An Overview of On-Device Deep-Neural-Network Training and Inference

Zhaojun Ni
School of Information Science and Technology
ShanghaiTech University
Shanghai,China
nizhj2022@shanghaitech.edu.cn

***Abstract*—Deep Neural Networks (DNN) have been widely used in various artificial intelligence (AI) applications due to their overwhelming performance. On-device training and inference enable the DNN model to adapt to new data collected from the sensors by fine-tuning a pre-trained model. Users can benefit from customized models without having to transfer the data to the cloud, protecting the privacy. However, the training memory consumption is prohibitive for IoT devices that have tiny memory resources. Therefore, analysis and optimization techniques targeting on-device DNN are required. This article aims to provide an overview of DNN processing that enables on-device training and inference. Specifically, it will provide hardware optimization techniques to overcomes the design challenges in terms of neural distinct dataflow, external memory access, and computation. In addition, this paper summarizes key schemes of recent efficient AI systems for edge devices. Moreover, we will also show a design example of on-device DNN with efficient optimization techniques.**

Keywords—deep neural network (DNN), DNN training and inference, on-device, machine learning (ML)

I. Introduction

Deep Neural Network (DNN) [1] has been widely studied in all technical fields due to its superior accuracy in computer vision, natural language processing (NLP), autonomous systems and other applications. In addition, with the explosive growth of hardware, algorithms and datasets, DNN provides high-performance artificial intelligence (AI) for users in real life.

DNN process has two stages: training and inference. On-device training allows users to adapt pretrained models to newly collected sensory data after deployment. Through local training and adjustment at the edge, the model can learn to improve prediction, and perform lifelong learning and user customization, as shown in Fig.1. For example, fine-tuning language models can be learned from users' typing and writing. Adjust the visual model so that new objects can be recognized from the mobile camera. By bringing training closer to sensors, it also helps protect user privacy when handling sensitive data.

However, on-device training on micro edge devices is extremely challenging, which is fundamentally different from cloud training. Micro IoT devices (e.g. microcontrollers) usually have a limited SRAM size, such as 256KB. Such a small memory budget is hardly enough to infer the deep learning model, let alone training [2]. It processes three different steps to find high-precision model parameters, including a large number of operations, external memory access and various data flows. Moreover, DNN training has distinct characteristics from inference, which limits the application of optimization techniques used in reasoning to training.

In this article, we discuss optimization techniques for on-device DNN training and inference. In particular, we analyze three design challenges for energy-efficient DNN: dataflow, external memory access, and computation [3]. First, the distinct dataflow of three DNN training steps are analyzed from the perspective of operational patterns and memory access patterns. Second, we break down external memory access and categorize compression methods for intermediate data that takes up the most significant amount. Third, computational optimization opportunities are summarized in terms of bit precision and sparsity. Moreover, we introduce the optimization techniques of recent DNN training hardware and describe an example of an efficient on-device DNN training and inference design. This paper is organized as follows. Section II describes hardware design challenges for on-device DNN. Section III introduces the corresponding key optimization techniques. Section IV shows a design example of an energy-efficient DNN training. Section V summarizes efficient AI systems for edge devices. Finally, Section VI provides the key takeaways in conclusion.

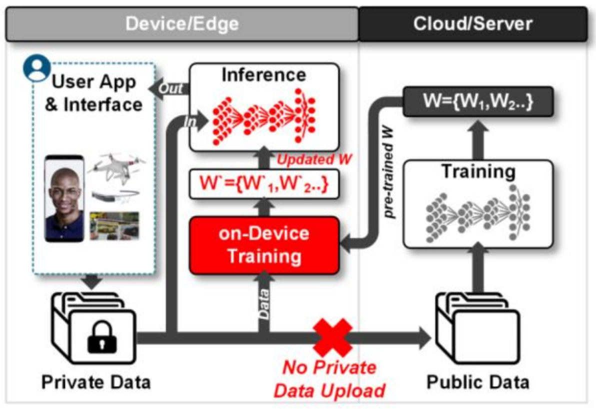


Fig.1. On-device DNN

II. Design Challenges for On-Device DNN

A. Dataflow

DNN training includes three steps, forward propagation (FP), backward propagation (BP) and weight gradient update (WG). First, FP uses small batches of given input data to generate DNN results. Secondly, BP calculates the error between the label and the FP result and transfers it to the previous layer. This is achieved by multiplying the weight of each layer by the error of the next layer. Then the calculated error is propagated to the previous layer. Third, weight gradient update (WG) is the process of updating the weight of each layer to the direction of minimizing the error. This process is performed by using the features and errors of each layer. The DNN training process repeats these three steps until the DNN accuracy converges, which leads to many calculations and memory accesses.

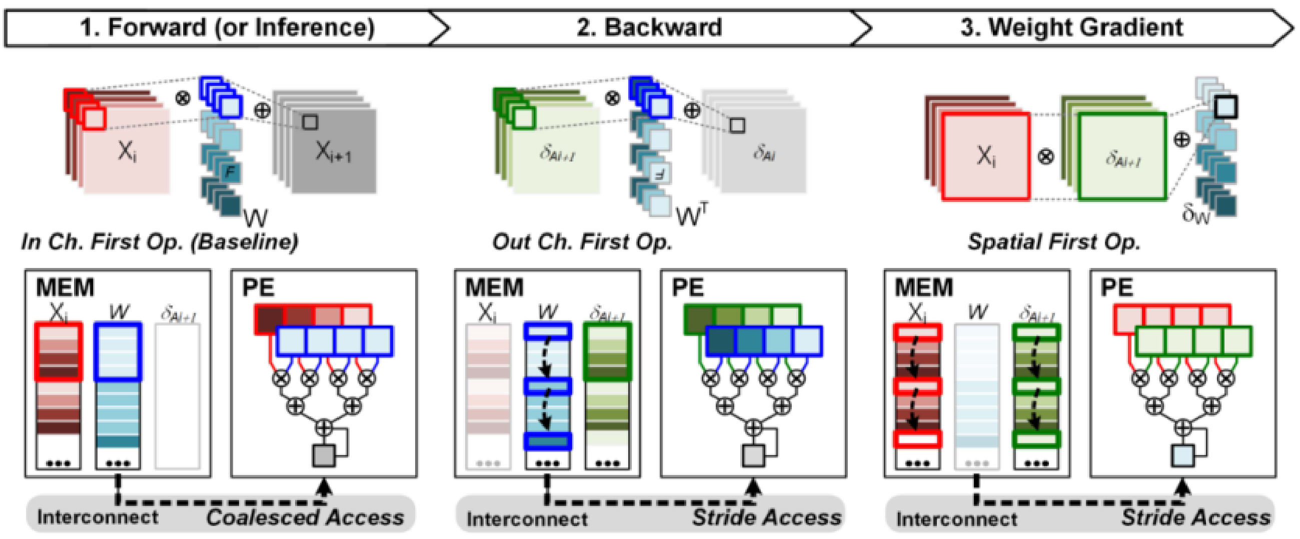


Fig.2. Dataflow of three DNN training steps and mapping to DNN device

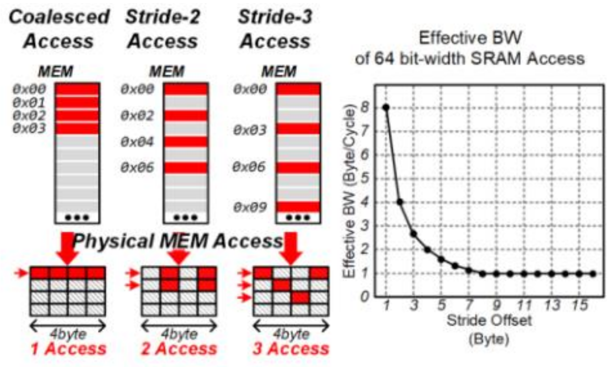


Fig.3. Effective bandwidth of different memory access patterns.

The operation of each training step is usually based on convolution or matrix multiplication, but each step has a different data flow. During the FP step, it convolves each input channel of the input feature (X) with the weight (W). The results are then added to produce the channels of the output feature (X). For the BP step, a channel of output gradients (δ) is convolved with rotated weights. Then, the output channel results are added to obtain the input gradient (δ). The channel of. BP is similar to FP except that the W tensor rotates in the channel and transposes between channels. During the WG step, it convolves the channel of the input feature with the channel of the output gradient to generate a weight gradient. Unlike FP and BP, there is no inter channel accumulation.

The operations for each training step are commonly based on convolution or matrix multiplication operations, but each step has a distinct dataflow. During the FP step, it convolves each input channel of input features (X) with weights (W). After then, the results are added to produce a channel of output features (X). For the BP step, a channel of output gradients (δ) is convolved with rotated weights. Then, the output channel results are added to obtain a channel of input gradient (δ). The BP is similar to the FP, except that the W tensor is intra-channel-wise rotated and inter-channel-wise transposed. During the WG step, it convolves a channel of input features with a channel of output gradients to produce a weight gradient. Unlike the FP and the BP, there is no channel-to-channel accumulation.

The three steps of DNN training share a data array as the operand, but have different dataflow. This also results in different memory access patterns for the same data array. For example, Fig.2 shows the dataflow of the DNN training step in PE. The PE has an output stationary data-path, and the memory layout is optimized only for FP. In order to maximize the output reuse in the FP step, the input channels of input features and weights must be loaded first. Therefore, for effective joint access, the input channel of the input feature and weight tensor is set to assign the input channel to the innermost dimension of adjacent addresses.

Weights and input features that have been stored using FP optimized memory layout can also be accessed in different patterns in BP and WG steps. In the BP step, because the transposed weight and output gradient are used to calculate the input gradient, the output channel of the weight must be fed first for output reuse. However, the innermost dimension of the weight array has been set as the input channel, so the output channel of the weight is located at the interval leading to step access. In the WG step, stride access is also caused by loading input features and output gradients, as shown in Fig2. Input features and output gradients must first be loaded spatially to maximize output reuse, but adjacent pixels in the same channel are far from each other in the memory address space.

Fig.3 shows the effective bandwidth based on the memory access pattern. Accessing the data array from consecutive addresses allows the use of all data in one row of the physical memory array, maximizing effective bandwidth and energy efficiency. However, discontinuous address access, such as step access can lead to bandwidth degradation and redundant memory array access. For example, to access 8×16 bit data, requiring only two memory accesses and data transfers. However, if allocated at an address interval of more than 8 bytes, 8 memory accesses and data transfers are required. In addition, since burst access is not possible when external memory is used, access to data arrays with discontinuous addresses leads to increased access latency. Therefore, the memory layout and data path must be determined by considering different dataflows to achieve efficient processing.

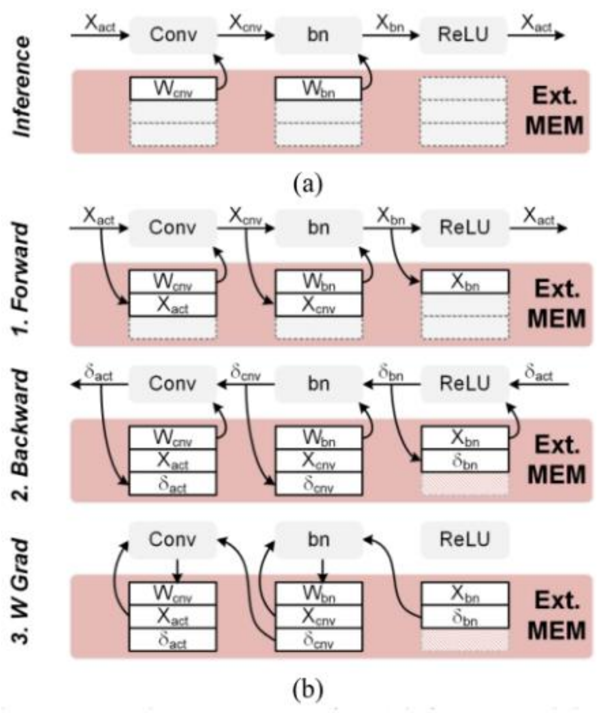


Fig.4. External memory usage for (a) inference, and (b) training.

B. External memory access

For on-device DNN training, reducing external memory access is one of the most important design challenges. In addition, the limited memory capacity makes DNN training more challenging. Many researches focus on reducing model parameters, such as weight quantization and pruning [4]. However, in DNN training, different methods from DNN inference are needed to reduce external memory access. This is because the external memory access pattern is different from inference. Fig.4 shows the difference between the DNN inference and the external memory access mode of the most commonly used CNN structure (Conv-BN-ReLU) in training.

In DNN inference, input features and weights are loaded from external memory to calculate output features. Then, the output features are normalized with the parameters of BN layer [5] and transferred to the next layer after ReLU activation. Once each layer operation is completed, the intermediate output features are no longer used, so there is no need to allocate and reserve memory space for the output features. In addition, BN and ReLU are element-wise operations in inference, so they can be combined with the convolution layer to eliminate external memory access of intermediate features of BN and ReLU layers.

Unlike the inference, BP and WG steps use intermediate features generated in FP for training. Therefore, memory space must be allocated for a batch of intermediate functions, resulting in a large number of memory accesses and space consumption. In addition, the mean and standard deviation of the output characteristics must be calculated for BN. Therefore, feature load/store accounts for the largest external memory access. The proportion of feature memory use in training is higher than that in inference. Therefore, it is necessary to reduce the size of intermediate features in training.

C. Computation

DNN training iterates FP, BP, and WG steps to optimize DNN parameters, resulting in many operations. Since each step contributes similarly to the total number of operations, the device must effectively calculate all three steps. Many works have explored sparsity [4,6-10] and bit precision to achieve efficient operation. However, due to the different computational characteristics of training and inference, there are limitations. Most of the work focuses on optimizing bit-precisions by using low bit width fixed points. However, it cannot cover the wide dynamic gradient range during the BP step.

In order to cover the gradient with wide dynamic range with standard bit-precision, at least IEEE half-precision is required. Otherwise, custom bit-precision is needed. When designing custom bit-precision, it is very important to reduce bit-width and determine the data format with limited bit-width. In addition, since the dynamic range and resolution depend on the data format, it is necessary to determine an effective format without losing training accuracy.

Sparsity allows the device to reduce the calculation pressure without loss of accuracy. However, DNN training and inference have different sparsity characteristics. Therefore, sparse sensing devices for inference cannot be used for training. For inference, algorithms such as weight pruning and quantization generate a large number of zeros in weights. In contrast, it is difficult to artificially create zero in the weights for DNN training, because the weights are updated every iteration. Although there is zero in the input feature of ReLU in FP and WG steps, there is no zero in BP step, which requires a lot of calculation. Therefore, it is necessary to explore the sparsity processing technology of DNN training.

III. On-device DNN Optimization Methods

There are several challenges to design an on-device DNN, because of distinct dataflow, large memory access and usage, and energy-consuming operators. What we discuss below can achieve better practical effect by applying the following methods to the real microcontrollers.

A. Dataflow Optimizations

In DNN training, the FP, BP, and WG have distinct dataflow. However, a tensor stored in the external memory are used for the three distinct dataflow, and this makes it difficult to optimize the operations for all three DNN training steps. Several methods can help to address this problem, and they can be divided into two categories: array re-arrangement and data-path configuration.

1) Array Re-arrangement

Fig.5 shows an example of transposed MatMul with data flow optimization technology. In Fig.5, a row of weight matrix is the most internal dimension of MatMul, which optimizes the parallel architecture of fixed output and input channels. However, the weight matrix with the innermost dimension of one row leads to inefficient step access and reduced PE utilization during the transpose of MatMul. First, memory re-arrangement can be applied to improve effective memory bandwidth and PE utilization. Before feeding to PE, memory re-arrangement changes the array structure to an optimized memory layout. In the example in Fig.8, the weight matrix can be accessed as a merge by re-arranging the

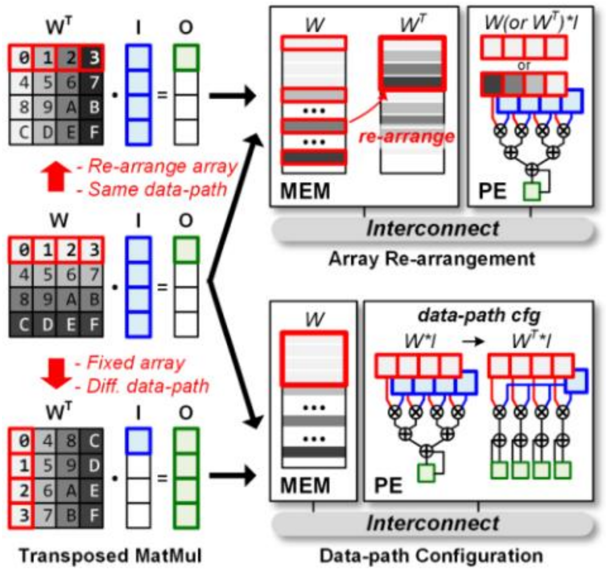


Fig.5. Transposed matrix multiplication with array re-arrangement and data-path configuration.

innermost dimensions of the weight array from a row to a column in advance to start MatMul. Memory re-arrangement is easy to achieve through DMA and software control. In addition, for small tensors, the memory re-arrangement delay can be hidden in the computation delay. However, it requires additional memory size and redundant memory access. In addition, for large tensors, array re-arrangement takes a long time, which reduces the overall performance.

2) Data-path Configuration

A configurable data-path can also optimize the different dataflow of DNN training steps. The weights in a column of the transposed matrix are loaded first for the transposed MatMul without array re-arrangement. Because the weights in a column cannot be accumulated, PE utilization is severely degraded when running on the channel parallel and output stationary PE architecture. However, the input stationary data-path allows high PE utilization thanks to each weight's independent output accumulation. As such, high performance can be maintained by the configuration of data-path without changing the memory layout.

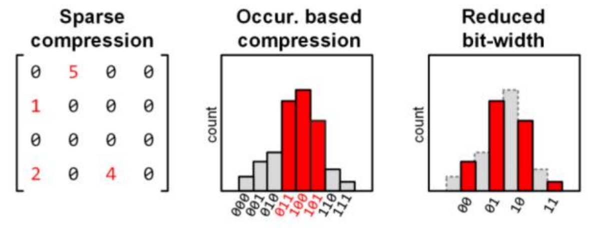
B. External Memory Access And Footprint Reduction

During DNN training, the intermediate features which account for the largest portion of memory access have characteristics of sparsity, locality, and loss robustness. The natures of the intermediate features allow the tensor to be compressed reducing memory access and footprint. Fig.6 shows three types of compression methods for DNN tensors: sparse compression, probability of occurrence based compression, and reduced bit-width.

1) Sparse Compression

ReLU, a derivative of ReLU, and a derivative of max-pooling cause redundant zeros in the intermediate features. Sparse compression that represents the tensors into a non-zero vector and an index vector reduce the byte size of data without any loss. There are several ways to present the index vectors.

- Zero-value compression (ZVC): The index vector for ZVC is a binary mask that has the same number of elements as the tensor. The index vector indicates whether the element is zero or not.
- Compressed sparse row (CSR): two index vectors, rows and column indices.
- Run-length encoding (RLE): The index vector of RLE represents a number of consecutive zeros between non-zero values.



	compressed vector
data	[5 1 2 4]
cidx	[1 0 0 2]
ridx	[0 1 3 3]

data	*p(%)*	code
011	35	01
100	32	10
101	31	11
000	0.5	00000
001	1.1	00001
...	...	...

data	reduced
000	00
001	01
010	
011	10
100	
...	...

Fig.6. Compression methods for the DNN tensor.

In addition, intermediate features can be compressed more efficiently using the locality characteristics. Because of the locality characteristic, zeros are concentrated. So, [11-13] utilized this characteristic and reduced the index size of sparse compression by hierarchically compressing features. References [11-13] divided the features into grids and compressed them in grid-level first. After that, only non-zero grids are compressed at a feature level. The hierarchical sparse compression can be effectively applied to training to reduce intermediate features which take most external memory access.

2) Probability of Occurrence based Compression

The probability of occurrence-based compression reduces data size by encoding more likely to occur values into low bit-width code. In the example of Fig. 6, only three values, 3b011, 3b100 and 3b101, account for 98% of the total data. And, encoding three values into 2bits and the remaining five values into 5bits can reduce the data size by 31%. There are several works that use the probability of occurrence-based compression to reduce the features. Reference [14] uses bfloat which allocates 8bit for an exponent value, and replaced 3 most-frequent exponent values with 2bit code. Although the bit-width of less frequent values is increased, the overall byte size of the intermediate feature size is decreased. Reference [4,15] adopts Huffman coding, which encodes high likely to occur values into low bits.

3) Reduced Bit-width

Due to the loss robustness, reduced bit-width can be used for DNN training and inference. Depending on the purpose of the stored data, the bit-width can be reduced without loss. For example, [17] uses 1 bit for ReLU-Pool layer pairs. Because in the BP, ReLU only needs whether the feature value is positive or not. Further from applying each three compression methods to DNN data, the compression methods can be combined to minimize the external memory access and footprint. In more detail, [15] quantizes the intermediate features to 8bit in the frequency domain, producing lots of zeros. Then, the zeros are eliminated by

RLE. Moreover, Huffman encoding is used to reduce the bit-width of frequent data at the end of the JPEG encoding.

C. Computation

1) Sparsity Handling

The zero input MAC operations that do not affect output can be eliminated leading energy efficiency improvement, and native zeros can be skipped. However, there is no zero in the BP which accounts for a large portion of the total number of operations. The derivative of ReLU is determined by the features calculated in the FP step. Therefore, regardless of the back-propagated gradients in the BP step, it is possible to know in advance whether the input gradient of the ReLU is zero. Thus, unnecessary convolution operations that results are zeros can be pre-detected by merging ReLU of the pre-layer, and eliminated with the help of output zero-skipping logics.

The algorithm and hardware co-design allows to produce more zeros for efficiency, but maintain the training accuracy. Reference [17] proposed a method of determining the weight to be pruned in the training phase and skipping all related operations. Reference [18] focuses on a partial sum of small values which does not significantly affect accumulation results, and proposed a way to skip the accumulation operations for small partial sum. Reference [19] proposed an algorithm that enables pruning in an earlier stage of DNN training, and enhanced performance with weight skipping.

2) Quantization

Most studies on bit-precision have focused on covering the wide data distribution within reduced bit-width such as 8bit and 16bit. A floating-point with custom bits assignment is most commonly adopted. Compared to conventional floating-point formats such as half-precision and mini-precision, more bits are assigned to exponent value expending dynamic range. In addition, there are studies to use fixed-point to take advantage of high energy-efficient hardware. Reference [18] tunes integer-fraction bits depending on data statistics to cover a wide dynamic range with minimum data loss.

There is a limit to reducing bit width by uniformly optimizing bit precision for all DNN training operations. To overcome the limitation, each DNN training data characteristic should be analyzed, and bit-precision should be optimized for each data depending on their characteristic. The bit-precision can be optimized at each training step, layer, or even neuron to minimize data loss. Reference [20-22] used a different bit-precision for each step of training. The weight-gradient, which has a relatively wide distribution and requires high precision, is calculated with fp32, and in FP and BP, fp16 is used to improve energy-efficiency. So, they store weight in high precision, fp32, in external memory, and quantize weight to fp16 for FP and BP steps. In particular, [21] assigned the same exponent bit-width as fp32, reducing the conversion cost between fp16 and fp32. Reference [23] optimizes bit-precision for FP and BP steps respectively to enable training with 8 bits. Data for FP step with a relatively narrow distribution is represented by 4bit-exponent and 3bit-mantissa, and in BP step, expended exponent bit-width format, 5bit-exponent and 2bit-mantissa,is used for wider dynamic range. References [18,23-24] perform quantization based on each layer's statistical analysis to minimize data loss. Reference [23] adjusts exponent bias of fp8 based on overflow and underutilization monitoring.

IV. ON-DEVICE DNN DESIGN EXAMPLE: MCUNET

Many optimization techniques for on-device DNN training and inference have been proposed to solve the design challenges. However, the dependences of optimization techniques have to be considered. In this Section, a design example of on-device DNN, MCUNet [25] will be introduced.

A. Efficient Neural Network Design

Network efficiency is very important for the overall performance of the deep learning system and neural architecture search (NAS) dominates efficient network design. The performance of NAS highly depends on the quality of the search space. TinyNAS [26] is a two-stage neural architecture search method that first optimizes the search space to fit the tiny and diverse resource constraints, and then performs neural architecture search within the optimized space. With an optimized space, it significantly improves the accuracy of the final model. It proposes to optimize the search space automatically at low cost by analyzing the computation distribution of the satisfying models. To fit the tiny and diverse resource constraints of different microcontrollers, the input resolution and the width multiplier of the mobile search space are also scaled.

B. Optimizing Real Quantized Graphs

Unlike fine-tuning floating-point model on the cloud, training with a real quantized graph is difficult: the quantized graph has tensors of different bit-precisions and lacks BN layers (fused), leading to unstable gradient update.

Consider a weight matrix of a linear layer $\mathrm{W} \in \mathrm{R}^{c1\times c2}$, where c1, c2 are the input and output channel. To perform per-tensor quantization, a scaling rate $s_\mathrm{W} \in \mathrm{R}$ is produced, such that $\bar{\mathrm{W}}$'s largest magnitude is $2^7 - 1 = 127$:

$$\mathrm{W} = s_\mathrm{W} \cdot \left(\frac{\mathrm{W}}{s_\mathrm{W}}\right) \approx s_\mathrm{W} \cdot \bar{\mathrm{W}},\ \mathrm{G}_{\bar{\mathrm{W}}} \approx s_\mathrm{W} \cdot \mathrm{G}_\mathrm{W}, \qquad (1)$$

When optimizing a quantized graph, the accuracy is lower compared to the floating-point counterpart. The quantization process distorts the gradient update and the ratio between weight norm and gradient norm for each tensor can verify the idea.

$$\frac{\left\|\bar{\mathrm{W}}\right\|}{\left\|\mathrm{G}_{\bar{\mathrm{W}}}\right\|} \approx \frac{\left\|\frac{\mathrm{W}}{s_\mathrm{W}}\right\|}{\|s_\mathrm{W} \cdot \mathrm{G}_\mathrm{W}\|} = s_\mathrm{W}^{-2}\|\mathrm{W}\|/\|\mathrm{G}\|. \qquad (2)$$

The ratio is much larger, which could be addressed by adjusting the learning rate, and has a different pattern after quantization. To solve the issue, Quantization-Aware Scaling (QAS) [24] is proposed by compensating the gradient of the quantized graph according to (3).

$$\bar{\mathrm{G}}_{\bar{\mathrm{W}}} = \mathrm{G}_{\bar{\mathrm{W}}} \cdot s_\mathrm{W}^{-2},\ \bar{\mathrm{G}}_{\bar{\mathrm{b}}} = \mathrm{G}_{\bar{\mathrm{b}}} \cdot s_\mathrm{W}^{-2} s_\mathrm{X}^{-2} = \mathrm{G}_{\bar{\mathrm{b}}} \cdot s^{-2}. \qquad (3)$$

After scaling, the gradient ratios match the floating-point counterpart. QAS enables fully quantized training (int8 for both forward and backward) while matching the accuracy of the floating-point training.

C. Memory-Efficient Sparse Update

Instead of pruning weights for inference, the gradient is pruned during BP, and update the model sparsely. Given a

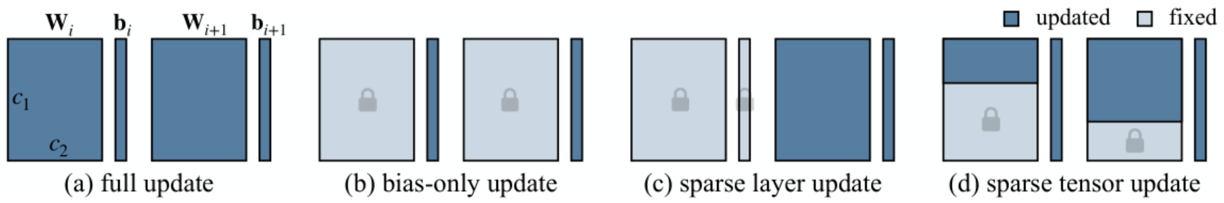


Fig.7. Different update paradigms of two linear layers in a deep neural network.

tight memory budget, the update of the less important parameters is skipped to reduce memory usage and computation cost. Given the different natures of updating rules, three aspects of the sparse update rule are considered, shown in Fig7.

- Bias-only update: bias update is cheap and biases are always updated if a layer is backpropagated to.
- Sparse layer update: select a subset of layers to update the corresponding weights.
- Sparse tensor update: a subset of weight channels are updated to reduce the cost.

The work proposes to automatically derive the sparse update scheme by contribution analysis to find the contribution of each parameter to the downstream accuracy.

D. *Efficient Training System*

An efficient training system, Tiny Training Engine (TTE) [25], is co-designed to transform the above algorithms into slim binary codes.

1) *Compile-time Differentiation and Code Generation*

TTE offloads the auto-differentiation from the runtime to the compile-time, generating a static backward graph which can be pruned and optimized to reduce the memory and computation. TTE is based on code generation: it compiles the optimized graphs to executable binaries on the target hardware, which minimizes the runtime library size and removes the need for host languages like Python.

2) *Backward Graph Pruning for Sparse Update*

The redundant nodes are pruned in the backward graph before compiling it to binary codes. For sparse layer update, the gradient nodes of the frozen weights are pruned away, only keeping the nodes for bias update. Afterwards, the graph are traversed to find unused intermediate nodes due to pruning and apply dead-code elimination (DCE) to remove the redundancy. For sparse tensor update, a sub-operator slicing mechanism is introduced to split a layer's weights into trainable and frozen parts; the backward graph of the frozen subset is removed. The compiler translates the sparse update algorithm into measured memory saving, reducing the training memory 7-9 × without losing accuracy.

3) *Operator Reordering and In-place Update*

The execution order of different operations affects the life cycle of tensors and the overall memory footprint. By reordering operators, the gradient update to a specific tensor (in-place update) can be immediately applied before back-propagating to earlier layers, so that the gradient can be released.

E. *Experimental Results*

The MCUNet overall framework is the first solution to enable tiny on-device training of convolutional neural networks under 256KB SRAM and 1MB Flash without auxiliary memory.

- The solution enables weight update not only for the classifier but also for the backbone, which provides a high transfer learning accuracy. For tinyML application VWW [27], the on-device finetuned model matches the accuracy of cloud training + edge deployment, and surpasses the common requirement of tinyML (MLPerf Tiny [28]) by 9%.
- The system-algorithm co-design scheme effectively reduces the memory footprint. The proposed techniques greatly reduce the memory usage by more than 1000× compared to PyTorch and Tensorflow.
- The framework also greatly accelerates training, reducing the per-iteration time by more than 20× compared to dense update and vanilla system design.
- The training system is deployed to a Cortex M7 microcontroller STM32F746 to demonstrate the feasibility, suggesting that tiny IoT devices can not only perform inference but also training to adapt to new data.

The study paves the way for lifelong on-device learning and opens up new possibilities for privacy-preserving device personalization.

V. Efficient AI Systems for edge devices

Due to the limited computation, storage, and communication resources of edge nodes, as well as the privacy, security, low-latency, and reliability requirements of AI applications, a variety of edge AI system architectures have been proposed and investigated for efficient training and inference. The main system architectures of edge AI can be summarized into four categories. According to the availability data and model parameters, data partition based edge training systems and model partition based edge training systems are two common system architectures for efficiently training at network edges. To achieve low-latency inference, computation offloading based edge inference systems is a promising approach by offloading the entire or a part of inference tasks from resource limited edge devices to proximate edge servers. There are also edge AI systems defined by general computing paradigms, which can be termed as general edge computing systems. [29]

A. *Data partition based edge training systems*

For data partition based edge training systems, the data is massively distributed over a number of edge devices, and each edge device has only a subset of the whole dataset. Then the edge AI model can be trained by pooling the computation capabilities of edge devices. During training, each edge device holds a replica of the complete AI model to compute a local update. This procedure often requires a centralized

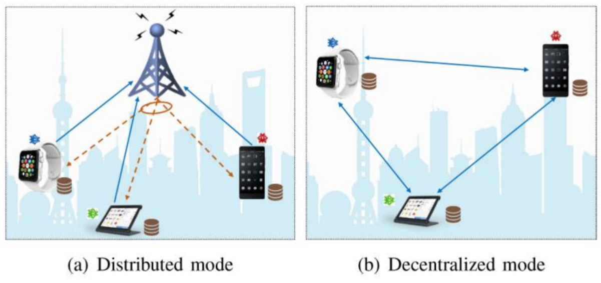


Fig.8. Two different types of edge training systems.

coordinating center (e.g., an edge server) for scheduling a number of edge devices, aggregating the local updates from edge devices, etc. There are also works considering decentralized systems where edge devices communicate with each other directly. Edge training systems with a central node are usually called distributed system modes, while systems without a central node are called decentralized system modes, as demonstrated in Fig. 8.

B. *Model partition based edge training systems*

In model partition based edge training systems, each node does not have replica of all the model parameters, i.e., the AI model is partitioned and distributed across multiple nodes. Model partition is needed when very deep machine learning models are applied. Some works proposed to balance computation and communication overhead via model partition for accelerating the training process. Furthermore, model partition based edge training systems garner much attention for preserving the data privacy during training when each edge node can only access to partial data attributes for a common set of user identities. It is often referred to as vertical federated learning. To preserve data privacy, it is proposed to train a model through the synergy of the edge device and edge server by performing simple processing at the device and uploading the intermediate values to a powerful edge server. This is realized by deploying a small part of model parameters on the device and the remaining part on the edge server to avoid the exposure of users' data.

C. *Computation offloading based edge inference systems*

To enable low-latency edge AI services, it is critical to deploy the trained model proximate to end users. Unfortunately, it is often infeasible to deploy large models, especially DNN models, directly on each device for local inference due to the limited storage, computation and battery resources. Therefore, a promising solution is to push the AI model and massive computations to proximate edge servers, which prompts the recent proposal of computation offloading based edge inference systems. The works on computation offloading based edge inference systems can be divided into two classes, i.e., deploying the entire model on an edge server, and partitioning the model and deploying across the edge device and edge server, as demonstrated in Fig.9.

D. *General edge computing systems*

There are also edge AI systems defined by general computing paradigms, e.g., MapReduce. The MapReduce-like frameworks often consider distributed data input and distributed model deployment jointly for accelerating distributed training or inference. In such systems, reducing the communication overhead for data shuffling between

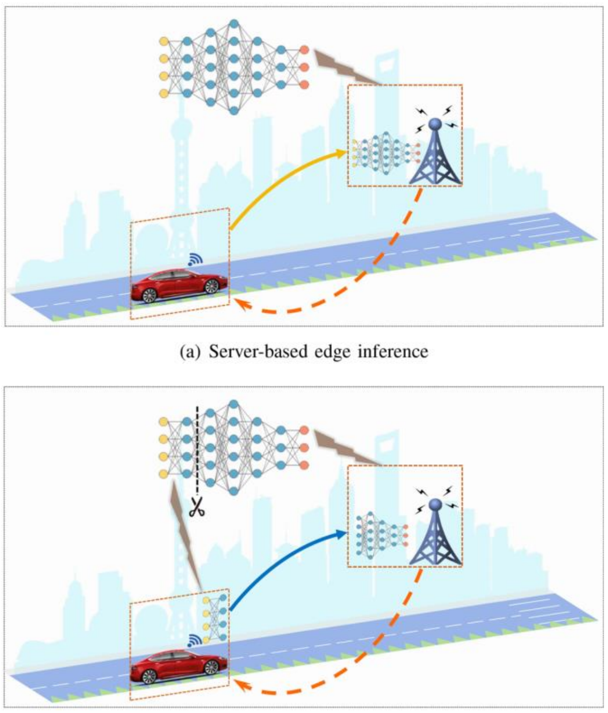


Fig.9. Computation offloading based edge inference systems.

multiple nodes becomes a critical task. Interestingly, coding technique plays a critical role in scalable data shuffling, as well as straggler mitigation.

VI. Conclusion

DNN training iteratively processes three different steps, resulting in a large number of external memory accesses and operations. Therefore, implementing DNN training on edge devices with limited computing power is necessary. Three challenges must be optimized: distinct dataflow, external memory access, and computation.

The three DNN training steps FP, BP and WG share a tensor, but have different data flows. In detail, the FP and BP steps share a weight tensor. However, the BP step uses the transposed weights in the out of channel and in channel dimensions. In addition, WG step generates weight gradient from features and output gradient obtained in FP and BP respectively, but unlike FP and BP, WG step does not perform channel accumulation. The different dataflows of the three training steps make DNN training and reasoning very challenging. This is because using the same data path architecture and memory layout to operate different data streams will reduce PE utilization and memory access efficiency. There are two ways to solve this problem: data array re-arrangement and data path configuration. Array re-arrangement modifies the storage layout before feeding to the PE. Many schemes have been proposed to achieve rearrangement more effectively, such as tile transpose, diagonal storage mode with bit rotator. On the contrary, the data path configuration modifies the data path of the PE array to maximize data reuse and PE utilization, while still using the same memory layout of tensors. The data path configuration uses the same memory layout for different training steps, but modifies the data path of the PE array to maximize data reuse and PE utilization.

External memory access must be considered. Unlike DNN inference, during the training, the intermediate features of all layers are stored in external memory. In addition, min-batch and batch normalization layer also make features a high proportion of external memory access. Therefore, compressing features must be considered to reduce external memory access. There are three methods of compression: sparse compression, occurrence-based compression, and bit-width reduction. First, sparse compression eliminates zeros by representing tensor with a non-zero value vector and an index vector. Second, occurrence-based compression reduces the tensor's byte size by encoding the most likely occurrence values to the lower bit-width. Third, reduced bit-width utilizes the loss robust property of DNN to lower bit-width of data while maintaining training accuracy. The compression methods can be used alone or combined depending on the feature's data characteristics.

The iterative operations of DNN training and the wide dynamic range of operands cause a lot of energy consumption, so computation has to be optimized to enable DNN on edge devices. In contrast to inference, in DNN training, the back-propagated gradients have a wide range of data distributions, so floating-point ALU is commonly used to handle them. However, floating-point ALU has high area consumption and energy consumption. There are two ways to implement high-energy efficient processing elements: zero-skipping operation and bit-precision optimization. Unlike inference, training has limitations in utilizing sparsity because there is no input sparsity in BP steps and no zeros in weights. Thus, using zero-skipping for DNN inference cannot significantly improve efficiency. So, many studies have explored output sparsity and detect unnecessary operations to skip. The bit-precision within the limited bit-width determines energy efficiency and accuracy. Bit precision can be optimized for each DNN training step, layer level, and neuron level. In addition, the PE must be configured to support various bit-precisions optimized to each level.

Section IV introduces an example of on-device DNN training and inference that considers dependencies between optimization techniques. Section V summarizes efficient AI systems for edge devices.

REFERENCES

[1] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," Nature,vol. 521, no. 7553, pp. 436–444, 2015.

[2] Tianqi Chen, Bing Xu, Chiyuan Zhang, and Carlos Guestrin. Training deep nets with sublinear memory cost. arXiv preprint arXiv:1604.06174, 2016.

[3] J. Lee and H. -J. Yoo, "An Overview of Energy-Efficient Hardware Accelerators for On-Device Deep-Neural-Network Training," in IEEE Open Journal of the Solid-State Circuits Society, vol. 1, pp. 115-128, 2021, doi: 10.1109/OJSSCS.2021.3119554.

[4] Song Han, Huizi Mao, and William J Dally. Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding. In ICLR, 2016.

[5] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," in Proc. Int. Conf. Mach. Learn., 2015, pp. 448–456.

[6] Yihui He, Ji Lin, Zhijian Liu, Hanrui Wang, Li-Jia Li, and Song Han. AMC: AutoML for Model Compression and Acceleration on Mobile Devices. In ECCV, 2018.

[7] Ji Lin, Yongming Rao, Jiwen Lu, and Jie Zhou. Runtime neural pruning. In NeurIPS, 2017.

[8] Yihui He, Xiangyu Zhang, and Jian Sun. Channel pruning for accelerating very deep neural networks. In ICCV, 2017.

[9] Zhuang Liu, Jianguo Li, Zhiqiang Shen, Gao Huang, Shoumeng Yan, and Changshui Zhang. Learning efficient convolutional networks through network slimming. In ICCV, 2017.

[10] Zechun Liu, Haoyuan Mu, Xiangyu Zhang, Zichao Guo, Xin Yang, Kwang-Ting Cheng, and Jian Sun. MetaPruning: Meta Learning for Automatic Neural Network Channel Pruning. In ICCV, 2019

[11] J. S. Park et al., "9.5 A 6K-MAC feature-map-sparsity-aware neural processing unit in 5nm flagship mobile SoC," in Proc. IEEE Int. Solid-State Circuits Conf. (ISSCC), vol. 64, 2021, pp. 152–154.

[12] K. Kanellopoulos et al., "SMASH: Co-designing software compression and hardware-accelerated indexing for efficient sparse matrix operations," in Proc. 52nd Annu. IEEE/ACM Int. Symp. Microarchit.,2019, pp. 600–614.

[13] Y. Park, Y. Kang, S. Kim, E. Kwon, and S. Kang, "GRLC: Grid-based RUN-length compression for energy-efficient CNN accelerator,"in Proc. ACM/IEEE Int. Symp. Low Power Electron. Design, 2020,pp. 91–96.

[14] C. Kim, S. Kang, D. Shin, S. Choi, Y. Kim, and H.-J. Yoo, "A 2.1 TFLOPS/W mobile deep RL accelerator with transposablePE array and experience compression," in Proc. IEEE Int. Solid-State Circuits Conf. (ISSCC), 2019, pp. 136–138.

[15] R. D. Evans, L. Liu, and T. M. Aamodt, "JPEG-ACT: Accelerating deep learning via transform-based lossy compression," in Proc. ACM/IEEE 47th Annu. Int. Symp. Comput. Architect. (ISCA), 2020,pp. 860–873.

[16] A. Jain, A. Phanishayee, J. Mars, L. Tang, and G. Pekhimenko, "GIST:Efficient data encoding for deep neural network training," in Proc.ACM/IEEE 45th Annu. Int. Symp. Comput. Architect. (ISCA), 2018,pp. 776–789.

[17] S. Kim, J. Lee, S. Kang, J. Lee, and H.-J. Yoo, "A 146.52 TOPS/Wdeep-neural-network learning processor with stochastic coarse-fine pruning and adaptive input/output/weight skipping," in Proc. IEEE Symp. VLSI Circuits, 2020, pp. 1–2.

[18] D. Han et al., "HNPU: An adaptive DNN training processor utilizing stochastic dynamic fixed-point and active bit-precision searching," IEEE J. Solid-State Circuits, vol. 56, no. 9, pp. 2858–2869, Sep. 2021.

[19] J. Zhang, X. Chen, M. Song, and T. Li, "Eager pruning: Algorithm and architecture support for fast training of deep neural networks,"in Proc. ACM/IEEE 46th Annu. Int. Symp. Comput. Architect. (ISCA),2019, pp. 292–303.

[20] BFloat16: The Secret to High Performance on Cloud TPUs.Accessed: Jun. 21, 2021. [Online]. https://cloud.google.com/blog/products/ai-machine-learning/bfloat16-the-secret-to-high-performance-on-cloud-tpus

[21] A. Agrawal et al., "DLFloat: A 16-b floating point format designedfor deep learning training and inference," in Proc. IEEE 26th Symp.Comput. Arithmetic (ARITH), 2019, pp. 92–95.

[22] P. Micikevicius et al., "Mixed precision training," 2017. [Online].Available: arXiv:1710.03740.

[23] J. Park, S. Lee, and D. Jeon, "A 40nm 4.81 TFLOPS/W 8b floating-point training processor for non-sparse neural networks using shared exponent bias and 24-way fused multiply-add tree," in Proc. IEEEInt. Solid-State Circuits Conf. (ISSCC), vol. 64, 2021, pp. 1–3.

[24] U. Köster et al., "FlexPoint: An adaptive numerical format for efficient training of deep neural networks," 2017. [Online]. Available: arXiv:1711.02213.

[25] Lin, Ji and Zhu, Ligeng and Chen, Wei-Ming and Wang, Wei-Chen and Gan, Chuang and Han, Song, "On-Device Training Under 256KB Memory". In NeurIPS 2022.

[26] Lin, Ji and Chen, Wei-Ming and Cohn, John and Gan, Chuang and Han, Song, "MCUNet: Tiny Deep Learning on IoT Devices". In NeurIPS 2020

[27] Aakanksha Chowdhery, Pete Warden, Jonathon Shlens, Andrew Howard, and Rocky Rhodes. Visual wake words dataset. arXiv preprint arXiv:1906.05721, 2019.

[28] Colby R Banbury, Vijay Janapa Reddi, Max Lam, William Fu, Amin Fazel, Jeremy Holleman, Xinyuan Huang, Robert Hurtado, David Kanter, Anton Lokhmotov, et al. Benchmarking tinyml systems: Challenges and direction. arXiv preprint arXiv:2003.04821, 2020.

[29] Y. Shi, K. Yang, T. Jiang, J. Zhang and K. B. Letaief, "Communication-Efficient Edge AI: Algorithms and Systems," in IEEE Communications Surveys & Tutorials, vol. 22, no. 4, pp. 2167-2191, Fourthquarter 2020, doi: 10.1109/COMST.2020.3007787.