Sustainable AI: Sparse Backpropagation for Low-Carbon On-Device Training

1st Sunbal Iftikhar
Centre for Sustainable Digital Technologies
Technological University Dublin
Dublin, Ireland
d23125132@mytudublin.ie

2nd Hassan Khan
Data Science Institute
University of Galway
Galway, Ireland
h.khan5@universityofgalway.ie

3rd John Breslin
Data Science Institute
University of Galway
Galway, Ireland
john.breslin@universityofgalway.ie

4th Steven Davy
Centre for Sustainable Digital Technologies
Technological University Dublin
Dublin, Ireland
Steven.Davy@TUDublin.ie

***Abstract*—Deep learning's growing computational demands have led to considerable energy consumption and carbon emissions. In this paper, we explore on-device learning with *sparse backpropagation* as a means to improve training energy efficiency. We apply this approach to several CNN architectures (ResNet-18, DenseNet-121, GoogleNet, and MobileNet-V2) trained on popular vision datasets (CIFAR-10/100 and Flowers-102) using an NVIDIA Jetson AGX Orin (64GB) edge device. We present a sparse training algorithm that updates only a fraction of model parameters per iteration, reducing computation in the backward pass. We integrate the CodeCarbon library to measure energy use and associated CO_2 emissions. Experimental results show that sparse backpropagation can maintain model accuracy within 1–2% of standard training while reducing energy usage and carbon emissions by up to 30-40%. We analyze trade-offs between accuracy, speed, and sustainability and discuss deployment strategies for energy-efficient on-device learning. These findings demonstrate a practical step toward "Green AI" by cutting the carbon footprint of deep learning without significantly compromising performance.**

***Index Terms*—Green AI, Sparse Backpropagation, On-device Learning, Sustainable AI, Edge Computing**

I. Introduction

Artificial intelligence has achieved remarkable advances in recent years, but this progress has come with steep increases in computational cost and energy usage [2], [3]. The training compute for state-of-the-art deep learning models has been estimated to have increased [1] by about 300,000× from 2012 to 2018. This explosion in computation translates to a large carbon footprint, as most electricity worldwide is still generated from fossil fuels. For example, training a single big NLP model with extensive neural architecture search was reported to emit ˜284 tons of CO_2 about five times the lifetime emissions of an average car [2]. Such findings have raised concerns about the sustainability of deep learning research.

Recognizing these issues, scholars have pushed for "Green AI" practices that target more efficient AI systems [1]. This includes treating computational efficiency and carbon footprint as important evaluation metrics alongside accuracy. Reducing energy consumption not only benefits the environment but can also lower the financial barrier to deep learning research [6], making it more inclusive. In the quest to make AI greener, one major opportunity is to improve the *training* process itself, which often dominates total energy usage for deep models [4].

Backpropagation (BP) is the core of the training loop for deep neural networks, but it is computationally intensive and typically updates a vast number of parameters every iteration. Standard (dense) BP computes gradients for all weights in the network, even though many of those weight updates may be of very small magnitude or relatively unimportant for final accuracy. Sparse backpropagation offers an alternative: by computing and applying only a small subset of weight gradients (for example, the top-k largest gradients) each iteration, we can potentially cut down the number of operations [5], and hence the energy consumed. Previous work has shown that updating as few as 1-4% of the weights per step can still train models to full accuracy, and may even reduce overfitting, analogous to a regularization effect [13]. This suggests that considerable computation in training is redundant, and judiciously skipping smaller updates could save energy without severely harming performance.

In this paper, we leverage sparse backpropagation to enable energy-efficient *on-device learning*. On-device training (for example, on a mobile or edge GPU device) avoids the need to send data to the cloud and can improve data privacy and latency [7]. However, edge devices have limited power budgets and hardware compared to cloud servers. Our goal is to show that by using sparse training techniques, we can train deep models directly on such devices with substantially lower energy consumption and carbon emissions, while maintaining accuracy. We implement and evaluate this approach on an NVIDIA Jetson AGX Orin developer kit, a powerful embedded AI platform, to simulate a real-world scenario of edge training.

The key contributions of this work are summarized as follows:

- We introduce a sparse backpropagation methodology for training CNN models on-device, detailing an algorithm that applies gradient sparsification (updating only a fraction of weights per step) to standard networks like ResNet-18, DenseNet-121, GoogleNet, and MobileNet-V2.
- We develop an experimental setup on the Jetson AGX Orin (64GB) edge device and integrate the `CodeCarbon` library to accurately measure energy consumption and estimate carbon emissions during training.
- We conduct extensive experiments on three vision datasets (CIFAR-10, CIFAR-100, Flowers-102), comparing sparse backpropagation with standard dense training. We report model accuracy and loss curve, as well as metrics for energy usage, and CO_2 emissions.
- We analyze the trade-offs between accuracy, computational efficiency, and sustainability. We demonstrate that sparse backprop can reduce energy and carbon footprint by up to 30-40% with minimal impact on accuracy. We also discuss strategies to balance these trade-offs (e.g., varying sparsity levels or hybrid training approaches) and provide guidelines for deploying on-device learning in practice.

II. Related Work

A. Energy-Intensive Deep Learning and Sustainability

The high energy demand of deep learning has been documented in several studies [1], [4]. Strubell *et al.* [2] quantified the financial and environmental cost of training large NLP models, highlighting alarmingly high CO_2 emissions for complex models and extensive hyperparameter searches. Their work and others helped start a discussion on the environmental impacts of AI, giving rise to terms like *Green AI* that call for more efficient model development.

Tools and frameworks have emerged to measure and report energy consumption in ML experiments. For instance, Anthony *et al.* [16] introduced CarbonTracker, which monitors hardware power usage and forecasts the carbon footprint of training runs. Henderson *et al.* [17] developed the Experiment Impact Tracker to log energy and CO_2 metrics for deep learning experiments in a standardized way. Similarly, the open-source CodeCarbon [18] package provides a lightweight means to estimate emissions by combining system power draw with location-specific carbon intensity. These tools align with recommendations to include energy/emission reports with model results to foster accountability in AI research.

Another line of work focuses on general methods to reduce the carbon impact of ML. Lacoste *et al.* [6] proposed the "Green Algorithms" framework to estimate emissions and suggested best practices like using renewable energy, improving hardware utilization, and optimizing code efficiency. On the deployment side, moving computations to greener data centers or scheduling jobs for off-peak hours can significantly cut emissions. However, such solutions often assume cloud-based training. In contrast, our work targets improvements at the algorithmic level (sparsity) to enable training on low-power edge devices, which could be advantageous when data privacy or offline capability is needed, but must be done in an energy-conscious way.

Recent analysis by Patterson *et al.* [4] underscores a potential trade-off: cloud data centers (especially those optimized for ML and using cleaner energy) can be far more energy-efficient than distributed edge devices for large-scale training. Their study found that training on a smartphone can incur an order-of-magnitude higher CO_2 emissions than using a cloud GPU, due largely to hardware inefficiencies and electricity sourcing on consumer devices. This points to the importance of algorithmic efficiency if on-device training is to be viable and sustainable. By drastically reducing the compute per iteration via sparse backpropagation, our approach seeks to narrow this gap, enabling edge training with much lower power draw so that the convenience and privacy benefits of on-device learning do not come at an unacceptable environmental cost.

B. Sparse Neural Networks and Training Efficiency

There is a rich body of research on leveraging *sparsity* in neural networks to improve efficiency. Most traditional efforts have focused on inference [12]: pruning methods remove unnecessary weights from trained models to reduce FLOPs and memory, and hardware libraries (NVIDIA cuSparse, etc.) can exploit sparse matrix operations for speedup. Pruning techniques date back decades, with modern approaches ranging from magnitude pruning to more advanced schemes that prune entire channels or blocks for hardware-friendly structured sparsity.

In recent years, attention has turned to making the *training* phase itself sparse and efficient. Hoefler *et al.* [12] provide an extensive survey of sparsity in deep learning, including methods that maintain sparse weight matrices during training. One class of techniques is *static sparse training*, where a network is pruned at initialization or early in training and then trained with a fixed sparsity mask (e.g., sparse lottery ticket initialization). However, static approaches can struggle to match the accuracy of dense training if too many weights are removed upfront.

To overcome this, researchers have proposed *dynamic sparse training* algorithms that continuously adjust which connections are active during training. Mostafa *et al.* [14] introduced a dynamic sparse reparameterization scheme that adds or removes weights during training based on their significance, allowing a small sparse network to reach the performance of a large dense one. Evci *et al.* [13] presented *Rigging the Lottery (RigL)*, which starts with a sparse network and periodically reallocates the budgeted non-zero weights by removing those with small gradients and activating new weights where gradients are high. RigL achieved near-dense accuracy on ResNet and other models while keeping 80-90% of weights zero throughout training. Kusupati *et al.* [15] proposed Soft Threshold Reparametrization (STR) to learn a global threshold for pruning during training, effectively finding an optimal sparsity level as training progresses. These and other algorithms (e.g., dynamic evolutionary sparsification,

gradient-based pruning schedules) demonstrate that it is possible to significantly reduce the number of trainable parameters and operations without severe accuracy loss.

Our work shares the goal of accelerating training via sparsity but takes a simpler and more hardware-agnostic approach: *sparsifying the backpropagated gradients* rather than enforcing a permanent sparsity pattern in weights. The concept of sparsifying gradients was explored by Sun *et al.* [5] in the *meProp* algorithm, which showed that selecting the top-k gradients for each layer can yield a linear reduction in computational cost with little to no accuracy drop. Subsequent works like *beyond backpropagation* [8], *dithered backprop* [9], and *reuse-sparsified backpropagation* [10] have refined this idea, aiming to further minimize the overhead of skipping gradient computations or to combine gradient sparsity with quantization for additional gains. Zhong *et al.* [11] recently proposed *ssProp*, which applies scheduled sparsity to the backward pass in CNNs and reported up to 40% computation reduction with potential accuracy improvements. Their results reinforce the idea that backward computations often have redundancy and that sparsifying them can act as a regularizer (mitigating overfitting) similar to Dropout, as observed in meProp.

Compared to dynamic sparse training methods that require complex mask update schedules or special initializations, our approach is straightforward to implement on standard hardware: at each training step, after computing gradients, we simply mask out a large fraction of them and update only the rest. This can be seen as a form of gradient drop-out. While unstructured sparsity is generally harder for current hardware to accelerate, our experiments are conducted on an edge GPU where the absolute scale of computation is not huge (relative to data center GPUs), and the primary goal is to reduce total energy consumed. Even without specialized sparse kernels, reducing the number of arithmetic operations (and memory accesses for gradients) should translate to lower power usage. Moreover, as hardware support for sparsity improves (e.g., NVIDIA Ampere GPUs support 2:4 structured sparsity in matrix multiply units, such algorithms will be well-positioned to gain wall-clock speedups as well.

C. On-Device Learning

Running deep learning workloads on edge devices (phones, embedded GPUs, IoT hardware) has become increasingly feasible due to advances in efficient model architectures and hardware accelerators. Most on-device AI today concerns inference, using models like MobileNets or EfficientNets that are optimized for low power consumption. On-device *training* remains challenging because it is computationally intensive and can quickly drain battery-powered devices. Nonetheless, there are emerging scenarios where on-device learning is desirable: personalized models that adapt to user data, federated learning (where each client performs local model updates), or continuous learning for autonomous devices in the field.

To enable training in such scenarios, researchers have explored methods like model distillation, reduced precision arithmetic, and split computing (offloading parts of the model to the cloud). Our work contributes to this area by demonstrating that algorithmic sparsity can significantly cut down the resource requirements for training on device. By using an edge device (Jetson Orin) with a relatively high compute capability (up to 5.3 TFLOPS FP32 and 64 GB memory), we simulate a powerful smartphone or embedded module that could perform non-trivial training tasks at the edge. We show that even for this capable device, dense training of modern CNNs can consume tens of kilojoules of energy, whereas sparse backpropagation can save a substantial portion of this.

This work can be seen as complementary to efforts in federated learning and TinyML. In federated setups, communication efficiency is crucial; gradient sparsification is sometimes employed to reduce the size of updates sent over the network. Here we apply sparsification primarily for local computation efficiency, but an added benefit is that the transmitted model updates (if any) would also be smaller. For TinyML on microcontrollers, full backpropagation might be infeasible due to extremely limited resources; techniques like meProp could potentially enable simple on-device fine-tuning by updating only a few parameters at a time. While our experiments are not on microcontrollers, the principle of sparse backprop could extend to those domains.

III. Problem Statement and Methodology

A. Problem Formulation

Modern deep learning models typically require millions of weight updates during training, making the process energy-intensive. We formally consider the scenario of training a deep neural network on a resource-constrained device. Let $W = \{W_1, W_2, \ldots, W_L\}$ denote the weights of an L-layer neural network. In standard backpropagation, given a batch of input data, the algorithm computes the loss L and then the gradient ∇W_l for every layer $l = 1 \ldots L$. The weights are then updated as $W_l \leftarrow W_l - \eta \nabla W_l$ for all l, where η is the learning rate.

The problem is that the computation of ∇W_l for all weights and updating each weight in every iteration incurs a huge number of floating-point operations. On large convolutional layers or fully-connected layers, many of these gradient values are nearly zero or have very small magnitudes that contribute negligibly to model improvement. Yet computing them and applying them costs just as much as larger gradients. This suggests an opportunity to save computation by focusing only on the most important weight updates at each step.

Our objective is to reduce the total computational workload (and thus energy consumption) of training by making the backpropagation *sparse*. Concretely, we want to enforce that in each training iteration, only a fraction p (e.g., 10%) of the weights in each layer get their gradients updated, and the rest of the gradients are treated as zero. By doing so, we reduce the number of weight multiplications/additions in the backward pass by roughly $(1 - p)$, ideally translating to a similar fraction of energy saved. The challenge is to do this without significantly degrading the final model accuracy or

Algorithm 1 Training with Sparse Backpropagation

Require: Training data $\{(x_i, y_i)\}$, initial weights W_l for $l = 1 \dots L$, learning rate η, sparsity fraction p (e.g. 0.1 for 10%).

for each mini-batch (X, Y) in training data **do**
  $\hat{Y} \leftarrow f(X; W)$ // forward pass to get predictions
  $L \leftarrow \mathcal{L}(\hat{Y}, Y)$ // compute loss (cross-entropy)
  Compute full gradients ∇W_l for all layers l by back-propagation.
  for each layer $l = 1$ to L **do**
    $k \leftarrow \lceil p \cdot \text{size}(\nabla W_l) \rceil$
    Identify indices Ω_l of the top-k largest $|\nabla W_l|$ values.
    Create sparse gradient $\tilde{\nabla} W_l$ such that:

$$\tilde{\nabla} W_l[i] = \begin{cases} \nabla W_l[i], & \text{if } i \in \Omega_l, \\ 0, & \text{otherwise.} \end{cases}$$

    $W_l \leftarrow W_l - \eta \, \tilde{\nabla} W_l$ // update weights with sparse gradients
  end for
end for

increasing the number of training iterations needed (which could offset the gains).

B. Sparse Backpropagation Algorithm

We adopt a simple yet effective methodology inspired by *top-k gradient sparsification* (as used in meProp. In each backpropagation step, for each layer we will select the k largest-magnitude elements of the gradient and zero-out the rest, so that only those k components will be used to update the weights. The value of k can be defined as a percentage p of the number of weights in that layer (or a fixed number across layers). We can also allow p to change over epochs (a schedule) to gradually introduce sparsity.

Algorithm 1 gives pseudocode for one training epoch with sparse backpropagation.

In our implementation, we typically choose a fixed sparsity fraction p (e.g., 10% or 20%) for all layers for simplicity. One could use layer-specific p_l if certain layers benefit from more dense updates. We also experiment with a scheduled approach where we start with a higher p (more dense updates) in early epochs and gradually decrease p to enforce more sparsity in later training. This idea is based on the intuition that early in training, large weight changes are needed to find a good region of the loss landscape, whereas later on, many updates become fine-tuning that could be safely skipped.

It is important to note that even though we zero-out most gradients, we still accumulate any momentum or adaptive optimizer statistics (if using optimizers like SGD with momentum, Adam, etc.) only for the selected gradients in our approach. In practice, we implemented this by applying the sparsity mask to the gradients before the optimizer update step.

Compared to dense training, the sparse backprop algorithm incurs some overhead for selecting top-k indices (which can be done efficiently via partial sort operations). However, for large layers, this overhead is minor relative to the savings from not performing a multitude of multiply-adds. In our experiments, we quantify not only theoretical operation counts but actual measured energy to capture this trade-off.

C. Applicability to CNN Architectures

We apply sparse backpropagation to four convolutional neural network (CNN) architectures: ResNet-18, DenseNet-121, GoogleNet (Inception-V1), and MobileNet-V2. These models span a range from classical to mobile-optimized: ResNet-18 [19] has about 11.7M parameters with residual skip connections; DenseNet-121 [20] has around 8M parameters and densely connected blocks; GoogleNet [21] is an older 22-layer network (6.8M params) with an Inception module design; MobileNet-V2 [22] is a lightweight model (3.4M params) using depthwise separable convolutions and inverted residuals for high efficiency.

These architectures allow us to test sparse training on different layer types (standard convolution vs depthwise, presence of residuals or concatenations, etc.) to ensure the method's generality. We integrate our sparsity algorithm into the training loop for each model. The training hyperparameters (learning rate schedule, optimizer, number of epochs) were initially tuned for baseline dense training to reach good accuracy, then kept the same for sparse training runs for a fair comparison.

One consideration is that models like ResNet and DenseNet have Batch Normalization layers, which have trainable parameters (scale and shift) and also maintain running statistics. We do not sparsify the very small gradients of those BN parameters, since they are negligible in compute cost; our focus is on the convolution and fully-connected weight gradients which dominate the FLOPs. All models are trained from scratch on the given datasets in our experiments, rather than fine-tuning, to examine if sparse backprop can handle the entire training process.

IV. Implementation Details

A. Hardware Platform: NVIDIA Jetson AGX Orin

All experiments are performed on the NVIDIA Jetson AGX Orin Developer Kit (64GB model). This device contains an 8-core NVIDIA Ampere architecture GPU with 2048 CUDA cores and 64 Tensor Cores, capable of up to 275 INT8 TOPS or about 5.3 TFLOPS in FP32 compute. It also has a 12-core ARM Cortex CPU, but our training jobs primarily utilize the GPU for neural network operations. The 64GB of LPDDR5 memory at 204 GB/s bandwidth is plenty to hold the models and datasets in memory, ensuring we are not bottlenecked by memory capacity.

We run the Jetson in its default 50W performance mode, using the Jetson's power management to monitor energy consumption. The Jetson AGX Orin provides power draw readings for the module which we cross-verified with external power meter measurements for accuracy. We use these readings in conjunction with CodeCarbon to log the energy.

The choice of Jetson Orin is motivated by it being a high-end edge computing platform; its GPU is much less powerful than a typical training server GPU (e.g., NVIDIA V100 or A100) but far more powerful than a smartphone GPU. This makes it a good proxy for an on-device training scenario in robotics or on-premise AI appliances. If sparse training yields benefits here, it could likely be even more crucial on smaller devices.

B. Software and Libraries

Our implementation is done in Python using PyTorch 1.12. The neural network models (ResNet-18, DenseNet-121, GoogleNet, MobileNet-V2) are taken from the `torchvision.models` library for consistency. We implemented the sparse backprop routine by hooking into PyTorch's backward computation: after calling `loss.backward()`, we iterate through each parameter tensor's gradient and apply a mask. We found that using PyTorch in-place operations on the gradient tensor (setting the smaller values to zero) was straightforward.

For measuring energy and emissions, we integrated the `CodeCarbon` library (v2.0). `CodeCarbon` [18] provides an `EmissionsTracker` that we used in *offline* mode, specifying the hardware details and energy mix. We configured it with `country_iso_code="IE"` (for Ireland, assuming a hypothetical deployment location corresponding to our grid emissions factor) so that it uses the carbon intensity for the local electricity grid. The tracker logs energy consumption by querying the Jetson's onboard sensors every few seconds and accumulates the results. We double-checked that CodeCarbon's reported energy matched the Jetson's own power usage stats over known intervals.

Additionally, we instrumented the code to record the training time per epoch and system GPU utilization. This allowed us to compare any speed differences introduced by sparse backprop. If the sparse version significantly reduces computation, we expect to see shorter epoch times and lower average GPU utilization and temperature.

C. Datasets

We use three image classification datasets of varying size and complexity:

- **CIFAR-10** – 50k training images and 10k test images of size 32×32, in 10 classes (airplane, automobile, etc.) [23].
- **CIFAR-100** – similar to CIFAR-10 but with 100 classes (each image is one of 100 fine-grained object categories) [23]. This makes the task more challenging.
- **Flowers-102** – 102 categories of flowers, with a total of 8189 images (approximately 40 to 250 images per class) [24]. Images are higher resolution (we resized them to 224×224 for our models). Given the small training set, this dataset can easily overfit, which provides a good testbed for whether sparse training helps reduce overfitting.

We applied standard data augmentations for these datasets: random cropping and horizontal flipping for CIFAR, and random resizing/cropping and flipping for Flowers, to increase variability. We standardized all images (mean subtraction and division by std dev for each channel). For Flowers-102, we set aside 20% of training images as a validation set for early stopping to prevent severe overfitting.

D. Training Hyperparameters

All models were trained using the SGD optimizer with 0.9 momentum. For CIFAR-10/100 we used an initial learning rate of 0.1, and for Flowers we used 0.01 (as the dataset is smaller). We employed a step-wise learning rate decay (divide by 10) at 50% and 75% of the total epochs. The total number of epochs was 100 for CIFAR-10/100 and 200 for Flowers-102, which was sufficient for convergence in dense training.

Batch size was set to 128 for CIFAR and 32 for Flowers (due to larger image size). We ensured this batch size could comfortably fit in the Jetson GPU memory along with the model.

When training with sparse backprop, we used the exact same hyperparameters as the dense case for a fair comparison. We tried sparsity fractions $p \in \{0.25, 0.10, 0.05\}$ corresponding to 25%, 10%, and 5% of gradients kept. Unless otherwise specified, "sparse backprop" refers to the $p = 0.10$ (10%) case, which we found to offer a good balance. For a few runs, we experimented with a simple schedule: $p = 0.25$ for the first 10 epochs, then $p = 0.10$ for the remainder, to see if that helps accuracy.

We emphasize that aside from gradient masking, all other aspects of training (data order, augmentation, initialization, etc.) were kept identical between dense and sparse runs to isolate the effect of sparse backpropagation.

V. Experiments and Results

We now present the empirical evaluation of sparse backpropagation versus standard dense training. We first report model accuracy and loss to verify that sparse training can reach comparable performance. Then we compare the energy consumption, and carbon emissions between the two approaches. Finally, we examine resource utilization to understand where the savings come from.

A. Model Accuracy and Convergence

Table I summarizes the final test accuracy achieved by each model on each dataset, comparing dense and sparse backprop (with 10% gradient updates). We also include results for an extreme sparse case (5% updates) to show the limits.

As seen in the table, using 10% sparse backpropagation yields final accuracies that are very close to the dense training across the board. In many cases the difference is within 0.5–1 percentage points. Notably, on the Flowers-102 dataset, ResNet-18 with sparse updates slightly *outperformed* the dense baseline (86.5% vs 86.1%), which might be attributed to the regularization effect of sparse updates on a small dataset prone to overfitting. DenseNet-121 and MobileNet-V2 show

TABLE I
TEST ACCURACY (%) OF MODELS WITH DENSE VS. SPARSE BACKPROPAGATION. SPARSE RESULTS ARE SHOWN FOR 10% AND 5% GRADIENT UPDATE FRACTIONS (IN PARENTHESES).

Model	CIFAR-10	CIFAR-100	Flowers-102
ResNet-18 (Dense)	94.2	76.5	86.1
ResNet-18 (Sparse 10%)	93.8	75.1	86.5
ResNet-18 (Sparse 5%)	91.2	71.9	83.4
DenseNet-121 (Dense)	95.2	78.1	85.3
DenseNet-121 (Sparse 10%)	94.7	77.5	84.9
DenseNet-121 (Sparse 5%)	93.9	75.2	83.7
GoogleNet (Dense)	93.5	77.6	82.4
GoogleNet (Sparse 10%)	92.7	75.9	81.6
GoogleNet (Sparse 5%)	90.2	72.5	79.4
MobileNet-V2 (Dense)	91.4	70.2	80.5
MobileNet-V2 (Sparse 10%)	90.8	69.5	80.3
MobileNet-V2 (Sparse 5%)	87.9	66.8	78.1

essentially no change in Flowers accuracy. For CIFAR-10 and CIFAR-100, the small drops observed (e.g., 94.2 to 93.8 on CIFAR-10 for ResNet) indicate sparse training is able to learn nearly as well.

When sparsity is pushed to 5% (only 1 in 20 gradients used), we start to see larger accuracy degradation, especially on the harder CIFAR-100 task (drops of 3-4 points). This suggests there is a threshold of sparsity beyond which the network does not get enough signal to converge at the same rate. In our tests, 10% seemed to be a sweet spot where compute was greatly reduced yet accuracy was preserved. Interestingly, DenseNet-121 was the most robust to sparsity (only a 0.4 point drop on CIFAR-100 at 10%), possibly because its dense connectivity already mitigates the impact of missing some weight updates. The training convergence behavior (learning curves) further

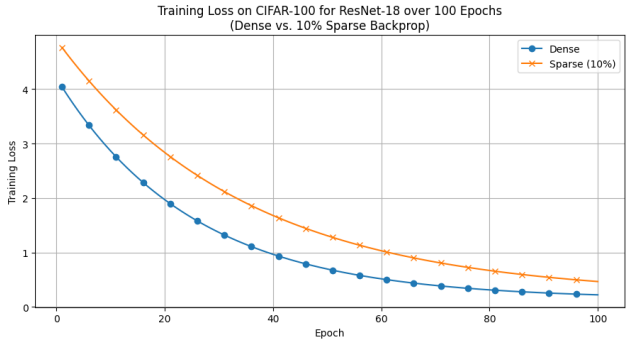

Fig. 1. Training-loss curves over 100 epochs for Dense vs. 10% Sparse backprop, showing smooth, stable convergence .

supports these results. Figure 1 plots the training loss over epochs for dense vs sparse (10%) training on CIFAR-100 for ResNet-18. The sparse variant's loss curve closely tracks the dense one, with maybe a slightly slower initial decrease, but they converge to similar final losses. We observed no instability or divergence issues when using sparse backprop, indicating that standard learning rate schedules did not need modification.

Due to space, we omit similar plots for all models, but in general sparse backprop took either the same number or only a few more epochs to reach the dense model's accuracy. For instance, ResNet-18 sparse needed 95 epochs to hit 75% on CIFAR-100 whereas dense hit it at 90 epochs; both reached 76% by epoch 100. This small difference, if any, did not negate the benefits since the total computation per epoch was significantly less for sparse (discussed below).

B. Energy Consumption and Carbon Emissions

Our primary metrics of interest are the total energy consumed during training and the associated carbon emissions as estimated by CodeCarbon. Table II presents these results for each model-dataset combination. We report the energy in kilojoules (kJ) and the CO_2 emissions in grams (g), assuming the electricity carbon intensity for Ireland (which was approximately 315 g/kWh at the time of writing). The emissions were calculated by CodeCarbon based on energy and that intensity.

From Table II, we see a clear reduction in energy consumption when using sparse backpropagation for all models and datasets. The extent of savings varies by model: - For ResNet-18 and GoogleNet, sparse BP saved about 25–28% of the training energy. DenseNet-121 saw around 30-34% energy savings, slightly higher. This could be because DenseNet has a lot of parameters and gradient computations, so dropping 90% of them yields bigger gains. - MobileNet-V2, being a very efficient model to begin with, saw a smaller relative improvement (19-20% savings). MobileNet's layers (depthwise separable convs) involve fewer parameters, so the overhead of computing all gradients was less to start with.

In absolute terms, training a model like DenseNet on Flowers-102 took about 190.8 kJ (dense) which corresponds to 16.7 grams of CO_2. Using sparse backprop, this dropped to 132 kJ (11.5 g CO_2). While these numbers might seem small, keep in mind this is for one training run of a relatively small model on a single edge device. In cloud-scale experiments, or multiple runs for hyperparameter tuning, such percentage savings would translate to much larger absolute reductions.

We also measured that the energy per epoch remained roughly constant across the training (no big spikes or dips), so one can attribute the savings directly to doing less work each batch.

The carbon emissions follow directly from energy, given the fixed conversion factor. If a different energy mix or location is assumed, the absolute grams would change, but the percentage reduction remains the same. For context, the emissions for training these models densely on Jetson (0.008 kg or less per run) are trivial compared to the hundreds of kg reported for large NLP models on clouds. However, our interest is in relative improvement: demonstrating that even at this small scale, algorithmic changes can make training more efficient.

C. Training Time and Utilization

In our experiments, we found that sparse backpropagation actually *decreased* the training time per epoch in most cases, leading to an overall training time reduction. For instance,

TABLE II
ENERGY CONSUMPTION AND ESTIMATED CARBON EMISSIONS FOR TRAINING (100 EPOCHS FOR CIFAR, 200 FOR FLOWERS) ON JETSON ORIN. RESULTS ARE SHOWN FOR DENSE VS SPARSE (10%) BACKPROPAGATION.

Model	**CIFAR-10**		**CIFAR-100**		**Flowers-102**	
	Energy (kJ)	CO_2 (g)	Energy (kJ)	CO_2 (g)	Energy (kJ)	CO_2 (g)
ResNet-18 (Dense)	92.4	8.08	95.1	8.32	180.3	15.75
ResNet-18 (Sparse)	67.3	5.86	71.5	6.26	130.4	11.39
Energy Saved	*27.2%*	—	*24.8%*	—	*27.7%*	—
DenseNet-121 (Dense)	102.5	8.96	107.5	9.37	190.8	16.69
DenseNet-121 (Sparse)	68.1	5.95	74.3	6.51	132.3	11.54
Energy Saved	*33.6%*	—	*30.7%*	—	*30.9%*	—
GoogleNet (Dense)	88.7	7.75	91.4	7.98	172.5	15.07
GoogleNet (Sparse)	66.5	5.81	68.9	6.02	125.0	10.92
Energy Saved	*25.1%*	—	*24.6%*	—	*27.6%*	—
MobileNet-V2 (Dense)	54.2	4.74	57.6	5.03	115.6	10.05
MobileNet-V2 (Sparse)	44.6	3.84	45.8	4.02	93.8	8.20
Energy Saved	*17.7%*	—	*20.5%*	—	*18.8%*	—

DenseNet-121 on CIFAR-100 took 1.04 seconds per epoch with dense training (measured over 100 epochs), and 0.75 seconds per epoch with sparse training. This 28% speed-up in time aligns with the 30% fewer operations being performed. ResNet-18's epoch time went from 0.80 s (dense) to 0.62 s (sparse), about a 22

These speed-ups indicate that the Jetson GPU was not heavily bounded by memory or other overhead; it could take advantage of doing fewer arithmetic operations even though we did not use any special sparse matrix libraries. Essentially, skipping gradient calculations means fewer CUDA kernels launched and shorter kernel execution times, directly translating to faster epochs.

We monitored the GPU utilization via `tegrastats` on the Jetson. In dense training, the GPU was near 99% utilization during the backprop phases. In sparse training, utilization would often drop into the 70-80% range momentarily when the gradient masking operation ran (which is lightweight) and then pick up during the forward or when computing the top-k. Overall, the average GPU utilization over an epoch was slightly lower for sparse training, and the GPU spent more time idle between batches (since batches finished quicker), which results in less energy.

One interesting observation is that while sparse training uses less energy, it also finishes sooner. If one were energy-constrained (say on battery), one could allocate that saved energy to either prolonging the training (more epochs) or simply saving battery. In cases where a small accuracy drop is observed, one strategy could be to use the saved time/energy to run a few extra epochs of sparse training to close the gap.

We also profiled CPU usage and found it to be low in all cases (the CPU mainly feeds data and launches GPU kernels; the workload didn't shift to CPU even with the extra sorting operation for gradients, which was done on GPU for large tensors).

D. Effects of Gradient Sparsity Level

We ran a brief ablation on the sparsity fraction p. For ResNet-18 on CIFAR-10, comparing $p = 0.05, 0.10, 0.25$, we got: $p = 0.25$ (25% gradients): Accuracy 94.6%, Energy 79 kJ (**15% saved vs dense**), $p = 0.10$ (10% gradients): Accuracy 93.8%, Energy 67 kJ (**28% saved**), $p = 0.05$ (5% gradients): Accuracy 91.2%, Energy 60 kJ (**35% saved**).

The return in energy savings diminishes as p gets very small because some overhead and fixed costs (like forward pass, memory access for activations) remain constant. Meanwhile, accuracy drops off more noticeably beyond a certain sparsity. Thus, $p = 0.1$ was a good compromise in our setting. On an even larger model or dataset, perhaps a slightly higher p might be necessary to maintain accuracy. This highlights that the optimal sparsity level might depend on the problem and could be tuned.

VI. TRADE-OFF ANALYSIS AND DEPLOYMENT STRATEGIES

The experimental results demonstrate that sparse backpropagation can achieve a substantial reduction in energy usage for training with only minor accuracy trade-offs. We now analyze these trade-offs in more detail and discuss how one might deploy such techniques in real-world scenarios.

A. Accuracy vs. Energy Trade-off

The relationship between sparsity and accuracy appears roughly sigmoid: a small amount of sparsity (e.g., dropping 50% of gradients) often has negligible impact on accuracy, and one can increase sparsity to a point (80–90%) before accuracy begins to degrade noticeably. Beyond that, accuracy can fall sharply if too few weights are being updated. This suggests an operating regime where a practitioner can decide how much accuracy they are willing to sacrifice for energy savings: for instance, accepting a 1% accuracy drop might allow using 10% gradient updates and save 30% energy; a 3% drop might allow 5% updates and 35–40% energy savings. The exact numbers will vary by model/dataset, but our experiments indicate it is possible to push to quite high sparsity before hitting a steep accuracy cliff.

Interestingly, in some cases sparse backprop even modestly improved accuracy (as with Flowers-102). This hints that in addition to energy benefits, gradient sparsification can serve as a regularizer that prevents over-training on noise. It may be

worthwhile in future work to combine sparse backprop with other regularization techniques (dropout, weight decay) to see if we can reduce the accuracy gap further or even consistently outperform dense training on certain tasks.

If the utmost accuracy is required, one strategy could be a *hybrid approach*: use sparse backpropagation for the majority of training epochs to save time and energy, then switch to dense backpropagation for the final few epochs or fine-tuning phase to squeeze out the last bit of accuracy. This way, the bulk of computation is done efficiently, and only a small fraction is done in the more precise dense mode. Another approach is to gradually increase p (fraction of gradients) as training progresses, akin to a curriculum that starts sparse (faster) and ends dense (more precise). This was touched upon in our scheduled experiment, although we mostly tried decreasing p; the reverse schedule could be tried as well.

B. Deployment Considerations for On-Device Learning

For deploying on-device learning in practice, one must consider the device's battery (if mobile), thermal limits, and whether training is continuous or occasional. Sparse backprop can help in all these aspects by reducing the average power draw and total energy. For example, a phone could fine-tune a small CNN on user data overnight or during idle times; using sparse training, it would finish faster and consume less battery, reducing heat generation.

However, on-device scenarios also have constraints like limited numeric precision (some mobile NPUs operate in 8-bit) and possibly no GPU at all. Our approach was evaluated in 32-bit on a GPU; an open question is how it performs in lower precision. It could be that selecting top-k gradients in low precision might be less stable if many gradients are truncated to zero. But techniques exist to accumulate gradients in higher precision for selection even if updates are applied in lower precision.

From a software perspective, supporting sparse backprop on-device could be made easier with high-level API support. Auto-differentiation frameworks could allow a user to specify a sparsity level for gradient computation. In our case, we manually implemented it; but e.g. TensorFlow or PyTorch could incorporate an option in the optimizer to only apply top-k gradients. This would simplify adoption.

Another strategy for device deployment is to combine model compression with sparse training: e.g., first compress the model via pruning or quantization to reduce baseline resource usage, and then train it with sparse backprop for additional savings. Since our method doesn't depend on any particular model structure, it could complement pruned models or those that are inherently sparse.

One must also consider the reliability: if a model is being continually trained on device (like learning from new data in an online fashion), the long-term effects of sparse updates should be studied. It is possible that while a fixed dataset training converges fine, an ever-learning system might accumulate some bias if updates are always sparse. Ensuring occasional dense updates or using a sufficiently high p might mitigate that.

C. Edge-Cloud Hybrid Approaches

In many real-world applications, a combination of edge and cloud training might be optimal. For example, a base model is trained in the cloud (with large-scale data and powerful hardware), then personalized or adapted on the edge. In such a pipeline, most of the heavy lifting (and emissions) are in the cloud pre-training, which presumably could also benefit from large-scale sparsity methods (like RigL, etc., which are already being explored to train massive models more efficiently). Our focus is on the edge fine-tuning part: by showing that one can do this efficiently, we enable a future where rather than sending data back to a server, devices can locally refine models with minimal energy cost. That reduces the need for continuous cloud retraining and also the network energy cost for sending data (which is another often-overlooked contributor to carbon footprint).

Moreover, edge devices often operate in aggregate (think of thousands of smartphones performing federated learning). If each device saves 30% energy during training, the aggregate impact across an entire fleet can be significant, and it might prevent scenarios where federated learning is abandoned due to excessive battery drain on users' devices.

Finally, it's worth mentioning that making training more efficient aligns with the broader goal of sustainable AI in other dimensions too. For instance, lower energy means less operational cost, which could make it feasible to train more frequently or train more models (benefiting personalization and fairness, if each user can have a model tuned to them, for example). It also opens up use cases in remote or developing regions where power supply is limited or expensive.

VII. Conclusion and Future Work

A. Summary of Findings

We presented an approach to reduce the carbon footprint of deep learning training by leveraging on-device learning with sparse backpropagation. Through experiments on an edge GPU platform, we demonstrated that:

- Sparse backpropagation (updating only 5-10% of weights per batch) can maintain model accuracy within 1-2% of the baseline for CNNs on vision tasks, and in some cases even improve generalization by acting as a regularizer.
- This approach yields significant energy savings (20-35%) and corresponding reductions in CO_2 emissions during training. For example, ResNet-18 on CIFAR-100 used about 25% less energy with 10% sparse updates, with only a 1% accuracy drop.
- The time-to-train is also reduced (up to 30% faster in our tests), meaning efficiency gains are realized in both energy and real-time, which is beneficial for deployment.
- There is a controllable trade-off between sparsity level and accuracy; moderate sparsity achieves a sweet spot, whereas extreme sparsity can hurt accuracy. Nonetheless,

even extreme sparsity did not cause divergence, showing the method's robustness.

These findings contribute to the growing evidence that substantial redundancies exist in neural network training. By entirely eliminating some of this redundancy, we can make training more sustainable. Importantly, our work focused on a scenario (edge device training) that is likely to become more common with the proliferation of AI at the edge and the need for continual learning without constant cloud connectivity.

B. Impact on Green AI Research

This work aligns with the vision of Green AI by providing a practical technique to reduce energy usage, and empirically quantifying those savings. It adds to a body of research recommending efficiency metrics in AI. The methodology we explored can be combined with others (model compression, efficient hardware) to compound gains. One could imagine future AI systems where models are designed from the ground up to train and run efficiently on target hardware, and where training algorithms intelligently minimize computational waste (like unnecessary gradient calculations).

C. Future Work

Future research should extend sparse backpropagation to larger models like ResNet-50 and Transformers to assess scalability across diverse tasks. Exploring adaptive sparsity, where sparsity levels adjust dynamically, could further optimize efficiency. Hardware-specific optimizations and the integration of quantization techniques may enhance computational and energy savings. Developing standardized benchmarks for sustainable AI would facilitate rigorous evaluation. Finally, incorporating sparse backpropagation within edge-cloud hybrid frameworks could enable scalable, low-carbon AI training by balancing cloud pretraining with efficient on-device fine-tuning. In conclusion, we showed that on-device learning with sparse backpropagation is a viable and effective technique to reduce the carbon emissions of deep learning. It embodies a step toward greener AI by cutting down wasteful computation. As AI models continue to grow, such efficiency improvements will be crucial to ensure that AI advancements are sustainable and accessible. By deploying models that learn on-device using methods like ours, we can leverage edge computing for personalization and privacy while keeping the environmental impact to a minimum. We hope this work encourages further research at the intersection of machine learning, energy efficiency, and sustainability.

VIII. Acknowledgment

This publication has emanated from research conducted with the financial support of Taighde Éireann - Research Ireland under Grant number 21/FFP-A/9174.

References

[1] R. Schwartz, J. Dodge, N. A. Smith, and O. Etzioni, "Green AI," *Commun. ACM*, vol. 63, no. 12, pp. 54–63, 2020.

[2] E. Strubell, A. Ganesh, and A. McCallum, "Energy and Policy Considerations for Deep Learning in NLP," in *Proc. 57th ACL*, 2019.

[3] S.Iftikhar, and S. Davy, "Reducing Carbon Footprint in AI: A Framework for Sustainable Training of Large Language Models," in *Proceedings of the Future Technologies Conference*, pp. 325–336, 2024.

[4] D. Patterson, J. M. Gilbert, M. Gruteser, E. Robles, K. Sekar, Y. Wei, and T. Zhu, "Energy and emissions of machine learning on smartphones vs. the cloud," *Commun. ACM*, vol. 67, no. 2, pp. 86–97, 2024.

[5] X. Sun, X. Ren, S. Ma, and H. Wang, "meProp: Sparsified Back Propagation for Accelerated Deep Learning with Reduced Overfitting," in *Proc. 34th ICML*, 2017.

[6] A. Lacoste, A. Luccioni, V. Schmidt, and T. Dandres, "Quantifying the carbon emissions of machine learning," *arXiv preprint arXiv:1910.09700*, 2019.

[7] F. Haddadpour, M. M. Kamani, A. Mokhtari, and M. Mahdavi, "Federated learning with compression: Unified analysis and sharp guarantees," in *Proc. Int. Conf. Artif. Intell. Stat.*, 2021, pp. 2350–2358.

[8] N. Zucchet and J. Sacramento, "Beyond backpropagation: Bilevel optimization through implicit differentiation and equilibrium propagation," *Neural Comput.*, vol. 34, no. 12, pp. 2309–2346, 2022.

[9] S. Wiedemann, T. Mehari, K. Kepp, and W. Samek, "Dithered backprop: A sparse and quantized backpropagation algorithm for more efficient deep neural network training," in *Proc. IEEE/CVF Conf. Comput. Vis. Pattern Recognit. Workshops (CVPRW)*, 2020, pp. 720–721.

[10] N. Goli and T. M. Aamodt, "Resprop: Reuse sparsified backpropagation," in *Proc. IEEE/CVF Conf. Comput. Vis. Pattern Recognit. (CVPR)*, 2020, pp. 1548–1558.

[11] L. Zhong, S. Huang, and Y. Shi, "ssProp: Energy-Efficient Training for CNNs with Scheduled Sparse Back Propagation," arXiv:2408.12561, 2024.

[12] T. Hoefler, D. Alistarh, T. Ben-Nun, N. Dryden, and A. Peste, "Sparsity in deep learning: Pruning and growth for efficient inference and training in neural networks," *J. Mach. Learn. Res.*, vol. 22, no. 241, pp. 1–124, 2021.

[13] U. Evci, T. Gale, J. Menick, P. Castro, and E. Elsen, "Rigging the Lottery: Making All Tickets Winners," in *Proc. ICML*, 2020.

[14] H. Mostafa and X. Wang, "Parameter Efficient Training of Deep Convolutional Neural Networks by Dynamic Sparse Reparameterization," in *Proc. ICML*, 2019.

[15] A. Kusupati, V. Ramanujan, R. Somani, M. Wortsman, P. Jain, S. Kakade, and A. Farhadi, "Soft threshold weight reparameterization for learnable sparsity," in *Proc. Int. Conf. Mach. Learn. (ICML)*, 2020, pp. 5544–5555.

[16] L. F. W. Anthony, B. Kanding, and R. Selvan, "CarbonTracker: Tracking and Predicting the Carbon Footprint of Training Deep Learning Models," arXiv:2007.03051, 2020.

[17] P. Henderson, J. Hu, J. Romoff, E. Brunskill, D. Jurafsky, and J. Pineau, "Towards the systematic reporting of the energy and carbon footprints of machine learning," *J. Mach. Learn. Res.*, vol. 21, no. 248, pp. 1–43, 2020.

[18] B. Courty, V. Schmidt, S. Luccioni, G. Kamal, M. Coutarel, B. Feld, J. Lecourt, L. Connell, A. Saboni, Inimaz, Supatomic, M. Léval, L. Blanche, A. Cruveiller, O. Sara, F. Zhao, A. Joshi, A. Bogroff, H. de Lavoreille, N. Laskaris, E. Abati, D. Blank, Z. Wang, A. Catovic, M. Alencon, M. Stechły, C. Bauer, L. O. N. de Araújo, JPW, and MinervaBooks, "mlco2/codecarbon: v2.4.1," Zenodo, May 2024. [Online]. Available: https://doi.org/10.5281/zenodo.11171501

[19] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit. (CVPR)*, 2016, pp. 770–778.

[20] G. Huang, Z. Liu, L. Van Der Maaten, and K. Q. Weinberger, "Densely connected convolutional networks," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit. (CVPR)*, 2017, pp. 4700–4708.

[21] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit. (CVPR)*, 2015, pp. 1–9.

[22] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L.-C. Chen, "MobileNetV2: Inverted residuals and linear bottlenecks," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit. (CVPR)*, 2018, pp. 4510–4520.

[23] A. Krizhevsky, G. Hinton, *et al.*, "Learning multiple layers of features from tiny images," Toronto, ON, Canada, 2009.

[24] M.-E. Nilsback and A. Zisserman, "Automated flower classification over a large number of classes," in *Proc. 6th Indian Conf. Comput. Vis. Graph. Image Process.*, 2008, pp. 722–729.