A. Detailed Related Work

The growing need for personalization, data privacy, and post-deployment adaptability underscores the importance of on-device training for DNNs, which necessitates computational resources to perform backpropagation locally, including the calculation and storage of intermediate gradients for updating the DNNs. To minimize resource consumption, existing approaches include techniques such as gradient checkpointing [7, 10], layer-wise training [5], split learning [30], and transfer learning [18]. Despite their resource efficiency, these approaches often suffer from reduced accuracy when faced with changes in data or downstream tasks.

Another approach is to co-optimally allocate resources alongside the accuracy of downstream tasks. However, such approaches [6, 25, 26, 37, 48] either (1) rely on server-side architecture search to select layers before deployment on the device, or (2) require at least one full backpropagation through the entire model (computing all gradients), which creates a significant bottleneck for resource-constrained devices.

PruneTrain [29] reduces training parameters through channel-level sparsity and structured pruning. It uses group lasso regularization to shrink the model while maintaining dense computation, enabling dynamic architecture adaptation during training. TinyTrain [21] performs an adaptive parameter selection method based on Fisher Information (FI) of activations to assess a layer's importance. This allows TinyTrain to be both compute- and memory-efficient on the device. However, to perform the online selection of layers, TinyTrain performs one round of backpropagation through the entire model. Since relying on a single round of backpropagation may result in suboptimal parameter selection [14, 19], TinyTrain performs multiple rounds of backpropagation during a server-side meta-training stage. Meta-training with data from similar distributions might not be realistic, as the distribution of the target private data could change significantly, which negatively impacts the effectiveness of the FI-based layer selection [42].

ElasticTrainer [16] dynamically selects parameters using a dynamic programming approach that minimizes resource consumption of on-device training. Unlike TinyTrain [21], ElasticTrainer eliminates the need for a meta-training phase by directly selecting important layers on the device. Such a selection process with dynamic programming involves multiple rounds of parameter tuning, each requiring backpropagation through the entire model to ultimately identify the optimal subset of layers. This makes the overall process of selection and training memory-intensive, at times even comparable to that of full training. A comparison between existing on-device layer selection approaches for DNNs and our proposed system, *AdaBet*, is summarized in Table 1.

B. Background on Betti Numbers

In mathematics, **topology** examines the invariant properties of spaces under continuous deformations, focusing on connectedness and compactness. In ML, topology allows us to study the robustness of learned representations to transformations such as scaling, rotation, and noise addition [9]. In algebraic topology, **homology** analyzes the structure of *holes* in a space across various dimensions to understand the shape and connectivity of high-dimensional spaces. Instead of relying on visualizations, homology captures topological features as algebraic invariants, enabling broader and more efficient analysis of complex spaces.

Formally, the n^{th} homology group of a topological space $\mathcal{X}$, denoted as $H_n(\mathcal{X})$, quantifies *n-dimensional holes* in $\mathcal{X}$; such that 0-dimensional holes indicate connected components, 1-dimensional holes indicate loops, and higher-dimensional holes indicate structures such as voids and cavities.

To calculate $H_n(\mathcal{X})$, we turn the geometric problem (e.g., holes in a space) into an algebraic one (i.e., computing groups) by building a chain complex $C(\mathcal{X})$: a sequence of abelian groups linked by boundary operators δ_n such that

$$\cdots \xrightarrow{\delta_{n+1}} C_n \xrightarrow{\delta_n} C_{n-1} \xrightarrow{\delta_{n-1}} \cdots \xrightarrow{\delta_2} C_1 \xrightarrow{\delta_1} C_0 \xrightarrow{\delta_0} 0.$$

Here, each δ_n satisfies the key property that consecutive boundaries vanish, i.e., $\delta_n \circ \delta_{n+1} = 0$, meaning that the boundary of a boundary is always zero to make sure that every boundary encloses a well-defined cycle. Having $C(\mathcal{X})$, the n^{th} homology group is:

$$H_n(\mathcal{X}) = Ker(\delta_n)/Im(\delta_{n+1}),$$

where Ker(δ_n) indicates the number of cycles (i.e., elements that map to zero under δ_n), and Im(δ_{n+1}) indicates the boundaries (i.e., elements that originate from a higher-dimensional space). *Thus, $H_n(\mathcal{X})$ captures nontrivial cycles that are not boundaries, effectively identifying the holes in $\mathcal{X}$.* Thanks to the computational technique called *persistent homology*, $H_n(\mathcal{X})$ is calculated to indicate how topological features, such as connected components, loops, and voids, emerge and persist across multiple scales in ML and data analysis.

An informative measure that quantifies the number of *independent n-dimensional holes* in a topological space $\mathcal{X}$ is the n^{th} **Betti Number** defined as the rank of $H_n(\mathcal{X})$:

$$b_n = \text{rank}(H_n(\mathcal{X})). \tag{1}$$

For example, if our topological space $\mathcal{X}$ is a circle, we have $b_0 = 1$, because the space is connected, and $b_1 = 1$, because the space contains a single one-dimensional hole (i.e., a loop). In this case, all higher Betti Numbers vanish: $b_n = 0$ for all $n >= 2$, as a circle has no higher-dimensional holes.

Table 4. Illustration of Betti Numbers for some common geometrical shapes across different dimensions.

Betti Number	·			
b_0	**1**	**1**	**1**	**1**
b_1	0	**1**	0	**2**
b_2	0	0	**1**	**1**
b_3	0	0	0	0
:	:	:	:	:
b_n	0	0	0	0

For an n-dimensional sphere, $b_0 = 1$ indicates a single connected component, and $b_n = 1$, indicates the n-dimensional void, while all intermediate Betti Numbers are zero (see Table 4). Such a multi-scale perspective, measured through Betti Numbers, helps in identifying topological structures while filtering out noise-induced artifacts. When $\mathcal{X}$ is a torus, persistent homology effectively identifies its two distinct one-dimensional holes and its two-dimensional void; regardless of transformations such as scaling, rotation, or translation.

C. Implementation Details

Baselines. We compare *AdaBet* against five baselines:
(1) Full Training: all layers of the DNN parameters are updated using the local dataset.
(2) Vanilla Transfer Learning: only the final layer is updated, following standard transfer learning approaches [18, 35].
(3) Last-K-Layers: The last k layers are updated, where k is chosen based on ρ or the percent of layers to be selected.
(4) ElasticTrainer: selected layers are updated during re-training, where selection is done using dynamic programming across multiple rounds of backpropagation [16].
(5) PruneTrain: structured pruning is applied to the DNN during on-device training to reduce training cost [29].
(6) Fisher Information: layers are selected based on their Fisher Information scores, which are computed from activations and gradients obtained in the first backpropagation round [21]. We select $\rho = 0.1$ (i.e., top 10%) of layers based on FI scores for re-training; the same default for *AdaBet*.

Note that baseline (5) differs from TinyTrain [21], which relies on meta-training. To ensure a fair comparison, we do not compare our approach with server-based meta-training approaches [6, 21] or compute-intensive selection methods that require neural architecture search [26]. We believe these approaches, while valuable, are not practical for constrained devices requiring efficient and timely model re-training. Furthermore, the code for TinyTrain [21] is not available, making it challenging to reproduce their methodology.

DNN Architectures. We evaluate *AdaBet* across four widely-used DNNs: (i) Vision Transformers (ViT-B16) [11], which employs an attention mechanism adopted to vision tasks and consists of 12 Transformer Encoder blocks, (ii) ResNet50 [15], a deep model comprising 50 convolutional blocks with residual connections, (iii) VGG16 [40], which includes 13 convolutional blocks with small 3×3 convolution filters, and (iv) MobileNetV2 [39], a lightweight architecture with inverted residual blocks designed for mobile deployment. All DNNs are initialized with pre-trained weights from the ImageNet dataset. These four architectures were chosen to represent a broad range of design principles. For example, while ViT requires less memory than MobileNetV2, it suffers from slower on-device inference, which makes it more challenging for on-device machine learning [36].

Datasets. As the local dataset $\mathbb{D}$, we use three datasets: (i) Stanford Dogs [20] with 12,000 training and 8,580 testing images of 120 dog breeds, (ii) Oxford-IIIT Pets [33] with 3,680 training and 3,669 testing images from 37 categories of pets, and (iii) CUB [47] with 5,994 training and 5,794 testing images from 200 bird species. (iv) Flowers102 [31] with 1020 training and 6149 test images from 102 flower categories. To maintain consistency with our primary baseline ElasticTrainer [16], prior to selection and training, we resize all the input images to 224×224 and apply default pre-processing steps such as centering and random flipping.

Hyperparameters. For MobileNetV2 model on all datasets we use a learning rate of $1 \times e^{-4}$, for all models on CUB dataset we use a learning rate of $1 \times e^{-2}$, for all other models and datasets we use a learning rate of $1 \times e^{-3}$ and with the SGDW optimizer [28], a cosine decay momentum of 0.9, and a weight decay of 5×10^{-4}. For all *AdaBet* experiments, we set $\rho = 0.1$, unless otherwise specified. The batch size is set to 8 for all experiments, and the impact of batch size is separately analyzed in Section D.5.

Libraries and Hardware. We use TensorFlow 2.15 and TensorFlow Addons 0.23, building on the codebase of ElasticTrainer [16]. For topological computations, we employ Ripser 0.6.10 [3, 46], which provides fast and memory-efficient calculation of Betti Numbers. Memory computation and profiling are conducted using benchmarking code adapted from [6, 26]. Our training and evaluation experiments (§6.2–6.3), unless otherwise specified, are conducted on an NVIDIA Tesla V100 GPU with 16 GB of VRAM.

D. More Quantitative Results

D.1. The Impact of Normalization.

Betti numbers capture the topological characteristics of activations, reflecting a layer's learning capacity. However, without normalization, layers with a larger number of activations naturally yield higher Betti numbers, making di-

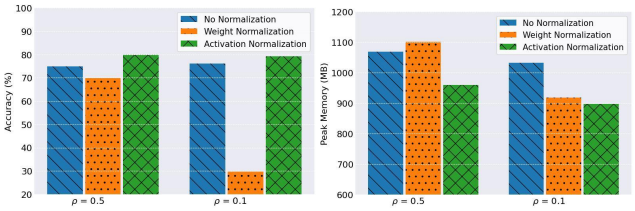

Figure D.8. (Left) accuracy and (right) peak memory consumption of *AdaBet* under three normalization strategies applied to Betti Numbers. Activation normalization (our choice) offers the best trade-off between accuracy and memory usage, particularly under tighter selection budgets ($\rho = 0.1$).

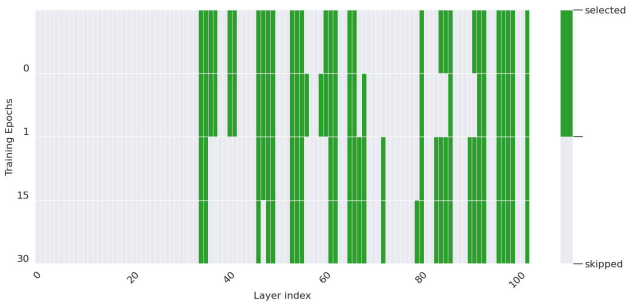

Figure D.9. Layers selected by *AdaBet* from a model before any training updates (epoch 0), from a partially trained model (epochs 1 and 15), and a trained model (epoch 30)

rect comparisons misleading. As discussed before, we normalize the Betti Numbers before layer selection to address this bias (see §4.2.3). Here, we compare our normalization strategy with two other alternatives: (i) no normalization, and (ii) normalization by the number of trainable parameters (weight size).

Figure D.8 compares the accuracy (left) and peak memory consumption (right) of ResNet50 across these normalization strategies for two values of ρ on Stanford Dogs dataset. Normalizing by activation size achieves the best trade-off, leading to higher accuracy and lower memory consumption. In contrast, weight normalization degrades accuracy, particularly at $\rho = 0.1$, while increasing peak memory usage. Activation normalization improves the performance, confirming its suitability for efficient layer selection.

D.2. Selection across Training Epochs.

Figure D.9, illustrates the dynamics of layer selection by *AdaBet* across different training stages on ResNet50 and Oxford-IIIT Pets. At epoch 0 (before any training updates), *AdaBet* already shows a selection pattern similar to epoch 1 and epoch 15. The distribution of selected layers shows that certain layers are consistently selected while others are skipped, indicating that *AdaBet* consistently keeps the focus on layers that contribute more effectively to learning.

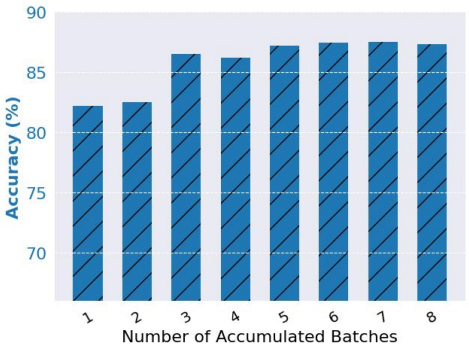

Figure D.10. Performance of *AdaBet* using VGG16 with increasing number of data batches (each of size 8) accumulated during selection on Oxford-IIIT Pets dataset.

D.3. Impact of Batch Accumulation during Selection.

To counter Betti Number's dependence on the volume of data used for its computation, *AdaBet* uses activation aggregation (see Section 4.2.1). To find the ideal number of data batches required for *AdaBet*, we run an ablation study as shown in Figure D.10. We observe that as the number of batches increases, the performance increases until activations from 5 batches are accumulated for Betti computation and layer selection. Beyond 5, the performance remained almost the same with consistent layer selection. This shows that *AdaBet* requires an aggregated equivalent of 5 batches, each with a batch size of 8, to capture robust layer importance and select layers to achieve optimal performance.

D.4. Convergence

Training dynamics are shown in Figure D.11, with test accuracy (left plot) and training loss curves (right plot) over 50 training epochs of ResNet50 on the Oxford-IIIT Pets dataset. ElasticTrainer shows the fastest convergence and ultimately matches the final accuracy achieved by *AdaBet*, while Full Training converges slowly and to a lower accuracy. Transfer Learning, which only updates the final layers, converges slowly and yields the lowest accuracy, highlighting the importance of informed layer selection. Fisher Information performs just better than Transfer Learning.

Although ElasticTrainer initially converges quickly for training loss, *AdaBet* achieves a lower final loss, indicating a better fit. In contrast, Full Training yields higher loss values, and Transfer Learning shows the slowest reduction with the highest final loss.

Overall, the training dynamics of *AdaBet* demonstrate a balance between fast convergence, lower loss, and competitive accuracy.

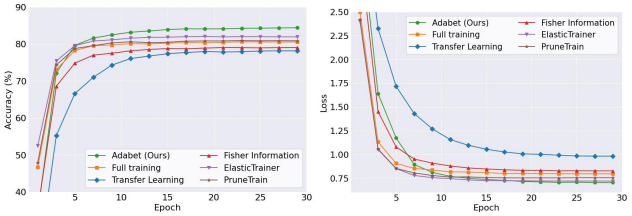

Figure D.11. (Left) classification accuracy on test set and (right) training loss of ResNet50 with Flowers102 dataset across baselines and *AdaBet* with $\rho = 0.1$.

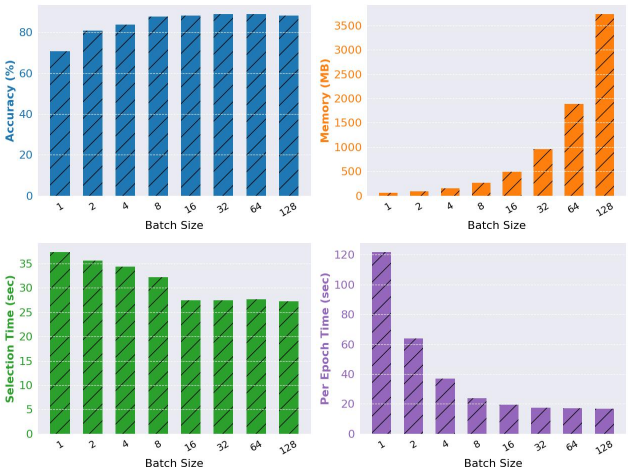

Figure D.12. Accuracy, peak memory consumption, selection time, and training time per epoch of ResNet50 in *AdaBet* with varying batch size.

D.5. Impact of Batch Size

Figure D.12 presents a performance study of *AdaBet* under varying batch sizes, evaluating its effect on accuracy, memory usage, selection time, and per-epoch training time. We observe that *AdaBet* maintains high accuracy across all configurations, with only a slight drop as batch size decreases beyond 8. This stability underscores its robustness to batch size variation—an important trait for practical deployment. Memory consumption, as expected, grows significantly with batch size, with a sharp increase beyond 8. This reinforces the importance of using smaller batch sizes for on-device scenarios, where memory resources are typically constrained.

Interestingly, while Betti Number estimations are generally expected to benefit from larger batch sizes due to better topological representation, we do not observe such a trend here. This is attributed to the accumulation strategy employed in *AdaBet* (see § 4.2.1), wherein activations from multiple smaller batches are aggregated to form an adequate batch size of 40 for layer selection. This allows us to retain the topological richness of larger batches while remaining within device constraints. Notably, the retraining phase continues to use the original smaller batch size.

Selection time across batch sizes decreases steadily until batch size 16, after which the accumulation strategy considers the input batch size as such to form an effective batch of size approximately 40 for selection. Beyond a batch size 16, the accumulation strategy fixes it at 16 to counter the memory consumption with increasing batch size. This effect can be observed at the constant selection time beyond batch size 16. Meanwhile, the time per training epoch decreases steadily with increasing batch size, following standard mini-batch processing behavior. Overall, these results validate that *AdaBet* is compatible with small-batch operation and performs reliably under varying resource budgets, with minimal accuracy degradation and strong efficiency characteristics.

D.6. Impact of ρ and ρ_{ch} on Time Efficiency.

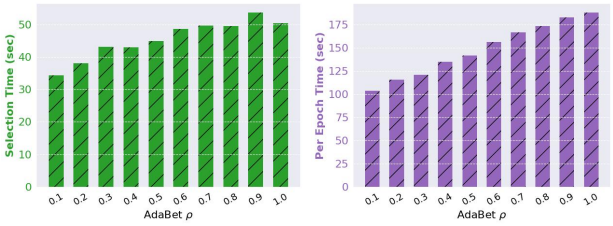

Figure D.13. Selection time and Time per epoch of ResNet50 in *AdaBet* with varying ρ on Stanford Dogs datasets.

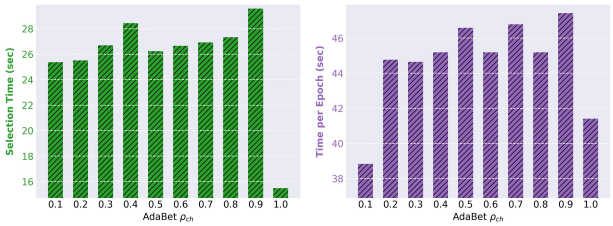

Figure D.14. Selection time and Time per epoch of ResNet50 in *AdaBet* with varying ρ_{ch} on Stanford Dogs datasets.

In *AdaBet*, ρ and ρ_{ch} acts as a hyperparameters that choose the proportion layers and channels per layer to be selected for training from the model. This parameter provides a trade-off between performance and peak memory consumption of the model. In this section, we are looking at the time efficiency brought about by *AdaBet* with varying ρ and ρ_{ch} in addition to peak memory and accuracy as shown in **??** and 7.

In terms of time, as ρ decreases, *AdaBet* selects and trains a smaller subset of the model layers, leading to an approximately 30% reduction in selection time and

approximately 45% reduction in training time per epoch for ResNet50, relative to full-model training as shown in Fig.D.13. However, the decrease in time is less prominent with channel selection. As ρ_{ch} decreases, a smaller subset of layers are selected per layer that was selected in the previous step. A $\rho_{ch} = 1$ indicate no layer selection. As shown in Fig.D.14, we observe that this consumes least time as expected and with decreasing ρ_{ch}, the selection time decreases by about 15%. Time per epoch decreases by about 6% as ρ_{ch} decreases from 0.9 to 0.2 with a more significant drop and ρ_{ch} is 0.1.

D.7. Controllability of ρ

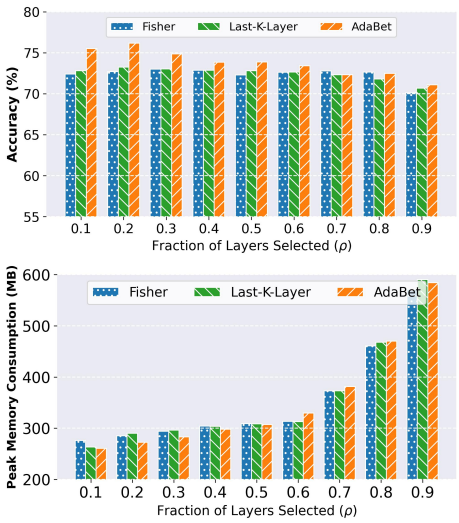

Figure D.15. Accuracy(%) and Peak Memory consumption (MB) of layer selection baselines with ResNet50 on Stanford Dogs.

For the same ρ, last-k-layer training requires comparable or slightly more memory yet consistently underperforms in accuracy, showing that heuristic selections discards learning utility, potentially never updating the key layers responsible for downstream feature extraction. Although accuracy is not strictly monotonic in ρ, experiments demonstrate that *AdaBet* never exhibits the failure mode of spending more memory for worse accuracy. Across all ρ, it consistently dominates last-k-layer and Fisher Information-based selection at equal or lower peak memory, defining a strictly better accuracy–memory frontier E.16. Once ρ is fixed, AdaBet produces a tightly bounded peak-memory footprint that remains well below full fine-tuning while exhibiting better accuracy.

E. On-Device Results

For on-device CPU based results we run *AdaBet* on a 2GB Raspberry Pi4. We observe performance, memory consumption, and time consumption similar to that of

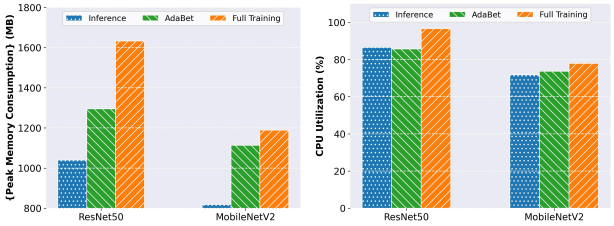

Figure E.16. Peak Memory consumption (MB) and CPU utilization (%) of ResNet50 and MobileNetV2 on Flowers102 when run on a Raspberry Pi4.

GPU based results. For the Pi based on-device results, we record peak memory consumption and CPU utilization for full training, *AdaBet*, and inference on ResNet50 and MobilenetV2 - two state-of-the-art architectures for image recognition - on Flowers102 dataset for 102 class image classifiaction task with a batch size of 2, $\rho = 0.1$ and $\rho_{ch} = 1.0$. All other hyperparameters were maintained the same as GPU as mentioned in 5. Due to limited compute availability on Raspberry Pi4, we were unable to run any methods on ViT. VGG16 was slow on a Raspberry Pi and we also observed minimal memory or cpu utilization gains. Unlike memory consumption recorded in GPU using methods similar to that in [26], we use total memory consumed by all process run on the CPU (measured using the library "psutils"). Although closer to what is required in real-world deployments, this is often influenced by other CPU processes that may be independent of the training or inference tasks. Similar influence can be expected on CPU utilization. We do not isolate the training process for the purposes of these results.

Fig. E.16 shows the peak memory consumption and CPU utilization of the models. As shown, we observe a 20.6% and 6.2% reduction in peak memory consumption on *AdaBet* in comparison to full training on ResNet50 and MobileNetV2 respectively. In terms of CPU utilization, we observe a reduction of 11.3% and 7.9% respectively for ResNet50 and MobileNetV2 in comparison to full training. **These results demonstrate that *AdaBet* can be effectively deployed on CPU based devices for efficient DNN tasks.**