`NeuroFlux`: Memory-Efficient CNN Training Using Adaptive Local Learning

Dhananjay Saikumar and Blesson Varghese
University of St Andrews, UK

Abstract

Efficient on-device deep neural network (CNN) training in resource-constrained mobile and edge environments is an open challenge. Backpropagation is the standard approach adopted, but it is GPU memory intensive due to its strong inter-layer dependencies that demand intermediate activations across the entire CNN model to be retained in GPU memory. This necessitates smaller batch sizes to make training possible within the available GPU memory budget, but in turn, results in a substantially high and impractical training time. We introduce `NeuroFlux`, a novel CNN training system tailored for memory-constrained scenarios. We develop two novel opportunities: firstly, adaptive auxiliary networks that employ a variable number of filters to reduce GPU memory usage, and secondly, block-specific adaptive batch sizes, which not only cater to the GPU memory constraints but also accelerate the training process. `NeuroFlux` segments the CNNs into blocks based on GPU memory usage and further attaches an auxiliary network to each layer in these blocks. This disrupts the typical layer dependencies under a new training paradigm - *'adaptive local learning'*. Moreover, `NeuroFlux` adeptly caches intermediate activations, eliminating redundant forward passes over previously trained blocks, further accelerating the training process. The results are twofold when compared to Backpropagation: on various hardware platforms, `NeuroFlux` demonstrates training speed-ups of 2.3× to 6.1× under stringent GPU memory budgets, and `NeuroFlux` generates streamlined models that have 10.9× to 29.4× fewer parameters without sacrificing accuracy.

Keywords: CNN training, Memory efficient training, Local learning, Edge computing

1 Introduction

Convolutional neural networks (CNNs) underpin a wide range of applications that run on resource-constrained environments, such as mobile and edge computing systems. These applications may be functionality-related or mission-critical and are required to operate within fixed resource constraints. Examples of the former type of applications include facial authentication [33, 48], speech recognition [1–3, 54], and gesture detection [17, 59] on home gadgets, and of the latter include advanced perception [25, 71] and decision-making [38, 40] for autonomous vehicles or robots.

High-throughput and budget-friendly inference of CNNs comprising millions of parameters has become feasible on mobile and edge due to two recent advances [42, 68]. Firstly, the availability of embedded hardware accelerators, such as Graphics Processing Units (GPUs) and Neural Processing Units (NPUs). Secondly, the development of techniques that reduce the complexity of CNNs via pruning [15, 23, 64], quantization [73], knowledge distillation [60], and neural architecture search [58, 76].

However, **CNN training is still computationally and memory intensive**, necessitating the reliance on servers found in high-performance computing sites or cloud data centers. The typical approach involves either training the CNN entirely on an external server and then deploying it onto the device solely for inference [40], or collaboratively training across both the device and the server by leveraging computational offloading [20, 22, 66, 70, 72].

Nonetheless, **on-device training remains a challenge**. This challenge arises from the memory-intensive nature of the widely used *Backpropagation* (BP) paradigm in CNN training. Within this paradigm, it is imperative that the intermediate outputs, from the layers of the CNN, referred to as activations, are retained in memory for calculating gradients (more in Section 2.2). The activations of deep and complex models require substantial amounts of GPU memory. For instance, a production-quality CNN model, such as ResNet-18, when used in real-world scenarios [28, 35, 49] may require up to 15 GB of GPU memory. While limited computing resources may allow for on-device training, albeit at the expense of training time, the lack of memory on mobile and edge resources is simply prohibitive for on-device CNN training. Consequently, on-device training is limited to smaller batch sizes, which in turn increases training time, or mandates the use of simpler models and datasets.

Our work is therefore positioned to achieve a breakthrough in on-device CNN training in resource-constrained environments by developing the `NeuroFlux` system. *We depart from Backpropagation and develop an adaptive layer-wise local learning approach.* The key advantage of the system is memory efficiency by: (a) segmenting the CNN into blocks for reducing the activation memory requirements as we eliminate inter-layer dependencies, and (b) implementing variable

batch sizes for different blocks to maximize memory utilization. Our **research contributions** are:

1) NeuroFlux, **a system designed for efficient on-device CNN training under memory constraints**. When compared to Backpropagation, NeuroFlux showcases speedups in training ranging from 2.3× to 6.1× and 3.3× to 10.3× when compared to Backpropagation and classic local learning [5], respectively, given a memory budget.

2) **Two novel adaptive strategies for local learning**: firstly, *adaptive auxiliary networks* with variable filters per CNN layer, and secondly, *adaptive batch sizes* that tailors batch size per layer. These strategies allow NeuroFlux to process larger training batches, consequently reducing the number of gradient descent steps. This reduction directly improves the training latency of NeuroFlux and enables it to achieve higher accuracy than BP and classic local learning in a given time frame.

3) **Block-based learning that groups CNN layers by memory needs**. Only the actively trained block occupies device memory, while the other blocks are located in storage. This combined with the adaptive strategies, allows for training to be carried out on a fixed memory budget.

4) **Streamlined early exit model**: The local learning strategy of NeuroFlux selects the optimal early exit model. CNNs generated by NeuroFlux are 10.9× to 29.4× more compact than those from BP and classic local learning. This naturally impacts inference speed, with NeuroFlux models showing throughput improvements of 1.61× to 3.95× across different hardware platforms.

The remainder of this paper is organized as follows. Section 2 presents the background and motivation for the research. Section 3 presents a case for adaptive local learning that underpins our work. Section 4 provides an overview of NeuroFlux, the system developed to leverage adaptive local learning. Section 5 considers the components of NeuroFlux and the underlying techniques. Section 6 evaluates NeuroFlux and highlights the results obtained. Section 7 considers related work. Section 8 concludes this paper.

2 Background and Motivation

This section further considers the memory-intensive nature of the Backpropagation paradigm and considers viable alternate training paradigms.

2.1 On-device Training

The need for on-device training is rising to enable personalized models and ensure data privacy. On-device training underpins recent training paradigms, such as federated learning [51] and continual learning [55] that make use of computational resources on edge devices while safeguarding user privacy. Furthermore, on-device training reduces the reliance on extensive backend infrastructure, which enables services to become more digitally sovereign and scale rapidly while also being less impacted by network variability [16]. This ensures user data, including sensitive information like voice recordings and facial images, remains on the device [31, 56]. On-device training is in contrast to traditional and resource-intensive model development methods, such as Neural Architecture Search (NAS) and structured pruning that require substantial computational resources - from several GPU days [9, 14] for NAS to GPU hours to days [52, 74] for structured pruning when utilizing server-grade hardware. These require significant energy and computational expenditure but also lead to further on-device fine-tuning costs to tailor the derived models for specific tasks. On-device training emerges as a more streamlined and cost-efficient alternative that can reduce the computational and carbon footprint associated with deep learning model training [57, 61].

2.2 Backpropagation is Memory Intensive

Backpropagation (BP)-based training operates in two phases: the forward pass and the backward pass. In the forward pass, the input data sequentially traverses through the layers of the CNN model, producing intermediate activations that are stored in the memory of the accelerator, such as that of the GPU. The output of the forward pass is fed into a function, such as Cross Entropy or Mean Squared Error, in order to evaluate a global loss. Conversely, in the backward pass, parameter gradients for the final layer are computed using both the loss and the activations from the preceding layer. The gradients are then propagated backwards, repeating this through the model. The gradient of each layer depends on the activations of its preceding layer, which makes it necessary to retain all activations in memory throughout the forward pass of the model. The necessity for retaining these activations is presented in Appendix A.

The two phases have computational similarities, with the backward pass requiring up to 3x the FLOPs (floating-point operations) as the forward pass. The forward pass during inference and training is computationally identical, differing only in GPU memory requirements, as the latter requires all intermediate activations for subsequent gradient calculations in the backward pass. Mobile and edge-based hardware accelerators can meet the compute demands of the forward and backward passes, but satisfying memory requirements is a challenge.

In Figure 1, we examine the GPU memory consumption and training time of two widely used CNN architectures on the Tiny ImageNet dataset: ResNet-18 and VGG-19. This analysis employs BP-based training across varying batch sizes. The memory footprint comprises three components: the inherent size of the model, the memory overhead of the optimizer, and the memory required for activations. It is immediately evident that the GPU memory usage during training arises from the memory required for retaining the intermediate activations. This poses challenges for training on memory-restricted devices. A similar trend is observed
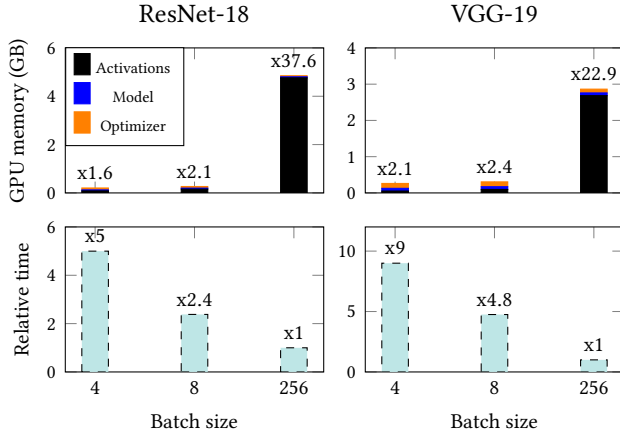
Figure 1. Comparison of GPU memory usage and relative training time for different architectures and batch sizes on the Tiny ImageNet dataset. The top row shows memory used by activations, the model, and the optimizer, with multipliers indicating memory required relative to inference. The bottom row highlights training time relative to batch size of 256.

for smaller models specifically designed for mobile environments, such as MobileNet [27], trained on the Tiny ImageNet dataset. For a batch size of 256, MobileNet requires 830MB of GPU memory to retain activations during BP-based training, while inference can be performed under 35MB. Therefore, even using smaller models tailored for mobile devices, the activations still dominate the GPU memory requirement for BP-based training.

While opting for smaller batch sizes can reduce memory demands, it increases the number of Stochastic Gradient Descent (SGD) steps, leading to larger training durations. For instance, VGG-19 trained on the Tiny ImageNet with a batch size of 4 takes over nine times longer than with a batch size of 256.

2.3 Training Paradigms Beyond Backpropagation

Several paradigms have been developed to address the limitations of BP-based training. Among them, Feedback Alignment, Signal Propagation, and Local Learning are noteworthy. Local Learning is considered further as it will be leveraged in our work given the opportunities it presents (see Section 3). The other paradigms are considered in Section 7.

1) *Feedback Alignment* (FA): FA provides a solution to the 'weight transport problem' in CNN training [44, 45]. While FA is shown to be effective for certain CNN models, it is not ideal for convolutional neural networks [37].

2) *Signal Propagation* (SP): By employing forward passes only, SP offers a unique layer-wise training methodology, akin to LL, but without making use of auxiliary networks [37].
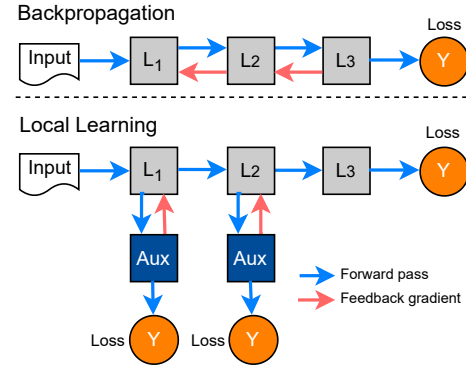

Figure 2. Comparison of BP and LL. BP relies on global loss, with each layer update dependent on subsequent layers. LL pairs each layer (excluding the last) with an auxiliary network for independent updates using local losses, eliminating backward dependencies.

While SP showcases a memory efficiency over BP, its accuracy is lower than BP and LL. Thus, it is not widely adopted for training.

3) *Local Learning* (LL) assigns an auxiliary network to each CNN layer to facilitate local prediction, rather than relying on a global loss derived from the final output of the CNN as in BP [5] (refer Figure 2). When a training batch is provided to the first CNN layer, it is processed to produce output activations. The activations are fed into the auxiliary network for prediction. Based on the prediction and the true labels, a local loss is computed. The loss is subsequently used to optimize the parameters in both the first layer and its auxiliary network. The activations of the first layer are then passed on to the next layer. This is sequential and continues for every layer, ensuring that each layer processes the data and undergoes optimization until the entire batch has traversed all layers [6]. Any feedback dependencies on subsequent layers are eliminated.

Recent empirical studies demonstrate that LL achieves similar performance as BP-based training for a range of datasets [5]. LL does not require the intermediate activations to be retained in memory for parameter adjustments [5], thereby offering the potential for memory-efficient training as the parameters are updated on a layer—by—layer basis (layer-wise training). However, in the existing work on classic LL, this memory gain is offset by the memory requirements of the auxiliary network to make predictions. The opportunities we will leverage within this training paradigm are considered in the next section.

Comparing GPU memory vs accuracy: Figure 3 captures the GPU memory required and accuracy of the training paradigms. An ideal paradigm will achieve high accuracy without significant memory use. Although LL achieves comparable accuracy to BP it has substantial memory requirements. *Currently there are no training paradigms that can*
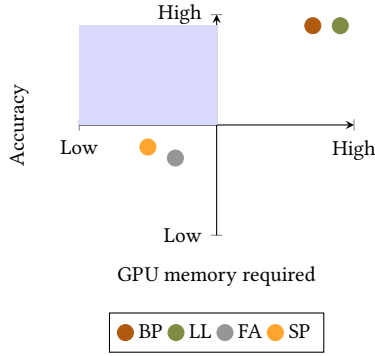
Figure 3. GPU memory required and accuracy achieved by different training paradigms. The blue-shaded quadrant represents the ideal zone for a training paradigm (low GPU memory utilization and high accuracy).

achieve high accuracy with a low GPU memory utilization. Therefore, we explore the potential to lower the memory utilization of LL into the blue-shaded area of Figure 3.

3 The Case for Adaptive Local Learning

We have identified **two opportunities** for reducing the GPU memory requirements while achieving comparable accuracy to BP. Here we introduce the notion of *'adaptivity'* for the first time within local learning that is considered below.

Opportunity 1: Adaptive Auxiliary Networks in Local Learning. In LL, each CNN layer (except the final layer) is accompanied by an auxiliary network, as shown in Figure 2. Classic LL employs a CNN classifier as the auxiliary network, which comprises a convolution layer, average pooling, and a fully-connected layer for prediction [5].

In the classic LL approach, auxiliary networks use a fixed number of convolutional filters (256 filters) [5]. However, the GPU memory used during classic LL training is noted to be surprisingly higher than BP, although in LL activations of only one given layer is retained in GPU memory. This is due to the number of convolutional filters in the auxiliary networks of the initial layers of the CNN, resulting in large activations that are not sufficiently downsampled. A uniform decrease in the number of convolutional filters for all auxiliary networks of the CNN leads to reduced GPU memory usage. However, the accuracy obtained is lower than BP.

Instead, we develop a new strategy referred to as **'Adaptive Auxiliary Networks based LL (AAN-LL)'** - employing a variable number of convolutional filters for the auxiliary networks for achieving both lower GPU memory consumption and comparable accuracy to BP. For the initial layers of the CNN, which in this context are defined as the layers before the first downsampling operation, the number of convolutional filters in the auxiliary network is halved with respect to the CNN layer with the smallest number of convolutional filters. For example, the smallest number of
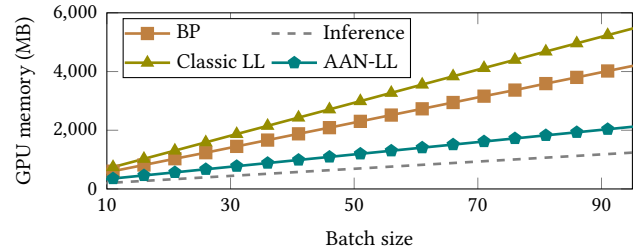

Figure 4. GPU memory usage of VGG-19 for inference, Back-propagation (BP), classic Local Learning (LL) with a constant number of 256 convolutional filters [5] and the proposed Adaptive Auxiliary Networks-based LL (AAN-LL) for different batch sizes.
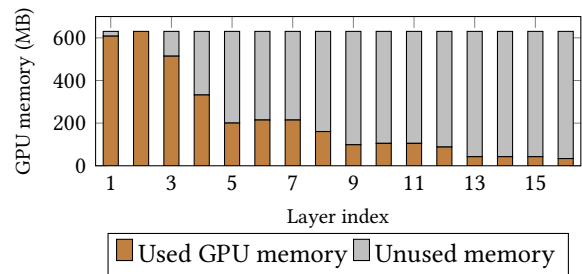

Figure 5. GPU memory usage for training VGG-19 with a batch size of 30 images using AAN-LL. 'Unused Memory' area refers to GPU memory not utilized by each layer.

convolutional filters for a VGG model is 64; the number of convolutional filters of the auxiliary network for the initial layers will therefore be 32. For the subsequent CNN layers, we reduce the number of convolutional filters again by half with respect to the CNN layer with the largest number of convolutional filters. For example, the largest number of convolutional filters for a VGG model is 512; therefore, the number of convolutional filters of the auxiliary network for the later layers is 256.

The GPU memory trends for inference, BP, classic LL with 256 convolutional filters and our strategy AAN-LL are shown in Figure 4 for VGG19; similar trends are observed for ResNet18. The success of the AAN-LL strategy is in reducing the size of the activations for the initial layers to reduce GPU memory consumption, while maintaining a sufficiently large number of convolutional filters in the later layers to ensure effective learning.

Opportunity 2: Adaptive Batch Sizes in Local Learning. Consider an example - the GPU memory using AAN-LL is approximately 630MB for training VGG19 using a batch size of 30; refer to Figure 4. The GPU memory use of AAN-LL arises from the initial layers of the CNN. The activations are gradually downsampled as they progress through the model, resulting in larger activations for the initial layers
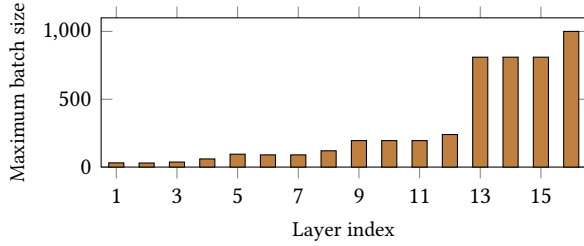
Figure 6. Maximum possible batch sizes per layer when training VGG-19 using AAN-LL while staying below a 630MB GPU memory limit.

compared to subsequent ones. Figure 5 highlights this observation; the second layer of the CNN dominates GPU memory. Thus, training the initial layers of a CNN is a GPU memory bottleneck. They also mandate a small batch size to be used for training the entire CNN under GPU memory constraints.

Due to the inherent design of BP and classic LL, they use a fixed batch size for training the entire CNN. However, we make two key observations. Firstly, the later layers of the CNN do not reach the GPU memory use of the initial layers, as shown in Figure 5. Secondly, the maximum possible batch size for a given GPU memory budget (630 MB in the case of VGG19 as it is the GPU memory use) is higher for later CNN layers than the initial layers as shown in Figure 6.

We leverage these observations to develop **'Adaptive Batch-based LL (AB—LL)'**. Contrary to conventional methods that use a static batch size, AB—LL employs dynamic batch sizes for training different layers. While the underlying algorithm for (LL) uses a fixed batch size for training the entire CNN, AB—LL dynamically adjusts the number of activations in a training batch in real-time. This adjustment effectively changes the number of batches consumed by each layer or groups of layers with similar GPU memory requirements. By capitalizing the unused GPU memory of the later CNN layers (shown as 'Unused Memory' in Figure 5), AB—LL enables the use of larger batch sizes specifically for these layers, thereby optimizing GPU memory use and accelerating training. Beyond mere efficiency enhancements, this approach offers a strategic shift in training by accommodating the fine-grained GPU memory requirements of individual CNN layers or their groups to maximize the utilization of available GPU memory resources.

4 NeuroFlux **Overview**

We develop NeuroFlux, *a CNN training system that leverages adaptive local learning by incorporating adaptive auxiliary networks and adaptive batch sizes within LL to facilitate GPU memory-efficient on-device training in resource-constrained edge environments.* NeuroFlux can be used to train large production quality CNNs on smaller devices with memory budgets that cannot be achieved using conventional BP.

The architecture of NeuroFlux is shown in Figure 7 and the individual modules are considered further in the next section. NeuroFlux takes four inputs (§0): an untrained CNN, a training set, a GPU memory budget, and a batch size limit. The high-level workflow of NeuroFlux is considered next.

Profiler: The Profiler first assigns auxiliary networks to the layers of an input CNN by utilizing the AAN-LL strategy presented in the previous section. Following this allocation, the GPU memory utilization required for training each layer is benchmarked for different batch sizes. These measurements subsequently facilitate the construction of layer-specific linear models for predicting GPU memory consumption based on the batch size (§1).

Partitioner: After obtaining the layer-specific linear models, the Partitioner systematically segments the layers of the given CNN into 'blocks' based on their projected GPU memory consumption patterns. Each block consists of contiguous layers with similar GPU memory footprints. Concurrently, the Partitioner determines the optimal training batch size for each block, ensuring the overall training remains within the stipulated GPU memory budget set by the user (§2).

Controller **and** ***Worker***: After receiving the blocks and their respective training batch sizes, the Controller activates CNN training by deploying the Worker (§3). The Worker loads the current block into GPU memory (§3.1). Subsequently, it trains the layers within this block via sequential LL using the pre-determined batch size for the given block (§3.2). Once the block is trained, the activations of its final layer are saved to a storage device (§3.3; they are not stored in GPU memory). After training, the current block is moved to storage, and the subsequent block is loaded into GPU memory (§3.1). The saved activations are inputs for the subsequent block, ensuring continuity in learning, and eliminating the need for forward passes over the trained block(s). This subsequently reduces the compute requirements, which accelerates the overall CNN training.

Additionally, NeuroFlux uses a prefetching mechanism to adjust the number of activations per batch on the fly, such that it has the same batch size recommended by the Partitioner (§3.2). This facilitates the training of different blocks with varied batch sizes via the AB-LL technique, eliminating the need for an entire CNN to be trained using small batch sizes due to the GPU memory demands arising from the initial layers. This process of blockwise training is repeated until all blocks have been trained.

Compact Trained CNN: The concept of early exits in CNNs allows for inferences to be made before reaching the final layer of a model by adding exit points selectively before the final layers [29, 34, 40, 43, 65]. These are known to offer higher inference efficiency with low latencies. In the context of NeuroFlux, the AAN-LL technique ensures that each CNN layer is paired with its corresponding auxiliary network, which effectively introduces a distinct prediction point
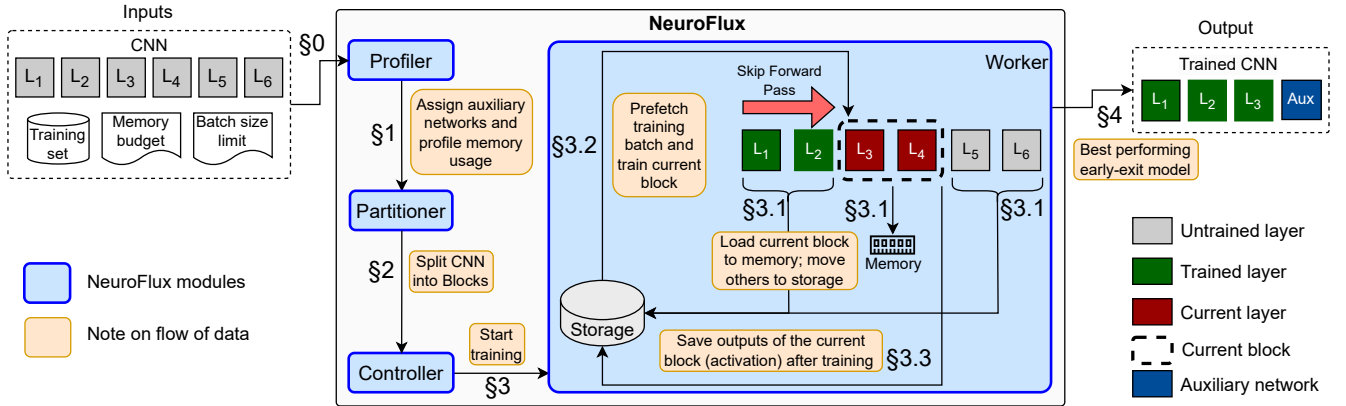
Figure 7. The NeuroFlux architecture.

for every layer. Following training, NeuroFlux systematically evaluates the performance of each layer by taking into account both accuracy and the number of parameters in a given early exit model. Finally, a model with an early exit point that balances the need for achieving high accuracy and has a small resource footprint is generated (§4).

5 Design

We first present the mathematical description of the LL paradigm. Then, we discuss the modules and the techniques employed by NeuroFlux.

5.1 Representing Local Learning

We denote a CNN of N layers as $F(\theta)$, where θ represents the set of parameters $(\theta_1, \theta_2, ..., \theta_N)$. θ_n represents the parameters of the nth layer. The CNN architecture integrates the rectified linear unit (ReLU) non-linearity and a downsampling operation, symbolised by α and P_n, respectively. An input training batch x_0 undergoes transformations across the layers of the CNN, resulting in an output, x_{n+1} at layer n. Subsequently, x_{n+1} is channelled into an auxiliary network, denoted as A_n, which generates an early prediction for the given depth, z_{n+1}. The transformations can be represented mathematically as:

$$\begin{cases} x_{n+1} = \alpha P_n \theta_n x_n \\ z_{n+1} = A_n x_{n+1} \end{cases} \quad (1)$$

The auxiliary network, A_n, is a CNN classifier, consisting of a convolution layer characterized by β_n, followed by a downsampling mechanism, F_n, such as max pooling or average pooling, followed by a linear layer γ_n for prediction. The auxiliary network carries out the following early prediction:

$$A_n x_{n+1} = \gamma_n F_n \beta_n x_{n+1} \quad (2)$$

The objective of training at depth n is based on refining parameters $\{\theta_n, \beta_n, \gamma_n\}$ by solving an auxiliary optimization problem. Given a dataset $\{x^k, y^k\}_{k<K}$, and a loss function ℓ

(e.g., Mean Squared Error or Cross Entropy Error), the goal is to minimize the loss:

$$\mathcal{L}(z_{n+1}; \theta_n, \beta_n, \gamma_n) = \frac{1}{K} \sum_k \ell(z(x^k; \theta_n, \beta_n, \gamma_n), y^k) \quad (3)$$

After the parameters of a layer are updated through the auxiliary optimization problem, the activations x_{n+1} serve as the input to the subsequent layer $n + 1$. This process of layerwise optimization continues sequentially until the input training batch progresses through the final layer.

5.2 Modules

The integrated working of the following modules, namely the *Profiler*, *Partitioner*, *Controller*, and *Worker* enable adaptive local learning in NeuroFlux.

Profiler: Given an input CNN, a training set, a GPU memory budget, and the batch size limit, the Profiler employs the AAN-LL strategy to designate auxiliary networks to the layers of the input CNN. Subsequently, it benchmarks the GPU memory required for training each layer for various batch sizes. We observe that the GPU memory used by each layer linearly correlates with the batch size as shown in Figure 8[1]. By leveraging this observation, the Profiler constructs layerwise linear models to predict the GPU memory requirements for a given batch size (§1).

Partitioner: This module segments the input CNN into distinct blocks by grouping layers based on their GPU memory consumption as shown in Algorithm 1 (§2). Initially, the Partitioner computes the maximum possible batch size each layer can accommodate while adhering to the GPU memory budget constraints, leveraging the linear models produced by the Profiler (refer to lines 2-3). Although the later CNN layers may be capable of supporting batch sizes in the order of thousands as illustrated in Figure 6, traditional CNN training often adopts batch sizes between 32 and 512 [36].

---

[1]A similar trend is noted for larger VGG and ResNet variants but we only provide results for VGG11 to clearly show the linear relationship.
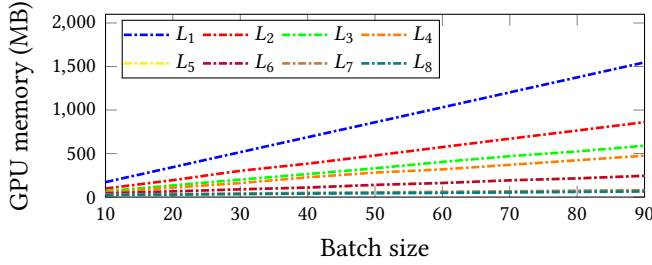
Figure 8. GPU memory required by different layers during training using AAN-LL for varying batch sizes on VGG-11.

Algorithm 1 CNN Partitioning in `NeuroFlux`

Input: GPU memory budget M
Input: Batch size limit B
Input: Layer-wise linear regression models R
Input: Grouping threshold ρ
Output: Partitioned CNN blocks and their respective batch sizes

1: Initialize empty list of blocks $Blocks$
2: **for** each layer l in CNN **do**
3:     $t \leftarrow$ max batch size from $R[l]$ such that GPU memory $\leq M$
4:     $b_l \leftarrow \min(t, B)$
5: **end for**
6: **for** $i = 1$ to number of layers in CNN **do**
7:     Initialize current block $Block \leftarrow \{\}$
8:     $Block$.layers $\leftarrow [i]$
9:     $Block$.batch_size $\leftarrow b_i$
10:    **while** $i+1 \leq$ number of layers **and** $|b_{i+1} - b_i| \leq \rho \times b_i$ **do**
11:        $Block$.batch_size $\leftarrow \min(Block$.batch_size$, b_{i+1})$
12:        Append $i + 1$ to $Block$.layers
13:        Increment i
14:    **end while**
15:    Append $Block$ to $Blocks$
16: **end for**
17: **return** $Blocks$

Employing excessively large batch sizes during training might inadvertently impair generalization [36, 50]. As a remedial strategy, the Partitioner caps the batch size at a predefined limit if its calculated value surpasses this threshold, as specified in line 4. Such a restriction on batch size not only guarantees the feasibility of on-device training but also preserves the ability of the model to generalize.

The Partitioner subsequently utilizes the calculated maximum batch size for each layer as a basis for grouping layers. If the variation in feasible batch sizes between successive layers remains below a 40% margin (defined as the grouping threshold ρ), then they are grouped into one block, as indicated in line 12. The batch size designated to this block defaults to the smallest batch size calculated for its constituent layers, as denoted in line 11. It was empirically found that the 40% threshold is the most effective in balancing training efficiency and model convergence across thresholds spanning 10% to 70%. This choice of threshold is based on an exhaustive analysis, which balances the equilibrium between layer grouping and GPU memory optimization during training. Importantly, blocks may consist of a single layer if the batch sizes of its neighbouring layers are above the threshold.

Controller and Worker: Following the partitioning phase, the Controller designates the training task of the primary block to a specific worker (§3). Once allocated, the worker imports the specified block into its working GPU memory to start training. The other blocks are stored in the storage device at this time (§3.1).

Considering a training set, denoted by $D \triangleq \{x^k, y^k\}_{k<K}$ and with a batch size ascertained by the Partitioner, the worker fetches the training batches into its active GPU memory (§3.2). In tandem, layer-wise training for the layers encapsulated within the block is initiated, adhering to the protocol described in Algorithm 2. The convergence analysis of adaptive local learning underpinning `NeuroFlux` is presented in Appendix B.

During this training phase, the training batches are provided to the block as indicated in lines 1-2. The training batch is processed by the initial layer and produces the activations as specified in line 3. Subsequently, these activations are relayed to the auxiliary network, resulting in the creation of

local predictions as shown in line 4. These predictions are used to compute the loss in line 5. The loss then determines the parameter gradients shown in line 6, which subsequently guide the required parameter updates shown in line 7. Following this update, the generated activations progress from the initial layer to the subsequent layer. This layer-specific procedure ensures that each layer not only processes the data but also refines its parameters. This iterative sequence is continued until the entire batch passes through every layer.

5.3 Efficient Forward Propagation via Prefetching and Adaptive Batching

In LL, forward propagation determines the data flow within the CNN architecture. Each individual block within this architecture inherently depends on the outputs generated by its immediate predecessor. Thus, a computational challenge arises: when a block is trained and requires inputs, the default mechanism is to execute forward propagation on the preceding blocks, even if they have already been trained. This process of generating inputs for an active training block from its already-trained predecessors is not only redundant, but also incurs significant computational overhead, resulting in longer training durations.

`NeuroFlux` addresses this challenge via a caching mechanism. When the training of a block is completed, the activations from the final layer of the block are transferred to

Algorithm 2 Block-wise Local Learning in NeuroFlux

Input: Current *Block* with N layers and batch size B (assigned by the partitioner)
Input: Training batches $D \triangleq \{x^k, y^k\}_{k \leq K}$, where each batch contains B samples / activations
Input: Loss function ℓ
1: **for** batch (x^k, y^k) in D **do**
2:    **for** $n = 1$ to N **do**
3:       $x_{n+1} \leftarrow \alpha P_n \theta_n x_n$ {Using Equation 1}
4:       Compute local prediction $z_{n+1} \leftarrow A_n x_{n+1}$ {Using Equation 2}
5:       Compute loss $\mathcal{L}_n \triangleq \ell(z_{n+1}, y^k)$
6:       Compute $\nabla_{(\theta_n, \beta_n, \gamma_n)} \mathcal{L}_n$
7:       $\theta_n, \beta_n, \gamma_n \leftarrow$ Update based on $\nabla_{(\theta_n, \beta_n, \gamma_n)} \mathcal{L}_n$
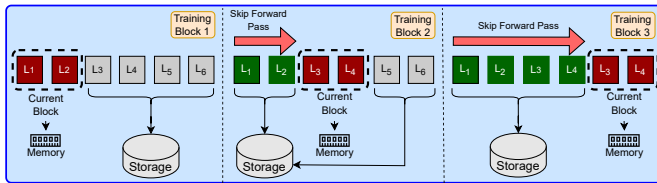8:   **end for**
9: **end for**

Figure 9. Example of skipping forward pass and managing GPU memory in the NeuroFlux training system for three time steps given a CNN with six layers. Red layers indicate the current block of layers being trained, green represents trained layers, and grey denotes untrained layers.

a storage device (§3.3). Subsequently, the trained block is moved to storage and the next block is loaded into GPU memory (§3.1). The cached activations then become the input for the succeeding block. This design eliminates the need for redundant forward passes through already-trained blocks, thereby optimizing the training speed. Figure 9 shows GPU memory management during the training of blocks, illustrating where each block is stored and how the forward pass is skipped as training advances to deeper blocks.

Furthermore, NeuroFlux integrates the AB-LL strategy, enhancing training performance. In this strategy (§3.2), as the activations are retrieved from storage in order to train a block, the number of activations in a given training batch is adjusted on-the-fly, making sure each block is trained with its designated batch size, allowing the current block to immediately use the training batch as input.

A key benefit of adaptive batch sizes is leveraging the unique GPU memory consumption patterns of each block. Initial blocks in a CNN tend to have higher GPU memory requirements, and thus, in edge computing environments with inherently limited working GPU memory, the batch sizes must be modest. However, as the training advances to subsequent blocks, which generally demand less GPU
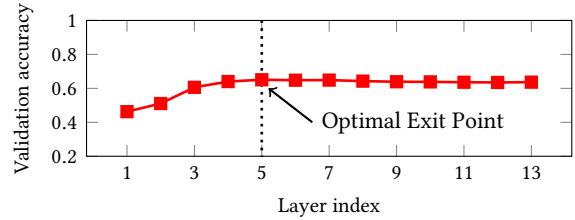

Figure 10. Layer-wise validation accuracy of VGG-16 on CIFAR-100 in NeuroFlux. Layer 5, the optimal exit point, achieves the highest validation accuracy with minimal parameters, making it the ideal early exit point.

memory, the AB-LL strategy permits these blocks to handle larger batch sizes.

AB-LL improves the computational efficiency by reducing the number of iterations required to process the entire dataset. By doing so, NeuroFlux *strikes a crucial balance: it significantly accelerates training while preserving the accuracy of the resultant model.*

5.4 Trained Output CNN

BP and classic LL produce a full-sized CNN after training that can be deployed for end-to-end inference. In contrast, NeuroFlux evaluates the performance of each layer within all possible combinations of early exit models by considering validation accuracy and the number of parameters employed in a given early exit model. The outcome is a model that has an early exit point and achieves the highest validation accuracy while maintaining the smallest parameter count.

The output CNN model from NeuroFlux is spatially efficient, as it contains only a fraction of the parameters compared to the original model. This is a consequence arising from the observation that the validation accuracy approaches a saturation point at a specific layer. Beyond this point, accuracy either remains consistent or decreases only trivially, as illustrated in Figure 10. Such behavior resonates with findings from existing literature, captured under the term *Overthinking* [34]. The concept highlights the diminishing returns of additional computational complexity in CNNs.

Conventionally, two approaches exist for early exit training. Each involves defining a few specific early exit points within a CNN. The first approach entails the simultaneous optimization of the entire CNN from its initialization, resulting in the computation of a weighted loss across all exit points [34]. In contrast, the second approach trains the base model until it converges. Then the early exit classifiers (auxiliary networks) undergo fine-tuning, while the parameters in the base CNN are frozen [40]. Both approaches require at least the same amount of GPU memory as end-to-end BP.

NeuroFlux detects overthinking at the layer level granularity, thereby mitigating the need for subsequent fine-tuning. In NeuroFlux, every CNN layer is a prospective exit point, but the most suitable exit tailored to the distinct
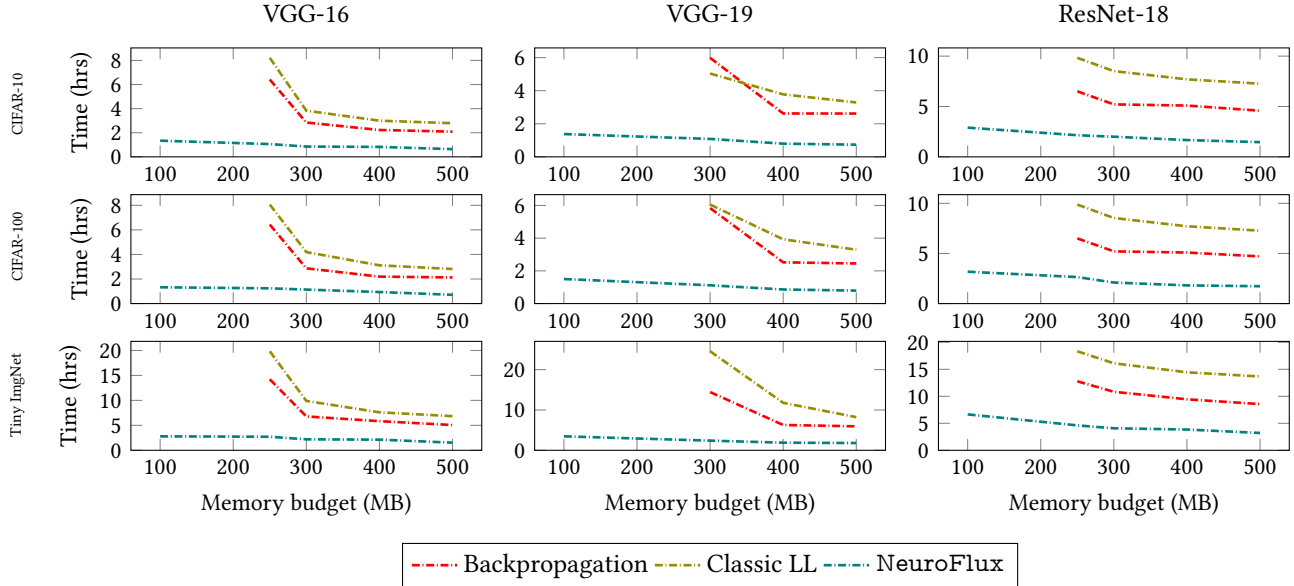
Figure 11. Training time for different memory budgets using different training methods for VGG-16, VGG-19, and ResNet-18 models trained on the CIFAR-10, CIFAR-100, and the Tiny ImageNet datasets trained on the Nvidia AGX Orin.
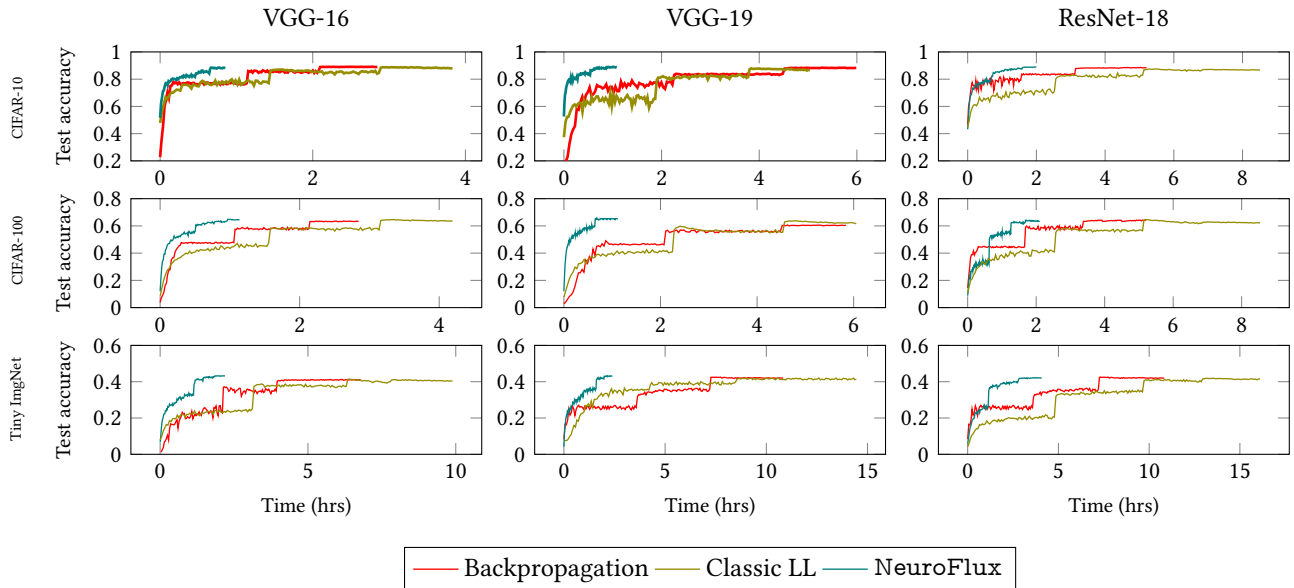
Figure 12. Comparison of test accuracy as training proceeds for different training methods for VGG-16, VGG-19, and ResNet-18 models, trained on the CIFAR-10, CIFAR-100, and Tiny ImageNet datasets. Results are shown for a 300MB memory budget trained on the Nvidia AGX Orin.

requirements of a given task is identified. Thus, the output CNN for on-device efficiency is optimized.

6 Evaluation

In this section, we evaluate the training performance of NeuroFlux against BP and classic LL for varying GPU memory budgets and the inference performance across a range of

edge devices. BP refers to vanilla Backpropagation, which includes no activation/gradient checkpointing [11].

6.1 Experimental Setup

In our experimental setup, we specifically target three platforms, each characterized by distinct resource specifications, as presented in Table 1: the Nvidia Jetson Nano, Nvidia Jetson Xavier NX, and Nvidia Jetson AGX ORIN. Additionally, we

Table 1. Target hardware platforms

Platform	CPU	# CPU Cores	Memory	GPU	# GPU Cores	Peak TFLOPs	TDP
Raspberry Pi 4B (Pi4B)	ARM Cortex-A72	4	4GB LPDDR4	N/A	N/A	0.00969	8W
Nvidia Nano	ARM Cortex-A57	4	4GB LPDDR4	Maxwell	128	0.472	5W
Nvidia Xavier NX	ARM Carmel	6	8GB LPDDR4x	Volta	384	1.33	15W
Nvidia AGX ORIN	ARM Carmel	12	64GB LPDDR5	Ampere	1,536	4.76	50W

employed the Raspberry Pi 4B to demonstrate the inference performance of the output model. To implement `NeuroFlux`, we utilize PyTorch version 2.0.0 and torchvision version 0.2.0, with Python 3.11.

Benchmarks: To assess the generalizability of our system, we conduct benchmark tests using a diverse set of CNNs that vary in depth and computational load. We include prominent VGG variants [63], namely, VGG-16, and VGG-19, which are large and computationally intensive networks with conventional single-layer connectivity. We also employ ResNet-18 [26], from the residual network family, to demonstrate the feasibility of `NeuroFlux`. We have selected large models to demonstrate the efficiency of our system under extreme memory and compute constraints.

Datasets: The effectiveness of `NeuroFlux` is evaluated on three widely used image classification datasets: (1) CIFAR-10, (2) CIFAR-100 datasets [39], and (3) Tiny ImageNet dataset [41]. The datasets are used for image classification and offer a collection of images with varying degrees of complexity for evaluating the performance and accuracy of `NeuroFlux`. The images of Tiny ImageNet were resized from 64×64 pixels to 32×32 pixels to match the input size of CIFAR datasets in order to consistently benchmark across the same CNNs. All measurements are averages of three independent experimental runs.

6.2 End-to-End Training Performance

In this section, we will present results to highlight the efficiency of `NeuroFlux` in training a diverse range of CNNs under various GPU memory constraints. We consider both training time and the accuracy of `NeuroFlux` relative to GPU memory budgets. Figure 11 shows the training time of BP, classic LL and `NeuroFlux` for VGG-16, VGG-19 and ResNet-18 on the Nvidia Jetson AGX ORIN for GPU memory budgets ranging from 100MB to 500MB[2].

> **Observation 1:** `NeuroFlux` outperforms BP and classic LL in training time for all GPU memory budgets.

It is immediately noted from Figure 11 that `NeuroFlux` achieves a reduction in training time - the speed-up ranges

from 2.3× to 6.1× compared to BP and 3.3× to 10.3× compared to classic LL for the same GPU memory budgets. Additionally, `NeuroFlux` can successfully train CNNs under tight GPU memory constraints where BP and classic LL fail to do so. Therefore, there are no data points for BP and classic LL when the GPU memory budget is below 250MB for VGG-16 and ResNet-18 and below 300MB for VGG-19. There is a performance gain when using `NeuroFlux` for more generous GPU memory budgets, such as a 500MB limit.

> **Observation 2:** The performance of `NeuroFlux` on lower GPU memory budgets is better than BP and classic LL on generous GPU memory budgets. It also successfully trains CNNs under GPU memory constraints that are unattainable by BP and classic LL.

`NeuroFlux` on a 100MB budget is 1.3× to 1.9× faster than BP and 2.1× to 2.5× faster than classic LL on a 500MB budget, hence reducing memory usage by 5×. These results validate the efficiency of `NeuroFlux` for CNN training in GPU memory-constrained environments. Figure 12 compares the test accuracy achieved in relation to the time taken for training using BP, classic LL, and `NeuroFlux`. `NeuroFlux` accelerates CNN training while maintaining the accuracy of the final model at the same level achieved by both BP and classic LL.

> **Observation 3:** For a given timeframe, `NeuroFlux` achieves higher accuracy than BP and classic LL.

The inherent GPU memory efficiency of `NeuroFlux` enables it to employ larger training batches compared to BP and classic LL, requiring fewer SGD steps across the entire training dataset. Thus, `NeuroFlux` can reach its peak accuracy faster than the other methods. `NeuroFlux` demonstrates better performance for VGG-19 over ResNet-18. VGG-19 frequently downsamples its activations, leading to smaller activation tensors, as shown in Figure 13 (left). The computational requirements are influenced by the size of activations as they are processed by the auxiliary network during LL. This is highlighted in Figure 13 (right) as the auxiliary networks of VGG-19 require fewer cumulative FLOPS compared to those of ResNet-18.
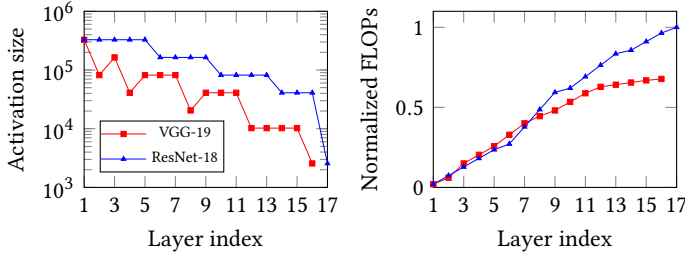
---

[2]Similar results are obtained on other hardware platforms but only show the results on one platform due to the limitation of space.

Figure 13. Comparison of the activation size (total number of elements in the activation tensor) for different layers of VGG-19 and ResNet-18 (left). Normalized Cumulative FLOPs of the auxiliary networks for VGG-19 and ResNet-18 (right).

6.3 Efficiency of Trained Output CNN

Table 2 compares the size of the output model from BP, classic LL and NeuroFlux when trained for different datasets. BP and classic LL produce CNNs of identical sizes. NeuroFlux consistently yields highly efficient early exit CNNs that have 10.9× to 29.4× fewer parameters relative to the full-sized models generated by both BP and classic LL. Anecdotally, the compression levels attained by NeuroFlux are similar to those achieved by structured pruning methods although the two are inherently different approaches [4, 14, 75].

> **Observation 4:** NeuroFlux generates efficient CNNs that are spatially compressed and thus smaller in size than those obtained from BP and classic LL.

Table 2. Parameter count of trained output CNNs.

Dataset	Model	# Parameters (10^6)		Compression Factor
		BP/LL	**NeuroFlux**	
CIFAR-10	VGG-16	14.7	**0.5**	29.4×
	VGG-19	20.0	**1.7**	11.8×
	ResNet-18	11.0	**0.8**	13.8×
CIFAR-100	VGG-16	14.7	**1.19**	12.3×
	VGG-19	20.0	**1.7**	11.8×
	ResNet-18	11.0	**0.96**	11.5×
Tiny ImageNet	VGG-16	14.7	**1.19**	12.3×
	VGG-19	20.0	**1.19**	16.8×
	ResNet-18	11.0	**1.01**	10.9×

This offers dual benefits: firstly, the parameter count of the model is smaller than BP and classic LL, and secondly, it provides an inference throughput gain. Thus, trained models from NeuroFlux are suited for edge deployments, where computational and GPU memory resources are often limited. In Figure 14, we illustrate that there is a significant inference throughput gain achieved by the early exit model produced by NeuroFlux when compared to the trained CNNs of BP and classic LL. The complete throughput data (images per second) is provided in Appendix C. It is immediately evident that the early exit model crafted by NeuroFlux outperforms

the conventional ones, delivering an increased throughput ranging from 1.61× to 3.95× on different target platforms.

6.4 System Overheads

The system overheads in NeuroFlux arise from:

***Profiler* and *Partitioner* modules** construct layer-specific linear models to predict GPU memory consumption based on batch size. Subsequently, the CNN is partitioned into blocks. This processing overhead constitutes less than 1.5% of the total CNN training time across all our experiments.

***Caching* and *Prefetching* techniques** that utilize storage space to retain intermediate activations of trained blocks. These serve as inputs for training subsequent blocks, thereby eliminating the need for a forward pass over trained blocks. Datasets, such as CIFAR-10/100 and Tiny ImageNet, require approximately 0.2GB and 0.5GB of storage, respectively. Our caching strategy increases the storage need for activations to about 1.5× to 5.3× the size of the original dataset. Nonetheless, these storage demands are generally manageable in contemporary edge computing hardware, given their ample storage relative to working memory. Moreover, affordable storage mediums, such as high-speed SD cards, further ease any potential storage constraints.

> **Observation 5:** The performance gain from using NeuroFlux substantially outweighs the overheads incurred by using its modules and underlying techniques.

7 Related work

Optimizing GPU memory usage during CNN training: Three strategies considered below are used for optimizing GPU memory usage in BP-based training:

Sparsity - Sparse tensors are used in this method for representing model parameters to reduce GPU memory requirements, thereby compressing the model without significantly impacting accuracy [10, 19, 23]. These methods either gradually train with sparsity [46, 53] or start with inherently sparse networks [14, 15, 64]. However, sparse tensor operations can be slow due to the irregular distribution of non-zero elements, leading to inefficient data transfers between global GPU memory and computational registers [62].

Microbatching - Smaller batch sizes are used in this method to reduce the GPU memory required for activations [30, 32]. This method is GPU memory efficient but requires precise hyperparameter tuning and increases the overall training time. This method may negatively impact the final accuracy of a model.

Gradient Checkpointing - GPU memory is conserved in this method by saving only certain model activations. The trade-off is the recalculation of activations that are not saved, which results in longer training times [11] than BP.
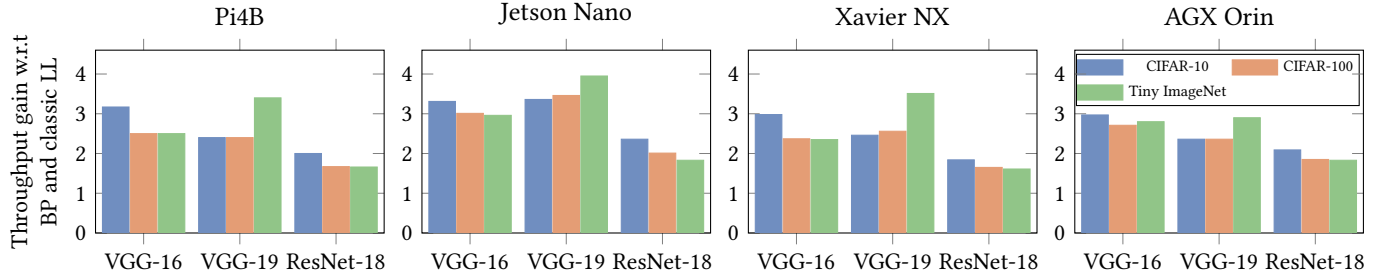
Figure 14. Inference throughput gain achieved by `NeuroFlux` over BP and classic LL for VGG-16, VGG-19 and ResNet-18 models on different hardware platforms using CIFAR-10, CIFAR-100 and Tiny ImageNet datasets.

Accelerating BP-based CNN Training: Two key approaches for enhancing the training speed and efficiency of CNNs are considered next.

Parameter Freezing - The concept of freezing parameters of the earlier layers of a CNN is known to improve training speed. For instance, SpotTune [18] employs pre-trained weights for CNN fine-tuning, and it determines which layers need to be trained on a per-sample basis. If a given sample does not require fine-tuning of the earlier CNN layers, then the parameter updates over those layers can be skipped, resulting in improved computational efficiency. However, the approach is limited to fine-tuning scenarios. In contrast, FreezeOut [8] accelerates CNN training by systematically freezing layers and does not require backward passes over the frozen layers. This results in a 1.2× training speedup for ResNets, but shows no improvement for VGG models.

Caching Activations - Egeria [69] combines layer-freezing and the caching of intermediate results, thereby eliminating forward passes through the frozen layers. It improves the runtime by 1.19×-1.43×. However, Egeria is designed for HPC settings. To illustrate this, for facilitating training on GPUs Egeria requires a CPU-bound quantized reference model that identifies the layers to be frozen — a configuration unsuited for resource-constrained edge or mobile environments. Also, Egeria requires unfreezing layers when the learning rate reduces, as it relies on end-to-end BP. In contrast, `NeuroFlux` underpinned by adaptive local learning ensures that the frozen layers are unaffected by learning rate changes of the subsequent layers. Lastly, PipeTransformer [24] is designed exclusively for Transformer models that utilize freezing progressive layers and caching intermediate results. The training pipeline is further optimized by omitting frozen layers, efficiently allocating active layers across fewer GPUs. This approach is suitable in HPC settings and is unattainable on edge or mobile devices due to inherent resource constraints, often limited to a single and small GPU.

Building on the concept of optimizing computational efficiency, Skip-Convolutions [21] re-engineer standard convolutions that are directly applied to residual frames. Using binary gates, the relevance of each layer to model prediction is evaluated — foreground elements are processed, while background elements may be bypassed to reduce computational overheads. Egeria [69] explores the application of Skip-Convolutions as a heuristic during training to determine which layers to freeze, thereby reducing training time over Backpropagation by approximately 1.28×. On the other hand, `NeuroFlux` reduces training time by 2.3× to 6.1× under stringent GPU memory budgets.

Comparing with Neural Architecture Search: Neural Architecture Search (NAS) is aimed at automating the design of neural network architectures. Leading NAS methods such as DARTS [47] and PreNAS [67] require extensive computational resources, typically between 2 to 10 GPU days on server-grade hardware for CIFAR-10 [14], which is a relatively small dataset, and produces models with an average parameter count ranging from 3.3M to 77M [14]. In comparison, `NeuroFlux` produces a trained model in under two hours on Nvidia Orin, an edge computing resource, with an average of 1M parameters on larger datasets, such as CIFAR-100 and Tiny ImageNet.

Alternate Training Paradigms: We focus on the below two paradigms that are developed as alternatives to BP-based training, which were introduced in Section 2.

Feedback Alignment (FA): FA tackles the 'weight transport problem' of CNN training [44, 45]. Traditional BP-based training presumes a weight symmetry, where the backward pass employs the transposed weights from the forward pass. FA disrupts this symmetry: during the backward pass, it employs fixed random weights distinct from the forward pass. Although FA and its variants match BP in performance for smaller, fully connected networks, they do not work as well with CNNs [37].

Signal Propagation (SP): SP employs a forward-pass training mechanism similar to the independent layer-wise approach used in LL, without the need for layer-specific auxiliary networks [37]. SP utilizes a target generator that recasts target labels into input dimensions, thus generating the 'context'. Each layer handles this context and the initial input, resulting in two discrete sets of activations. Predictions within layers are subsequently derived from the dot product

of these activations, and the layer parameters are adjusted accordingly. While SP is over three times more GPU memory efficient than BP, it has lower training accuracy on small datasets, such as CIFAR-10. The inability to scale effectively to larger datasets, such as CIFAR-100 and Tiny ImageNet, limits its widespread adoption for training.

Both FA and SP have limitations, as mentioned in Section 2 and highlighted in Figure 3. This positions LL, despite its GPU memory requirements, as the most viable alternative to traditional BP. NeuroFlux builds upon this foundation by developing adaptive local learning described in Section 3. This approach not only addresses the GPU memory requirements inherent to LL but also significantly accelerates training compared to both classic LL and BP, while maintaining accuracy across multiple datasets and CNN architectures, as empirically validated in Section 6.

8 Future Directions and Conclusions

This paper demonstrates the feasibility of on-device CNN training using NeuroFlux, both theoretically and experimentally. Its efficacy is not limited to vision-based tasks; NeuroFlux also exhibits significant potential in training speech recognition models [1–3, 54]. Another area is extending NeuroFlux to support transformers, which are pivotal to NLP tasks. The complexity of training transformers is underscored by the GPU memory requirement (at least 8GB) for training BERT-small [13].

Additionally, with the rise of federated learning for decentralized model training, NeuroFlux offers a compelling advantage of improved training time. Efficient on-device training is essential to making federated learning feasible for edge and mobile devices. We envision that the training and memory efficiency offered by NeuroFlux will enable the convergence of the global model more rapidly. This direction will extend the utility of NeuroFlux *in the context of distributed machine learning*.

While NeuroFlux is designed for memory-constrained edge and mobile environments, it also offers benefits for desktop and server systems. These systems can also benefit from the reduced memory usage of NeuroFlux and faster training times in scenarios when training large models on large datasets, such as ResNet-50 [26] on ImageNet [12]. However, the extreme memory constraints that NeuroFlux can operate in may not be of concern in larger desktop and server systems.

In summary, we introduced NeuroFlux, a system optimized for on-device CNN training in memory-limited environments. Unlike traditional Backpropagation, NeuroFlux uses adaptive local learning, showing consistent advancements across various CNNs and datasets. It reliably outperforms Backpropagation and classic local learning, with training speed improvements of 2.3× to 6.1× against Backpropagation and 3.3× to 10.3× against classic local learning for the

same GPU memory budgets. NeuroFlux generates a trained early exit model that reduces parameter counts by 10.9× to 29.4×, boosting inference throughput by 1.61× to 3.95× on our experimental setup comprising diverse hardware platforms. NeuroFlux provides a disruptive memory-efficient training solution that will enable the vision of practical on-device training in memory-constrained environments.

References

[1] Babak Joze Abbaschian, Daniel Sierra-Sosa, and Adel Said Elmaghraby. 2021. Deep Learning Techniques for Speech Emotion Recognition, from Databases to Models. *Sensors*.

[2] Ahmed M. Abdelmoniem, Atal Narayan Sahu, Marco Canini, and Suhaib A. Fahmy. 2023. REFL: Resource-Efficient Federated Learning. In *European Conference on Computer Systems*.

[3] Samson Akinpelu, Serestina Viriri, and Adekanmi Adegun. 2023. Lightweight Deep Learning Framework for Speech Emotion Recognition. *IEEE Access*.

[4] Milad Alizadeh, Shyam A. Tailor, Luisa M Zintgraf, Joost van Amersfoort, Sebastian Farquhar, Nicholas Donald Lane, and Yarin Gal. 2022. Prospect Pruning: Finding Trainable Weights at Initialization using Meta-Gradients. In *International Conference on Machine Learning*.

[5] Eugene Belilovsky, Michael Eickenberg, and Edouard Oyallon. 2019. Greedy Layerwise Learning Can Scale To ImageNet. In *International Conference on Machine Learning*.

[6] Eugene Belilovsky, Michael Eickenberg, and Edouard Oyallon. 2020. Decoupled Greedy Learning of CNNs. In *International Conference on Machine Learning*.

[7] Léon Bottou, Frank E. Curtis, and Jorge Nocedal. 2018. Optimization Methods for Large-Scale Machine Learning. *SIAM Rev.*

[8] Andrew Brock, Theodore Lim, J. M. Ritchie, and Nick Weston. 2017. FreezeOut: Accelerate Training by Progressively Freezing Layers. *arXiv:abs/1706.04983*.

[9] Han Cai, Ligeng Zhu, and Song Han. 2019. ProxylessNAS: Direct Neural Architecture Search on Target Task and Hardware. In *International Conference on Learning Representations*.

[10] Miguel A. Carreira-Perpinan and Yerlan Idelbayev. 2018. "Learning-Compression" Algorithms for Neural Net Pruning. In *IEEE Conference on Computer Vision and Pattern Recognition*.

[11] Tianqi Chen, Bing Xu, Chiyuan Zhang, and Carlos Guestrin. 2016. Training Deep Nets with Sublinear Memory Cost. *arXiv:abs/1604.06174*.

[12] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. 2009. ImageNet: A Large-Scale Hierarchical Image Database. In *IEEE Conference on Computer Vision and Pattern Recognition*.

[13] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *Proceedings of the Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*.

[14] Bailey J. Eccles, Philip Rodgers, Peter Kilpatrick, Ivor Spence, and Blesson Varghese. 2024. DNNShifter: An Efficient DNN Pruning System for Edge Computing. *Future Generation Computer Systems*.

[15] Jonathan Frankle and Michael Carbin. 2019. The Lottery Ticket Hypothesis: Finding Sparse, Trainable Neural Networks. In *International Conference on Learning Representations*.

[16] In Gim and JeongGil Ko. 2022. Memory-efficient DNN training on mobile devices. In *International Conference on Mobile Systems, Applications and Services (MobiSys '22)*.

[17] Junyao Guo, Unmesh Kurup, and Mohak Shah. 2021. Efficacy of Model Fine-Tuning for Personalized Dynamic Gesture Recognition. In *Deep Learning for Human Activity Recognition*.

[18] Yunhui Guo, Honghui Shi, Abhishek Kumar, Kristen Grauman, Tajana Rosing, and Rogerio Feris. 2019. SpotTune: Transfer Learning Through Adaptive Fine-Tuning. In *IEEE Conference on Computer Vision and Pattern Recognition*.

[19] Yiwen Guo, Anbang Yao, and Yurong Chen. 2016. Dynamic Network Surgery for Efficient DNNs. In *International Conference on Neural Information Processing Systems*.

[20] Otkrist Gupta and Ramesh Raskar. 2018. Distributed Learning of Deep Neural Network over Multiple Agents. *Journal of Network and Computer Applications*.

[21] Amirhossein Habibian, Davide Abati, Taco Cohen, and Babak Ehteshami Bejnordi. 2021. Skip-Convolutions for Efficient Video Processing. In *IEEE Conference on Computer Vision and Pattern Recognition*.

[22] Dong-Jun Han, Do-Yeon Kim, Minseok Choi, Christopher G. Brinton, and Jaekyun Moon. 2022. SplitGP: Achieving Both Generalization and Personalization in Federated Learning. *IEEE Conference on Computer Communications*.

[23] Song Han, Huizi Mao, and William J. Dally. 2016. Deep Compression: Compressing Deep Neural Network with Pruning, Trained Quantization and Huffman Coding. In *International Conference on Learning Representations*.

[24] Chaoyang He, Shen Li, Mahdi Soltanolkotabi, and Salman Avestimehr. 2021. PipeTransformer: Automated Elastic Pipelining for Distributed Training of Large-scale Models. In *International Conference on Machine Learning*.

[25] Kaiming He, Georgia Gkioxari, Piotr Dollár, and Ross Girshick. 2020. Mask R-CNN. *IEEE Transactions on Pattern Analysis and Machine Intelligence*.

[26] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep Residual Learning for Image Recognition. In *IEEE Conference on Computer Vision and Pattern Recognition*.

[27] Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. 2017. MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications. *arXiv:abs/1704.04861*.

[28] Baojin Huang, Zhongyuan Wang, Guangcheng Wang, Kui Jiang, Zheng He, Hua Zou, and Qin Zou. 2021. Masked Face Recognition Datasets and Validation. In *2021 IEEE/CVF International Conference on Computer Vision Workshops*.

[29] Gao Huang, Danlu Chen, Tianhong Li, Felix Wu, Laurens van der Maaten, and Kilian Weinberger. 2018. Multi-Scale Dense Networks for Resource Efficient Image Classification. In *International Conference on Learning Representations*.

[30] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Mia Xu Chen, Dehao Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V. Le, Yonghui Wu, and Zhifeng Chen. 2019. GPipe: Efficient Training of Giant Neural Networks Using Pipeline Parallelism. *International Conference on Neural Information Processing Systems*.

[31] Sinh Huynh, Rajesh Balan, and Jeonggil Ko. 2021. iMon: Appearance-based Gaze Tracking System on Mobile Devices. In *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies*.

[32] Sergey Ioffe. 2017. Batch Renormalization: Towards Reducing Minibatch Dependence in Batch-Normalized Models. In *Advances in Neural Information Processing Systems*.

[33] Joseph Bailey Luttrell Iv, Zhaoxian Zhou, Chaoyang Zhang, Ping Gong, and Yuanyuan Zhang. 2017. Facial Recognition via Transfer Learning: Fine-Tuning Keras_vggface. *International Conference on Computational Science and Computational Intelligence*.

[34] Yigitcan Kaya, Sanghyun Hong, and Tudor Dumitras. 2018. Shallow-Deep Networks: Understanding and Mitigating Network Overthinking. In *International Conference on Machine Learning*.

[35] Ira Kemelmacher-Shlizerman, Steven M Seitz, Daniel Miller, and Evan Brossard. 2016. The MegaFace Benchmark: 1 Million Faces for Recognition at Scale. In *IEEE Conference on Computer Vision and Pattern Recognition*.

[36] Nitish Shirish Keskar, Dheevatsa Mudigere, Jorge Nocedal, Mikhail Smelyanskiy, and Ping Tak Peter Tang. 2017. On Large-Batch Training for Deep Learning: Generalization Gap and Sharp Minima. In *International Conference on Learning Representations*.

[37] Adam Kohan, Edward A. Rietman, and Hava T. Siegelmann. 2023. Signal Propagation: The Framework for Learning and Inference in a Forward Pass. *IEEE Transactions on Neural Networks and Learning Systems*.

[38] Alexandros Kouris and Christos-Savvas Bouganis. 2018. Learning to Fly by MySelf: A Self-Supervised CNN-Based Approach for Autonomous Navigation. In *IEEE/RSJ International Conference on Intelligent Robots and Systems*.

[39] Alex Krizhevsky. 2009. Learning Multiple Layers of Features from Tiny Images. *Technical Report*. https://www.cs.toronto.edu/ kriz/cifar.html.

[40] Stefanos Laskaridis, Stylianos I. Venieris, Hyeji Kim, and Nicholas D. Lane. 2020. HAPI: Hardware-Aware Progressive Inference. In *International Conference on Computer-Aided Design*.

[41] Ya Le and Xuan S. Yang. 2015. Tiny ImageNet Visual Recognition Challenge. http://vision.stanford.edu/teaching/cs231n/reports/2015/pdfs/yle_project.pdf

[42] Juhyun Lee, Nikolay Chirkov, Ekaterina Ignasheva, Yury Pisarchyk, Mogan Shieh, Fabio Riccardi, Raman Sarokin, Andrei Kulik, and Matthias Grundmann. 2019. On-Device Neural Net Inference with Mobile GPUs. *arXiv:abs/1907.01989*.

[43] H. Li, H. Zhang, X. Qi, Y. Ruigang, and G. Huang. 2019. Improved Techniques for Training Adaptive Deep Networks. In *IEEE/CVF International Conference on Computer Vision*.

[44] Qianli Liao, Joel Z. Leibo, and Tomaso Poggio. 2016. How Important is Weight Symmetry in Backpropagation?. In *AAAI Conference on Artificial Intelligence*.

[45] Timothy P. Lillicrap, Daniel Cownden, Douglas Blair Tweed, and Colin J. Akerman. 2016. Random Synaptic Feedback Weights Support Error Backpropagation for Deep Learning. *Nature Communications*.

[46] Tao Lin, Sebastian U. Stich, Luis Barba, Daniil Dmitriev, and Martin Jaggi. 2020. Dynamic Model Pruning with Feedback. In *International Conference on Learning Representations*.

[47] Hanxiao Liu, Karen Simonyan, and Yiming Yang. 2019. DARTS: Differentiable Architecture Search. In *International Conference on Learning Representations*.

[48] Joseph Luttrell, Zhaoxian Zhou, Yuanyuan Zhang, Chaoyang Zhang, Ping Gong, Bei Yang, and Runzhi Li. 2018. A Deep Transfer Learning Approach to Fine-Tuning Facial Recognition Models. In *IEEE Conference on Industrial Electronics and Applications*.

[49] Bishwas Mandal, Adaeze Okeukwu, and Yihong Theis. 2021. Masked Face Recognition using ResNet-50. *arXiv:abs/2104.08997*.

[50] Dominic Masters and Carlo Luschi. 2018. Revisiting Small Batch Training for Deep Neural Networks. *arXiv:abs/1804.07612*.

[51] Brendan McMahan, Eider Moore, Daniel Ramage, Seth Hampson, and Blaise Aguera y Arcas. 2017. Communication-Efficient Learning of Deep Networks from Decentralized Data. In *International Conference on Artificial Intelligence and Statistics*.

[52] Pavlo Molchanov, Stephen Tyree, Tero Karras, Timo Aila, and Jan Kautz. 2017. Pruning Convolutional Neural Networks for Resource Efficient Inference. In *International Conference on Learning Representations*.

[53] Hesham Mostafa and Xin Wang. 2019. Parameter Efficient Training of Deep Convolutional Neural Networks by Dynamic Sparse Reparameterization. In *International Conference on Machine Learning*.

[54] Sarala Padi, Seyed Omid Sadjadi, Dinesh Manocha, and Ram D. Sriram. 2021. Improved Speech Emotion Recognition using Transfer Learning and Spectrogram Augmentation. *Proceedings of the International Conference on Multimodal Interaction*.

[55] German I. Parisi, Ronald Kemker, Jose L. Part, Christopher Kanan, and Stefan Wermter. 2019. Continual Lifelong Learning with Neural Networks: A Review. *Neural Networks* 113 (2019), 54–71.

[56] HyeonJung Park, Youngki Lee, and JeongGil Ko. 2021. Enabling Real-time Sign Language Translation on Mobile Platforms with On-board Depth Cameras. In *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies*.

[57] David Patterson, Joseph Gonzalez, Urs Hölzle, Quoc Le, Chen Liang, Lluis-Miquel Munguia, Daniel Rothchild, David R. So, Maud Texier, and Jeff Dean. 2022. The Carbon Footprint of Machine Learning Training Will Plateau, Then Shrink. *Computer* (2022).

[58] Hieu Pham, Melody Guan, Barret Zoph, Quoc Le, and Jeff Dean. 2018. Efficient Neural Architecture Search via Parameters Sharing. In *International Conference on Machine Learning*.

[59] Jaya Prakash Sahoo, Allam Jaya Prakash, Paweł Pławiak, and Saunak Samantray. 2022. Real-Time Hand Gesture Recognition Using Fine-Tuned Convolutional Neural Network. *Sensors*.

[60] F. Sarfraz, E. Arani, and B. Zonooz. 2021. Knowledge Distillation Beyond Model Compression. In *International Conference on Pattern Recognition*.

[61] Roy Schwartz, Jesse Dodge, Noah A. Smith, and Oren Etzioni. 2020. Green AI. *Commun. ACM* (2020).

[62] Shaohuai Shi, Qiang Wang, and Xiaowen Chu. 2020. Efficient Sparse-Dense Matrix-Matrix Multiplication on GPUs Using the Customized Sparse Storage Format. In *IEEE International Conference on Parallel and Distributed Systems*.

[63] Karen Simonyan and Andrew Zisserman. 2015. Very Deep Convolutional Networks for Large-Scale Image Recognition. In *International Conference on Learning Representations*.

[64] Hidenori Tanaka, Daniel Kunin, Daniel L Yamins, and Surya Ganguli. 2020. Pruning neural networks without any data by iteratively conserving synaptic flow. In *Advances in Neural Information Processing Systems*.

[65] Surat Teerapittayanon, Bradley McDanel, and H.T. Kung. 2016. BranchyNet: Fast inference via early exiting from deep neural networks. In *International Conference on Pattern Recognition*.

[66] Praneeth Vepakomma, Otkrist Gupta, Tristan Swedish, and Ramesh Raskar. 2018. Split Learning for Health: Distributed Deep Learning Without Sharing raw patient data. *arXiv:abs/1812.00564*.

[67] Haibin Wang, Ce Ge, Hesen Chen, and Xiuyu Sun. 2023. PreNAS: Preferred One-Shot Learning Towards Efficient Neural Architecture Search. In *International Conference on Machine Learning*.

[68] Siqi Wang, Anuj Pathania, and Tulika Mitra. 2020. Neural Network Inference on Mobile SoCs. *IEEE Design & Test*.

[69] Yiding Wang, Decang Sun, Kai Chen, Fan Lai, and Mosharaf Chowdhury. 2022. Egeria: Efficient DNN Training with Knowledge-Guided Layer Freezing. *European Conference on Computer Systems*.

[70] Zhiyuan Wang, Hongli Xu, Yang Xu, Zhida Jiang, and Jianchun Liu. 2023. CoopFL: Accelerating Federated Learning with DNN Partitioning and Offloading in Heterogeneous Edge Computing. *Comput. Netw.*

[71] Bichen Wu, Forrest Iandola, Peter H. Jin, and Kurt Keutzer. 2017. SqueezeDet: Unified, Small, Low Power Fully Convolutional Neural Networks for Real-Time Object Detection for Autonomous Driving. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops*.

[72] Di Wu, Rehmat Ullah, Paul Harvey, Peter Kilpatrick, Ivor Spence, and Blesson Varghese. 2022. FedAdapt: Adaptive Offloading for IoT Devices in Federated Learning. *IEEE Internet of Things Journal*.

[73] Jiaxiang Wu, Cong Leng, Yuhang Wang, Qinghao Hu, and Jian Cheng. 2016. Quantized Convolutional Neural Networks for Mobile Devices. In *IEEE Conference on Computer Vision and Pattern Recognition*.

[74] Fang Yu, Li Cui, Pengcheng Wang, Chuanqi Han, Ruoran Huang, and Xi Huang. 2021. EasiEdge: A Novel Global Deep Neural Networks Pruning Method for Efficient Edge Computing. *IEEE Internet of Things*

Journal (2021).

[75] Ruizhe Zhao and Wayne W. C. Luk. 2018. Efficient Structured Pruning and Architecture Searching for Group Convolution. *2019 IEEE/CVF International Conference on Computer Vision Workshop*.

[76] Barret Zoph and Quoc Le. 2017. Neural Architecture Search with Reinforcement Learning. In *International Conference on Learning Representations*.

Appendix

A Backpropagation is Memory Intensive

Consider an N-layer CNN, denoted by $F(\theta)$, where θ belongs to the set $\{\theta_1, \theta_2, ..., \theta_N\}$ and θ_n specifies the parameters in the nth layer. Backpropagation-based training operates in two phases: the forward pass and the backward pass.

During the forward pass, the training input x traverses through the CNN $F(\theta)$ with all intermediate activations $a \in \{a_1, a_2, ..., a_N\}$ being retained, thereby yielding the output $\hat{y} = F(\theta; x)$. This output $\hat{y}$, in conjunction with the target output y, is fed into a global loss function $L(y, \hat{y})$ that represents the quality of the predicted output.

Computation of the gradients $\nabla L(\theta_n)$ proceeds via the chain rule, as illustrated below:

$$\frac{\partial L}{\partial \theta_n} = \frac{\partial L}{\partial a_n} \times \frac{\partial a_n}{\partial z_n} \times \frac{\partial z_n}{\partial \theta_n} \qquad (4)$$

The output of a neuron before activation is computed as:

$$z_n = a_{n-1} \cdot \theta_n \qquad (5)$$

Differentiating this w.r.t. θ_n gives:

$$\frac{\partial z_n}{\partial \theta_n} = a_{n-1} \qquad (6)$$

The output after the activation function, σ, is given by:

$$a_n = \sigma(z_n) \qquad (7)$$

Considering the dependence on the activation function, the gradient of the loss with respect to the pre-activation output is:

$$\frac{\partial a_n}{\partial z_n} = \sigma'(z_n) \qquad (8)$$

Substituting into Equation 4 gives:

$$\frac{\partial L}{\partial \theta_n} = \frac{\partial L}{\partial a_n} \times \sigma'(z_n) \times a_{n-1} \qquad (9)$$

After computing the gradients for the final layer, the backward pass begins to systematically propagate gradients from the last to the initial layer. The backward pass necessitates the preservation of all activations during the forward pass. This is because the gradient computation for layer n in the backward pass depends on the activations from the prior layer $(n-1)$ as shown in Equation 9. These gradients guide the parameter updates aimed at reducing the training loss.

B   Theoretical Analysis

The theoretical foundations of end-to-end CNN training using classic local learning have been presented in the literature [6]. The NeuroFlux approach advances local learning by offering a strategy to segment the CNN into M distinct blocks and train each block sequentially with unique batch sizes. Our analysis builds on existing work [6], but our objective is to explore adaptive local learning underpinning NeuroFlux that has not been considered elsewhere. Convergence is guaranteed under reasonable assumptions [7]. Our starting point is the convergence dynamics of the first block. These are akin to patterns observed in smaller CNNs when trained using local learning [6].

B.1   Convergence of the First Block

Consider a block at iteration t with N layers. The parameters of the nth layer (including the auxiliary parameters) are represented by Ψ_n^t. While conventional end-to-end BP-based training maintains a fixed input distribution to the model, this is not the case with local learning. The input distribution to subsequent layers in local learning is time-varying, except for the first layer, which maintains a fixed input distribution. This time-varying nature of the input distribution is attributed to the fact that each layer, (except the first), is trained based on outputs generated from its preceding layer. As data is propagated and the layers are updated, the distribution of the outputs change over iterations.

The training batches can be represented as:

$$D_{j_n}^t \triangleq \{x_{j_n}^t, y_j\}_{j \leq J} \tag{10}$$

where $x_{j_n}^t$ denotes the jth output batch of the nth layer (input batch to the $(n+1)^{th}$ layer) at iteration t, and its corresponding target batch y_j remains invariant over all iteration t and layers n. Meanwhile, J represents the cumulative number of training batches. Each training batch follows the density $\rho_n^t(k)$. For layers with $n > 1$, the convergence density of its previous layer is $\rho_{n-1}^*(k)$. The *drift* of the preceding layer is described as:

$$s_{n-1}^t = \int |\rho_{n-1}^t(k) - \rho_{n-1}^*(k)| dk \tag{11}$$

The *drift* for the initial layer ($n = 1$) is zero, since its input distribution remains constant. The expected risk is:

$$\mathcal{L} = \mathbb{E}_{\rho_{n-1}^*}[\ell(D_{j_{n-1}}; \Psi_n)] \tag{12}$$

where ℓ represents an invariably non-negative loss function.

Consider the following assumptions. Assumption 1 to Assumption 3 are standard non-convex setting [7] and Assumption 4 considers the time-varying input distribution [6]

- **Assumption 1** (L-smoothness and Differentiability): The gradient of $\mathcal{L}$ is L-Lipschitz continuous. The learning rate of SGD is represented as η_t and for $D_{j_{n-1}}^t \sim \rho_{n-1}^t(k)$. Our parameter sequence is:

$$\Psi_n^{t+1} = \Psi_n^t - \eta_t \nabla_{\Psi_n} \ell(D_{j_{n-1}}^t; \Psi_n^t) \tag{13}$$

- **Assumption 2** (Robbins-Monro Criterion):

$$\sum_t \eta_t = \infty \quad \text{and} \quad \sum_t \eta_t^2 < \infty \tag{14}$$

- **Assumption 3** (Bounded Variance): A boundary $G > 0$ exists such that for every t and Ψ_n:

$$\mathbb{E}_{\rho_{n-1}^t}[||\nabla_{\Psi_n} \ell(D_{j_{n-1}}; \Psi_n)||^2] \leq G \tag{15}$$

- **Assumption 4** (Stabilization of Prior Layer):

$$\sum_t s_{n-1}^t < \infty \tag{16}$$

Given Assumption 3 and Assumption 4 and as already demonstrated in Lemma 4.1 presented in the literature [6]:

$$\mathbb{E}_{\rho_{n-1}^*}[||\nabla_{\Psi_n} \ell(D_{j_{n-1}}; \Psi_n)||^2] \leq G \tag{17}$$

Additionally, from Lemma 4.2 presented in the literature [6] and given Assumption 1, Assumption 3 and Assumption 4:

$$\mathbb{E}[\mathcal{L}(\Psi_n^{t+1})] \leq \mathbb{E}[\mathcal{L}(\Psi_n^t)]$$
$$+ \frac{LG}{2} \eta_t^2 - \eta_t (\mathbb{E}[||\nabla \mathcal{L}(\Psi_n^t)||^2] - \sqrt{2} G s_{n-1}^t) \tag{18}$$

Furthermore, based on the above four assumptions and using Proposition 4.2 presented in the literature [6], each term in the following equation converges.

$$\sum_{t=0}^T \eta_t \mathbb{E}[||\nabla \mathcal{L}(\Psi_n^t)||^2]$$
$$\leq \mathbb{E}[\mathcal{L}(\Psi_n^0)] + G \sum_{t=0}^T \eta_t (\sqrt{2 s_{n-1}^t} + \frac{L \eta_t}{2}) \tag{19}$$

Hence, the first block exhibits convergence. This is in alignment with the convergence criteria detailed in the literature [7] and resembles end-to-end local learning of a small CNNs [6]. Upon completing the training of the first block, which comprises N layers, the trained output batches (activations) from the final layer, denoted as $D_{j_N}^* \triangleq \{x_{j_N}^*, y_j\}_{j<J}$, are saved to storage.

B.2   Convergence analysis of the Subsequent Blocks

B.2.1   Setting for the Second Block: Consider the second block at iteration t, comprised of M layers. While the first layer of the second block can be perceived as the $N + 1$ layer from the perspective of the previous block, we will use the index m to clarify layers within the second block, where $m = 1, \ldots, M$. The outputs (activations) from the final layer of the initial block become the inputs for this block.

Notably, this block is trained using a distinct batch size, leading to a different number of training batches, Q. These training batches are represented as:

$$Z_{q_m}^t \triangleq \{x_{q_m}^t, y_q\}_{q<Q} \tag{20}$$

The input distribution for the layer $m = 1$ is static, reflecting the training dynamics of the first layer in the primary block. This is because the inputs for this layer are obtained

from the outputs of the last layer of the preceding block, fetched directly from storage.

Each training batch conforms to the density $\gamma_m^t(p)$. For layers with $m > 1$, we define the convergence density of its preceding layer as $\gamma_{m-1}^*(p)$. This gives rise to the drift of the previous layer, defined as:

$$s_{m-1}^t = \int |\gamma_{m-1}^t(p) - \gamma_{m-1}^*(p)| dp \qquad (21)$$

The resultant expected risk is:

$$\mathcal{L} = \mathbb{E}_{\gamma_{m-1}^*}[\ell(Z_{q_{m-1}}; \Psi_m)] \qquad (22)$$

B.2.2 Analytical Similarity with the First Block: The analytical approach to understand the convergence of the second block shares resemblance with that of the first block. The distinction arises from the unique training dataset and batch size of the current block. The mathematical representations and assumptions applied to the first block can similarly be applied to the second block, establishing convergence on the second block. The analysis reinforces that the second block can be conceptualized as an independent CNN with its own dataset and batch size.

B.2.3 Extension to Subsequent Blocks: Drawing upon the understanding of the first two blocks, it becomes clear that the analytical framework can be applied to all subsequent blocks within the CNN when utilizing adaptive local learning. The convergence of the entire CNN using NeuroFlux can be understood as training several smaller, discrete, and independent CNNs via local learning [6], with each having its own dataset and batch size.

C Inference Throughput

Table 3 shows the inference throughput measured as images per second (images/s) of BP, classic LL and NeuroFlux on different hardware platforms for VGG-16, VGG-19 and ResNet-18 on three different datasets. The speed-up NeuroFlux achieves is shown alongside. BP and classic LL have the same throughput since the trained DNNs produced by these methods have the same number of parameters.

Table 3. Inference throughput (images per second; images/s) of the trained output DNN on different platforms, including the Tiny ImageNet dataset.

Platform	Dataset	Model	images/s		Speedup
			BP/classic LL	NeuroFlux	
Pi4B	CIFAR-10	VGG-16	6	**19**	3.17x
		VGG-19	5	**12**	2.40x
		ResNet-18	3	**6**	2.00x
	CIFAR-100	VGG-16	6	**15**	2.50x
		VGG-19	5	**12**	2.40x
		ResNet-18	3	**5**	1.67x
	Tiny ImageNet	VGG-16	6	**15**	2.50x
		VGG-19	5	**17**	3.40x
		ResNet-18	3	**5**	1.66x
Jetson Nano	CIFAR-10	VGG-16	213	**706**	3.31x
		VGG-19	164	**551**	3.36x
		ResNet-18	180	**424**	2.36x
	CIFAR-100	VGG-16	212	**638**	3.01x
		VGG-19	160	**554**	3.46x
		ResNet-18	181	**364**	2.01x
	Tiny ImageNet	VGG-16	213	**632**	2.96x
		VGG-19	160	**632**	3.95x
		ResNet-18	180	**330**	1.83x
Xavier NX	CIFAR-10	VGG-16	1278	**3811**	2.98x
		VGG-19	1053	**2590**	2.46x
		ResNet-18	870	**1600**	1.84x
	CIFAR-100	VGG-16	1272	**3015**	2.37x
		VGG-19	1034	**2642**	2.56x
		ResNet-18	899	**1481**	1.65x
	Tiny ImageNet	VGG-16	1270	**2997**	2.35x
		VGG-19	842	**2960**	3.51x
		ResNet-18	885	**1430**	1.61x
AGX Orin	CIFAR-10	VGG-16	3706	**10995**	2.97x
		VGG-19	3512	**8271**	2.36x
		ResNet-18	2761	**5779**	2.09x
	CIFAR-100	VGG-16	3600	**9746**	2.71x
		VGG-19	3442	**8126**	2.36x
		ResNet-18	2578	**4766**	1.85x
	Tiny ImageNet	VGG-16	3450	**9688**	2.8x
		VGG-19	3320	**9647**	2.9x
		ResNet-18	2392	**4391**	1.83x