Memory-Efficient Neural Network Training for Wearable Devices

Maxim Nitsenko

***Abstract*—Neural network training on wearable devices is severely constrained by prohibitive memory requirements. This project implements five complementary memory optimization techniques to enable on-device training for health monitoring applications. The processing pipeline combines dynamic sparse training, gradient checkpointing, microbatching, efficient architectures, and patch-wise training to systematically reduce memory footprint. Evaluated on a 17,092-image blood cell classification dataset, this approach achieves an 82x peak memory reduction (from 9.3 GB to 113 MB) with only 3.3% accuracy loss compared to baseline ResNet-50 performance.**

I. Introduction

The rapid advancement of deep learning has produced increasingly sophisticated neural network architectures, but at the cost of extreme parameter counts and high memory demands during training. While most research targets memory reduction for inference, the much larger requirements of training remain a critical bottleneck [1]. Today, such training is almost exclusively performed in large datacenters with specialized GPUs or TPUs, where abundant memory and compute resources are available. These constraints are especially severe for highly overparameterized models in domains like computer vision. Large memory footprints often force practitioners to downsample inputs or use smaller models, sacrificing accuracy. The challenge intensifies with body-worn devices such as smartwatches, heart-rate monitors, and smart glasses, which operate under strict limits on memory, computation, and battery life. Traditional training methods are infeasible in these environments. On-device training for such platforms offers key advantages: personalization, reduced data transfer costs, and complete data privacy since sensitive information remains local. This is particularly important in health monitoring applications, where model customization and privacy is essential. This work develops and evaluates five practical techniques for low-memory neural network training: gradient checkpointing, efficient mobile-oriented architectures, model sparsity, microbatching, and patch training. Combined, these approaches achieve an **82x reduction in peak memory** usage with only a **3.3% accuracy drop**, demonstrating the feasibility of on-device learning for wearable applications. By addressing the training memory bottleneck, this project enables advanced machine learning capabilities to run directly on resource-constrained health and fitness devices.

II. Related Work

This project is primarily built around Dynamic Sparse Training (DST), the most extensive and time-consuming part of the work, leveraging recent advances in sparse neural network training to maintain performance while greatly reducing computational requirements. Dynamic Sparse Training has evolved through several key methods.

Sparse Evolutionary Training (SET) [2] introduced training neural networks with dynamic sparsity from scratch by replacing fully connected layers with sparse layers following an ER topology. Connectivity evolves through pruning small-magnitude weights and randomly regrowing connections. While SET achieves substantial parameter reductions with competitive accuracy, it suffers from unprincipled regrowth and no cross-layer weight redistribution, leading to inefficiencies in deeper networks.

Dynamic Sparse Reparameterization (DSR) [3] addressed these issues via adaptive global thresholding and cross-layer parameter redistribution proportional to gradient magnitudes, allowing layers with larger gradients to acquire more weights. This improves deeper network performance but retains two constraints: some layers must remain dense, and regrowth still depends on random selection.

Sparse Momentum [4] uses exponentially smoothed gradients (momentum) to guide both pruning and regrowth, prioritizing weights that consistently contribute to error reduction. It supports fully sparse training and redistributes weights across layers based on mean momentum magnitude, enabling significant training speedups-up to 5.61x while matching dense network performance. In experimental evaluations on MNIST, CIFAR-10, and ImageNet, Sparse Momentum consistently outperforms prior methods, including SET and DSR, in both accuracy and efficiency across a range of sparsity levels.

This work builds primarily on Sparse Momentum but integrates two design choices from DSR: (1) within-layer regrowth uses random selection instead of momentum-guided regrowth to reduce memory cost, and (2) normalization layers remain dense for training stability.

III. Processing Pipeline

This section presents a sequence of techniques designed to reduce memory consumption during neural network training while preserving good performance. Starting from a ResNet-50 baseline, a series of cumulative memory optimization strategies was implemented. Each subsequent model configuration includes all preceding optimizations, resulting in six distinct configurations (C1-C6).

Baseline: ResNet-50 (C1)

Standard supervised training using the ResNet-50 architecture on input images of size 224×224, without any memory optimization techniques.

+ Dynamic Sparse Training (C2)

The baseline model is enhanced with Dynamic Sparse Training (DST) which trains a sparse network, maintaining a fixed number of active parameters by pruning low-importance weights and regrowing new ones.

+ Gradient Checkpointing (C3)

Gradient checkpointing is added to the sparse model, further reducing peak memory usage. This technique trades computational overhead for memory savings by recomputing intermediate activations during the backward pass instead of storing them.

+ Micro-Batching (C4)

To further reduce memory usage, micro-batching with gradient accumulation is employed. Smaller sub-batches are processed sequentially, and their gradients are accumulated before performing a parameter update. This enables training with effectively larger batch sizes without exceeding memory constraints.

MobileNet Backbone + Prior Optimizations (C5)

To achieve additional memory savings, the ResNet-50 backbone is replaced with the MobileNet architecture, while retaining all prior optimizations (DSR, checkpointing, and micro-batching). MobileNet employs depthwise separable convolutions, which reduce both parameter count and activation size.

+ Patch-wise Training (C6)

Finally, instead of using full-resolution 224×224 images, each image is divided into smaller 32×32 patches. These patches are used to train an auxiliary lightweight MobileNet model, which incorporates all previous memory optimizations.

IV. Signals and Features

The dataset [5] used in this study consists of 17,092 microscopic peripheral blood cell images originally acquired using the CellaVision DM96 analyzer at the Hospital Clinic of Barcelona. The images were resized to 224×224 pixels in RGB format and represent eight different types of normal peripheral blood cells: neutrophils (19.48%), eosinophils (18.24%), basophils (7.13%), lymphocytes (7.1%), monocytes (8.31%), immature granulocytes (including promyelocytes, myelocytes, and metamyelocytes) (16.94%), erythroblasts (9.07%), and platelets (13.74%). All images were annotated by expert clinical pathologists and collected from individuals without infection, hematologic or oncologic diseases, and free of pharmacologic treatment at the time of blood collection.

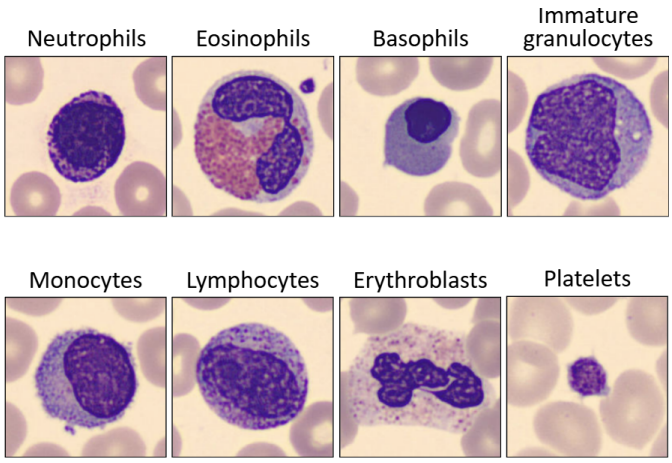

Fig. 1: Example images of the eight peripheral blood cell classes in the dataset.

V. Learning Framework

A. Types of Memory

The primary sources of memory consumption during the training of a neural network can be categorized as follows:

1) Model Memory: Model memory refers to the memory allocated for storing the model parameters, including the weights and biases of each layer. As illustrated in Figure 2, model memory constitutes the smallest portion of the overall memory usage among the three main categories discussed.

2) Optimizer Memory: Optimizer memory corresponds to the memory required for storing gradients and any associated momentum buffers during training. The optimizer memory usage ranges between two to three times that of model memory, depending on the optimizer employed. For example, using SGD with Nesterov momentum results in optimizer memory approximately twice the size of the model memory, due to the need for one buffer to store gradients and an additional buffer for momentum values.

3) Activation Memory: Activation memory includes the storage of intermediate outputs (activations) produced during the forward pass of the network. These activations, including the input data, must be retained because they are required to compute gradients during the backward pass of backpropagation. As shown in Figure 2, activation memory accounts for the largest portion of memory consumption during standard training.

4) Other Memory Contributions: In addition to the primary sources described above, other factors contribute to the overall memory footprint during training. These include memory utilized by the deep learning framework for internal operations, such as the computational graph used for automatic differentiation, temporary buffers for intermediate computations, and overhead associated with parallelism or device communication. Nonetheless, these contributions are minimal compared to model, optimizer, and activation memory.

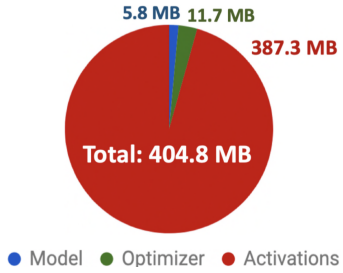

Fig. 2: Memory usage breakdown of training memory consumption for WideResNet on CIFAR-10. Adapted from [1].

B. Memory Reduction Techniques and Tradeoffs

This section reviews five complementary techniques for reducing model, optimizer, and activation memory, each with specific tradeoffs in accuracy and computation time (see Table 1).

1) Dynamic Sparse Training: Modern deep neural networks are typically highly overparameterized. A common approach to reduce this redundancy is the dense-to-sparse paradigm, also known as network pruning. In this approach, a dense model is trained first, and then unimportant weights are removed to obtain a sparsely connected network. While this can reduce memory usage and improve inference efficiency, it offers no memory savings during training.

An alternative approach, sparse-to-sparse training, has recently gained attention. This method trains sparse models from scratch, aiming to improve both memory and computational efficiency throughout training. This project implementation is based on Dynamic Sparse Traing [6], where the sparse structure of the network is continuously updated during training. Layers are initialized with some sparsity, say 80%, then at scheduled intervals, the algorithm prunes weights with the smallest magnitudes and regrows an equal number of new connections, maintaining a constant number of non-zero weights throughout training. Dynamic sparse training is applied to all trainable weights, excluding batch normalization parameters and biases. To guide weight redistribution during regrowth, the efficiency of each layer is estimated using the mean magnitude of the momentum of its weights. Layers with lower momentum are considered less effective and donate capacity to more efficient layers. This adaptive allocation enables the network to focus resources where they are most useful. The prune rate is gradually decayed over time to stabilize training. See Algorithm 1.

While sparsity reduces memory usage for both the model and optimizer by pruning parameters, activations typically remain dense, limiting the overall memory benefits. Additionally, the theoretical reduction in floating-point operations (FLOPs) does not always lead to faster execution in practice. The actual speedups depend heavily on both hardware and software support for sparse computation a limitation that is present in TensorFlow, which lacks native support for sparse convolutions which are the most frequent and computationally expensive operations in modern vision architectures. Despite experimenting with several sparse 2D convolution implementations, it was challenging to outperform highly optimized dense convolutions. Therefore, sparse tensors need to be converted back to dense format to perform convolutional operations efficiently. This conversion introduces additional overhead and negates the intended performance benefits.

Algorithm 1 Prune and Regrow Logic

Require:
Model layers $\mathcal{W} = \{W_1, W_2, \ldots, W_n\}$
Global sparsity s
Pruning ratio $\rho \in (0, 1)$

Pruning Phase:
total_pruned $\leftarrow 0$
for each layer $W_i \in \mathcal{W}$ **do**
  $p_i \leftarrow \lceil \rho \times \|W_i\|_0 \rceil$
  Remove p_i smallest-magnitude weights in W_i
  total_pruned $\leftarrow$ total_pruned $+ p_i$
end for
Regrowth Phase:
Retrieve momentum: $M_i \leftarrow \beta M_i + (1-\beta)\nabla W_i$ for all i
$\bar{m}_i \leftarrow \|M_i\|_1 / \sum_j \|M_j\|_1$ ▷ normalized average momentum
for each layer $W_i \in \mathcal{W}$ **do**
  $r_i \leftarrow \lfloor \bar{m}_i \times$ total_pruned$\rfloor$
  Initialize random r_i new wights in W_i
end for
Decay pruning ratio: $\rho \leftarrow \rho/2$

Note: Some implementation edge cases are omitted for clarity. These include: (i) attempting to regrow more weights than there are available zero positions in a layer, (ii) imperfect redistribution due to fractional allocations that require an additional correction step, and (iii) pruning rates becoming too small to have an effect.

2) Gradient Checkpointing: Gradient checkpointing [7] is a memory-saving technique that reduces the need to store all intermediate activations during the forward pass of a neural network. Instead of keeping the activations of every layer, it stores only a subset, known as checkpoints. During backpropagation, the discarded activations are recomputed as needed by re-running parts of the forward pass.

One of the main advantages of gradient checkpointing is that it does not alter the outcome of training. The forward and backward passes remain mathematically equivalent to standard training, so there is no tradeoff between memory usage and model accuracy.

This technique targets activation memory and can, in principle, reduce it to near zero if applied extensively. The tradeoff is increased computation time, which grows with the number of discarded activations that must be recomputed.

3) Microbatching: Reducing memory usage during training is often necessary, and one common approach is to reduce the minibatch size, since the number of stored activations scales proportionally with it. However, decreasing the batch size can significantly change the training dynamics and may hurt performance, often requiring careful hyperparameter tuning.

A better alternative in many cases is microbatching [8]. Here, the original minibatch is split into several smaller microbatches. Each microbatch is processed independently through the forward and backward passes, and gradients are accumulated across microbatches. After all microbatches have been processed, the accumulated gradients are used to update the model parameters.

If the model does not rely on inter-example operations, this procedure is mathematically equivalent to using the full batch. For models with operations like batch normalization, which compute statistics across the batch dimension, the equivalence breaks down, since each microbatch computes its own normalization statistics. This can introduce small differences in training behavior, particularly for very small microbatches. In practice, when the total batch size remains above 10, the final accuracy is usually unaffected [3].

Microbatching reduces peak activation memory approximately in proportion to the microbatch size, at the cost of increased computation time. It allows training large models within memory constraints while preserving the benefits of larger effective batch sizes.

4) Efficient Architectures: Efficient neural network architectures are a major research focus aimed at making deep learning viable on resource-constrained devices such as smartphones, embedded systems, and edge hardware. These models are designed to reduce memory usage, parameter count, and computational cost, while maintaining competitive accuracy.

Architectures like MobileNet, SqueezeNet, and ShuffleNet exemplify this trend. By prioritizing compactness and speed, they enable real-time inference and low-latency applications in environments where traditional models are too large or computationally intensive. These designs generally reduce all forms of memory consumption, including model size, activation memory, and intermediate buffers.

The main tradeoff is a potential drop in accuracy, which is usually modest. At the same time, compute time is typically reduced, making these models well-suited for real-time deployment in constrained environments.

5) Patching: Patching is a technique used to reduce memory usage during training by dividing a high-resolution input image (e.g. 224×224) into smaller, non-overlapping sub-images or patches (e.g. 32×32). Each patch is treated as an independent training example, allowing the model to process smaller inputs and lower memory consumption. During inference, predictions from all patches need to be aggregated to form a single overall classification for the image.

This method requires architectural modifications. A standard model designed for full-sized images cannot be used directly, as it must be adapted to operate on smaller, patch-sized inputs.

Unlike downsampling, patching does not discard information. The full content of the original image is preserved across the set of patches. Only the spatial structure is fragmented. However, since patches are processed independently, the model loses access to the global context of the full image. This can degrade performance by limiting the model's ability to capture long-range dependencies. Patching also increases computational cost. More forward and backward passes are needed per full image, as each patch is treated separately. Despite these tradeoffs, patching is an effective way to reduce memory usage during training without losing image content.

C. Model Architectures

The following models were selected to evaluate memory-efficient training under strict device constraints. ResNet-50 serves as the baseline for full-image training. MobileNet introduces architectural efficiency via depthwise separable convolutions. Finally, a patched MobileNet variant is used for 32x32 input patches, enabling great memory reduction by processing small image regions instead of full inputs.

1) Resnet-50: Resnet-50 introduced in [9] is a deep convolutional neural network based on residual learning, designed to ease the training of very deep architectures. The key innovation of ResNet-50 is the use of residual connections or "skip connections" that allow gradients to flow directly through shortcut paths, enabling the training of much deeper networks than previously possible. This design allows the network to learn residual mappings rather than direct mappings, making it easier to optimize and enabling the construction of networks with over 100 layers without degradation in performance.

The network begins with an initial 7×7 convolution with stride 2, followed by batch normalization, ReLU activation, and 3×3 max pooling with stride 2. This initial processing reduces the spatial dimensions while extracting low-level features from the input.

ResNet-50 then consists of four main stages (or "stacks") of residual blocks with increasing depth and feature map channels. The blocks are organized in a 3-4-6-3 configuration across the four stages, totaling 16 residual blocks. Each stage operates at progressively smaller spatial resolutions (56×56, 28×28, 14×14, and 7×7 for 224×224 input) while increasing the number of feature channels (64, 128, 256, and 512 respectively).

Each residual block uses a bottleneck design consisting of three consecutive convolutions:

1. A 1×1 convolution that reduces the number of channels.
2. A 3×3 convolution that performs the main feature extraction.
3. A 1×1 convolution that expands the channels back to the original or desired width.

The first two convolutions are followed by batch normalization and ReLU activation. The critical component is the shortcut connection that bypasses these three convolutions, allowing the input to be added directly to the output of the third convolution. When the input and output dimensions don't match (typically at the beginning of each stage), a 1×1 convolution with batch normalization is applied to the shortcut path to match the dimensions. The final ReLU activation is applied after adding the shortcut connection to the main path output.

The network concludes with global average pooling, which reduces each feature map to a single value, followed by a fully

TABLE 1: Summary of memory reduction techniques and trade-offs

		Sparsity	Microbatching	Checkpointing	MobileNet	Patching
Memory	Model	↓			↓	↓
	Optimizer	↓			↓	↓
	Activation		↓	↓	↓	↓
Computation Time		↑	↑	↑	↓	↑
Accuracy		↓			↓	↓

connected layer with softmax activation for classification. This structure enables ResNet-50 to reach 50 layers (hence the name) while maintaining computational efficiency and stable optimization through the residual learning framework.

2) MobileNet: MobileNet [10] is an efficient convolutional neural network specifically designed for mobile and embedded vision applications where computational resources and power consumption are limited. The key feature of MobileNet is the use of depthwise separable convolutions, which dramatically reduce computational cost and model size while maintaining reasonable accuracy.

The core building block of MobileNet is the depthwise separable convolution, which factorizes a standard convolution into two separate operations:

1. Depthwise convolution: Applies a single 3×3 filter to each input channel independently, performing spatial filtering without mixing channels.
2. Pointwise convolution: Uses 1×1 convolutions to linearly combine the outputs from the depthwise convolution, creating new feature representations.

Each depthwise separable convolution block follows a specific structure:

1. 3×3 depthwise convolution
2. Batch normalization
3. ReLU activation
4. 1×1 pointwise convolution
5. Batch normalization
6. ReLU activation

This factorization reduces both computational complexity and the number of parameters significantly. A standard convolution requires

$$K \times K \times C_{in} \times C_{out} \times D \times D$$

operations, where

- K is the kernel size (e.g., 3 for a 3×3 kernel),
- D is the spatial width and height of the feature map,
- C_{in} is the number of input channels,
- C_{out} is the number of output channels.

In contrast, depthwise separable convolutions only requires

$$K \times K \times C_{in} \times D \times D + C_{in} \times C_{out} \times D \times D$$

operations. The parameter count follow a similar pattern. A standard convolution has

$$K \times K \times C_{in} \times C_{out}$$

parameters, while a depthwise separable convolution has

$$K \times K \times C_{in} + C_{in} \times C_{out}$$

The ratio of parameters (and computational cost) between depthwise separable and standard convolutions is

$$\frac{1}{C_{out}} + \frac{1}{K^2}$$

For 3×3 kernels, the ratio approaches $\frac{1}{9} \approx 0.11$ for large C_{out}, resulting in approximately 8-9 times fewer computations and parameters.

The MobileNet architecture begins with a standard 3×3 convolution followed by 13 depthwise separable convolution blocks. The network progressively reduces spatial dimensions:

$$224\times224 \rightarrow 112\times112 \rightarrow 56\times56 \rightarrow 28\times28 \rightarrow 14\times14 \rightarrow 7\times7$$

while increasing the number of feature channels:

$$32 \rightarrow 64 \rightarrow 128 \rightarrow 256 \rightarrow 512 \rightarrow 1024$$

Downsampling is handled through strided convolutions in both the initial layer and specific depthwise convolution layers. The architecture concludes with global average pooling and a fully connected layer with softmax activation.

3) Patched MobileNet: This variant is specifically optimized for 32×32 iput resolutions by reducing the number of spatial downsampling steps and adjusting the network depth accordingly. Compared to the standard MobileNet (designed for 224×224 inputs), the patched version employs only three downsampling steps ($32\rightarrow16\rightarrow8\rightarrow4$) instead of five ($224\rightarrow112\rightarrow56\rightarrow28\rightarrow14\rightarrow7$), preventing excessive feature map reduction that could degrade performance on low-resolution inputs. To maintain efficiency, the architecture omits the five repeated middle blocks present in the original model, resulting in a shallower network (9 layers vs. 14) while retaining the same 1024-channel final representation before classification. The final feature map resolution is reduced to 4×4 (vs. 7×7 in the standard version), ensuring sufficient spatial context is preserved for smaller inputs. These modifications preserve the computational benefits of depthwise separable convolutions while better adapting the network to lower-resolution tasks.

D. Experimental Setup

All experiments were performed on an NVIDIA L4 GPU with 24 GB of memory.

TABLE 2: ResNet-50 architecture

Layer	Input Size	Output Size	Filter Shape; Stride
Convolution	$224 \times 224 \times 3$	$112 \times 112 \times 64$	$7, 7, 3, 64; 2$
Max Pool	$112 \times 112 \times 64$	$56 \times 56 \times 64$	$3, 3; 2$
Residual Stack 1	$56 \times 56 \times 64$	$56 \times 56 \times 256$	$\begin{bmatrix} 1, 1, 64 \vert 256, 64; 1 \\ 3, 3, 64, 64; 1 \\ 1, 1, 64, 256; 1 \end{bmatrix} \times 3$
Residual Stack 2	$56 \times 56 \times 256$	$28 \times 28 \times 512$	$\begin{bmatrix} 1, 1, 256 \vert 512, 128; 2 \vert 1 \\ 3, 3, 128, 128; 1 \\ 1, 1, 128, 512; 1 \end{bmatrix} \times 4$
Residual Stack 3	$28 \times 28 \times 512$	$14 \times 14 \times 1024$	$\begin{bmatrix} 1, 1, 512 \vert 1024, 256; 2 \vert 1 \\ 3, 3, 256, 256; 1 \\ 1, 1, 256, 1024; 1 \end{bmatrix} \times 6$
Residual Stack 4	$14 \times 14 \times 1024$	$7 \times 7 \times 2048$	$\begin{bmatrix} 1, 1, 1024 \vert 2048, 512; 2 \vert 1 \\ 3, 3, 512, 512; 1 \\ 1, 1, 512, 2048; 1 \end{bmatrix} \times 3$
Average Pool	$7 \times 7 \times 2048$	2048	$7, 7$
Fully Connected	2048	8	$2048, 8$

TABLE 3: MobileNet architecture

Layer	Input Size	Output Size	Filter Shape; Stride
Convolution	$224 \times 224 \times 3$	$112 \times 112 \times 32$	$3, 3, 3, 32; 2$
Depthwise Separable 1	$112 \times 112 \times 32$	$112 \times 112 \times 64$	$3, 3, 32, 1$ dw; 1 $1, 1, 32, 64; 1$
Depthwise Separable 2	$112 \times 112 \times 64$	$56 \times 56 \times 128$	$3, 3, 64, 1$ dw; 2 $1, 1, 64, 128; 1$
Depthwise Separable 3	$56 \times 56 \times 128$	$56 \times 56 \times 128$	$3, 3, 128, 1$ dw; 1 $1, 1, 128, 128; 1$
Depthwise Separable 4	$56 \times 56 \times 128$	$28 \times 28 \times 256$	$3, 3, 128, 1$ dw; 2 $1, 1, 128, 256; 1$
Depthwise Separable 5	$28 \times 28 \times 256$	$28 \times 28 \times 256$	$3, 3, 256, 1$ dw; 1 $1, 1, 256, 256; 1$
Depthwise Separable 6	$28 \times 28 \times 256$	$14 \times 14 \times 512$	$3, 3, 256, 1$ dw; 2 $1, 1, 256, 512; 1$
Depthwise Separable 7–11	$14 \times 14 \times 512$	$14 \times 14 \times 512$	$\begin{bmatrix} 3, 3, 512, 1 \text{ dw}; 1 \\ 1, 1, 512, 512; 1 \end{bmatrix} \times 5$
Depthwise Separable 12	$14 \times 14 \times 512$	$7 \times 7 \times 1024$	$3, 3, 512, 1$ dw; 2 $1, 1, 512, 1024; 1$
Depthwise Separable 13	$7 \times 7 \times 1024$	$7 \times 7 \times 1024$	$3, 3, 1024, 1,$ dw; 1 $1, 1, 1024, 1024; 1$
Average Pool	$7 \times 7 \times 1024$	1024	$7, 7$
Fully Connected	1024	8	$1024, 8$

The dataset was divided into 11,959 training images, 1,712 validation images, and 3,421 test images. No preprocessing beyond pixel-value normalization to the range [0,1] was applied, in order to isolate the effects of memory-efficient training methods.

Because the dataset is only moderately imbalanced and the focus of this study is memory efficiency rather than class-specific performance, classification accuracy was adopted as the sole evaluation metric.

Each configuration was trained for a fixed duration of 2 hours to ensure comparability. The maximum number of epochs and iterations was set sufficiently high such that training terminated after the 2-hour mark. Testing time was excluded from this budget. During training, test accuracy was sampled every 2.5 minutes, and the best value observed up to that point was recorded. To reduce variance, each experiment was repeated with four independent random seeds, and reported results are means with standard deviations across these runs.

At fixed intervals, the average loss over the preceding

TABLE 4: Patched MobileNet architecture

Layer	Input Size	Output Size	Filter Shape; Stride
Convolution	$32 \times 32 \times 3$	$32 \times 32 \times 32$	$3, 3, 3, 32; 1$
Depthwise Separable 1	$32 \times 32 \times 32$	$32 \times 32 \times 64$	$3, 3, 32, 1$ dw; 1 $1, 1, 32, 64; 1$
Depthwise Separable 2	$32 \times 32 \times 64$	$16 \times 16 \times 128$	$3, 3, 64, 1$ dw; 2 $1, 1, 64, 128; 1$
Depthwise Separable 3	$16 \times 16 \times 128$	$16 \times 16 \times 128$	$3, 3, 128, 1$ dw; 1 $1, 1, 128, 128; 1$
Depthwise Separable 4	$16 \times 16 \times 128$	$8 \times 8 \times 256$	$3, 3, 128, 1$ dw; 2 $1, 1, 128, 256; 1$
Depthwise Separable 5	$8 \times 8 \times 256$	$8 \times 8 \times 256$	$3, 3, 256, 1$ dw; 1 $1, 1, 256, 256; 1$
Depthwise Separable 6	$8 \times 8 \times 256$	$4 \times 4 \times 512$	$3, 3, 256, 1$ dw; 2 $1, 1, 256, 512; 1$
Depthwise Separable 7	$4 \times 4 \times 512$	$4 \times 4 \times 1024$	$3, 3, 512, 1$ dw; 1 $1, 1, 512, 1024; 1$
Depthwise Separable 8	$4 \times 4 \times 1024$	$4 \times 4 \times 1024$	$3, 3, 1024, 1$ dw; 1 $1, 1, 1024, 1024; 1$
Average Pool	$4 \times 4 \times 1024$	1024	$4, 4$
Fully Connected	1024	8	$1024, 8$

interval was computed. If this average failed to improve for a specified number of tests (patience), the learning rate was halved.

All hyperparameters, including prune-and-regrow frequency T, initial learning rate, patience, patch size, and stride, were selected using the validation set.

The Adam optimizer was employed with a batch size of 32 and a microbatch size of 16. Sparsity was fixed at 80%.

In the patched variant (C6), each image was divided into non-overlapping 32×32 patches (stride = 32), retaining only those containing violet-stained cellular material for training. During testing, patches were extracted in the same manner but with a smaller stride of 8 to allow overlap. For patch-wise inference, two aggregation methods were evaluated: violet-pixel-based weighting and uniform weighting. Contrary to the initial hypothesis, uniform weighting yielded higher accuracy, indicating that contextual information from all patches contributed to the final classification.

For gradient checkpointing, all intermediate activations were recomputed to maximize memory reduction.

VI. Results

Six configurations (C1–C6) were evaluated to quantify the trade-off between training accuracy, memory usage, and training speed under a fixed 2-hour budget. Results are reported in Tables 5 and 6.

The baseline (C1) achieved 98.5±0.2% accuracy with a peak memory usage of 9337.8 MB. Dynamic Sparse Reparameterization (C2) reduced memory to 8678.3 MB (1.08× reduction) with a 0.9% accuracy drop. Gradient checkpointing (C3) lowered peak usage to 2592.2 MB (3.6× reduction) while maintaining 97.8% accuracy. Microbatching (C4) further reduced memory to 1807.5 MB (5.17× reduction) with a 1.1% drop. Replacing the backbone with MobileNet (C5) cut memory to 775.6 MB (12.04× reduction) while retaining 98.2% accuracy. Patch-wise training (C6) achieved the largest reduction, 113.8 MB (82.05×), at the cost of a 3.3% accuracy loss.

Table 6 reports the time required to reach 95%, 96%, 97%, and 98% accuracy. This training time is a proxy for energy consumption and battery life on wearable devices, as longer computation times directly correlate with higher energy drain. The baseline (C1) is the fastest to 98% at about 5450 seconds. Configurations with sparsity, checkpointing, and microbatching (C2–C4) generally required more time to reach the same thresholds. MobileNet (C5) was competitive, in some cases matching or exceeding the baseline at higher accuracy levels, indicating that efficient architectures can offset the computational overhead of memory-saving techniques. This makes C5 a strong candidate for energy-efficient on-device training, offering a favorable balance of memory savings, accuracy, and battery consumption. Patch-wise training (C6) was markedly slower, reaching only about 95% within the same budget. This slow convergence would result in significantly higher battery consumption, making C6 suitable only for applications where minimal memory footprint is the absolute priority, regardless of energy cost. The variability of these measurements was substantial, with standard deviations often comparable to or larger than the mean differences across configurations. Consequently, Table 6 indicates general trends rather than precise performance rankings.

VII. Concluding Remarks

This work demonstrates the feasibility of neural network training on severely memory-constrained devices through a

TABLE 5: Accuracy and peak memory usage per configuration

Configuration	Accuracy	Peak Memory (MB)	Reduction Memory	Accuracy Drop
C1	0.985 ± 0.002	9337.8	1	0
C2	0.976 ± 0.002	8678.3	1.08	0.009
C3	0.978 ± 0.004	2592.2	3.6	0.007
C4	0.974 ± 0.002	1807.5	5.17	0.011
C5	0.982 ± 0.003	775.6	12.04	0.003
C6	0.952 ± 0.004	113.8	82.05	0.033

TABLE 6: Time (in seconds) to reach specific accuracy levels

Configuration	95%	96%	97%	98%
C1	3550.7 ± 311.9	3550.7 ± 311.9	4500.6 ± 600.2	5450.4 ± 458.3
C2	3400.6 ± 1801.8	5100.4 ± 1649.9	6100.9 ± 976.2	
C3	3650.8 ± 606.2	5101.1 ± 654.3	6600.6 ± 299.8	
C4	3551.3 ± 825.7	4401.2 ± 998.5	5350.9 ± 952.4	
C5	4100.5 ± 173.6	4200.831 ± 149.6	4500.6 ± 397.1	5000.5 ± 606.3
C6	5925.364 ± 317.9			

systematic combination of five complementary optimization techniques. The most significant achievement is configuration C6, which reduces peak memory usage from 9.3 GB to just 113.8 MB, an 82-fold reduction that enables deployment on wearable devices with less than 1GB of RAM.

The results highlight a trade-off between memory use, accuracy, and training time. For practitioners, this work suggests a clear strategy: begin with efficient architectures designed for mobile deployment (e.g., MobileNet), then apply gradient checkpointing, microbatching, and patch-based training as needed to meet memory constraints. Dynamic Sparse Training should be considered only when simpler techniques prove insufficient, given its implementation complexity and modest practical benefits in current software frameworks.

Several promising extensions could further enhance this work. Exploring more recent MobileNet variants (v2 and v3) would likely yield additional efficiency gains. Half-precision training represents another significant opportunity for memory reduction; unfortunately, extensive attempts to implement it were unsuccessful.

Finally, this project was not only a technical demonstration but also a deeply educational experience. The challenge of implementing a neural network at such a low level, particularly the convolutions, provided an invaluable insight into the core mechanics of deep learning. This hands-on understanding is, in my view, what made the project so rewarding and effective.

REFERENCES

[1] N. S. Sohoni, C. R. Aberger, M. Leszczynski, J. Zhang, and C. RÃ©, "Low-memory neural network training: A technical report," 2022.

[2] D. C. Mocanu, E. Mocanu, P. Stone, P. H. Nguyen, M. Gibescu, and A. Liotta, "Scalable training of artificial neural networks with adaptive sparse connectivity inspired by network science," *Nature Communications*, vol. 9, June 2018.

[3] H. Mostafa and X. Wang, "Parameter efficient training of deep convolutional neural networks by dynamic sparse reparameterization," 2019.

[4] T. Dettmers and L. Zettlemoyer, "Sparse networks from scratch: Faster training without losing performance," 2019.

[5] A. Acevedo, A. Merino, S. Alférez, Á. Molina, L. Boldú, and J. Rodellar, "A dataset of microscopic peripheral blood cell images for development of automatic recognition systems," *Data Brief*, vol. 30, p. 105474, June 2020.

[6] S. Liu and Z. Wang, "Ten lessons we have learned in the new "sparseland": A short handbook for sparse neural network researchers," 2023.

[7] T. Chen, B. Xu, C. Zhang, and C. Guestrin, "Training deep nets with sublinear memory cost," 2016.

[8] Y. Huang, Y. Cheng, A. Bapna, O. Firat, M. X. Chen, D. Chen, H. Lee, J. Ngiam, Q. V. Le, Y. Wu, and Z. Chen, "Gpipe: Efficient training of giant neural networks using pipeline parallelism," 2019.

[9] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," 2015.

[10] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam, "Mobilenets: Efficient convolutional neural networks for mobile vision applications," 2017.