samet, and H. P. Graf. Pruning filters for efficient convnets. *arXiv:1608.08710*, 2016.

[21] J. Lin, W. Chen, Y. Lin, J. Cohn, C. Gan, and S. Han. Mcunet: Tiny deep learning on iot devices. In *Advances in Neural Information Processing Systems, NeurIPS*, NeurIPS '20, 2020.

[22] S. Lin, R. Ji, C. Yan, B. Zhang, L. Cao, Q. Ye, F. Huang, and D. Doermann. Towards optimal structured cnn pruning via generative adversarial learning. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2019.

[23] C. Liu, B. Zoph, M. Neumann, J. Shlens, W. Hua, L.-J. Li, L. Fei-Fei, A. Yuille, J. Huang, and K. Murphy. Progressive neural architecture search. In *Proceedings of the European Conference on Computer Vision (ECCV)*, September 2018.

[24] Nordic Semiconductor technical documentation, 2019. https://infocenter.nordicsemi.com/.

[25] S. J. Pan, J. T. Kwok, Q. Yang, et al. Transfer learning via dimensionality reduction. In *Proceedings of the AAAI Conference on Artificial Intelligence*, 2008.

[26] H. Pham, M. Guan, B. Zoph, Q. Le, and J. Dean. Efficient neural architecture search via parameters sharing. In *Proceedings of the 35th International Conference on Machine Learning*. PMLR, 10–15 Jul 2018.

[27] Nordic Semiconductorpower profiler kit, 2019. https://www .nordicsemi.com/.

[28] C. Profentzas, M. Almgren, and O. Landsiedel. Performance of deep neural networks on low-power iot devices. In *ACM CPS-IoTBench '21*, 2021.

[29] H. Ren, D. Anicic, and T. Runkler. Tinyol: Tinyml with online-learning on microcontrollers. *arXiv:2103.08295*, 2021.

[30] J. Ren, H. Wang, T. Hou, S. Zheng, and C. Tang. Federated learning-based computation offloading optimization in edge computing-supported internet of things. *IEEE Access*, 7:69194–69201, 2019.

[31] F. Sung, Y. Yang, L. Zhang, T. Xiang, P. H. Torr, and T. M. Hospedales. Learning to compare: Relation network for few-shot learning. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2018.

[32] Tensorflow methods, 2021. https://www.tensorflow.org/.

[33] P. Warden. Speech Commands: A Dataset for Limited-Vocabulary Speech Recognition. *arXiv:1804.03209*, 2018.

[34] Q. Wu, X. Chen, Z. Zhou, and J. Zhang. Fedhome: Cloud-edge based personalized federated learning for in-home health monitoring. *IEEE Transactions on Mobile Computing*, pages 1–1, 2020.

[35] R. Zhao, Y. Hu, J. Dotzel, C. De Sa, and Z. Zhang. Improving neural network quantization without retraining using outlier channel splitting. In *International Conference on Machine Learning*, ICML '17, 2019.