Enhancing Split Computing and Early Exit Applications through Predefined Sparsity

Luigi Capogrosso*, Enrico Fraccaroli*†, Giulio Petrozziello‡, Francesco Setti*,
Samarjit Chakraborty†, Franco Fummi*, Marco Cristani*
*Dept. of Engineering for Innovation Medicine, University of Verona, Italy, name.surname@univr.it
‡Dept. of Computer Science, University of Verona, Italy, name.surname@studenti.univr.it
†Dept. of Computer Science, University of North Carolina at Chapel Hill, USA, samarjit@cs.unc.edu

***Abstract*—In the past decade, Deep Neural Networks (DNNs) achieved state-of-the-art performance in a broad range of problems, spanning from object classification and action recognition to smart building and healthcare. The flexibility that makes DNNs such a pervasive technology comes at a price: the computational requirements preclude their deployment on most of the resource-constrained edge devices available today to solve real-time and real-world tasks. This paper introduces a novel approach to address this challenge by combining the concept of predefined sparsity with Split Computing (SC) and Early Exit (EE). In particular, SC aims at splitting a DNN with a part of it deployed on an edge device and the rest on a remote server. Instead, EE allows the system to stop using the remote server and rely solely on the edge device's computation if the answer is already good enough. Specifically, how to apply such a predefined sparsity to a SC and EE paradigm has never been studied. This paper studies this problem and shows how predefined sparsity significantly reduces the computational, storage, and energy burdens during the training and inference phases, regardless of the hardware platform. This makes it a valuable approach for enhancing the performance of SC and EE applications. Experimental results showcase reductions exceeding $4\times$ in storage and computational complexity without compromising performance. The source code is available at https://github.com/intelligolabs/sparsity_sc_ee.**

***Index Terms*—Split Computing, Early Exit, Deep Neural Networks, Predefined Sparsity, Edge Devices.**

I. Introduction

Remarkable improvements in areas like computer vision, speech recognition, and autonomous systems are significantly supported thanks to the rise of Deep Neural Networks (DNNs) [1], [2]. At the same time, over the last decade, DNNs have grown significantly in size and complexity; some now have millions or even billions of trainable parameters [3]. However, these powerful models have a significant drawback: they demand substantial computational resources and storage, making deploying them directly on edge devices impractical.

This study was carried out within the PNRR research activities of the consortium iNEST (Interconnected North-Est Innovation Ecosystem) funded by the European Union Next-GenerationEU (Piano Nazionale di Ripresa e Resilienza (PNRR) – Missione 4 Componente 2, Investimento 1.5 – D.D. 1058 23/06/2022, ECS_00000043), and by the European Union's Horizon Europe research and innovation programme under the Marie Sklodowska-Curie grant agreement No. 101109243. This manuscript reflects only the Authors' views and opinions. Neither the European Union nor the European Commission can be considered responsible for them. This work was also partially supported by the US NSF grant 2038960.

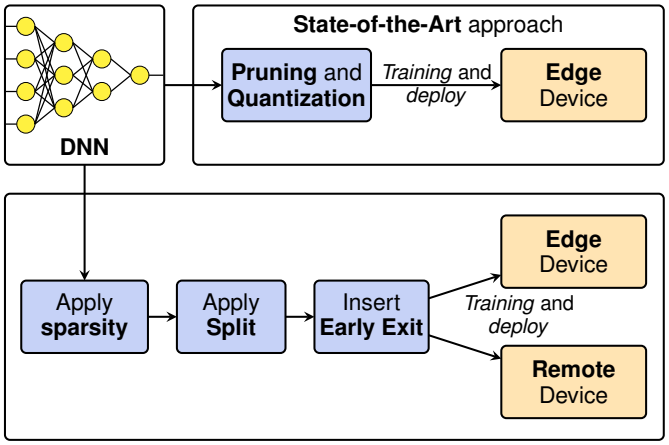


Figure 1. Difference between our approach of predefined sparsity applied to SC and EE, against the state-of-the-art pruning and quantization.

Nowadays, the most commonly used approach is to train huge DNNs in server computing settings using high-performance hardware accelerators like Graphical Processing Units (GPUs) [4], and Tensor Processing Units (TPUs) [5]. Then, once trained, the DNN model is used for inference, a less computationally expensive activity that, in the case of real-time and real-world problems, is often performed on edge devices, and more in general on embedded systems [6].

Motivations for this paper: As a result, a significant amount of research effort has been directed toward improving embedded technologies. This focus has enabled the development of real-time solutions for a wide range of real-world complex applications. In this regard, hardware-specific (e.g., edge TPUs) and Micro-Controller Unit (MCU)-based embedded systems have earned a lot of attention, primarily due to their low power requirements and high performance, and secondarily for their maintainability, adaptability, and reliability [7].

In particular, the importance of *model reduction* techniques for accelerating DNNs is widely acknowledged nowadays, aiming to optimize their performance [8]. These approaches focus on reducing the number of trainable parameters within a DNN, resulting in substantial advantages across various aspects: computational resources needed, storage requirements, and energy consumption. Since efficiency is paramount, improvements in this area are crucial if we aim to deploy DNNs

on edge devices for solving real-world applications [9].

At the same time, current state-of-the-art approaches for efficient edge computing rely on advanced Machine Learning (ML) paradigms, such as SC and EE [10], [11], [12], [13], [14]. In particular, SC, where a DNN is intelligently split with a part of it deployed on an edge device, and the rest on a remote server, and EE, where the model is built with multiple "exits" across the layers and each exit can produce the model output, represents the state-of-the-art framework for structuring distributed deep learning applications [15]. These approaches allow DNNs to be leveraged for latency-sensitive applications in scenarios where deploying the entire DNN remotely is impractical due to limited local computation bandwidth, and also locally since DNN would require higher memory requirements than those available on the edge device. They combine local and remote computing advantages, leading to lower latency and, more importantly, drastically reducing the required transmission bandwidth.

Innovations in this paper: Figure 1 exemplifies the proposal of this paper, where we explore the application of *predefined sparsity* as a model reduction method within the context of SC and EE. We aim to showcase how, in the context of SC and EE, predefined sparsity can significantly reduce computational demands, storage requirements, and energy consumption compared to state-of-the-art approaches. Furthermore, regardless of the underlying implementation platform, our approach yields advantages for both the training and inference stages.

In particular, our pipeline is exemplified by the flow presented in Figure 2. Our strategy for producing sparsity involves defining a preset set of sparse neuron connections before the training process, i.e., we eliminate a specified set of connections inside the neural network, and we keep this configuration constant throughout both the training and inference phases. Specifically, we apply this strategy within an SC and EE scenario, illustrating how predefined sparsity can enhance performance in applications operating within this context.

As a killer application where this proposal can bring real benefits, consider a common smart manufacturing setting, such as a real-time quality control system on a production line. The edge devices need to assess whether a product has a defect on time and catch all the defective ones. It can base its decision on the inference provided by the early exit strategy, which is fairly accurate but not as accurate as the inference produced by the full DNN. As such, it will move the defective product to a buffer on the side of the line while the remote device runs the remaining layers of the DNN. Once the remote device completes the inference, a worker or a conveyor belt system can reintroduce the falsely marked non-defective products inside the production line.

In summary, the main contributions of this paper are:

- A strategy for implementing predefined sparsity, where a predefined set of sparse neuron connections is established before training and remains constant throughout training and inference.
- Our approach involves the removal of specific connections within the neural network, reducing computational and storage complexity during inference and throughout the training process.
- We apply this methodology in an SC and EE scenario, demonstrating its potential to further enhance application performance within this context.

The rest of the paper is organized as follows. Section II presents background information. Our proposal's details are outlined in Section III, followed by experiments in Section IV. Finally, conclusions are drawn in Section V.

II. Related Work

This section provides an overview of *i)* DNNs sparsity methods, and *ii)* distributed deep learning applications with a specific focus on SC and EE.

A. Deep Neural Networks (DNNs) sparsity

While numerous methods concerning sparse DNNs can be found in the literature [16], [17], [18], most of these do not reduce the computation and storage complexity associated with training, but only with inference.

An example is the dropout. This technique is used to prevent overfitting. It randomly drops some neurons out (i.e., ignoring) during each training iteration. This helps the network learn more robust and generalized features, as it can't rely too much on any one neuron [19]. Due to lack of space, and since it is only a related topic, refer to [20] for more details.

There are other approaches, such as pruning and trimming methods, that process the trained DNN to generate a sparse DNN for inference. For example, in [21], the authors observe that a large fraction of the computations performed by DNNs are intrinsically ineffectual as they involve a multiplication where one of the inputs is zero. This observation motivates "Cnvlutin", a value-based approach that eliminates most of the ineffectual operations. Furthermore, in [22], the authors present an algorithm that prunes (specifies) a trained network layer-wise, removing connections at each layer by solving a convex optimization program. This program seeks a sparse set of weights at each layer, keeping the layer inputs and outputs consistent with the originally trained model.

As mentioned in Section I, other methods aim to reduce the complexity of performing inference on a trained DNN. These methods encompass techniques like quantization [16] and compression [23]. It's important to note that all these methods primarily target reducing complexity in the inference models rather than significantly simplifying the training process.

One approach that strives to minimize complexity in both training and inference involves using neural networks with structured weight matrices that are not necessarily sparse [24].

Lastly, it's worth mentioning that several authors have recently introduced the concept of predefined sparse neural networks [25]. This surge in research activity is motivated by the recognition that specialized hardware is typically required to run large and complex neural networks effectively.

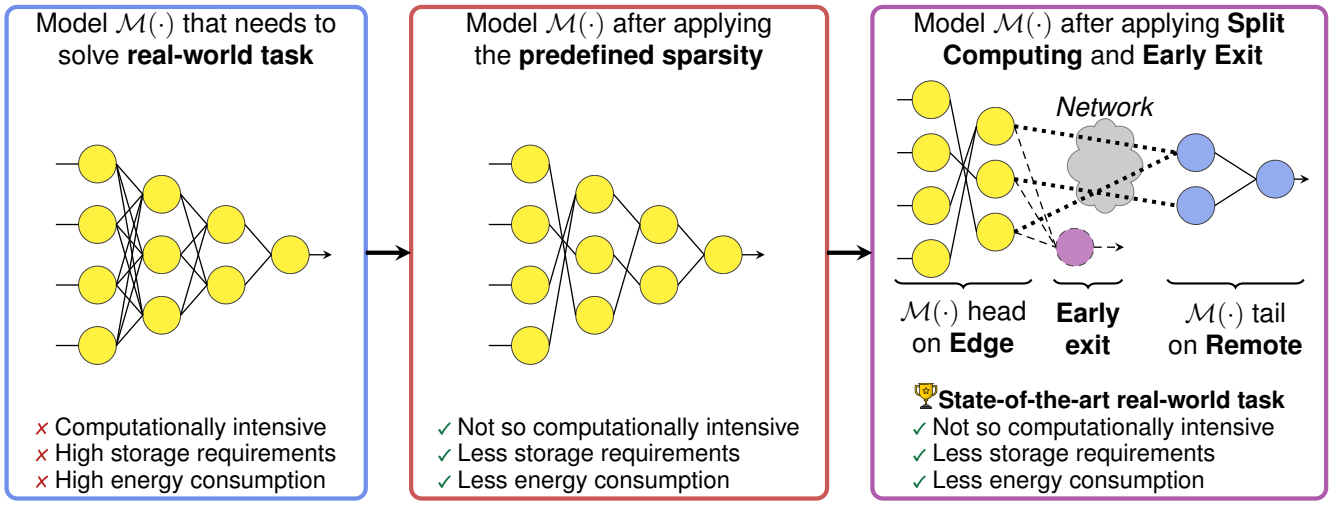


Figure 2. Starting from a DNN $\mathcal{M}(\cdot)$, we first apply the *predefined sparsity*, and then we train the network. After the training stage, we split the network following the SC and EE paradigm. As a result, the final architecture is not so computationally intensive, doesn't require huge storage spaces, and has less energy consumption, all without compromising the overall performance.

In this article, we leverage the previously mentioned *predefined sparsity*. Specifically, our approach offers two key benefits. Firstly, it is hardware agnostic. This is in contrast to quantization, which relies on specific hardware capabilities. Secondly, it allows us to streamline our model before the training begins. This differs from pruning, which tackles complexity reduction after training. As a result, our model has a smaller footprint on the GPU right from the start of training.

Our work goes a step further. While default sparsity has been explored previously, we are the first to investigate its advantages within the context of SC and EE. This exploration aims to unlock even greater efficiency for applications that utilize this technique.

B. Distributed deep learning

We focus on architectures operating through a DNN model $\mathcal{M}(\cdot)$, whose task is to produce the inference output y from an input x. We can identify three major types of architectures used for distributed deep learning applications in the literature: Local-only Computing (LoC), Remote-only Computing (RoC), and SC.

Local-only Computing (LoC): Under this policy, the entire computation is performed on the sensing devices. Therefore, the edge device entirely executes the function $\mathcal{M}(x)$. Its advantage lies in offering low latency due to the proximity of the computing element to the sensor [7]. However, it may not be compatible with DNN-based architectures that demand robust hardware capabilities. Usually, simpler DNN models $\bar{\mathcal{M}}(x)$ that use specific architectures (e.g., depth-wise separable convolutions) are used to build lightweight networks, such as MobileNetV3 [26].

Besides designing lightweight neural models, in the last few years, great progress has been made in the area of DNN compression. Compression techniques, such as network pruning and quantization [27], or knowledge distillation [28] achieve a more efficient representation of one or more layers of the neural network, but with a possible quality degradation.

Remote-only Computing (RoC): The input x is transferred through the communication network and then is processed at the remote system through the function $\mathcal{M}(x)$. This architecture preserves full accuracy considering the higher power budget of the remote system, but it leads to high latency and bandwidth consumption due to the input transfer.

Split Computing (SC): A typical SC scenario is discussed in [10], where the authors show that neither LoC nor RoC approaches are optimal, and a split configuration is an ideal solution. The SC paradigm divides the DNN model into a head, executed by the local sensing device, and a tail, executed by the remote system. It combines the advantages of both LoC and RoC thanks to the lower latency and, more importantly, drastically reduces the required transmission bandwidth by compressing the input to be sent x through the use of an autoencoder [29]. We define the encoder and decoder models as $z_l = \mathcal{F}(x)$ and $\bar{x} = \mathcal{G}(z_l)$, which are executed at the edge, and remotely, respectively. The distance $d(x, \bar{x})$ defines the performance of the encoding-decoding process.

One of the earliest works on SC is the study by Kang *et al.* [30], in which the authors show that the initial layers of a DNN are the most suitable candidates for partitioning, as they optimize both latency and energy consumption. Additionally, latency reduction is usually achieved through quantization, as explored in [31], and the utilization of lossy compression techniques prior to data transmission, as investigated in [32]. In addition to lossy compression techniques, in [33], the authors also explore lossless techniques to encode intermediate results without modifying the ML model. Instead, the concept of employing autoencoders to compress the data further to be transferred is discussed in various studies, such as [34].

The prevalent methods for identifying potential splitting points have evolved from architecture-based techniques to more refined neuron-based methods. Within the domain of

architecture-based approaches, in [12], the authors state that candidate split locations are where the size of the DNN layers decreases: the rationale is that compressing information by autoencoders, where compression would still occur due to the shrinking of the architecture, certainly seems reasonable. In [13] and [14], the authors show that not only the architecture of the layers but also the saliency of individual layers is a crucial factor when deciding where to split. A neuron's saliency is determined by its gradient in relation to the accurate decision. Consequently, optimal splitting points should be strategically positioned following layers housing a concentration of impactful neurons to preserve the information flowing until then.

At the same time, current state-of-the-art approaches in different ML applications rely on advanced learning procedures, such as the *Multi-Task Learning (MTL)* [35]. In particular, MTL is a paradigm in which multiple related tasks are jointly learned to improve the generalizability of a model by using shared knowledge across different aspects of the input. As a result, in [36], the authors propose, for the first time ever, how to partition multi-tasking DNN to be deployed within a SC framework. With this design, the authors can handle multiple tasks concurrently instead of the current focus on Single-Task Learning (STL) in SC, and through MTL, they increase task performance, overcoming the challenge of preserving only the performance of the main task.

***Early Exit (EE)**:* This scenario adds an early exiting branch to a standard SC architecture. Formally, we can define $B_i, i = 1 \dots N$ (with $N = L$, and L is the number of layers of the DNN) as the branch model that takes as input z_l and produces an estimate of the desired output y. In practice, the EE architecture is a modification of an existing neural network, adding one or more classification branches where, before the computation of all network's layers, the confidence of the intermediate result is checked to be enough to be considered the final result [37].

EE architecture can be exploited in a distributed deep learning application where the intermediate result can be directly transmitted, as in local computing, or refined at the remote side, as in SC. In this scenario, the level of transmission traffic depends on the input, thus varying stochastically. Therefore, the interdependencies between computation and communication cannot be analytically modeled, and real experiments are needed to validate a given implementation.

In this paper, we're not looking to develop a new method for finding the best layers in a DNN to use SC or the most efficient way to put EE into action. We aim to explore and show how the predefined sparsity can improve these existing frameworks. Thus, instead of comparing our work to SC and EE methods, we see it as a way to make them even better, investigating how sparsity can add benefits on top of what SC and EE already offer.

III. Method

To understand the core concepts of our research, let us delve into the mathematical background. These concepts were initially presented in [25].

Let us take a $(L+1)$-layer MultiLayer Perceptron (MLP), described collectively by the following *neuronal configuration*:

$$N_{net} = (N_0, \dots, N_{i-1}, N_i, \dots, N_L) \, ,$$

where N_i represents the number of nodes in the i-th layer. We use the convention that layer i is to the right of layer $i-1$. Given L *junctions* between layers, with junction i connecting the N_{i-1} nodes of its left layer $i-1$ with the N_i nodes of its right layer i.

We can define *predefined sparsity* as simply not having all $N_{i-1} \cdot N_i$ edges (or weights) present in junction i. Furthermore, we can define *structured predefined sparsity* so that for a given junction i, each node in its left layer has fixed out-degree, i.e., $d_i^{out} \in \mathbb{N}$ connections to its right layer, and each node in its right layer has fixed in-degree, i.e., $d_i^{in} \in \mathbb{N}$ connections from its left layer.

In particular, a MLP have $d_i^{out} = N_i$ and $d_i^{in} = N_{i-1}$ with $N_{i-1} \cdot N_i$ edges present in the i^{th} junction. While a sparse MLP has at least one junction with less than this number of edges. The number of edges in junction i is given by the formula:

$$|W_i| = N_{i-1} \cdot d_i^{out} = N_i \cdot d_i^{in} \, . \tag{1}$$

The density of junction i is measured relative to MLP, and is denoted by the function:

$$\rho_i = \frac{|W_i|}{N_{i-1} \cdot N_i} \, . \tag{2}$$

In our structured predefined sparse network, the density of the i-th junction ρ_i cannot be arbitrary. By replacing Equation (1) in Equation (2), we can define:

$$\rho_i = \frac{d_i^{in}}{N_i} = \frac{N_i}{d_i^{out}} \, , \tag{3}$$

where d_i^{out} and d_i^{in} are natural numbers. The number of possible ρ_i values is the same as the number of (d_i^{out}, d_i^{in}) values satisfying the structured predefined sparsity constraints:

$$d_i^{out} = \frac{N_i \cdot d_i^{in}}{N_{i-1}}, \qquad d_i^{in} \leq N_{i-1} \, . \tag{4}$$

Now, the smallest value of d_i^{in} which satisfies the assignment to d_i^{out} in Equation (4), and $d_i^{out} \in \mathbb{N}$, is the following:

$$\frac{N_{i-1}}{\gcd(N_{i-1}, N_i)} \, ,$$

and other values are integer multiples. Since d_i^{in} is upper bounded by N_{i-1}, the total number of possible (d_i^{out}, d_i^{in}) is $\gcd(N_{i-1}, N_i)$. We can now define the set of possible densities ρ_i, as follows:

$$\left\{ \rho_i = \frac{k}{\gcd(N_{i-1}, N_i)}, \quad \rho_i \in (0, 1], \quad k \in \mathbb{N} \right\} . \tag{5}$$

Specifying N_{net} and the *out-degree configuration* $d_{net}^{out} = (d_1^{out}, \dots, d_L^{out})$ determines the density of each junction and the overall density, defined as:

$$\rho_{net} = \frac{\sum_{i=1}^{L} |W_i|}{\sum_{i=1}^{L} N_{i-1} \cdot N_i} \, . \tag{6}$$

It's worth noting that, for an MLP using structured predefined sparsity, only the weights corresponding to connected edges are stored in memory and used in the computation. Specifically, the parameters are updated based on the gradient of a loss function with respect to the parameters. Let's denote the parameters of the network as θ and the loss function as $\mathcal{L}(\theta)$. The gradient of the loss function for the parameters is denoted as $\nabla_\theta \mathcal{L}$. Then, the parameters are updated using a gradient descent step:

$$\theta_{\text{new}} = \theta_{\text{old}} - \eta \cdot \nabla_\theta \mathcal{L} \,, \tag{7}$$

where η is the learning rate, controlling the step size in the direction opposite to the gradient.

IV. Experiments

This section describes the experimental trials performed to validate our proposal, along with their implementation details and results.

Models details: We use three types of MLPs in our experiments: shallow, deep, and sparse. We can characterize them as follows:

- **Shallow**: This is the simplest type. It contains only one hidden layer besides the input and output layers and represents the base case. The hidden layer neurons receive information from the input layer and process it before transmitting it to the final output layer;
- **Deep**: Each neuron within a hidden layer is fully connected to all neurons in the subsequent layer;
- **Sparse**: The *predefined sparsity* pattern is applied. This sparsity pattern essentially removes certain connections between neurons in adjacent layers. Specifically, the sparse MLP aims to balance the simplicity of the shallow MLP and the learning capacity of the deep MLP.

Each network configuration is defined by using two lists. The first list identifies the number of neurons in the hidden layers:

$$H = [H_0, \ldots, H_n] \,,$$

where n is the number of hidden layers, and intuitively, H_0 and H_n represent the number of neurons in the input and output layers, respectively. The second list identifies the out-degree of each neuron for each layer:

$$G = [G_0, \ldots, G_{L-1}] \,.$$

Specifically, G_0 is the out-degree of each input layer node, and the output layer G_L is not reported because it is zero.

Then, the shallow, deep, and sparse MLP has been split. Our research primarily focuses on using existing sparsity patterns in split computing applications for these networks. Inspired by the approach presented in [30], we opt to split the network at the midpoint, ensuring uniformity in our approach. While more recent studies, such as [12], [13], have proposed advanced techniques for selecting optimal splitting points, exploring these methods will be part of future investigations.

Finally, the inserted EEs are composed of linear layers followed by Rectified Linear Unit (ReLU) activation functions, where the dimension of the last layer matches the desired output size of the DNN.

Even with the rise of specialized architectures like Convolutional Neural Networks (CNNs), MLPs remains a valuable tool for researchers. Their fundamental structure allows for in-depth exploration of core deep learning concepts without the added complexity of specialized architectures. This focus on MLPs aligns with the experimental nature of this research, where we aim to isolate the effects of the proposed technique and gain a deeper understanding of its fundamental workings. Furthermore, recent advancements in research in MLP have demonstrated their continued effectiveness in various real-world applications, highlighting their ongoing relevance in the field [38], [39].

Datasets: In this research, we focus on the image classification task. We utilize the MNIST dataset [40], a well-known collection of handwritten digits. This contains 60,000 training and 10,000 testing images for the multi-class image classification task. The images were centered in a 28×28 image by computing the center of mass of the pixels and translating the image to position this point at the center of the 28×28 field.

MNIST has to be considered as a placeholder for bigger datasets (e.g., ImageNet [41]); nonetheless, the focus here is to show the potentialities of the predefined sparsity applied in an SC and EE and not beating the state-of-the-art in a specific computer vision challenge.

While this article focuses on a computer vision task, the concepts explored here remain valid also for broader applications. For example, they can be effectively used for other tasks, such as time series forecasting.

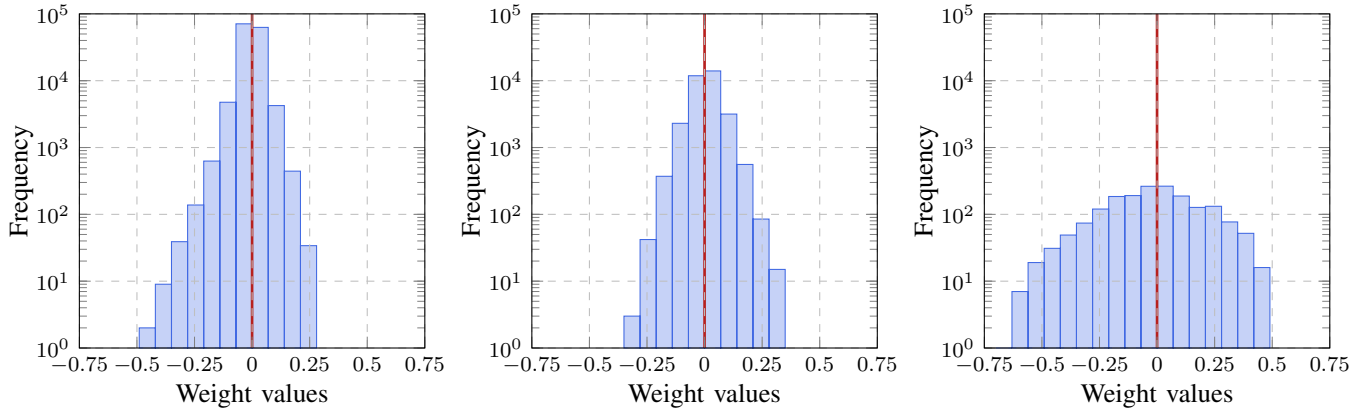
Training details: All the source code is implemented in TensorFlow [42]. We train our models for 50 epochs, with a learning rate of 1×10^{-5}, using Adam [43] as an optimizer, on an NVIDIA RTX 3060 Ti.

A. Why predefined sparsity in SC and EE?

The motivation behind predefined sparsity can be exemplified by examining the weights histograms of a trained MLP shown in Figure 3. Specifically, the three histograms show the weight distributions for each layer in a 3-layer MLP model trained on the MNIST dataset with hidden layers having the following number of neurons $H = [800, 180, 180, 10]$.

As we can see from Figures 3(a) and 3(b), the first layers of the network have a significant concentration of weights around zero, suggesting that these weights might not be crucial for the network's performance. While Figure 3(c) highlights how the weights in the last layer assume a broader spectrum of values.

This finding suggests that the benefits of *predefined sparsity* can be especially pronounced in the earlier layers of the network. As a result, in resource-constrained environments, like those encountered in SC and EE applications, predefined sparsity offers significant advantages because by reducing the number of connections, we can decrease the size and complexity of the network portion deployed on the edge



(a) Weights distribution in layer 1 [800, 180]. (b) Weights distribution in layer 2 [180, 180]. (c) Weights distribution in layer 3 [800, 10].

Figure 3. Histograms of weights in each junction resulting from training a deep MLP on the MNIST dataset. The network configuration $H = [H_0, \ldots, H_n]$ used is [800, 180, 180, 10].

Table I
RESULTS REGARDING THE ACCURACY AND THE NUMBER OF PARAMETERS WITH DIFFERENT CONFIGURATIONS OF HEAD MLPS.

Type	Neurons per Layer	Out-degree per Node	Accuracy (%)	Number of Parameters
Deep	[800 **3 3** 10]	[3 3 10]	76.58	2443
Shallow	[800 **3** 10]	[3 10]	80.89	2455
Sparse	[800 **40 40** 10]	[2 9 10]	93.93	2450
Deep	[800 **5 5** 10]	[5 5 10]	90.69	4095
Shallow	[800 **5** 10]	[5 10]	89.84	4065
Sparse	[800 **40 40** 10]	[4 10 10]	95.30	4090
Deep	[800 **7 7** 10]	[7 7 10]	92.85	5743
Shallow	[800 **7** 10]	[7 10]	92.22	5687
Sparse	[800 **40 40** 10]	[6 11 10]	95.89	5730
Deep	[800 **14 14** 10]	[14 14 10]	95.35	11574
Shallow	[800 **14** 10]	[14 10]	95.44	11364
Sparse	[800 **40 40** 10]	[13 17 10]	96.54	11570
Deep	[800 **20 20** 10]	[20 20 10]	96.03	16650
Shallow	[800 **21** 10]	[21 10]	96.03	17041
Sparse	[800 **40 40** 10]	[19 24 10]	96.98	16650

Table II
RESULTS REGARDING THE ACCURACY AND THE NUMBER OF PARAMETERS WITH DIFFERENT CONFIGURATIONS OF TAIL MLPS.

Type	Neurons per Layer	Out-degree per Node	Accuracy (%)	Number of Parameters
Deep	[800 **6 6** 10]	[6 6 10]	90.68	4918
Shallow	[800 **6** 10]	[6 10]	91.03	4876
Sparse	[800 **40 40** 10]	[5 11 10]	95.65	4930
Deep	[800 **10 10** 10]	[10 10 10]	94.59	8230
Shallow	[800 **10** 10]	[10 10]	94.00	8120
Sparse	[800 **40 40** 10]	[9 14 10]	96.24	8250
Deep	[800 **14 14** 10]	[14 14 10]	95.43	11574
Shallow	[800 **14** 10]	[14 10]	94.88	11364
Sparse	[800 **40 40** 10]	[13 17 10]	96.72	11570
Deep	[800 **28 28** 10]	[28 28 10]	96.79	23530
Shallow	[800 **29** 10]	[29 10]	96.79	23529
Sparse	[800 **80 80** 10]	[26 22 10]	97.33	23530
Deep	[800 **39 39** 10]	[39 39 10]	97.35	33199
Shallow	[800 **41** 10]	[41 10]	97.34	33261
Sparse	[800 **80 80** 10]	[37 33 10]	97.75	33210

device. This translates to lower memory requirements and faster processing times during training and inference.

Furthermore, the storage footprint is directly proportional to the number of edges. Operating at a sparsity level of, for example, 50% results in a two-fold reduction in complexity. This translates to significant efficiency gains, making deploying more complex models on devices with limited resources possible. These findings led us to want to study them in the context of SC and EE.

B. Results

Tables I and II provide a comparative overview of the three neural network configurations, detailing their architectural details and performances. Specifically, each table reports the number of neurons per layer, focusing on the number of neurons in hidden layers (in bold), the out-degree for each node, their accuracy, and the number of parameters.

Figures 4 and 5 show the accuracy plots against the number of parameters for the three neural network configurations. These two plots reveal the advantage of sparse split models: they present remarkable stability in accuracy even with significant reductions in trainable parameters. This characteristic results in two key advantages, particularly desired and pursued in resource-constrained settings like SC and EE.

Unlike traditional deep and shallow DNN, where accuracy improvements often depend on a significant increase in the number of parameters, sparse split models achieve optimal inference performance without requiring massive parameter

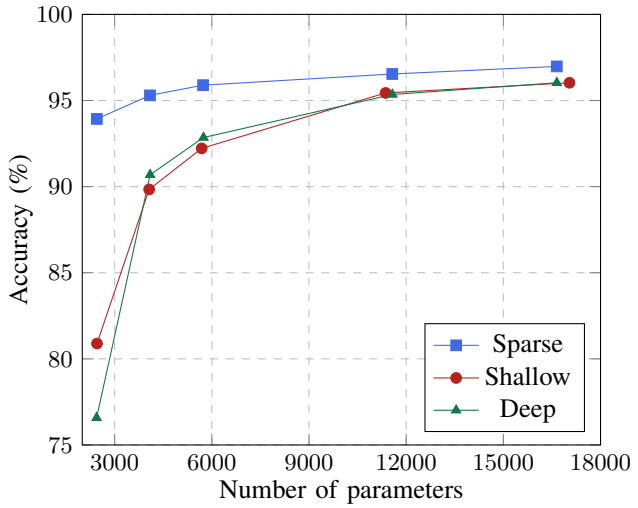


Figure 4. Accuracy tests by the number of parameters of deep, shallow, and sparse head MLPs.

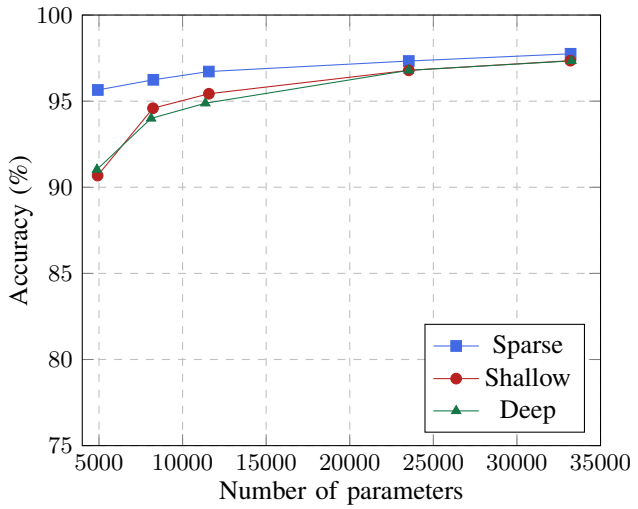


Figure 5. Accuracy tests by number of parameters of deep, shallow, and sparse tail MLPs.

expansions. As a result, they are much more efficient when using the limited processing power and storage space available on edge devices, and so very suitable for distributed deep leasing scenarios through SC and EE.

Furthermore, the predefined sparsity of these networks leads to decreased memory usage, both during training and inference. Regarding the training, this is an advantage because researchers with limited resources can also engage in cutting-edge research in deep learning without the need for expensive, high-end computational resources. Instead, the reduced complexity translates into faster processing times, enabling real-time operation on edge devices with limited processing power.

C. Discussion

Early Exiting and reduced communication: In SC and EE applications, where data security and bandwidth limitations are paramount, sparse split models represent an ideal solution.

The early exiting strategy applied to the head provides an acceptable level of accuracy with a smaller subset of parameters, as highlighted by the achieved accuracy shown in Figure 4. This allows the network to terminate computations early, significantly reducing the amount of data transmitted between the edge device and the server. This not only minimizes the risk of data breaches but also conserves precious network bandwidth, enabling efficient communication even in low-connectivity environments.

Balancing training time and efficiency: While the benefits of sparse split models are evident, it's important to acknowledge the trade-off with training time. When we introduce predefined sparsity into a MLPs, we aim to train the model to learn a function using a significantly reduced number of connections compared to a non-sparse network. This approach creates a more lightweight structure for the model but also leads to a more challenging optimization problem. Consequently, the gradient descent algorithm may require more iterations to converge due to the increased complexity introduced by the sparsity constraints. In particular, sparse split models in our experiments exhibited a $2\times$ slowdown in training time compared to their dense counterparts.

However, this drawback needs to be considered in the context of the specific application and its resource constraints. In many cases, the significant gains in memory usage, communication efficiency, and inference speed during deployment far outweigh the potential increase in training time.

Potential further exploration: While this work demonstrates the effectiveness of predefined sparsity with SC and EE, further exploration is possible. First, future research could look at applying this technique to different DNN architectures on top of existing SC and EE methods.

It would also be interesting to see how different hardware edge devices can benefit from this. For example, we know that devices like GPUs and Field Programmable Gate Arrays (FPGAs) handle sparse data well. So, using this technique with these devices on the edge (like GPU-based or FPGA-based hardware) could be a promising direction.

Finally, further exploration could involve applying these techniques to real-world applications. This would allow us to assess the effectiveness of the approach in practical scenarios and identify any challenges that may arise.

V. Conclusion

In this paper, we presented the effect of predefined sparsity within the SC and EE paradigm. This approach, demonstrably effective for the first time in an SC and EE scenario, significantly reduces the computational, storage, and energy demands during training and inference, regardless of the hardware platform. The experimental results showcase impressive reductions exceeding $4\times$ in both storage and computational complexity while maintaining comparable accuracy. This paves the way for deploying complex DNN models on edge devices for real-world SC and EE applications.

REFERENCES

[1] S. Pouyanfar, S. Sadiq, Y. Yan *et al.*, "A survey on deep learning: Algorithms, techniques, and applications," *ACM Computing Surveys (CSUR)*, 2018.

[2] S. Dong, P. Wang, and K. Abbas, "A survey on deep learning and its applications," *Computer Science Review*, 2021.

[3] S. Smith, M. Patwary, B. Norick *et al.*, "Using deepspeed and megatron to train megatron-turing nlg 530b, a large-scale generative language model," *arXiv preprint arXiv:2201.11990*, 2022.

[4] K. Y. Chan, B. Abu-Salih, R. Qaddoura *et al.*, "Deep neural networks in the cloud: Review, applications, challenges and research directions," *Neurocomputing*, 2023.

[5] N. P. Jouppi, C. Young, N. Patil *et al.*, "In-datacenter performance analysis of a tensor processing unit," in *Proceedings of the 44th annual international symposium on computer architecture*, 2017.

[6] R. Singh and S. S. Gill, "Edge ai: a survey," *Internet of Things and Cyber-Physical Systems*, 2023.

[7] L. Capogrosso, F. Cunico, D. S. Cheng *et al.*, "A machine learning-oriented survey on tiny machine learning," *IEEE Access*, 2024.

[8] G. Menghani, "Efficient deep learning: A survey on making deep learning models smaller, faster, and better," *ACM Computing Surveys*, 2023.

[9] E. Fragkou, M. Koultouki, and D. Katsaros, "Model reduction of feed forward neural networks for resource-constrained devices," *Applied Intelligence*, 2023.

[10] A. E. Eshratifar, M. S. Abrishami, and M. Pedram, "Jointdnn: An efficient training and inference engine for intelligent mobile cloud computing services," *IEEE Transactions on Mobile Computing*, 2019.

[11] M. Jankowski, D. Gündüz, and K. Mikolajczyk, "Joint device-edge inference over wireless links with pruning," in *2020 IEEE 21st International Workshop on Signal Processing Advances in Wireless Communications (SPAWC)*. IEEE, 2020.

[12] M. Sbai, M. R. U. Saputra, N. Trigoni *et al.*, "Cut, distil and encode (cde): Split cloud-edge deep inference," in *2021 18th Annual IEEE International Conference on Sensing, Communication, and Networking (SECON)*. IEEE, 2021.

[13] F. Cunico, L. Capogrosso, F. Setti *et al.*, "I-split: Deep network interpretability for split computing," in *2022 26th International Conference on Pattern Recognition (ICPR)*. IEEE, 2022.

[14] L. Capogrosso, F. Cunico, M. Lora *et al.*, "Split-et-impera: A framework for the design of distributed deep learning applications," in *2023 26th International Symposium on Design and Diagnostics of Electronic Circuits and Systems (DDECS)*. IEEE, 2023.

[15] Y. Matsubara, M. Levorato, and F. Restuccia, "Split computing and early exiting for deep learning applications: Survey and research challenges," *ACM Computing Surveys*, 2022.

[16] Y. Gong, L. Liu, M. Yang *et al.*, "Compressing deep convolutional networks using vector quantization," *arXiv preprint arXiv:1412.6115*, 2014.

[17] W. Chen, J. Wilson, S. Tyree *et al.*, "Compressing neural networks with the hashing trick," in *International conference on machine learning*. PMLR, 2015.

[18] W. Wen, C. Wu, Y. Wang *et al.*, "Learning structured sparsity in deep neural networks," *Advances in neural information processing systems*, 2016.

[19] N. Srivastava, G. Hinton, A. Krizhevsky *et al.*, "Dropout: a simple way to prevent neural networks from overfitting," *The journal of machine learning research*, 2014.

[20] A. Labach, H. Salehinejad, and S. Valaee, "Survey of dropout methods for deep neural networks," *arXiv preprint arXiv:1904.13310*, 2019.

[21] J. Albericio, P. Judd, T. Hetherington *et al.*, "Cnvlutin: Ineffectual-neuron-free deep neural network computing," *ACM SIGARCH Computer Architecture News*, 2016.

[22] A. Aghasi, A. Abdi, N. Nguyen *et al.*, "Net-trim: Convex pruning of deep neural networks with performance guarantee," *Advances in neural information processing systems*, 2017.

[23] S. Han, H. Mao, and W. J. Dally, "Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding," *arXiv preprint arXiv:1510.00149*, 2015.

[24] V. Sindhwani, T. Sainath, and S. Kumar, "Structured transforms for small-footprint deep learning," *Advances in Neural Information Processing Systems*, 2015.

[25] S. Dey, K.-W. Huang, P. A. Beerel *et al.*, "Pre-defined sparse neural networks with hardware acceleration," *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, 2019.

[26] A. Howard, M. Sandler, G. Chu *et al.*, "Searching for mobilenetv3," in *Proceedings of the IEEE/CVF international conference on computer vision*, 2019.

[27] T. Liang, J. Glossner, L. Wang *et al.*, "Pruning and quantization for deep neural network acceleration: A survey," *Neurocomputing*, 2021.

[28] J. Gou, B. Yu, S. J. Maybank *et al.*, "Knowledge distillation: A survey," *International Journal of Computer Vision*, 2021.

[29] Y. Matsubara, S. Baidya, D. Callegaro *et al.*, "Distilled split deep neural networks for edge-assisted real-time systems," in *Proceedings of the 2019 Workshop on Hot Topics in Video Analytics and Intelligent Edges*, 2019.

[30] Y. Kang, J. Hauswald, C. Gao *et al.*, "Neurosurgeon: Collaborative intelligence between the cloud and mobile edge," *ACM SIGARCH Computer Architecture News*, 2017.

[31] G. Li, L. Liu, X. Wang *et al.*, "Auto-tuning neural network quantization framework for collaborative inference between the cloud and edge," in *Artificial Neural Networks and Machine Learning–ICANN 2018: 27th International Conference on Artificial Neural Networks, Rhodes, Greece, October 4-7, 2018, Proceedings, Part I 27*. Springer, 2018.

[32] H. Choi and I. V. Bajić, "Deep feature compression for collaborative object detection," in *2018 25th IEEE International Conference on Image Processing (ICIP)*. IEEE, 2018.

[33] D. Carra and G. Neglia, "Dnn split computing: Quantization and run-length coding are enough," in *GLOBECOM 2023-2023 IEEE Global Communications Conference*. IEEE, 2023.

[34] A. E. Eshratifar, A. Esmaili, and M. Pedram, "Bottlenet: A deep learning architecture for intelligent mobile cloud computing services," in *2019 IEEE/ACM International Symposium on Low Power Electronics and Design (ISLPED)*. IEEE, 2019.

[35] R. Caruana, "Multitask learning," *Machine learning*, 1997.

[36] L. Capogrosso, E. Fraccaroli, S. Chakraborty *et al.*, "Mtl-split: Multi-task learning for edge devices using split computing," *arXiv preprint arXiv:2407.05982*, 2024.

[37] C. Lo, Y.-Y. Su, C.-Y. Lee *et al.*, "A dynamic deep neural network design for efficient workload allocation in edge computing," in *2017 IEEE International Conference on Computer Design (ICCD)*. IEEE, 2017.

[38] C. Turetta, G. Skenderi, L. Capogrosso *et al.*, "Towards deep learning-based occupancy detection via wifi sensing in unconstrained environments," in *2023 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2023.

[39] S.-A. Chen, C.-L. Li, N. Yoder *et al.*, "Tsmixer: An all-mlp architecture for time series forecasting," *arXiv preprint arXiv:2303.06053*, 2023.

[40] L. Deng, "The mnist database of handwritten digit images for machine learning research," *IEEE Signal Processing Magazine*, 2012.

[41] J. Deng, W. Dong, R. Socher *et al.*, "Imagenet: A large-scale hierarchical image database," in *2009 IEEE conference on computer vision and pattern recognition*. IEEE, 2009.

[42] M. Abadi, A. Agarwal, P. Barham *et al.*, "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015, software available from tensorflow.org. [Online]. Available: https://www.tensorflow.org/

[43] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.