A New Frontier of AI: On-Device AI Training and Personalization

Jijoong Moon
jijoong.moon@samsung.com

Hyun Suk Lee
hs89.lee@samsung.com

Jiho Chu
jiho.chu@samsung.com

Donghak Park
donghak.park@samsung.com

Seungbaek Hong
sb92.hong@samsung.com

Hyungjun Seo
hyungjun.seo@samsung.com

Donghyeon Jeong
dhyeon.jeong@samsung.com

Sungsik Kong
ss.kong@samsung.com

MyungJoo Ham
myungjoo.ham@samsung.com

Samsung Research, Samsung Electronics
Seoul, Republic of Korea

ABSTRACT
Modern consumer electronic devices have started executing deep learning-based intelligence services on devices, not cloud servers, to keep personal data on devices and to reduce network and cloud costs. We find such a trend as the opportunity to personalize intelligence services by updating neural networks with user data without exposing the data out of devices: on-device training. However, the limited resources of devices incurs significant difficulties. We propose a light-weight on-device training framework, \textit{NNTrainer}, which provides highly memory-efficient neural network training techniques and proactive swapping based on fine-grained execution order analysis for neural networks. Moreover, its optimization does not sacrifice accuracy and are transparent to training algorithms; thus, prior algorithmic studies may be implemented on top of \textit{NNTrainer}. The evaluations show that \textit{NNTrainer} can reduce memory consumption down to 1/20 (saving 95%) and effectively personalizes intelligence services on devices. \textit{NNTrainer} is cross-platform and practical open-source software, which is being deployed to millions of mobile devices.

KEYWORDS
On-Device AI, Neural Network, Personalization, Training, Software Framework

1 INTRODUCTION
We have witnessed the rapid proliferation of deep neural networks for a wide range of consumer electronics products in the industry. Their intelligence services provide key features to consumer electronics: semantic segmentation [30] for smartphone cameras, super-resolution [48] for TVs, object detection for robotic vacuums, image classification [20, 47] for smart ovens, speech recognition [7, 14] for smartphones and TVs, and ASR [24] and TTS [33] for real-time translations. The quality of such services has become significantly important for consumer electronics; thus, a lot has been invested to develop and optimize deep neural networks.

Since the proliferation of intelligence services in consumer electronics, the need for adaptation to different environments and requirements of each individual user, “personalization”, has arisen with additional technical difficulties [18]. Because we use general data to train neural network models for the general public, not personal data for a personal model, for an individual user, the model may look over-parameterized or its quality of service is lower compared to the size of the model.

To personalize intelligence services, we target to update models on devices with data available on devices: “on-device training”. Note that we do not need to train the whole models on devices; with personal data, we can update, fine-tune, or adjust models pre-trained by general data. We may increase the accuracy of a user with the user’s data. We may add classes defined by a user to personalize the service for the user. We may reduce latency and energy consumption for a user by reducing or skipping over-parameterized parts [41].

Training models on devices instead of clouds provides significant advantages. By running intelligence services on devices, “on-device AI” [12, 26], we can save cloud operating costs (we have hundreds of millions of active mobile phones and consumer electronics deployed!), and we can keep personal data and privacy without exposing them externally. By training on devices, not clouds, we can achieve the same advantages. Besides, regulations and consumers’ expectations on personal data and privacy are becoming more strict; e.g., the General Data Protection Regulation (GDPR) [45] makes it extremely difficult to gather personal activity records in clouds. A frequently used device (e.g., a mobile phone) usually generates personal data continuously, which can enable continuous personalization with online or continual learning techniques [8, 38, 42] on...
the device. Federated learning [10, 21] usually requires on-device training mechanisms, too.

Limited resources of devices pose challenges to training on devices especially if the accuracy cannot be sacrificed for resources; i.e., business units usually prohibit such trade-off in the authors’ affiliation. For example, studies to reduce the memory overhead emerge as larger neural networks become popular: dynamic sparse reparameterization [27], low precision training [28], reduced batch sizes [15], and gradient checkpointing [4, 6], which address algorithmic aspects of models. However, the system software aspect for training optimization is seldom studied, and we can improve significantly by addressing the structure and complexity of training software implementations. We address the system software aspect (i.e., how memory and computation are allocated and scheduled) to optimize, which has been often overlooked in neural network frameworks. Our approach does not sacrifice accuracy to conserve resources, which is often prohibited by applications. Moreover, the algorithmic improvements mentioned above can be applied transparently on top of the proposed mechanism.

We address the execution orders of training procedures based on the observation that we can statically calculate computation and memory requirements from the model structure. Thus, by controlling such execution orders, we can optimize resource utilization. First, we divide a training session into fine-grained procedures and determine the life cycles of memory blocks assigned for procedures so that controlling execution orders of the procedures can determine the memory consumption accordingly. Then, we identify possible execution orders for each neural network layer type and schedule them to minimize peak memory consumption. Finally, we apply another optimization technique, Proactive Swap; i.e., we know when a buffer is read or written (clairvoyant!); thus, we can swap in and out proactively, minimizing its performance impacts.

Our contributions can be summarized as follows:

- We propose a highly memory-efficient on-device training technique without sacrificing accuracy and latency, exploiting the novel observation on the nature of neural network training, and Proactive Swapping. The peak memory consumption is dramatically reduced, realizing training on embedded devices. Moreover, the techniques are general; conventional machine learning frameworks may adopt them for servers, and neural network algorithmic optimization may be applied simultaneously.
- We implement and release the proposed techniques as an open source, cross-platform, and commercialization-ready framework, NNTrainer, which is already applied to actual products.
- We evaluate NNTrainer with various models and conventional frameworks, and show that it is highly efficient and effective. We demonstrate that complex encoder-decoder models based on Tacotron2 [37] and Transformer [44] can be personalized with sufficient batch sizes on mobile phones, which are deployed for products in early 2023.

2 RELATED WORK

With larger neural networks being popular, studies on algorithms to consume less resources to train have been published, which usually trade-off between resources and accuracy with neural network algorithmic approaches. A study [27] proposes dynamic sparse reparameterization to reduce the memory requirements by making the weight and activation values sparse during training. It generates smaller model sizes that incur less peak memory consumption and computation with the sparsity by sacrificing the accuracy. By adopting 16 bits float precision instead of 32 bits, [28] reduces the memory and computation requirements, which also reduces the model size; however, it also sacrifices the accuracy. The activation values occupy a significant part of memory and microbatching technique [15] helps reduce the size of such values. However, it alters the statistical properties of batch normalization, which affect the accuracy. There are studies [4, 6] to reduce memory consumption without sacrificing accuracy by storing the activation values partially and recomputing the activation values not stored; however, obviously, this increases the computation cost by about 30%. ZeRO-Offload [35] and ZERO-Infinity [34] propose software optimization mechanisms without sacrificing the accuracy by swapping the memory block to the secondary storage. However, unlike [34, 35], which swaps in and out reactively, the swapping mechanism of NNTrainer swaps in and out according to the execution order; i.e., the swapping becomes proactive, which minimizes performance deterioration. Another difference is that they focus on offloading the sliced chunks of activation or weight; thus, it is difficult for [34, 35] to support gradient clipping or non-trainable layers.

There are several popular software frameworks to train the neural network model: TensorFlow [1] and PyTorch [32]. TensorFlow is a popular open-source machine learning framework from Google. It includes Keras and various tools for developers. A lightweight variant of TensorFlow, called TensorFlow-Lite [11], targets on-device AI and training features have been enabled recently. It is written in C/C++ rather than Python. However, unlike NNTrainer, developers cannot update model structures with the lightweight TensorFlow-Lite on device; the full TensorFlow is required for such tasks.

PyTorch [32], including Caffe2 [17], is another popular open-source machine learning framework from Facebook, which includes highly intuitive APIs. It has an experimental lightweight variant, PyTorch Mobile, which runs mobile models converted from PyTorch models. Note that PyTorch Mobile targets inference and does not support training.

TensorFlow, TensorFlow-Lite, and PyTorch consume an excessive amount of memory to train models (shown in §5); thus, consumer electronics cannot use them to train models. For example, the quality assurance team of the authors’ affiliation will drop in-house applications consuming an excessive amount of memory because the main memory may be crowded by a large number of applications preloaded [23]. Note that reducing the memory consumption is beneficial for conventional server-based machine learning; i.e., we may increase the batch size with less cost.

3 ANALYSIS

This section elaborates on the observation and analysis of the neural network training processes and their resource requirement and
which takes an input image of \( X \) with the same padding, \( W \). Where the loss is calculated and propagated. A backward process calculates the gradient \( W \) with the input derivative \( D \) and \( X \) saved in a forward process, and updates the weight \( W \) using an optimizer. Then, it calculates the output derivative, \( D \), to propagate the input derivative \( D \) to \( L \).

Depending on the layer type, a layer requires buffers for weight \( W \), gradient \( D \), input \( X \), and output \( Y \). Usually, the sizes of such buffers depend on the input size and the depth of the neural network. When the input size and the network configuration are determined, the amount of memory required for training can be roughly calculated. Assume that there is a convolution 2D layer with the same padding, 1 \( \times \) 1 stride, and 64 filters with \( 3 \times 3 \) size, which takes an input image of \( 32 \times 32 \times 3 \). Then, the input size is 0.39 MiB for 32 batch size (width \( \times \) height \( \times \) channels (3) \( \times \) sizeof (float) \( \times \) batch size), and the output buffer size is 8.3 MiB (width \( \times \) height \( \times \) channels (64) \( \times \) sizeof (float) \( \times \) batch size). We also need derivatives \( (\Delta D_i, \Delta D_{i+1}) \), weight \( (W_i) \) and gradient \( (\Delta W_i) \) for backward process. Therefore, we need about 16.6 MiB of heap memory to train a single layer. For "deep" neural networks, we need multiple times, often over a hundred times, of such an amount of memory depending on the depth of the given neural network model.

Figure 1.b shows the memory buffer configurations for neural network training. It repeatedly reuses the memory buffer configuration of a single layer depicted in Figure 1.a, which is equivalent to dotted rectangles in Figure 1.b. Conducting forward and backward processes for \( n \) layers requires memory space of about \( n/2 \) times the memory space required for a single layer. Because the output of \( L_i \), \( X'_i \), and the input of \( L_{i+1} \), \( X_{i+1} \), represent the same data and layers do not modify input data, the two may share the same buffer instance. Besides, we can reduce memory further as shown in Figure 1.c. Although the output of each layer, \( X'_i \), is stored for the backward process, its derivatives, \( \Delta D_i \) and \( \Delta D'_i \), are not required after the completion of the layer’s backward process. Therefore, memory space for the derivatives can be shared. The same optimization can also be applied to the gradients because they are not required once the weight is updated with the gradients.

There are layers that may reuse input buffers, \( X_i \), as output buffers; e.g., activation layers. For example, let \( X'_i \) be the output of a sigmoid activation function, then, its derivative is \( \Delta D_i = X_i (1 - X'_i) \). During a backward process, computing \( \Delta D_i \) requires \( X'_i \), which is the output of a corresponding forward process, not the input of it. Therefore, only one intermediate activation is required to store the output \( X'_i \); i.e., an in-place computation shown as \( L_3 : \text{InplaceOp} \) in Figure 1.c. This allows for freeing the memory space storing inputs of activation layers. Because activation layers are applied after most operations, including convolution and linear layers, this method reduces the memory requirement of inputs by almost half. This can be applied to batch normalization as well.

From the analysis, we find that different granularity of training procedure is possible, reducing the memory consumption with its own pros and cons as in Figure 2. Coarse-grained procedure as in conventional training frameworks (TensorFlow and PyTorch) costs less to develop a new layer because only its forward process is required to be implemented with Automatic Differentiation [3];
This section describes the design and implementation of NNTrainer. NNTrainer is a modular framework written in C++ with many user-extensible components, whose structure is shown in Figure 4. It is cross-platform; we release to Ubuntu, Tizen, Android, and Windows.

The training processes of NNTrainer can be categorized as Load, Configure, Compile, Initialize, setData, and Train, in the order we elaborate.

4 DESIGN

The proposed mechanism supports a mixed network of trainable layers and non-trainable layers, which is common in on-device AI personalization [25, 40, 43]. In non-training layers, forwarding processes do not need to store $X_0$ because of skipping gradient-related processes shown in Figure 3b. Although conventional frameworks support non-trainable layers, they still allocate memory for activation inputs, $X_0$. We evaluate with a fully connected layer having 131,584 parameters (514 kB of float32) in the middle of a model with GPUs and confirm that the memory of gradient is reduced as in Figure 3b. Moreover, the re-ordering of fine-grained training procedures enables the Gradient Clipping [49] easily as in Figure 3a. Therefore, the fine-grained training procedure is more robust and efficient for supporting various training processes and resource utilization optimizations.

Figure 3: Different types of training procedures.

Figure 4: Abstract architecture of NNTrainer.

Load: the output of Load is an instance of Model, which has all the information of a neural network model, including hyper-parameters, and orchestrates all the components and processes. Users may describe a neural network as an instance of Model with an initialization file [29] or APIs in C, C++, .NET, and Web. The interpreter in Model Loader parses model descriptions and generates intermediate graph representations. Developers may extend Interpreters to support model files of other frameworks; we provide an extension for TensorFlow-Lite [1]. The output of Load describes layers of the model as tuples of [{Layer type}, {Properties (key, value)}].

Configure creates layer objects with those tuples, and the layer objects construct a Graph. We construct Graph and Layer instances with delayed allocation based on the information provided by Context, which has parameters of Graph and Layer instances. It conserves memory by delaying Tensor buffer allocations until the buffers are actually required.

Compile: before Initialize, Layer Context exists as an initContext instance storing the information as strings. Later, initContext becomes runContext by Initialize process, converting string descriptions to Tensor objects.

It is lighter and more efficient to conduct the Compile process with initContext; it handles strings, not Tensor instances. Each Layer subclass provides forward and backward functions that calculate gradients and derivatives, respectively. Each Optimizer subclass provides a function applying gradients. Users can implement new layers and optimizers by adding these subclasses.

For higher memory efficiency, we need to manage Tensor data independently; thus, we have separated specifications (dimensions and status) and data stored in buffers.

The Model’s Graph, created by Configure process, can operate after Realizer of Compiler applies lowering operations; i.e., the
Realizer adds layers or changes the order based on the analysis of graph structures. Table 1 shows the default Realizer subclasses of NNTrainer. Compile process computes connection and ordering with the provided Graph Optimizer subclasses, which are critical for memory saving. Users may extend both Graph Optimizer and Realizer by adding subclasses.

**Initialize**: in Initialize process, a finalize method of a Layer subclass is executed while visiting each Layer instance in Graph, and Tensor instances are requested to Tensor Pool. Because specification and data of Tensor are managed independently, NNTrainer manages memory separated by Tensor Pool and Memory Pool. After Tensor Pool creates all requested Tensors instances throughout the Model, Memory Planner plans with Tensor instances given by Tensor Pool, computes the offset of the Memory Pool for each Tensor instances, and assigns the memory for Tensor data. NNTrainer provides the Memory Swap using Cache Pool and Loader. Loader creates an independent Task Executor and monitors the existence of the tensor data in memory. It loads and unloads proactively using execution order (EO) to hide the data transfer overhead (§3).

**setData**: to generate batch-sized data for training in setData process, NNTrainer provides DataSet, and users can provide training data using DataProducer, which is extendable. DataProducer generates data for training and accumulates the data in the Batch Queue up to the batch size. After the described processes, we are finally ready for Train process.

As an example of resources, the uniqueness of NNTrainer over the conventional is that Tensors in NNTrainer are prioritized based on EO and in-place operation. This allows Memory Planner to maximize the reusability of Memory Pool; thus, reducing memory consumption.
Algorithm 1 Compute Execution Order.

Input: Layer List, L = {L₀, L₁, ..., Lₙ₋₁}

1: \( EO_{max} = N \times 4 \)
2: Initialize \( T_{map} \)
3: for \( i = 0, 1, ..., N − 1 \) do
4: \( EO_Pi = 1 \)
5: \( EO_{CG} = N + i \times 3 \)
6: \( EO_{AG} = EO_{CD} + 1 \)
7: if Gradient Clipping then
8: \( EO_{CG} = N + i \times 2 \)
9: \( EO_{AG} = N + 3 \times 3 \)
10: else if
11: \( EO_{CD} = EO_{CG} + 1 \)
12: else if
13: \( EO_{CG, CD, AG} = N + i \times 3 \)
14: end if
15: \( EO_{Li} = (EO_{P}, EO_{CG}, EO_{CD}, EO_{AG}) \) if \( T \) type
16: for \( T_L \in T_{map} \) do
17: \( EO_{Li} \) is Tensor list requested in \( L_i \)
18: Add \( EO \) to \( EO_{L_i} \) in \( T \) w.r.t Temporal of \( T_r \)
19: end for
20: end for
21: Sort \( T_{map} \) by ascending with \( min(EOs \ of \ T_i \in \ T_{map}) \)
22: for \( T_i \in T_{map} \) do
23: \( C_{Mt} \) = Tensor create mode of \( T_i \)
24: if \( C_{Mt} = MV_{Ts} \) then
25: if \( min(EOs \ of \ T_i) \geq max(EOs \ of \ T_j) \) then
26: Merge \( T_j \) to \( T_i \) including EOs
27: end if
28: else if \( C_{Mt} = RV_{Ts} \) or \( E_{Ts} \) then
29: Merge \( T_i \) to \( T_j \) including EOs
30: end if
31: end for

Algorithm 2 Simple sorting-based Memory Planner.

Input: Tensor List, \( T = \{T_1, T_2, ... , T_N\} \)

1: Sort \( T \) by ascending order based on \( min(EOs \ of \ each \ T_i) \)
2: if \( min(EOs \ of \ T_1) \rightarrow min(EOs \ of \ T_j) \) then
3: Sort by descending order based on \( max(EOs \ of \ T) \)
4: end if
5: for \( i = 1, 2, ..., N \) do
6: \( EO_{min} = min(EOs \ of \ T_i) \)
7: last = i
8: for \( j = i, 1, 2, ..., N \) do
9: if \( EO_{max} = max(EOs \ of \ T_j) \)
10: \( EO_{max} = min(EOs \ of \ T_j) \)
11: last = j
12: end if
13: end for
14: if last = i then
15: Calculate offset for \( T_i \) and Assign to \( T_i \) data
16: else
17: Assign data(offset) of \( T_{last} \) to \( T_i \) data
18: end if
19: end for

Figure 5: Execution orders and temporal-spatial relations of an example model, where only \( X_0, X_1, D_3, \Delta W_0, \) and \( W_0 \) are required. Refer to Figure 1, Table 2 and 3 for the notations.

Figure 6: Memory planning without spatial relation for three linear layers.

Tensor cannot be guaranteed because the target Tensor is accessed after the merge; thus, it cannot be combined and a new Tensor needs to be created. Next, the largest order of \( X_1 + X_2 \) is greater than the smallest order of the merged Tensor, \( X_3 \), 2; however, it can still be merged because the integrity of data is guaranteed with \( RV \). This reduces memory requirement by removing another intermediate activation as described in Figure 1.

4.2 Memory Pool and Memory Planner

With EOS assigned, Memory Planner allocates buffers from Memory Pool with the planner algorithm. Algorithm 2. Algorithm 2 is simple sorting-based; an algorithm minimizing fragmentation for higher utilization is future work. Figure 6 shows how Algorithm 2 works for a case without spatial relations as a simple example, where the four iterations of the for-loop, \( i = 7..10 \), are shown, reallocating a few tensors to share and reuse the memory space.

For each Tensor data, we calculate the memory offset to assign. In Figure 6, when we calculate the offset of \( W_1 \) after calculating the offset of \( W_0 \), because there are no assigned Tensors whose largest \( EO \) is less than the smallest \( EO \) of \( W_1 \), a new offset is calculated and assigned to \( W_1 \). It keeps assigning new offsets until \( X_3 \) as shown in the first row of Figure 6. As in the second row of the figure, when we calculate for \( D_3 \), the largest \( EO \) of \( X_3, 2 \), is less than the smallest \( EO \) of \( D_3 \). It means that \( X_3 \) can be reused; thus, the same offset is assigned to the data of \( D_3 \) as in line 17 of Algorithm 2. To
works with three linear layers. In Figure 8.a, the memory scheduler to schedule swapping, the validity of tensors, which differs for each training procedure, should be considered. The weight tensors for activation which occupy a huge part of memory during training.

Most of the previous works focus on reducing the intermediate computation beforehand, engineers can plan machine learning tasks more effectively and try more diverse hyper-parameters, different model structures, or higher resource utilization (e.g., increased batch sizes).

### 4.3 Reduced and Proactive Swap

Most of the previous works focus on reducing the intermediate activation which occupies a huge part of memory during training. In order to resolve this problem, NNTrainer proposes the Proactive Swap scheme on top of NNTrainer memory planning based on EO. To schedule swapping, the validity of tensors, which differs for each training procedure, should be considered. The weight tensors should be valid throughout the whole training procedure without any allocation and deallocation (Light blue area in Figure 7). The input and output tensors for activation should be valid until gradient computation whose inputs are the results of forwarding computation (Light green area in Figure 7). NNTrainer uses individual I/O streams for each Tensor Pool Stream (TPS) and Weight Pool Stream (WPS) considering different characteristics of tensors. The simplest swap scheme is On-Demand in Figure 7 which conducts swap-in/out when required without any concern about EOs. Figure 8 shows an example of how Reduced and Proactive Swap works with three linear layers. In Figure 8.a, the memory scheduler of NNTrainer counts temporal and spatial relations, and reduces the number of swapping by reusing input tensors ($X_n$) for input derivatives ($D_3$) between $CG_3$ and $CD_3$. Figure 8.b shows a Proactive Swap example with +1 lookahead from $AG_3$ to $CD_2$; Proactive Swap at the $n$-th EO hides I/O overhead by pre-loading tensors of $n$-lookahead EO and offloading tensors of $n$-1-th. For each EO, an I/O stream swaps out tensors used by the previous EO and swaps in tensors for the next (lookahead +1) EO. We can reduce the swap overhead further by merging conflicting swap requests; e.g., $D'_3$ and $D_3$ at $AG_3$. Then, we can compute the memory overhead of Proactive Swap. Tensors in memory at $AG_3$ are $X_n$, $D'_3$, $AW_2$, and $W_3$ after Proactive Swap. Only $W_3$ is the memory required additionally from the on-demand swap, and it is the bearable cost for the reduced latency.

The lookahead is a hyper-parameter determined statically at the initialization, and its default value is one. However, the optimal lookahead differs by network configurations and execution orders; e.g., gradient clipping, distribution of non-trainable layers, and layer types. Thus, we can further reduce memory and computation overhead by assigning lookahead for each EO dynamically; i.e., let each lookahead value converge throughout iterations. This is left for future work.

## 5 EVALUATION

We evaluate both small experimental neural networks and large practical neural networks. We analyze and compare latency and memory consumption of NNTrainer and conventional frameworks: PyTorch 1.13.1, TensorFlow 2.11.0, TensorFlow-Lite 2.11.0 (both C++ and Python implementations); recent TensorFlow-Lite supports training. For memory consumption, we also compare with the theoretical memory requirement based on Analysis (§3). We demonstrate on-device training applications on devices: a complex text-to-speech (TTS) application with multiple LSTM layers on Galaxy S21 Ultra (Exynos 2100, 1×ARM Cortex-X1 2.9 GHz, 3×ARM Cortex-A78 2.9GHz, 4×Cortex-A55 2.2GHz) and a Transformer model with huge memory requirements, virtually impossible for embedded devices to run. Except for TTS, we evaluate with Raspberry pi 4, which has 4×ARM Cortex-A72 (1.5GHz) CPU cores and 8 GiB RAM running Ubuntu 22.04. We evaluate NNTrainer with CPUs as its computation backend. Note that most NPUs and DSPs of embedded devices do not support floating points and GPUs of consumer electronics are often not available for training; e.g., GPUs of TV are supposed to be fully dedicated to video streams and GUIs.

### 5.1 Component Evaluation

Table 4 describes 5 test cases of small neural networks with different dimensions of inputs and labels. The test cases include an LSTM layer (popular for time series including voice models) and linear and Conv2D layers (popular for vision models). We evaluate a case of three linear layers (FC-FC-FC) with non-trainable layers in the middle and a case of Figure 5 with Conv2D layers (Conv-AC-FL). We have applied mean squared error (MSE) [2] and stochastic gradient descent (SGD) [19] to train the test cases.

We ensure the correctness of NNTrainer by comparing every activation and weight value of models trained by NNTrainer with the same models and data trained by TensorFlow. The two frameworks result in equivalent neural network models with errors at $10^{-4}$ level. Note that an automated test suite [22] ensures the correctness and equivalence for each pull request in different environments.
As explained in Section 3, once the input sizes and configurations are determined, the theoretical memory requirements can be calculated based on the sizes of intermediate activations, weights, and gradients as in Table 4.

Figure 9 shows the evaluation results of the baseline and peak memory consumption. The baseline implies memory consumed by the framework itself (e.g., codes and libraries), which is constant for different test cases. The peak implies the memory consumed by training a model in addition to the baseline. TensorFlow consumes x2.52 of baseline memory (329.8 MiB) than NNTrainer (11.3 MiB) consumes, and PyTorch consumes x8 (103.4 MiB). TensorFlow-Lite in C++ consumes 39.6 MiB, which is three times. Such excessive baseline memory consumption of TensorFlow, TensorFlow-Lite (C++) consumes three times. Even TensorFlow-Lite in C++ consumes x4 of that NNTrainer consumes. The theoretical memory requirement in Figure 9 suggests that NNTrainer is extremely efficient in using memory by sharing the most shareable tensors with ignorable memory overhead. The baseline is required to load essential libraries, and additional heaps attributing to the peak are required for some layers; e.g., NNTrainer’s Conv2D layer adds “Image to Column” (im2col) [9] operator for computation efficiency, which requires additional buffers.

We evaluate how much memory can be reduced by NNTrainer with FC-FC-FC case by setting the second layer non-trainable. Figure 10 shows the memory placement along with EOs by NNTrainer’s memory planner. $X_1$ is the input tensor of the second layer which needs to be saved for computing gradient ($L_1 : CG$) for the training case (Figure 10.a). For the non-trainable case, the validity of $X_1$ is during forwarding as in Figure 10.b so that the memory space is utilized by other tensors. The results show the memory reduction at $L_1 : CG$ (red rectangle), the non-trainable layer. However, the peak memory is measured at a different EO and 24.4 MiB is expected to be reduced. The experimental results show that 20.9 MiB is reduced: 146.1 MiB vs. 125.2 MiB.

Figure 11 shows the training latency of 1 epoch, 640 dataset size with the test cases in Table 4. This experimental result shows that NNTrainer does not sacrifice the latency or the accuracy to conserve memory. Although consuming significantly less memory, in most cases, NNTrainer is evaluated to be faster than or equivalent to the conventional frameworks.

5.2 On-Demand, Reduced, Proactive Swap

Swapping allows additional memory saving selectively; users may turn it on and off. We compare the peak memory consumption and the number of swapping for different schemes, On-Demand, Reduced, and Proactive Swap with one lookahead parameter, in Figure 12. Proactive Swap includes Reduced Swap. We train the VGG16 model with 64 batches of $32 \times 32 \times 3$ inputs. Figure 13 shows the latency of each EO for each scheme. As expected, without swapping, the largest size of memory is consumed, although it is still the smallest among the compared frameworks in Figure 14. The evaluations show that Proactive Swap reduces the peak memory consumption from 181 MiB to 71 MiB successfully with 20.1% latency overhead.
and On-Demand Swap reduces to 51 MiB with 35.5% latency overhead. The peak memory consumption of On-Demand Swap is 59 MiB, the same as Reduced Swap at c2:CD. The memory size of input derivative of $c2:CD, \Delta D'$ is $64 \times 32 \times 32 \times 64 \times \text{sizeof(float)}$ and output derivative $\Delta D$ has the same size. The memory size for weight $W$ is $(64 \times 3 \times 3 \times 64 \times 64) \times \text{sizeof(float)}$. Then, the theoretical memory size, which needs to be kept allocated at c2:CD, is 32.14 MiB. Considering the baseline (11 MiB), this is close to the peak memory consumption with swapping. Thus, we can conjecture that NNTrainer's swap scheduling utilizes memory extremely efficiently.

There is almost the same memory reduction achieved for both On-Demand and Reduced Swap; however, Reduced Swap reduces the latency by 10% with reduced number of swapping by the EO-based memory planning proposed in §4. Proactive Swap consumes more memory while reducing latency by 10%; it further reduces swapping as shown in Figure 8.b and Figure 12. Note that Proactive Swap does not request swapping at a convolution layer during forwarding (a red box in Figure 12). It is because its activation layer incurs in-place computation, which does not need additional memory swap-in. However, we need a swap-in after the activation before its next layer; thus, the swapping count increases. The latency of $c9\_layer:CD$ (green boxes with red borders) in Figure 13 compares the latency of the swap schemes.

5.3 Applications

Figure 14 shows the peak memory consumption of training with the batch size of 64, trained by NNTrainer (w/o Swap, Reduced Swap), TensorFlow, TensorFlow-Lite (Python, C++), and PyTorch. The models are trained from scratch: AlexNet [20], VGG16 [39], and Resnet18 [13]. In every case, NNTrainer consumes much less memory. Without swap, it consumes 23% (AlexNet), 15% (VGG16), and 35% (ResNet18) of what the other uses. With Reduced Swap, it consumes 18% (AlexNet) and 5% (VGG16 and ResNet18) of what the other uses; the most efficient of others, PyTorch, consumes 20 times more! AlexNet shows less improvement with swapping is because the last two linear layers consume peak memory; the weight dimension is $4096 \times 4096$, which is 62 MiB. The theoretical peak memory of Apply Gradient is two times of its weight size due to the gradient (124 MiB). TensorFlow-Lite, with both C++ and Python, consumes almost x46 times more memory for ResNet18 even if it is for embedded devices.

Tacotron2 [37] is a sequence-to-sequence text-to-speech model based on attention, which enables end-to-end training with dataset...
of sentence and voice. Developers can create a personalized text-to-speech (TTS) application by running Tacotron2 on NNTrainer so that an application may read books for kids with voices of their parents or translate with the user’s voice. In our application, a user reads 18 sentences to create a user dataset for personalization, which is combined with the pre-installed dataset. Tacotron2 consists of an Encoder, Attention, and Decoder, and the Decoder generates voice outputs; thus, we apply on-device training to the Decoder, and other parts are kept static for inferences.

NNTrainer does time-unrolling to perform complex time iteration, and the entire memory is statically allocated based on the maximum time iteration declared by developers. Weights of the same layers that are time-unrolled incur no additional memory or computation because of the Tensor sharing. Because of time iteration, forward process is performed for all unrolled layers, and gradients are accumulated in the backward process without updating weights. The optimizer updates weights only once per layer, which requires additional memory to store the gradient during time iterations. Gradient Clipping [31] and Teacher Forcing [46] are also supported.

In Figure 15, peak memory consumption and latency with the different numbers of threads for training with 26 samples on Galaxy S21 Ultra are measured to evaluate the effect of parallelism with OpenBLAS [50]. Overheating is a major issue of mobile phones, and to mitigate it, CPU cores are usually throttled by limiting their frequencies. Training neural networks heavily utilize CPUs; thus, we observe how the CPU throttling affects the latency. Figure 15 shows the latency and frequency of each core, and we can see that the CPU frequencies are continuously throttled down. Another finding is that multi-core parallelism is not helpful for this model, and if we can cool CPUs properly, the training performance will be improved significantly. Only 364 MiB of memory is requested for the case without swap to personalize the Tacotron2 and NNTrainer completes within ten minutes even with throttled CPU cores, which has been enough to pass the quality assurance team.

Transformer [44] is a popular neural network model throughout various applications. Transformer consists of Encoder and Decoder, and each of them has stacked multi-head attention layers, which require a huge amount of memory: over 6 GiB for 128 batch size, 6 stacked 8 multi-headed attention layers in both encoder and decoder to train. Although it requires memory-intensive computation, each multi-head attention layer is independent, promoting easier and more efficient parallelism. Conventional frameworks intentionally allocate weights of \( Q, K, V \) for each head and try to achieve shorter latency with parallelism. It is definitely efficient in clouds, which have abundant resources; however, it is usually not desirable on devices because the size of the internal activation of the attention layer increases. For swapping, the smaller the size of the layer intermediate activation or weight, the more reduction in the memory consumption. Therefore, NNTrainer implements the multi-head attention using Concat(Attention\((QV_i^Q, KW_i^K, VW_i^V)\)) as proposed in [44]. Figure 16 shows the size of memory consumed to train Transformer with different batch sizes. With relatively small batch sizes, the effect of memory planner based on EOs is noticeable compared to PyTorch; it is almost half (575.2 vs 256.9 MiB). However, as the batch size increases, the size of intermediate activation saved for gradient computing grows exponentially and the effect of the memory planer diminishes. Proactive Swap reduces the memory consumption dramatically from 4.4 GiB to 0.22 GiB (down to 1/20) for 64 batch size. Moreover, only Proactive Swap can train with 128 batch size in a machine with 8 GiB RAM, utilizing only 420 MiB.

6 FUTURE WORKS

Our future plan is to make NNTrainer more widely available and expand for wider range of machine learning techniques popular in embedded device applications. Also, we do have a plan to develop new techniques to utilize resources more efficiently.

- Extend the reach of NNTrainer. This includes providing an interface with other frameworks, where model importers will be our first priority followed by model exporters, and expand platform support for Android with Java APIs, support Yocto/OpenEmbedded and TIZEN with C# and Web APIs.
- Extend support to various computation backends and accelerators for embedded devices such as GPUs, DSPs and NPUs to expand computational capacity with higher energy efficiency. Integer-based training or quantization-aware training [16] is the next step.
- Support advanced few-shot learning techniques such as meta-learning and adaption problems.
- Support half precision during training. This is more efficient on-device at a point of latency and memory consumption.

7 CONCLUSIONS

We propose a highly efficient and light-weight on-device neural network training framework, NNTrainer, with techniques based on novel observations on neural network training mechanisms. We significantly reduce memory consumption so that embedded devices can practically train neural networks without deterioration of accuracy or modification of model architecture. The evaluation results show the efficiency of NNTrainer for various neural network models and applications. The proposed framework, NNTrainer, is practical open-source software that personalizes complex AI services on mass-produced devices.

REFERENCES

[1] Martin Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray,
Authorized licensed use limited to the terms of the applicable license agreement with IEEE. Restrictions apply.