Truly Sparse Neural Networks at Scale

Selima Curci[1], Decebal Constantin Mocanu[2,1], and Mykola Pechenizkiyi[1]

[1] *Department of Mathematics and Computer Science, Eindhoven University of Technology*
[2] *Faculty of Electrical Engineering, Mathematics and Computer Science, University of Twente*

Abstract

Recently, sparse training methods have started to be established as a de facto approach for training and inference efficiency in artificial neural networks. Yet, this training efficiency is just in theory. In practice, everyone uses a binary mask to simulate sparsity since the typical deep learning software and hardware are optimized for dense matrix operations. In this paper, we take an orthogonal approach, and we show that we can train truly sparse neural networks to harvest their full potential. To achieve this goal, we introduce three novel contributions [1], specially designed for sparse neural networks: (1) a parallel training algorithm and its corresponding sparse implementation from scratch, (2) an activation function with non-trainable parameters to favour the gradient flow, and (3) a hidden neurons importance metric to eliminate redundancies. All in one, we are able to break the record and to train the largest neural network ever trained in terms of representational power – reaching the bat brain size. The results show that our approach has state-of-the-art performance while opening the path for an environmentally friendly artificial intelligence era.

Keywords: Deep Learning, Sparse Neural Networks, Parallel Algorithms, Bio-inspired optimization, Activation Functions, Connection Importance, Breaking Symmetry

arXiv:2102.01732v2 [cs.LG] 12 Jul 2022

1 Introduction

Artificial Neural Networks (ANNs) succeeded in a broad range of application domains (LeCun et al. (2015)) due to their ability to learn complex transformations from data while achieving superior generalisation performance. However, current state-of-the-art networks are typically highly overparameterised (e.g. Zhang et al. (2017)) and demand extensive computational resources to be trained, which become a bottleneck where such resources are limited (Kepner et al. (2018)). Reducing the memory footprint and training time of ANNs are active areas of research, crucial to handle the rapid expansion of machine learning which has resulted in enormous datasets, with millions to billions of examples and features, but also to decrease the high environmental impact of the energy-hungry deep learning algorithms. Taking inspiration from nature, a solution to improve neural network scaling is to use sparse connectivity. The traditional *dense-to-sparse* training paradigm (known mainly as network pruning) (Mozer & Smolensky, 1989; LeCun et al., 1989; Han et al., 2015; Frankle & Carbin, 2019) offer computational benefits just in the inference phase as first, it trains a dense network in order to prune unimportant connections and to obtain a sparsely connected neural network.

Therefore, to obtain scalable and efficient ANNs, contrary to general practice, artificial neural networks, like biological neural networks, should not have fully connected layers also in the training phase. Recently, a new *sparse-to-sparse* training paradigm (or simply, sparse training) began to establish inside the research community, with several studies focus on developing memory and computational efficiency from the start by training directly sparse neural networks from scratch. The first attempt (Mocanu et al., 2016) has used just static sparsity limiting the capacity of the model sparse connectivity graph to fit the data distribution. This concept has been revised and drastically improved by introducing the Sparse Evolutionary Training (SET) algorithm with dynamic (or adaptive) sparsity in (Mocanu et al. (2018)). Currently, the sparse training concept has started to be a *de facto* approach for efficient training of ANNs, as demonstrated in (Bellec et al., 2018; Dettmers & Zettlemoyer, 2019;

1 The code is available at https://github.com/SelimaC/large-scale-sparse-neural-networks

Mostafa & Wang, 2019b; Evci et al., 2019, 2020; Jayakumar et al., 2020; Raihan & Aamodt, 2020). These algorithms search for an optimal *sparse topology* according to some salience criteria, while simultaneously optimising the model weights. Here, the *topology* of a neural network refers to the way the neurons are connected, and it is a crucial factor in network functioning, and learning (Miikkulainen (2010)). The resulting networks have a significantly lower number of parameters by design, and they have empirically shown to outstretch higher generalisation power than their dense counterparts in a number of cases, especially in the case of multilayer perceptrons and recurrent neural networks (Evci et al. (2020); Bourgin et al. (2019); Liu et al. (2020c, 2021)). Besides this, intrinsically sparse models allow, in theory, real scalable deep learning solutions in low-resource devices, standard computers, and in the cloud.

The main limitation to achieve this theoretical scalability level is given by the fact that all state-of-the-art deep learning frameworks are based on very well-optimised dense matrix multiplications on Graphics Processing Units (GPUs), while sparse matrix operations are practically ignored. The only notable exception is given by the NVIDIA A100 GPU which was released in 2020 (Jeff Pool (2020)) and supports a hardware fixed 2:4 sparsity level (i.e. 50% sparsity level). Within these frameworks, one can only simulate the sparsity by using a binary mask over the connections; therefore, the model will carry on training dense matrices. As follows, until optimised hardware for sparse operations appears, one would have to focus on optimising the algorithms.

In (Liu et al. (2020b)), the authors developed an efficient implementation of sparse multilayer perceptrons (MLPs) trained with SET. For the first time, they built sparse MLP models with over one million artificial neurons on commodity hardware, only utilising one CPU core. Still their sparse framework is completely sequential and cannot yet compete against advanced professional frameworks designed to accelerate the learning of dense neural networks.

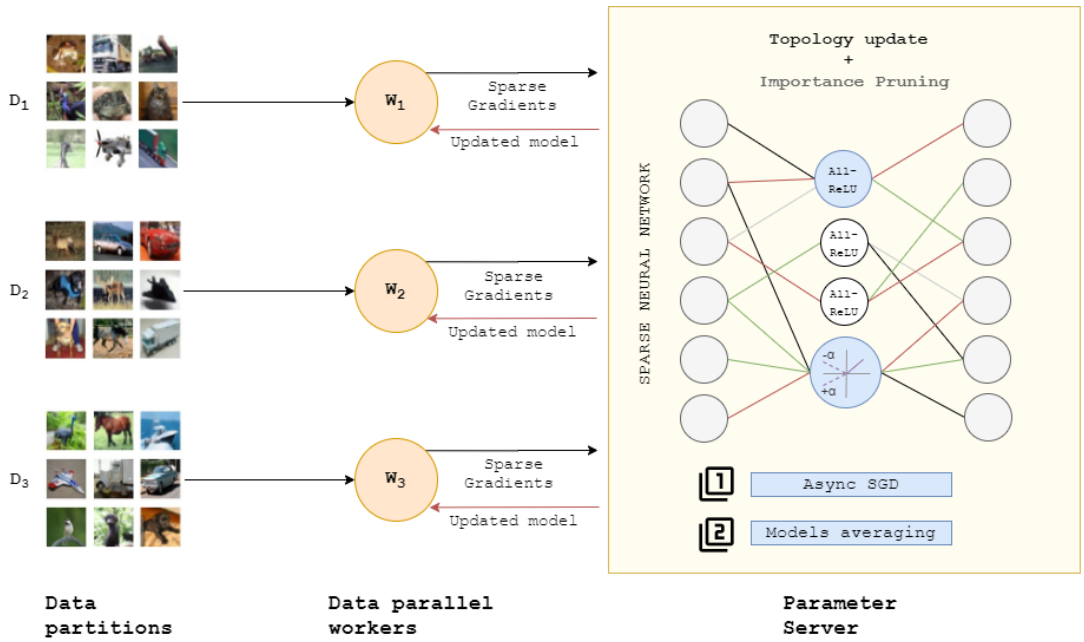


Figure 1. A graphical high-level overview of the proposed methods to efficiently train truly sparse neural networks.

Additionally, there is the need for revisiting various aspects (e.g. optimizers and activation functions) of sparse neural networks training since most of the literature has mainly focused on dense models. The choice of the activation function for deep neural networks has a critical impact on the performance of the training procedure. An inappropriate selection can lead to the loss of information of the input during forward propagation and the exponential vanishing/exploding of gradients during backpropagation (Hayou et al. (2019)). It is crucial to question weather the activation functions currently used for densely connected networks still behave reliably

in the sparse context. SReLU is a relatively little-known activation function suggested in (Jin et al. (2016)) and it has proven to outperform ReLU (Agarap (2018)) when training sparse networks over different datasets (Mocanu et al. (2018); Dubowski (2020)) as it improves the networks gradient flow (ab Tessera et al. (2021)). However, this activation function requires to learn four additional parameters per neuron, which becomes a non-negligible number and introduces a serious computational overhead if we want to train models with millions or billions of hidden units.

To alleviate the aforementioned limitations, this paper proposes four new contributions which advance the scalability of neural networks by exploiting sparsity:

- We introduce *Weight Averaging Sparse Asynchronous Parallel SGD* (*WASAP-SGD*), a parallel algorithm to train truly sparse neural networks and expand their feasible size on commodity hardware, without any GPU support.
- We propose a variant of ReLU, called *ALternated Left ReLU* (*All-ReLU*), to achieve performance comparable to SReLU for SET without the additional overhead for training its associated parameters.
- We introduce the concept of neuron importance and a method (*Importance Pruning*) to blend it into the sparse training procedure, which allows us to shrink even more the number of weights and to accelerate sparse training considerably.
- We developed a customised and modularized software framework for sparse neural networks to test our theoretical contributions. It allows us to break the record and to train as a proof-of-concept the largest neural network model ever trained, i.e. 50 million neurons.

The three approaches (*WASAP-SGD*, *All-ReLU* and *Importance Pruning*) represent independent contributions to sparse neural network literature, but also they can be used together as complementary methods to improve further the performance of sparse models, as we illustrate in our experimental results.

2 Results

Our work is focused on the most straightforward type of neural networks, MLPs, as they count for 61% of a typical Google TPU (Tensor Processing Unit) workload for production neural networks applications, while convolutional neural networks represent merely 5% (Jouppi et al. (2017)). Despite the numerous algorithms available to train sparse neural networks from scratch, we decided to base our evaluation on the SET algorithm, given its simplicity and good performance on a broad range of domains. Unlike the other sparse training techniques mentioned in section 4 that calculate and store information for all the parameters, including the non-existing ones, SET is memory-efficient because it uses information just from the existing parameters, and it does not require high computational complexity. These are all favourable advantages to our goal of developing large scale sparse neural networks. We evaluate and discuss the performance of our proposed methods on sparse MLP models by considering five publicly available datasets listed in Table 1.

Problem formulation

Given a dataset $\mathcal{D} = (x_i, y_i)_{i=1}^{n}$ and a network $f(x;\theta)$ with L layers parameterized by θ (weights $\mathbf{w}$ and biases $\mathbf{b}$). We train the network to minimize the loss function $\sum L(f(x;\theta), y)$. The motivation of sparse neural networks is to use a fraction of parameters to reparameterize the whole network, while preserving the performance as much as possible. Hence, a sparse neural network can be denoted as $f_s(x;\theta_s)$ with a given sparsity level. Initially, the network is uniformly initialized with a sparse distribution in which the sparsity level $\mathcal{S}_l$ of each layer l is controlled by a parameter ϵ (see Mocanu et al. (2018) for details) and stays constant during the training. More precisely, for each layer l the connections are defined in a sparse adjacency weights matrix $\mathbf{W}^{(l)} = [[w_{11}, w_{12}, \ldots, w_{1n_l}], \ldots, [w_{n_{l-1}1}, w_{n_{l-1}2}, \ldots, w_{n_{l-1}n_l}]]$ in which the elements are either *null* ($w_{ij} = 0$) when there is no connection between neuron i and neuron j or have a connection weight ($w_{ij} \neq 0$) when the connection between i and j exists. Initially each $\mathbf{W}^{(l)}$ is a Erdős-Rényi random graph (Erdös & Rényi (1959)).

2.1 Proposed contributions

WASAP-SGD method

We propose a novel parallel training method with two phases based on a data parallelism strategy (where the learning phase of a model is partitioned by input samples) to improve the scalability of sparse neural networks. The algorithm, called **WASAP-SGD**, is based on *asynchronous SGD* (Dean et al. (2012)) training for the first phase and *Stochastic Weight Averaging* (Izmailov et al. (2018)) for the second phase. This two-phase method helps in filling the gap between sparse and dense neural networks' performances (accuracy, running time and generalization).

We consider a system with K workers, which repeatedly compute gradient contributions based on independently drawn data mini-batches from the given dataset $\mathcal{D}$. We also consider a *shared parameter server*, which communicates with each of the workers independently, to give state information and get updates that it applies according to the algorithm it follows. The master and each worker have a replica of the sparse model to be trained. Moreover, each worker has access to a subset of the training data, as depicted in Figure 2.

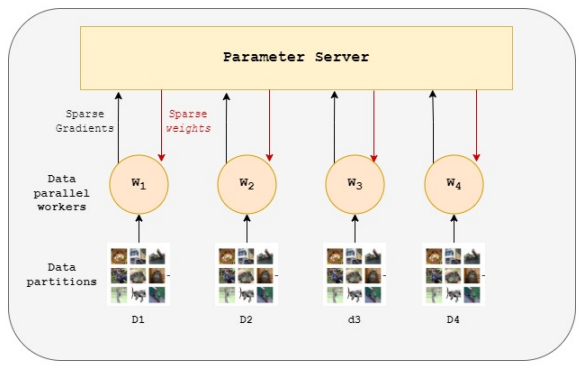


Figure 2. Sparse model replicas asynchronously fetch sparse parameters and push sparse gradients to the parameter server with atomic read and write operations.

In Algorithm 1 we show the pseudocode for *WASAP-SGD*, describing how standard *asynchronous SGD* using a parameter server is extended with a local training phase followed by a *sparse* model averaging step to improve its generalization performance. Moreover, it is designed to include the topology adaptation step of sparse networks. The training is carried out *asynchronously* by all workers. We adopt a simple SGD update rule with momentum, which has shown to be effective for training intrinsically sparse models:

$$\mathbf{w_{t+1}} = \mathbf{w_t} + \mu(\mathbf{w_t} - \mathbf{w_{t-1}}) - \eta_t \nabla \mathbf{w_t} \tag{1}$$

The master must periodically pause the asynchronous update to carry on the weight evolution algorithm on the sparse model to generate the new topology. Each update must include a minor modification, since individual weights may be outdated due to the topology evolution (as illustrated in Figure 3).

Then, to improve the model generalization performance, each worker locally updates its sparse replica for the next phase (*phase two*). Once *phase two* is concluded, the K different models are averaged:

$$\theta_s^f = \frac{1}{K} \sum_{i=1}^{K} \theta_s^i \tag{2}$$

The averaging step does not preserve the sparsity level $\mathcal{S}$, since each worker has updated its topology independently from each other. Hence, the final model θ_s^f will have a different sparsity level $\mathcal{S}'_{(l)}$, for each layer l, where $\mathcal{S}'_{(l)} \geq \mathcal{S}, \forall l = 1, \ldots, L$. Thus, unimportant connections, accounting for a certain fraction $\mathcal{S}'_{(l)} - \mathcal{S}$, will be pruned in each layer. More precisely, the unimportant connections are pruned based on their magnitude,

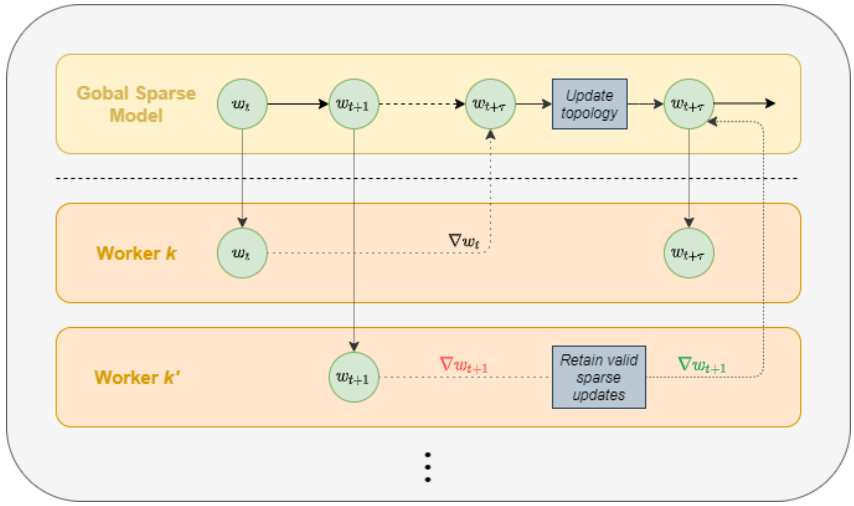


Figure 3. Worker k' fetches parameters w_{t+1} and push gradients Δw_{t+1} to the parameter server. These gradients may contain non-valid updates, since in that time frame the global model may have performed the topology evolution, hence they need to be corrected.

corresponding to the largest negative weights and the smallest positive weights in $\mathbf{W}^{(l)}$.

All-ReLU

The new proposed activation function, **All-ReLU** (Alternated Left ReLU), is designed for training sparse MLPs and is able to accelerate training, without adding any additional computational complexity. All-ReLU is inspired by the S-shaped rectified linear activation unit (SReLU) presented in Jin et al. (2016). The intuition behind it came from analysing the SReLU parameters as well as the input distribution of the learned topology. Like SReLU, this function can improve the networks gradient flow, and consequently achieve better accuracy. However, since All-ReLU does not require to train additional parameters, it can be considered as simple and fast as ReLU to use.

Given an ANN with L layers, our proposed Alternate Left ReLU (All-ReLU) is defined as follows for each layer l:

$$f_l(x_i) = \begin{cases} -\alpha x_i & x_i \leq 0 \ \& \ l\%2 == 0 \\ \alpha x_i & x_i \leq 0 \ \& \ l\%2 == 1 \\ x_i & x_i > 0 \end{cases} \tag{3}$$

where x_i is the input value, α is the slope for the negative side of the input and % represent the *modulo* operation. The input layer ($l = 1$) and the output layer ($l = L$) are excluded. We believe that the proposed activation function can accelerate convergence by breaking symmetry during training and preserving the gradient flow through the network, hence leading to better performance for sparse models.

Importance Pruning

To substantially reduce the size of neural networks, we propose a novel method, for selecting the most important neurons, based on their strength (importance). In graph theory, the node strength is the sum of weights of links connected to the node (Barrat et al. (2004)). Taking inspiration from this graph measure, we determine the importance of each neuron based on the summation of absolute weights of its incoming connection. For each neuron j in layer l we define its importance as follows:

$$I_j^{(l)} = \sum_{i \in \Gamma_j^{(l-1)}} |w_{ij}^{(l)}| \tag{4}$$

where $\Gamma_j^{(l-1)}$ is the set of all neurons connected to neuron j, i.e. $\Gamma_j^{(l-1)} = \{i | 1 \leq i \leq n^{(l-1)}, \forall i \in \mathbb{N} \ \wedge \ w_{ij}^{(l)} \neq 0\}$, given that $n^{(l-1)}$ is the number of hidden neurons from the previous layer $l-1$ and $w_{ij}^{(l)}$ denotes the weight of connection linking neuron i to j in two consecutive layers as defined in $\mathbf{W}^{(l)}$. This neuron importance metric can rapidly identify the main *hubs* of the sparse network, i.e. nodes that are positioned to make strong contributions to global network performance.

This metric can be easily integrated during training and we named this procedure *Importance Pruning*, once the topology is stable, to reduce overfitting with almost no loss in accuracy and substantially lessen the number of parameters up to 80% with respect to the initial sparse network and, as a consequence, decrease the overall training running time. In Algorithm 2, we provide an example of how *Importance Pruning* can be integrated with dynamic sparse training. The pseudocode refers to the SET algorithm; however, it could be easily replaced by any other sparse-to-sparse training technique.

Large scale Sparse Neural Network framework

We extended the sparse framework presented in Liu et al. (2020b) by implementing the theoretical contributions presented in this paper. The initial implementation was sequential and it was not able to obtain the same accuracy as Keras on some datasets such as CIFAR10. Our focus was on MLP as we did not have the human resources to develop RNNs and CNNs from scratch and we let this as future work. With respect to the speed of our approaches, it is worth to highlight that first we substantially improved the training time of a truly sparse MLP from its previous implementation in Liu et al. (2020b) with no parallelisation by replacing Cython with Numba (Lam et al. (2015)) and adopting `32-bit` float precision instead of `64-bit`. These minor changes ensure minimum resource requirements. WASAP-SGD is designed using Python 3.7, one of the most popular Machine Learning languages, combined with Message Passing Interface (MPI).

With our framework, we were able to build the largest Sparse MLP model, in terms of the number of neurons, ever trained on a single machine on the cloud (with no GPU). Note that the enormous models mentioned in literature have been usually trained in a distributed fashion or on several GPUs. Since a high-dimensional dataset for this task is hard to find, we use an artificial one to train a model with 50 million neurons on a machine with 96 virtual cores and 768 GB of RAM. This experiment demonstrates how the Sparse MLP can achieve what its dense counterpart cannot, due to memory error.

Experiment	Dataset	Dataset properties						
		Domain	Features	Train samples [#]	Train samples [MB]	Test samples [#]	Test samples [MB]	Classes
MLPs	Leukemia (Edgar & Lash, 2002)	Microarray	54675	1397		699		18
	Higgs (Baldi et al., 2014)	Physics particles	28	105000	12	50000	6	2
	Madelon (Guyon et al., 2005)	Artificial data	500	2000		600		2
	FashionMNIST (Xiao et al., 2017)	Images	784	60000	188	10000	32	10
	CIFAR10 (Krizhevsky, 2009)	RGB Images	3072	50000	615	10000	123	10

Table 1. List of dataset used for the experiments.

2.2 Performance on Sequential Trained Sparse MLPs

This section summarises the performance for the comparison between All-ReLU and ReLU on sparse MLP models, and their integration with *Importance Pruning* to speed up the training. All SET-MLP variants have been run using our own truly sparse implementation framework and just one CPU core. Table 2 lists the maximum accuracy for each method/dataset combination, along with the total training time (expressed in minutes), the number of parameters at the beginning ($start_n^W$) and at the end of the training (end_n^W). This is particularly interesting to report when *Importance Pruning* is applied to the sparse models for understanding the benefits in terms of memory footprint. The resulting learning curves and errors of our experiments are shown in Figure 6 and Figure 7, for both testing and training sets. For all figures, we obtain the mean accuracy by averaging the best test accuracy from 5 trials over 500 epochs. Moreover, for each model, we display the resulting number of

parameters for the dense MLP version, the basic SET-MLP and SET-MLP where the neuron importance metric is adopted for taking out unimportant hidden units. Since a key result is the relative space improvement from the *Importance Pruning*, in Figure 4, we plot the relative model size (expressed in number of parameters) against the relative error to highlight the differences with and without *Importance Pruning* (where Figure 4a summarizes the final training results and Figure 4b for the final testing results).

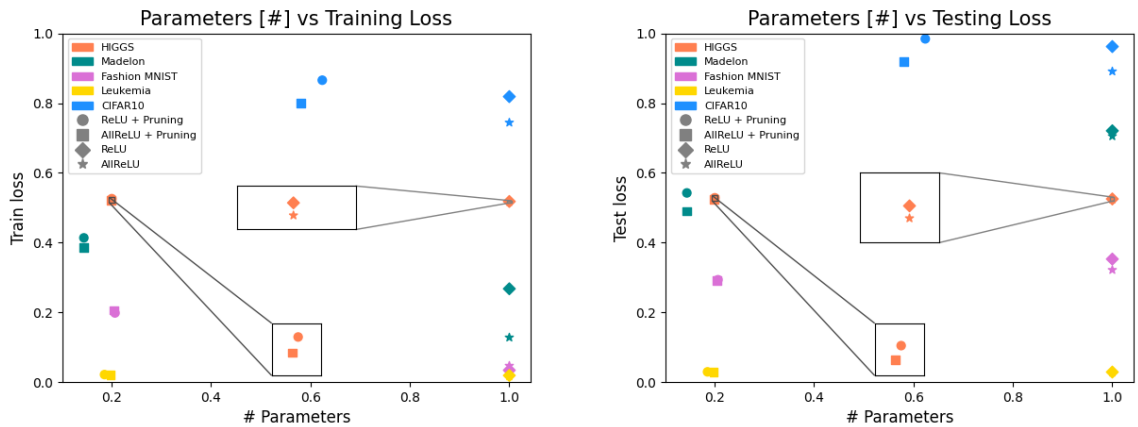
Dataset	**Architecture**	**NN model**	**CPU cores**	**Activation**	**Importance Pruning**	**Results**			
						Accuracy [%]	**start_n^W [#]**	**end_n^W [#]**	**Training [min]**
Leukemia	54675-27500-27500-18	**SET-MLP**	1	ReLU	no	85.98	1684944	1684944	~ 375
				ReLU	yes	85.40	1582214	312900	~ 266.5
				All-ReLU ($\alpha = 0.75$)	no	**86.42**	1582518	1582518	~ 375
				All-ReLU ($\alpha = 0.75$)	yes	85.69	1582581	313047	~ 266.5
		Dense MLP [2]	12	ReLU	-	n/a	2260362518	2260362518	n/a
				All-ReLU	-	n/a	2260362518	2260362518	n/a
Higgs	28-1000-1000-1000-2	**SET-MLP**	1	ReLU	no	73.59	50224	50244	~ 216.67
				ReLU	yes	73.50	50246	10065	~ 182.3
				All-ReLU ($\alpha = 0.05$)	no	73.67	50165	50165	~ 216.67
				All-ReLU ($\alpha = 0.05$)	yes	**73.76**	50165	9992	~ 140.3
		Dense MLP	12	ReLU	-	70.59	2033002	2033002	~ 182.3
				All-ReLU	-	70.10	2033002	2033002	~ 140.13
Madelon	500-400-100-400-2	**SET-MLP**	1	ReLU	no	68.50	19000	19000	~ 3.6
				ReLU	yes	75.00	19000	2739	~ 3.2
				All-ReLU ($\alpha = 0.5$)	no	71.33	19011	19011	~ 3.6
				All-ReLU ($\alpha = 0.5$)	yes	**77.00**	19000	2737	~ 3.2
		Dense MLP	12	ReLU	-	59.66	281702	281702	~ 3.62
				All-ReLU ($\alpha = 0.5$)	-	62.00	281702	281702	~ 3.62
FashionMNIST	784-1000-1000-1000-10	**SET-MLP**	1	ReLU	no	90.48	126302	126302	~ 137.5
				ReLU	yes	89.43	126302	26111	~ 96.25
				All-ReLU ($\alpha = 0.6$)	no	**91.38**	126302	126302	~ 137.5
				All-ReLU ($\alpha = 0.6$)	yes	90.12	126302	25759	~ 96.25
		Dense MLP	12	ReLU	-	90.85	2797010	2797010	~ 95.05
				All-ReLU ($\alpha = 0.25$)	-	90.73	2797010	2797010	~ 95.05
CIFAR10	3072-4000-1000-4000-10	**SET-MLP**	1	ReLU	no	67.05	381758	381758	~ 590
				ReLU	yes	65.21	381758	238114	~ 530
				All-ReLU ($\alpha = 0.75$)	no	**69.83**	381425	381425	~ 590
				All-ReLU ($\alpha = 0.75$)	yes	68.55	380318	221323	~ 530
		Dense MLP	12	ReLU	-	64.94	20337010	20337010	~ 530.7
				All-ReLU ($\alpha = 0.25$)	-	67.96	20337010	20337010	~ 530.7

Table 2. On each dataset, we report the best classification accuracy and error obtained by each model on the test data over five different runs for 500 epochs. $start_n^W$ represents the number of weights in the model at the beginning of the training, while end_n^W represents the number of parameters in the final model. *Importance Pruning* (y/n) indicates if the proposed pruning strategy based on our neuron importance metric is activated. *Training* reports the overall running time needed for training the models. Furthermore, the table reports the performance of fully connected MLPs (Dense MLPs) with both activation functions. The networks have been trained using momentum SGD in its standard sequential version. It is worth mentioning that the Dense MLP has been run using Keras using all CPU cores and SET-MLP using our own implementation and just one CPU core.

We can observe that All-ReLU consistently outperforms ReLU on all datasets, indicating that the novel activation function associated with a sparse-to-sparse training algorithm helps to model better the data distribution. Moreover, when *Importance Pruning* is activated, we can notice a significant reduction in the number of parameters, which leads to a remarkable speedup in running time, with almost no loss in terms of accuracy. Looking at the CIFAR10 dataset (Figure 6e), we can see that the new activation function is capable of boosting the accuracy on test data by more than 2%. SET-MLP on CIFAR10, after 500 epochs when using SReLU reaches about 70.30 % accuracy. This result suggests that All-ReLU indeed fills the performance gap with SReLU successfully. The model version with *Importance Pruning* can achieve comparable performances while training roughly 40% fewer parameters (where the reduced number of parameters refers to the final model and it is obtained by gradually reducing the connections during training) and gaining a speedup of 60 minutes. In this case, the importance

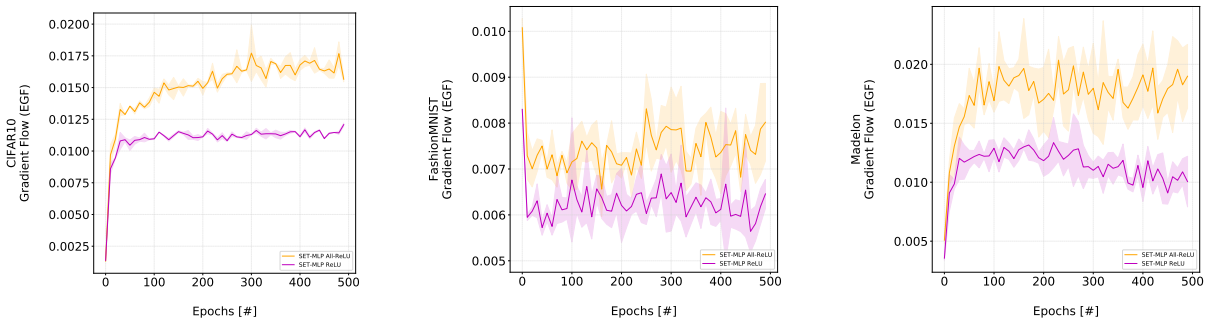
2 With our hardware settings (CPU Intel Core i7-9750H, 2.60 GHz × 6, RAM 32 GB, Hard disk 1000 GB, NVIDIA GeForce GTX 1650 4GB), it was not possible to train the dense MLP on Leukemia due to memory error.

metric seems to be more stable when adopting All-ReLU, resulting in a minor loss in performance. A similar outstanding result is obtained with Madelon (Figure 6c), where All-ReLU obtains 3% increase in accuracy with no *Importance Pruning* and about 2% with *Importance Pruning* (where the model uses 80% fewer parameters). Here, the *Importance Pruning* method has improved the performance significantly.



(a) Train results (b) Test results

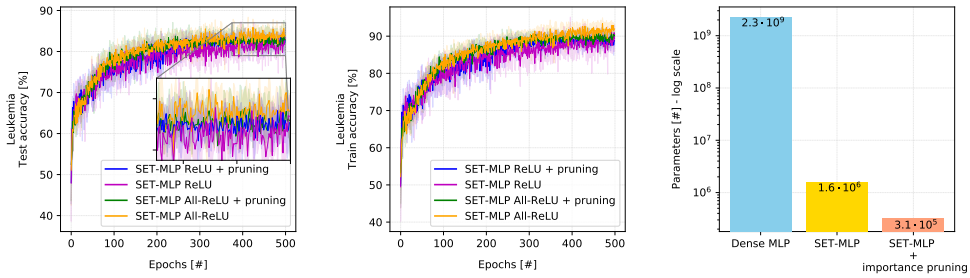
Figure 4. Relative model size (expressed in number of parameters [#]) against the relative error (loss) to highlight the differences with and without *Importance Pruning*.



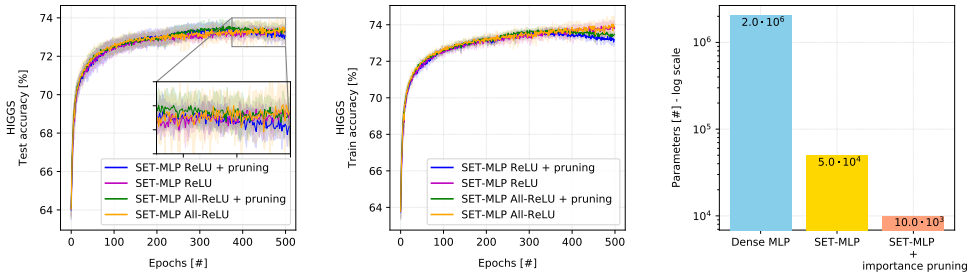
(a) CIFAR10 (b) FashionMNIST (c) Madelon

Figure 5. Gradient Flow for sparse MLPs with three hidden layers on CIFAR10 (a), FashionMNIST (b) and Madelon (c) trained with All-ReLU and ReLU.

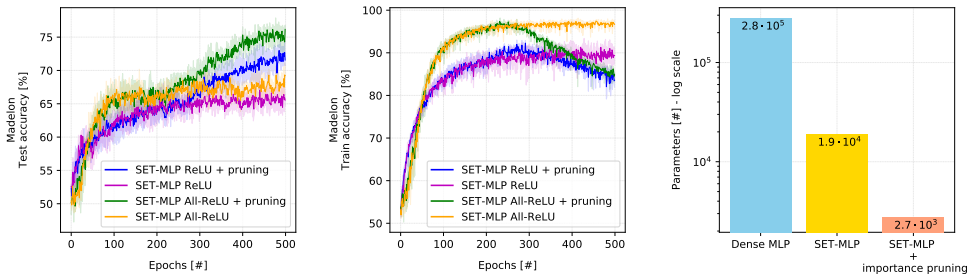
For FashionMNIST, All-ReLU can surpass both ReLU (Figure 6d) and SReLU. The latter achieves around 90.10 % accuracy when trained for 500 epochs with the same settings, while All-ReLU achieves 91.38 %. The smaller version obtained via Importance Pruning ends with 20% of the original parameters, gaining a 41 minutes speedup. We attain similar performance for Leukemia and Higgs, although the increase in accuracy is more predominant for image data and Madelon. Lastly, in Figure 5, we show the gradient flow for the sparse models trained with All-ReLU and ReLU on CIFAR10 (Figure 5a), FashionMNIST (Figure 5b) and Madelon (Figure 5c). We recall that gradient flow is the first-order approximation of the decrease in the loss expected after a gradient step, hence the higher, the better. All-ReLU visibly improves this metric, which is associated with efficient training of sparse neural networks (ab Tessera et al. (2021); Wang et al. (2020); Evci et al. (2020)).



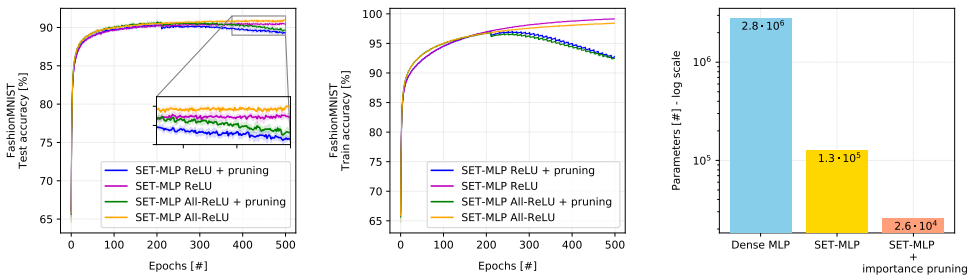
(a) Learning curves and parameter numbers comparison for Leukemia dataset.



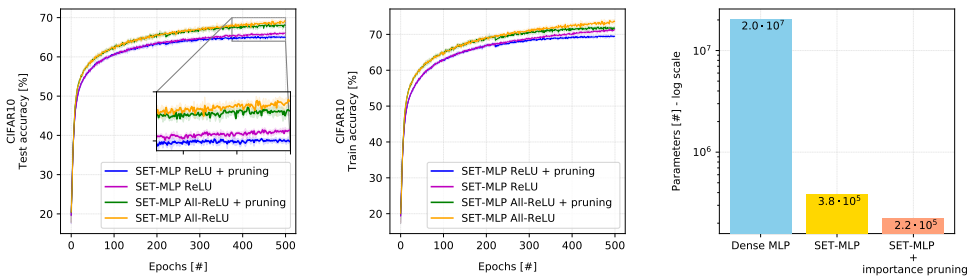
(b) Learning curves and parameter numbers comparison for HIGGS dataset.



(c) Learning curves and parameter numbers comparison for Madalon dataset.

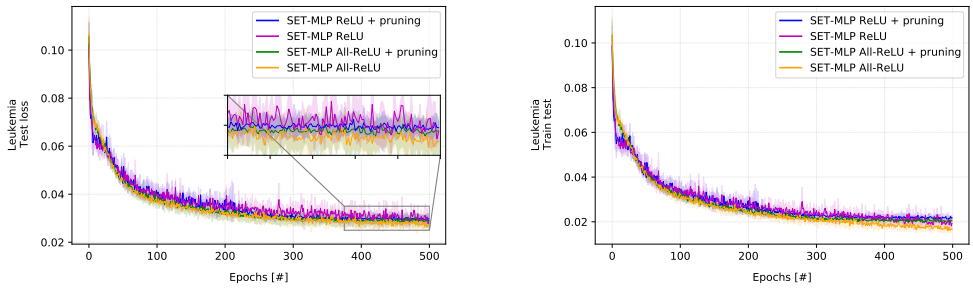


(d) Learning curves and parameter numbers comparison for FashionMNIST dataset.

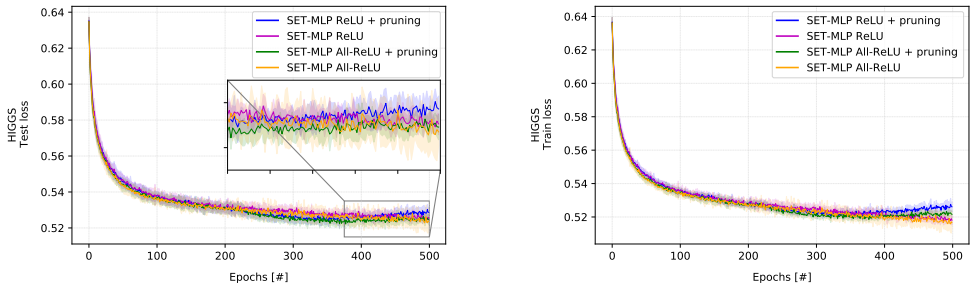


(e) Learning curves and parameter numbers comparison for CIFAR10 dataset.

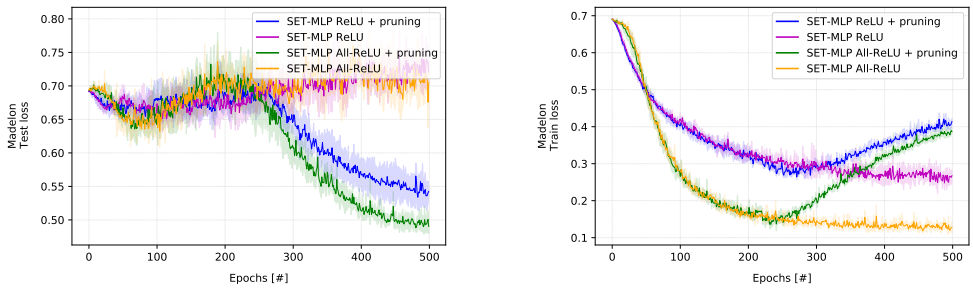
Figure 6. Evaluation of the proposed methods on five different datasets. These results are obtained by standard sequential training with *momentum* SGD.



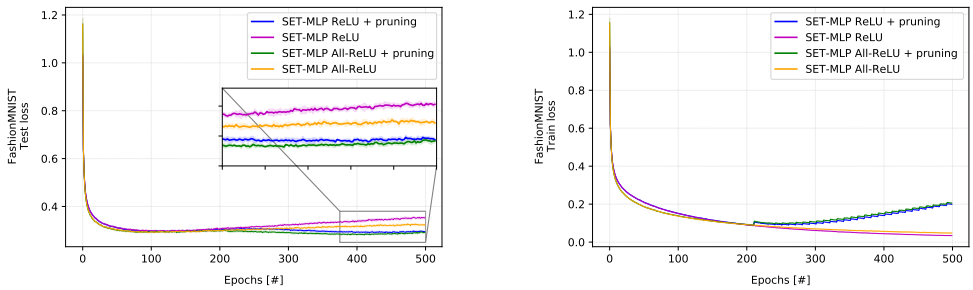
(a) Train and test error for Leukemia dataset.



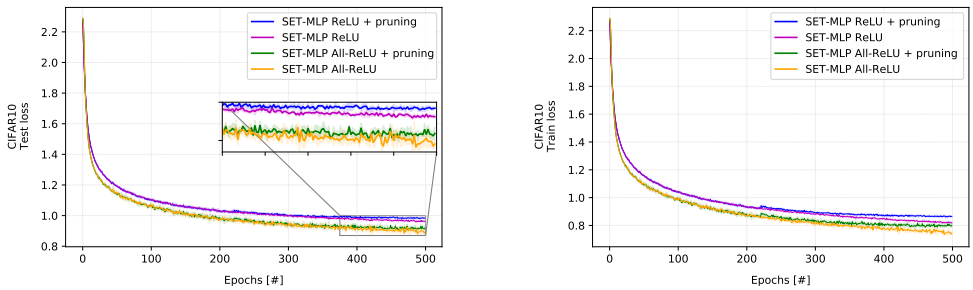
(b) Train and test error for HIGGS dataset.



(c) Learning curves and parameter numbers comparison for Madalon dataset.



(d) Train and test error for FashionMNIST dataset.



(e) Train and test error for CIFAR10 dataset.

Figure 7. Train and test errors of the experiments results presented in Figure 6 (sequential training with *momentum* SGD).

2.3 Performance on Parallel Trained Sparse MLPs

The available algorithms for parallelisation of dense neural networks are not suitable for sparse neural networks training. Hence they are not considered in our experimental evaluation of the proposed method. We did not test WASAP-SGD with Madelon and Leukemia because there is no added value in parallelising the training under the conditions where the data and/or the model size are minimal. In Table 3 we summarise the performance of the proposed parallel algorithm on three different datasets where All-ReLU is adopted, with and without *Importance Pruning*. For completeness, we report the results for a variant where *phase one* is synchronous, called WASSP-SGD (*Weight Averaging Sparse Synchronous Parallel SGD*). In this way, we want to empirically demonstrate that our asynchronous version is more suitable to train sparse models. Furthermore, for easy comparison, we again report the accuracy and training time for the sequential version (baseline). Since we run the experiments on a machine with six physical cores, we employed five workers and one master (parameter server). The time reduction does not improve significantly as the number of workers surpasses the number of physical processors.

The synchronous variant WASSP-SGD is implemented by following the suggestions from (Goyal et al. (2017)), such as their gradual warmup and linear scaling rule for the learning rate. Conversely, for our WASAP-SGD, where the first phase is asynchronous, we observed it benefits from larger learning rates for the first few epochs, followed by fixed learning rates.

In Table 3, we also show the average running time using the Keras implementation of SET-MLP with a mask over the parameters. The statistics are provided for three different configurations of the hardware: training on one CPU core only, training with no constrains (all 12 CPUs cores) and GPU training. These numbers allow us to make a comparison between our proposed sparse framework and a popular deep learning library like Keras. By looking at the results in Table 3, we can observe that the proposed parallel algorithm exhibits persistently better convergence when the first phase is carried out asynchronously (WASAP-SGD). The same outcome holds in terms of training time. If we compare the running times of *WASAP-SGD* against the one from the sequential version, we gain an improvement of about half among all datasets, without introducing a notable increase in memory footprint. Our parallel method is able to outperform Keras CPU-based wall-clock running time for training of sparse MLPs significantly. We would like to emphasise that our sparse implementation together with parallelisation and *Importance Pruning* gets quite close to GPU training time (see Table 3). Plus, this is the kind of comparison that people typically do not make, because (for dense networks) does not make sense to compare CPU with GPU. Moreover, as we could notice from the results in subsection 2.2 for the Leukemia dataset, this computational advantage of GPUs manifests its limitation when it comes to training huge models. In this case, Keras is not able to allocate the dense tensors, resulting in a memory error. Additionally, it is worth mentioning that GPU training utilises more resource than CPU training and classic GPU training, as it also uses the CPU besides GPU. Hence, we believe that our approach has more potential to tackle large scale deep learning models.

2.4 Extreme large sparse MLPs

To the best of our knowledge, the largest public (claimed) dense neural network has 160 billion (B) parameters, where a parameter roughly corresponds to a synapse in the human brain. Given the human brain is estimated to have about 100 trillion synapses, that neural network could be said to be about 0.16% of the human brain. State-of-the-art networks in a generic sense contain around 16M neurons. For comparison, the 16 million (M) neurons number when compared with the 100B neurons in a human brain—only represents 0.016% of human brain size. Now, it is worth noting that size is not the only thing that matters. Most advanced models today perform worse when their network size is increased blindly. While it is true that DNN capabilities increase with their network size, there is also a fair amount of engineering work that goes into making larger networks accurate. Hopefully, sparsity will help in overcoming some challenges when training such large models.

In this scenario, we tried to push the limit of ANNs on a virtual machine with 96 cores and 768GB of RAM to train extreme large sparse MLP models. We are attempting to enter a region where neural networks have never been explored. Hence, we believe that any small finding is important. Because of limited time and resources,

Dataset	Framework	Importance Pruning	CPU cores	GPU	Accuracy [%]	Training time [min]	CPU usage [%]	Memory usage [MB]
Higgs	WASSP-SGD	no	12	no	74.26	~ 142.52	~ 62%	~ 2800 MB
	WASSP-SGD	yes	12	no	74.16	~ 113.33	~ 62%	~ 2800 MB
	WASAP-SGD	no	12	no	74.47	~ 138.52	~ 62%	~ 2800 MB
	WASAP-SGD	yes	12	no	74.88	~ 108.33	~ 62%	~ 2800 MB
	Sequential	no	1	no	73.67	~ 216.67	~ 24%	~ 1700 MB
	Sequential	yes	1	no	73.76	~ 182.3	~ 24%	~ 1700 MB
	Keras CPU	no	1	no	74.28	~ 350	~ 27 %	~ 2100 MB
	Keras CPU	no	12	no	74.28	~ 191.7	~ 65 %	~ 2300 MB
	Keras GPU	no	12	yes	74.28	~ 78.3	~ 25 %	~ 2400 MB
FashionMNIST	WASSP-SGD	no	12	no	90.52	~ 90.6	~ 65%	~ 3300 MB
	WASSP-SGD	yes	12	no	89.96	~ 77.6	~ 65%	~ 3300 MB
	WASAP-SGD	no	12	no	91.23	~ 87.25	~ 65%	~ 3300 MB
	WASAP-SGD	yes	12	no	90.15	~ 74.85	~ 65%	~ 3300 MB
	Sequential	no	1	no	91.38	~ 137.5	~ 20 %	~ 1900 MB
	Sequential	yes	1	no	90.12	~ 96.25	~ 20 %	~ 1900 MB
	Keras CPU	no	1	no	-	~ 316	~ 20 %	~ 2800 MB
	Keras CPU	no	12	no	-	~ 135.6	~ 76 %	~ 2800 MB
	Keras GPU	no	12	yes	-	~ 65	~ 26 %	~ 3000 MB
CIFAR10	WASSP-SGD	no	12	no	67.13	~ 309.9	~ 70 %	~ 6500 MB
	WASSP-SGD	yes	12	no	66.88	~ 279.9	~ 70 %	~ 6500 MB
	WASAP-SGD	no	12	no	69.03	~ 281.9	~ 70 %	~ 6500 MB
	WASAP-SGD	yes	12	no	68.51	~ 246.5	~ 70 %	~ 6500 MB
	Sequential	no	1	no	69.83	~ 590	~ 25 %	~ 2200 MB
	Sequential	yes	1	no	68.55	~ 530	~ 25 %	~ 2200 MB
	Keras CPU	no	1	no	-	~ 1000	~ 20 %	~ 4700 MB
	Keras CPU	no	12	no	-	~ 530.7	~ 80 %	~ 4700 MB
	Keras GPU	no	12	yes	-	~ 195	~ 20 %	~ 4800 MB

Table 3. The table reports the accuracy, average running time and average resource utilisation over five different runs for 500 epochs when using parallel training with WASAP-SGD and our proposed sparse implementation framework. For completeness, we report the performance for the synchronous (phase 1) version of the algorithm as well. Here, this synchronous version is called WASSP-SGD (*Weight Averaging Sparse Synchronous Parallel SGD*). Moreover, we include the performance of sequential training for facilitating the comparison. *Importance Pruning* (y/n) indicates if the proposed pruning strategy based on our neuron importance metric is activated. Note that in this setting, the memory usage will decrease during training. To have an idea on how these performances can be compared against a state-of-the-art framework, we report the running time and memory statistics when using the Keras implementation of SET-MLP with a mask over the parameters. These statistics are provided for three different configurations of the hardware.

we did not repeat experiments to get statistical confidence. We just run them once for few epochs, on different regimes, to collect statistics such as matrix initialisation time, training time per epoch, inference time and topology evolution time.

Our main goal was to train a sparse model with more than 125 million neurons because we believe that this is the latest state-of-the-art (just for inference) size, according to (Mohammad Hasanzadeh Mofrad & Hammoud (2021)). For the sake of clarity, we mention that these results ignore entirely the training focusing just on inference. We want to go one step further and train such models. Herein, we have discovered that we are very close of reaching the limits of our proposed implementation framework of training extremely large sparse neural networks, but it is good to know the limits in order to know how to proceed further.

There are not many available datasets with a high number of features and a reasonable amount of data points to exploit data parallelism. For this reason, we created an artificial dataset by adopting the function `make_classification` [3] from Scikit-learn, a free software machine learning library for the Python programming language. The algorithm is adapted from (Guyon (2003)) and was designed to generate the "Madelon" dataset. We generated a binary classification task with 10000 samples, where each sample has 65536 features. We used 30% of the data as test data and 70% as train data. The models are trained using our parallel algorithm WASAP-SGD with momentum (set to 0.9), weight decay and dropout (set to 0.4). Moreover, the batch size is set to 128, and the learning rate is 0.01. The statistics for different architectures are presented in table Table 4. Moreover, we report the number of workers, the number of parameters and the value for ϵ, which controls the sparsity level. We stress that the training is performed on a single machine with no GPU, while popular state-of-the-art models are usually trained with distributed algorithms on multi GPUs.

Architecture	Epsilon ϵ	Neurons [#]	Parameters [#]	Workers [#]	Weight initialization [min]	Training [min]	Testing [min]	Weight evolution [min]
65536-0.5M-0.5M-2	10	1 M	20.6 M	16	~ 1	~ 6	~ 2.5	~ 1
65536-2.5M-2.5M-2	5	5 M	50.3 M	16	~ 2	~ 10	~ 6	~ 2
65536-5M-5M-2	5	10 M	100.3 M	16	~ 3	~ 20	~ 11	~ 5
65536-5M× 4-2	1	20 M	40.3 M	8	~ 5	~ 26	~ 18	~ 6
65536-5M × 10-2	1	50 M	100.3M	8	~ 9	~ 52	~ 20	~ 6

Table 4. The table reports the running time (per epoch) when training extreme large sparse models with WASAP-SGD on the big artificial dataset. The training is performed with 16 workers. We trained the model for a few epochs, to make sure the loss is decreasing, hence the networks are learning.

While trying to build these large sparse models, multiple challenges and technical difficulties have emerged:
Inference bottleneck: Up to this moment, our work was focused on the training phase of neural networks since inference did not play an important role for common sized networks. However, once we started building extreme large models, we immediately noticed how important it is to optimise this phase as well. We tried to overcome the bottleneck by parallelising the inference in batches with python `mutiprocessing`. We are aware that there exist more sophisticated approaches, but this simple solution could already decrease the running time considerably.
MPI overflow: Parallelization in Python integrates Message Passing Interface via `mpi4py` module. When we first selected this framework, we did not consider that mpi4py does not support the parallelisation of objects greater than 2^{31} bytes, limiting the size of sparse matrices to be created. This limitation is probably becoming more noticeable nowadays, given the importance of Big Data analysis, and in Ascension & Araúzo-Bravo (2020), the authors developed BigMPI4py, a Python module that wraps mpi4py, supporting object sizes beyond this boundary.
Matrix initialisation time: When building such larges models, weights initialisation starts to play a significant role. If the sparse matrix initialisation is not implemented efficiently, this may cause a bottleneck. In this regard, we had to vectorise this step in order to reduce the weight initialization running time.

3 More details about the function are available here.

Memory allocation issues: Sparsity allows for creating a bigger model (in terms of the number of hidden units) than when one adopts fully connected layers. However, this advantage has its limitations as well. When adopting a data parallelism strategy, the sparse model is replicated among all workers to accelerate the training procedure. When the model becomes extremely large, the number of workers which can fit in memory decreases (as can be noticed in Table 4). At this point, the training should become distributed by having each workers running on a different node in a network. Alternatively, other strategies may be explored to overcome the memory issues. Our implementation adopts MPI standard; hence it could be automatically run in a distributed fashion. Nevertheless, the feasible size of sparse models is much larger than their dense counterparts. In this regard, the largest dense network we can build on the virtual machine has around 600000 neurons.

We note that the weight evolution step is able to scale successfully without adding too much overhead. Moreover, we want to briefly outline that we managed to train a model with 10 million neurons ($\epsilon = 1$) via sequential training on Leukemia. Each training epoch takes around 33 minutes, and 18 seconds, inference takes 12 minutes and 9 seconds, and evolution time 30 seconds. Similarly, we could train a model with 50 million neurons where one epoch takes about 1 hour and 15 minutes. In the end, we were not able to train sparse models with more than 50 million neurons due to the challenges listed above. After a point, to grow the number of neurons, we had to increase the sparsity as well as decrease the number of workers. Nonetheless, from these extreme experiments, we could learn some of the limitations of our approach and the chosen technologies. Lastly, it is important to highlight that these limitations come from the implementation itself and they are not limitations of the three theoretical contributions.

3 Discussion

To improve the scalability of ANNs by exploiting sparsity, three main contributions have been introduced in this work. Each of them has been implemented in a truly sparse manner with sparse matrices and operations. First, we introduced *WASAP-SGD*, a new parallel algorithm to efficiently train sparse neural networks asynchronously. This type of communication protocol has proven to be more effective than synchronous training for sparse models, both from a running time and convergence point of view. Furthermore, the last training phase ensures higher generalization performance. For example, for CIFAR10, we could bring down the sequential training time of *590* minutes to *282* without *Importance Pruning* and to *246* with *Importance Pruning* while the GPU training time is around *200*. It should be pointed out that all communications are intrinsically sparse, reducing the overhead significantly. From our experimental evaluation, we argue that sparsity could more easily overcome the typical negative traits of asynchrony. At the same time, the communication overhead is mitigated since the processes in the system exchange sparse updates. Lastly, the concept of staleness-adaptive AsyncPSGD and delayed compensation strategies have been explored, but they did not improve statistical efficiency.

Secondly, we proposed a new activation function called *All-ReLU* to boost sparse MLPs performances. All-ReLU has shown promising results, outperforming ReLU in all five datasets, across various domains, without adding any extra computational complexity to the training procedure. To present some outstanding results, it has shown to significantly increase accuracy on test data for CIFAR10 and Madalon by more than 2% when compared to ReLU. The benefit on CIFAR10 becomes even more visible if we train the network longer. In this case, All-ReLU increases the accuracy by 3.4 % when compared to the classic ReLU. Moreover, if the model is trained longer, All-ReLU gets on par results (72-73%) with SReLU, achieving an accuracy close to the 73-74% reported in original SET (Mocanu et al., 2018). For image datasets, we hypothesized that the higher performance of our activation function might be caused by its ability to capture the feature shift that is common in this data. Our results are in line with independent parallel literature on sparse neural networks, as we demonstrate that All-ReLU achieves better performance by enriching the gradient flow during the training process. These findings make stronger the contribution of both works and the generality of the concept.

Lastly, we proposed a metric to define neuron importance which can be employed to remarkably shrink the number of parameters via *Importance Pruning*, an active pruning strategy. The sparsity level of the network can be increased without performance loss using our proposed method, which reduces computation time and

memory requirements. The combination of the new activation and *Importance Pruning* has been tested across all datasets, resulting in better or comparable outcomes when pruning is included. The most interesting case has been revealed for Madelon data, where the combination of the two methodologies has significantly improved performance up to 77%. This happens because the artificial dataset contains many redundant features which are eliminated, and it might imply that the neuron importance metric is useful for implicit feature selection.

The three methods are complementary and can be combined to obtain large scale sparse MLPs. In subsection 2.4 we performed some experiments to train extremely large networks (in terms of neurons), and we managed to train a sparse MLP with 50M neurons on a single machine. Thanks to this investigation, we could identify many important limitations in order to open several new research directions. Our contributions allow advancing the state-of-the-art in representational power (i.e. number of neurons) of artificial neural networks. Currently, up to our knowledge, the largest ANNs, built on supercomputers, accommodate the size of a frog's brain (about 16 million neurons)(Goodfellow et al. (2016)). After some technical challenges are overcome, with sparse neural networks, we may create on the same supercomputers ANNs close to the human brain size (about 80 billion neurons).

There are several directions for future work. The concept of staleness-adaptive AsyncPSGD for the first training phase has been under-explored for a high number of workers. Although these adaptations do not seem to help in our experiments, continuing to investigate asynchrony-aware SGD is of interest for very sparse large models. Future research directions also include investigating the nature of sparse training with more extensive experiments on various model architecture, as CNNs and Transformers. The latter would likely benefit the most since they use large dense layers. Additionally, it would be intriguing to consider a decentralized architecture with no parameter server involved. From an implementation point of view, it would be great to develop the parallelization in C++, in order to achieve better performance and overcome some sloppy characteristics of Python, or to incorporate customized sparse matrix operations developed for GPUs (Gale et al., 2020). Lastly, we need to adopt distributed settings if we want to outstretch the size of ANNs and approach the human brain's size. A clear limitation for *All-ReLU* consists in choosing the slope α. Although we provide some practical advice (see subsection 5.2), it would be interesting to find a way to tune this parameter before training automatically.

We believe that our research opens the path for obtaining better performance for current state-of-the-art sparse training research in terms of accuracy, computational requirements, and energy costs. With regard to the latter, people use artificial intelligence for climate change, but they do not improve deep learning to save energy. With our work, we hope to raise more awareness concerning this problem and show that it is possible to pursue sustainable supercomputing. Finally, it can pave the way to develop much larger neural networks with billion of neurons which can help us to tackle challenging problems in complex domains such as health care.

4 Methods

Sparse Neural Networks Training

Fully connected neural networks have been shown to have a substantial number of redundant parameters, and, in some cases, more than 95% of the parameters can be predicted from the remaining ones without accuracy loss (Denil et al. (2013)). In early work on sparsification, Optimal Brain Damage LeCun et al. (1989) and Optimal Brain Surgeon (Hassibi et al. (1993)) use gradient methods to sparsify networks during training. They observed that a sparse network demonstrated several advantages over its dense counterparts, such as better generalisation, reduced memory footprint and faster inference time. Additionally, the methodology of sparsification includes Hebbian pruning (Srivastava et al. (2014)), matrix decomposition (Liu et al. (2015)), and graph techniques (Kepner & Gilbert (2011); Kepner et al. (2017, 2018); Kumar et al. (2018); Prabhu et al. (2018)).

Dense-to-Sparse Training

Recently, more and more studies attempted to obtain memory and computational efficiency methods for the inference phase of deep neural networks. Numerous post-pruning techniques (dense-to-sparse training) have been proposed to reduce the number of parameters and speed up the inference phase across a broad range

of neural network architectures (Han et al., 2015; Srinivas & Babu, 2015; Han et al., 2016; Narang et al., 2017; Zhu & Gupta, 2017; Zhou et al., 2019); yet, these approaches require to fully train the dense model first. Several methods strove to learn the sparse networks during training (Louizos et al., 2018; Wen et al., 2018; Liu et al., 2020a; Kusupati et al., 2020; Yuan et al., 2021). However, these techniques begin with (or, are using at some moment during training) a fully-connected model, and as a consequence, they are not memory efficient. Another viable way is *one-shot pruning*, which aims to find sparse neural networks by pruning once before the main training phase (Lee et al., 2019, 2020; Wang et al., 2020). In this setting, at least one iteration of the dense model requires to be trained to identify the sparse sub-networks, and therefore the pruning process is unfeasible for memory-limited scenarios. Additionally, this method cannot meet the performance of dynamic sparse training, especially at extreme sparsity levels (Wang et al. (2020)).

Sparse-to-Sparse Training

The aforementioned issues can be naturally overcome by training intrinsically sparse neural networks from scratch to obtain the efficiency for both the training and inference phases. With respect to *Static Sparse Training*, Mocanu et al. (2016) introduced intrinsically sparse networks by exploring the scale-free and small-world topological properties in Restricted Boltzmann Machines. Later, some works focus on designing sparse CNNs based on Expander graphs and show comparable performance (Prabhu et al. (2018); Kepner & Robinett (2019)). In (Alford et al. (2019)) they tests pruning-based topologies, as well as RadiX-Nets from (Kepner & Robinett (2019)). Once more, results show that these static sparse networks obtain accuracies comparable to dense networks, but extreme levels of sparsity cause instability in training. A training technique that allows for sparsity throughout the entire training process in a *dynamic* fashion (sparse-to-sparse training) was first introduced in Mocanu (2017); Mocanu et al. (2018), with a simple and effective procedure called Sparse Evolutionary Training (SET) which uses magnitude-based pruning and random growth at the end of each training epoch. After that, in Deep Rewiring (DeepR) (Bellec et al. (2018)), the authors rigorously combined dynamic sparse parameterisation with stochastic parameter updates for training; however, this approach is computationally expensive and difficult to deploy on large models. More recent work like (Mostafa & Wang (2019a)) includes the cross-layer redistribution of weights, while in (Dettmers & Zettlemoyer (2019)) they present a similar approach which uses gradient information and momentum, significantly improving performances on various Convolutional Neural Networks (CNNs) models. Nevertheless, these additional calculations can result in a significant amount of extra computation. Very recently, based on the Lottery Ticket Hypothesis (Frankle & Carbin (2019)), RigL (Evci et al. (2019); Jayakumar et al. (2020)) was introduced as a novel method for training sparse models without the need of a "lucky initialisation"; it can match and sometimes exceed the performance of pruning based approaches. Lastly, in (Liu et al. (2021)), the authors propose an approach to successfully train sparse Recurrent Neural Networks (RNNs) with a stable number of floating-point operations (FLOPs) and a fixed parameters count.

Parallel Training of Deep Neural Networks

Accelerating training for Deep Neural Networks (DNNs) is a daunting challenge and techniques range from distributed algorithms to hardware optimisations. Stochastic Gradient Descent (SGD) (Robbins & Monro (1951)) together with backpropagation, and in particular, its mini-batch variants (Bottou (2010)) are the de-facto methods to train DNNs. SGD, however, is inherently sequential with a dependency across iterations and, this dependency limits parallelism. In (Ben-Nun & Hoefler (2018)), the authors provide an extensive survey about the vast catalogue of parallelisation approaches in deep learning. There are three prominent strategies to partition the learning phase of a model: partitioning by input samples (*data parallelism*), by network structure (*model parallelism*), and by layer (*pipelining*). Data parallelism can be easily implemented, and it is, therefore, the most widely used implementation strategy on multi-GPUs (Li et al. (2016)). We have explored this option focusing on CPUs only, where each core utilises the same sparse model to train on different data subsets. The replicas communicate updates through a centralised *parameter server* (shared memory) which maintains the current state of all parameters for the sparse model. In this architecture, there is no synchronisation between CPU cores

during the forward pass, but the gradients must be synchronised for the weights update. As for the parallelisation of SGD algorithms, one can choose to do it in either a synchronous or asynchronous way (Figure 8).

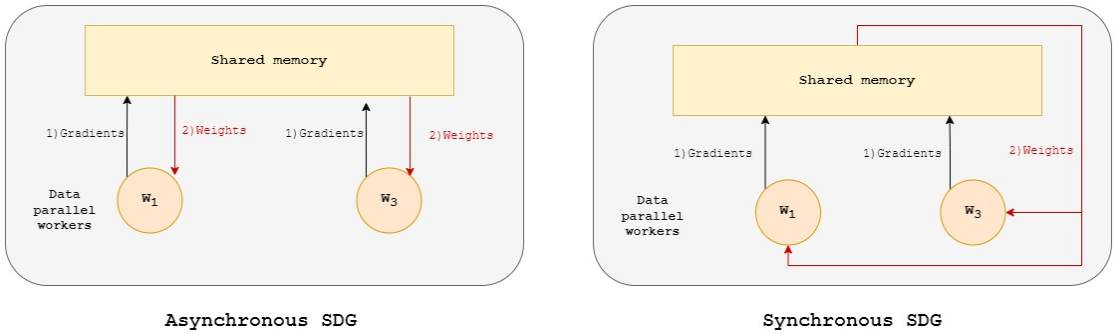


Figure 8. Asynchronous vs. Synchronous SGD in standard shared memory architecture.

Synchronous Parallel SGD

In a *shared parameter server* system, the local workers can compute the gradients over their mini-batch of data, and then add the gradients to the global model; this approach is commonly referred to as *Synchronous* Parallel SGD (SyncPSGD) due to its barrier-based nature. In its standard form, SyncPSGD has scalability issues due to the waiting time that is inherent in the aggregation when independent workers compute with different speed (Bäckström et al. (2019)).

Recent work in Goyal et al. (2017); You et al. (2017), explores the various limitation of this approach, as in general models trained with large mini-batches often do not generalise well. To overcome the communication overhead, local SGD has recently attracted increasing research interest Zhang et al. (2016); McMahan et al. (2017); Yu et al. (2019); Stich (2019) where data is partitioned among computation nodes, and the nodes compute local updates with periodically exchanging the model among the workers to perform averaging. To address the current generalisation issue of large-batch training and improve the scalability of SyncPSGD, in Lin et al. (2020) the authors proposed Post-Local SGD, where the classic large mini-batch SGD is followed by a local SGD phase which allows workers to independently update their models for a few steps before synchronising. Moreover, (Gupta et al. (2020)) introduces a variant where they adapt Stochastic Weight Averaging Dean et al. (2012) to accelerate DNNs training. In the second phase, each worker refines its network separately, and at the end, they average the weights of the resulting models to produce the final result.

Asynchronous Parallel SGD

An alternative type of parallelisation is Asynchronous Parallel SGD (AsyncPSGD) Dean et al. (2012), in which workers fetch and update the shared model independently of each other. Hence, the training procedure has no barrier imposed. Notably, for sparse problems, Hogwild method (Recht et al. (2011)) shows that updating only the relevant parameters without any synchronisation could guarantee a nearly linear speedup with the number of processors (Recht et al. (2011)). Although AsyncPSGD can achieve faster speed due to the absence of waiting overhead, the lack of coordination implies that gradients may be computed on *stale* (old) version of the weights, which leads to statistical inefficiency. This problem is well-known, and some researchers have analysed its negative effect on the convergence speed Avron et al. (2014); Lian et al. (2015). Another critical factor to monitor when considering different scales of asynchrony is that it introduces momentum to the SGD update, called the *implicit momentum* (Mitliagkas et al. (2016)).

Recent studies in Lan & Zhou (2018); Bäckström et al. (2019); Zheng et al. (2020) proposes different staleness-adaptive SGD algorithms to reduce the negative impact of asynchrony and approach the performance of sequential SGD. Moreover, they allow for fine-tuning the implicit *momentum* and increase the number of workers

while maintaining statistical efficiency.

Parallel Training of Sparse Networks

To reduce the communication overhead in parallel distributed DNN training, various quantisation techniques and sparse gradient updates have been developed Wangni et al. (2018); Alistarh et al. (2017); Stich et al. (2018). In this regard, one significant advantage of sparse models is that the sparse gradient communication is automatically at hand. Related work on parallelisation for sparse DNN is presented in (Sattar & Anfuzzaman (2020)) as a solution to the Sparse DNN Challenge posed by MIT/IEEE/Amazon (Kepner et al. (2019)). However, their work is focused on sparse neural networks created using RadiX-Net (Kepner & Robinett (2019)) and trained on GPU which do not evolve the topology over time. Moreover, their solution is implemented in Tensorflow, where sparse layers are represented by dense matrices with a mask over weights. Similarly, in (Mohammad Hasanzadeh Mofrad & Hammoud (2021)), the authors devised a parallel strategy for large sparse neural networks (up to 125 millions of neurons), but even if they employ truly sparse matrices their approach is focused on the inference phase only. Further related work for sparse neural network inference is presented in (Hidayetoğlu et al. (2020); Pawłowski et al. (2020); Hasanzadeh-Mofrad et al. (2020)). In short, none of the available parallel training algorithms is designed for training truly sparse dynamic neural networks.

Activation Functions

Activation functions determine the output of a deep learning model, its accuracy, and also the computational efficiency, which can make or break a large scale neural network. They are essential for an artificial neural network to help the model learn complex non-linear patterns in the data.

ReLU

Rectified Linear Unit (ReLU) is one of the most widely used activation functions in neural networks (Glorot et al. (2011); Nair & Hinton (2010)), which can effectively solve the problem of vanishing gradient (small gradient prevents the weight from altering its value) and slow training time of saturated activation function. Its mathematical expression is:

$$f(x_i) = \begin{cases} 0 & x_i \leq 0 \\ x_i & x_i > 0 \end{cases} \tag{5}$$

and its derivatives can be easily calculated with a meagre computational cost; this is a desirable advantage for choosing ReLU to speed up the training. When the input value x_i is lower than zero, the resulting derivative will also be zero, leading to a disconnection of the neuron (zero-sparsity). Disconnecting some neurons may reduce overfitting; however, this will hinder the neural network from learning in some cases due to the neurons death problem. The ReLU function also keeps the mean activation value to be greater than zero, which makes it difficult for the network to determine the direction with the fastest gradient drop in the backpropagation process, thus affecting the network convergence. This "free" sparsity (in terms of neuron activations) obtained by adopting ReLU may represent an advantage for training fully-connected layers. However, this might be untrue for sparsely connected neurons since, in these settings, the fact they are not capable of capturing some significant aspect of the data on the negative side and the dead neurons problem could lead to a higher impact on performance overall.

Leaky ReLU

Leaky ReLU (LReLU) is a modification of ReLU which replaces the zero part of the domain in $[-\inf, 0]$ by a low slope α. The reason for using LReLU instead of ReLU is that constant zero gradients can also result in slow learning, as when a saturated neuron uses a sigmoid activation function. Additionally, others do not even activate. According to the authors in (Maas (2013)), this sacrifice of the zero-sparsity might bring worse results than when the neurons are entirely deactivated, suggesting the leaky rectifiers' non-zero gradient does not

substantially impact training optimisation.

PReLU

Parametric Rectified Linear Unit (PReLU) is proposed by (He et al. (2015)) and generalizes the traditional rectified unit. The authors reported that its performance was considerably better than ReLU in large-scale image classification tasks. It is the same as Leaky ReLU with the exception that the slope parameter is learned during training via backpropagation.

Activation Functions in Sparse Networks

It is crucial to question weather the activation functions currently used for densely connected networks still behave reliably in the sparse context. S-shaped rectified linear activation unit (SReLU) is a relatively little-known activation function suggested in (Jin et al. (2016)) and used by Mocanu et al. (Mocanu et al. (2018)) in their implementation of the SET algorithm. When compared with the most common activation functions, SReLU in SET models has shown to perform best at all sparsity levels (Dubowski (2020)) in various domain.

Very recently, in (ab Tessera et al. (2021)) the authors have shown that SReLU and PReLU are more suitable activation function for sparse networks as they improve the networks gradient flow and achieve better accuracy when compared to the other activations. In this regard, they proposed a normalised measure of gradient flow called *Effective Gradient Flow* (EGF), which is better suited to examining the training dynamics of sparse networks. Their results related to activation functions are in line with our findings. Previous papers were already suggesting that low gradient flow is an exacerbated issue in sparse networks Wang et al. (2020); Evci et al. (2020), but they did not investigate its relation with activation functions.

Neuron Importance

The importance of hidden units in neural networks is still an open problem, crucial to understand neural networks' behaviour and to enhance the explainability of these black-box models. Many papers on dense deep networks speculate about the significance of a neuron towards a prediction. They tend to use the activation value of the hidden unit or its product with the gradient as a proxy for feature importance (e.g. Zagoruyko & Komodakis (2017)); however, both metrics can have undesirable outcomes. To overcome these problems, in (Dhamdhere et al. (2019)), the authors proposed the notion of *conductance* to gain a better understanding of neuron relevance through extensive ablation studies.

We argue that a neuron importance metric can be straightforwardly identified in sparse neural networks trained with a sparse training algorithm where the topology evolves overtime to find the best weight configuration. The proposed metric is shown to be valuable since it can remove a big chunk of unimportant units and related connections with almost no loss in accuracy. More importantly, this metric can be simultaneously derived for all neurons without requiring expensive computations. To define the importance of a neuron, we borrow some terminology and ideas from graph theory and hence, we introduce them here. In network science, a hub is a high-degree node that occupies a central role in the overall organisation of a network. Hubs have a significantly larger number of links in comparison with other nodes in the network (Barabási & Pósfai (2016)). They can be found in many real networks, such as the brain (van den Heuvel & Sporns (2013)) or the Internet. The loss of such well-connected hubs can be extremely devastating to network function. Given the role of hubs and their significance to networks, their locations and functions in the brain are of clear interest to neuroscientists. Accordingly, we would expect to find a similar biological structure in sparse ANNs as well.

Data availability

The data used in this paper are public datasets, freely available online, as reflected by their corresponding citations from Table 1.

Code availability

Prototype software implementations of the models used in this study are freely available online.

Acknowledgement

We thank the Google Cloud Platform Research Credits program for granting us the necessary resources to run the Extreme large sparse MLPs experiments.

Competing interests

The authors declare no competing interests.

Algorithm 1: WASAP-SGD

Result: Trained sparse model θ_s^f
Input: Number of workers K, $t = 0$, $t' = 0$, $phase = 1$, $epoch = 0$, sparsity level $\mathcal{S}$
Step size η and momentum μ
Training dataset $\mathcal{D}$ wit labels $\mathcal{I}$; Mini-batch size $\mathcal{B}$
SGDSparseUpdate(·), a function that updates the weights using momentum SGD
TopologyEvolutionStep(·), a function that updates the sparse topology
RetainValidUpdates(·), a function that retain only gradients applicable to the topology defined by θ_s^t
Epoch τ_1 and τ_2, at which to exit phase one and phase two respectively

Phase 1:
Worker k in [1,...,K]
/* Each worker shuffle its data partition $\mathcal{I}_t^{(k)}$ after each local epoch */
while *phase == 1* **do**
  Sample a mini-batch B^k from $\mathcal{I}_t^{(k)}$
  Calculate worker gradient: $\nabla w_t^{(k)} = \frac{1}{\mathcal{B}} \sum_{i \in \mathcal{B}^k} \nabla w^i$
  Send gradients $\nabla w_t^{(k)}$ and time step t to PS
  Receive updated model from PS and time step t'
  Update time step: $t = t'$
end
Parameter server PS
while $epoch \leq \tau_1$ **do**
  Receive gradients $\nabla w_t^{(k)}$ and time step t from a ready worker k
  Retain valid updates: g = RetainValidUpdates($\nabla w_t^{(k)}, \theta_s^{t'}$)
  Update model: $\theta_s^{t'+1}$ = $\theta_s^{t'}$ + SGDSparseUpdate(g, η, μ)
  if $t'\%(n \div \mathcal{B}) == 0$ **then**
    Update sparse topology: $\theta_s^{t'+1}$ = TopologyEvolutionStep($\theta_s^{t'}$)
    Update epoch: $epoch = epoch + 1$
  end
  Send updated model $\theta_s^{t'+1}$ and time step t' to worker k
  Update time step: $t' = t' + 1$
end
Switch phase: $phase = 2$

Phase 2: ;
/* Local training of K sparse models that evolve their topology separately */
while $epoch \leq \tau_2$ **do**
  Each worker shuffles its data partition $\mathcal{I}_t^{(k)}$
  for *k in [1,...,K] in parallel* **do**
    Sample a mini-batch B^k from $\mathcal{I}_t^{(k)}$
    Calculate worker gradient: $\nabla w_t^{(k)} = \frac{1}{B} \sum_{i \in B^k} \nabla w^i$
    Update model: θ_s^{t+1} = θ_s^t + SGDSparseUpdate($\nabla w_t^{(k)}, \eta, \mu$)
    Update sparse topology: θ_s^{t+1} = TopologyEvolutionStep(θ_s^t)
  end
  Update time step: t=t+1; epoch=epoch+1
end
We get K different models at the end of phase 2
Produce averaged model θ_s^f and select a fraction $\mathcal{S}$ of weights with bigger magnitude for each layer

Algorithm 2: Sparse evolutionary training (SET) with Importance Pruning

Result: Trained sparse model
Input: An ANN model with L layers
Weight θ_s, sparsity $\mathcal{S}$
pruning rate ζ, pruning step p, starting pruning epoch τ, and threshold t

```
% Sparse initialization;
for each fully-connected (FC) layer l do
    replace l with a Sparse Connected Layer having a Erdős-Rényi topology
end
% Training;
for each training epoch e do
    Perform standard training procedure;
    Perform weights update;
    if e % p == 0 and e ≥ τ then
        % Perform Importance Pruning;
        for each SC layer of the ANN do
            Calculate importance (I) per each neuron;
            Remove incoming weights of neurons where I < t
        end
    end
    % Weight pruning-regrowing cycle;
    for each SC layer of the ANN do
        Remove a fraction ζ of the smallest positive weights;
        Remove a fraction ζ of the largest negative weights;
        Add randomly new weights in the equivalent amount as the one removed previously;
    end
end
```

References

Kale ab Tessera, Sara Hooker, and Benjamin Rosman. Keep the gradients flowing: Using gradient flow to study sparse network optimization. *ArXiv*, abs/2102.01670, 2021.

Abien Fred Agarap. Deep learning using rectified linear units (relu). *ArXiv*, abs/1803.08375, 2018.

Simon Alford, Ryan A. Robinett, Lauren Milechin, and J. Kepner. Training behavior of sparse neural network topologies. *2019 IEEE High Performance Extreme Computing Conference (HPEC)*, pp. 1–6, 2019.

Dan Alistarh, Demjan Grubic, Jerry Li, Ryota Tomioka, and M. Vojnovic. Qsgd: Communication-efficient sgd via gradient quantization and encoding. In *NIPS*, 2017.

Dustin Anderson, Jean-Roch Vlimant, and Maria Spiropulu. An mpi-based python framework for distributed training with keras, 2017.

Alex M. Ascension and Marcos J. Araúzo-Bravo. Bigmpi4py: Python module for parallelization of big data objects. *bioRxiv*, 2020. doi: 10.1101/517441. URL https://www.biorxiv.org/content/early/2020/03/18/517441.

H. Avron, Alex Druinsky, and A. Gupta. Revisiting asynchronous linear solvers: Provable convergence rate through randomization. In *IPDPS*, 2014.

Karl Bäckström, M. Papatriantafilou, and P. Tsigas. Mindthestep-asyncpsgd: Adaptive asynchronous parallel stochastic gradient descent. *2019 IEEE International Conference on Big Data (Big Data)*, pp. 16–25, 2019.

P. Baldi, P. Sadowski, and D. Whiteson. Searching for exotic particles in high-energy physics with deep learning. *Nature Communications*, 5(1), Jul 2014. ISSN 2041-1723. doi: 10.1038/ncomms5308. URL http://dx.doi.org/10.1038/ncomms5308.

Albert-László Barabási and Márton Pósfai. *Network science*. Cambridge University Press, Cambridge, 2016. ISBN 9781107076266 1107076269. URL http://barabasi.com/networksciencebook/.

Alain Barrat, Marc Barthelemy, Romualdo Pastor-Satorras, and Alessandro Vespignani. The architecture of complex weighted networks. *Proceedings of the National Academy of Sciences of the United States of America*, 101:3747–52, 04 2004. doi: 10.1073/pnas.0400087101.

Guillaume Bellec, David Kappel, Wolfgang Maass, and Robert A. Legenstein. Deep rewiring: Training very sparse deep networks. *ICLR*, abs/1711.05136, 2018.

Tal Ben-Nun and Torsten Hoefler. Demystifying parallel and distributed deep learning: An in-depth concurrency analysis. *ACM Comput. Surv.*, 52:65:1–65:43, 2018.

Léon Bottou. Large-scale machine learning with stochastic gradient descent. *Proc. of COMPSTAT*, 01 2010. doi: 10.1007/978-3-7908-2604-3_16.

David Bourgin, Joshua C. Peterson, Daniel Reichman, Thomas L. Griffiths, and Stuart J. Russell. Cognitive model priors for predicting human decisions. In *ICML*, 2019.

Jeffrey Dean, Greg S. Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Quoc V. Le, Mark Z. Mao, Marc'Aurelio Ranzato, Andrew Senior, Paul Tucker, Ke Yang, and Andrew Y. Ng. Large scale distributed deep networks. pp. 1223–1231, 2012.

Misha Denil, Babak Shakibi, Laurent Dinh, Marc'Aurelio Ranzato, and Nando de Freitas. Predicting parameters in deep learning, 2013.

Tim Dettmers and Luke Zettlemoyer. Sparse networks from scratch: Faster training without losing performance. *ArXiv*, abs/1907.04840, 2019.

K. Dhamdhere, M. Sundararajan, and Qiqi Yan. How important is a neuron?, 2019.

A. Dubowski. Activation function impact on sparse neural networks. *ArXiv*, abs/2010.05943, 2020.

Ron Edgar and Alex Lash. 6. the gene expression omnibus (geo): A gene expression and hybridization repository. 01 2002.

P Erdös and A Rényi. On random graphs i. *Publicationes Mathematicae Debrecen*, 6:290–297, 1959.

Utku Evci, Trevor Gale, Jacob Menick, Pablo Samuel Castro, and Erich Elsen. Rigging the lottery: Making all tickets winners. *ArXiv*, abs/1911.11134, 2019.

Utku Evci, Yani A. Ioannou, Cem Keskin, and Yann Dauphin. Gradient flow in sparse neural networks and how lottery tickets win. *ArXiv*, abs/2010.03533, 2020.

Jonathan Frankle and Michael Carbin. The lottery ticket hypothesis: Finding sparse, trainable neural networks, 2019.

Trevor Gale, Matei A. Zaharia, Cliff Young, and Erich Elsen. Sparse gpu kernels for deep learning. *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–14, 2020.

Xavier Glorot, Antoine Bordes, and Yoshua Bengio. Deep sparse rectifier neural networks. In *AISTATS*, 2011.

Ian Goodfellow, Yoshua Bengio, Aaron Courville, and Yoshua Bengio. *Deep learning*, volume 1. MIT Press, 2016.

Priya Goyal, Piotr Dollár, Ross Girshick, Pieter Noordhuis, Lukasz Wesolowski, Aapo Kyrola, Andrew Tulloch, Yangqing Jia, and Kaiming He. Accurate, large minibatch sgd: Training imagenet in 1 hour, 2017.

Vipul Gupta, Santiago Akle Serrano, and Dennis DeCoste. Stochastic weight averaging in parallel: Large-batch training that generalizes well, 2020.

I. Guyon. Design of experiments for the nips 2003 variable selection benchmark. 2003.

Isabelle Guyon, Steve Gunn, Asa Ben-Hur, and Gideon Dror. Result analysis of the nips 2003 feature selection challenge. In L. K. Saul, Y. Weiss, and L. Bottou (eds.), *Advances in Neural Information Processing Systems 17*, pp. 545–552. MIT Press, 2005. URL http://papers.nips.cc/paper/2728-result-analysis-of-the-nips-2003-feature-selection-challenge.pdf.

Song Han, Jeff Pool, John Tran, and William J. Dally. Learning both weights and connections for efficient neural networks, 2015.

Song Han, Huizi Mao, and W. Dally. Deep compression: Compressing deep neural network with pruning, trained quantization and huffman coding. *arXiv: Computer Vision and Pattern Recognition*, 2016.

M. Hasanzadeh-Mofrad, R. Melhem, M. Y. Ahmad, and Mohammad Hammoud. Studying the effects of hashing of sparse deep neural networks on data and model parallelisms. *2020 IEEE High Performance Extreme Computing Conference (HPEC)*, pp. 1–7, 2020.

Babak Hassibi, David G.Stork, and Gregory Wolff. Optimal brain surgeon and general network pruning. pp. 293 – 299 vol.1, 02 1993. ISBN 0-7803-0999-5. doi: 10.1109/ICNN.1993.298572.

Soufiane Hayou, A. Doucet, and J. Rousseau. On the impact of the activation function on deep neural networks training. In *ICML*, 2019.

Kaiming He, X. Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. *2015 IEEE International Conference on Computer Vision (ICCV)*, pp. 1026–1034, 2015.

M. Hidayetoğlu, C. Pearson, V. S. Mailthody, E. Ebrahimi, J. Xiong, R. Nagi, and W. m. Hwu. At-scale sparse deep neural network inference with efficient gpu implementation. In *2020 IEEE High Performance Extreme Computing Conference (HPEC)*, pp. 1–7, 2020. doi: 10.1109/HPEC43674.2020.9286206.

Pavel Izmailov, D. Podoprikhin, T. Garipov, D. Vetrov, and A. Wilson. Averaging weights leads to wider optima and better generalization. *ArXiv*, abs/1803.05407, 2018.

Siddhant Jayakumar, Razvan Pascanu, Jack Rae, Simon Osindero, and Erich Elsen. Top-kast: Top-k always sparse training. In *Advances in Neural Information Processing Systems, 33*, 2020. URL https://proceedings.neurips.cc/paper/2020/file/ee76626ee11ada502d5dbf1fb5aae4d2-Paper.pdf.

Jeff Pool. Accelerating sparsity in the nvidia ampere architecture, 2020. URL https://developer.download.nvidia.com/video/gputechconf/gtc/2020/presentations/s22085-accelerating-sparsity-in-the-nvidia-ampere-architecture%E2%80%8B.pdf.

X. Jin, Chunyan Xu, Jiashi Feng, Yunchao Wei, Junjun Xiong, and S. Yan. Deep learning with s-shaped rectified linear activation units. *ArXiv*, abs/1512.07030, 2016.

N. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, Raminder Bajwa, S. Bates, Suresh Bhatia, Nan Boden, Al Borchers, R. Boyle, P. Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, M. Daley, M. Dau, J. Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Ho, Doug Hogberg, John Hu, R. Hundt, D. Hurt, J. Ibarz, A. Jaffey, A. Jaworski, Alexander Kaplan, Harshit Khaitan, Daniel Killebrew, Andy Koch, N. Kumar, S. Lacy, J. Laudon, J. Law, Diemthu Le, C. Leary, Z. Liu, K. Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, K. Miller, R. Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, A. Phelps, J. Ross, Matt Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, J. Souter, Dan A. Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, B. Tian, H. Toma, Erick Tuttle, V. Vasudevan, R. Walter, Walter Wang, E. Wilcox, and D. H. Yoon. In-datacenter performance analysis of a tensor processing unit. *ACM SIGARCH Computer Architecture News*, 45:1 – 12, 2017.

J. Kepner, M. Kumar, J. Moreira, P. Pattnaik, M. Serrano, and H. Tufo. Enabling massive deep neural networks with the graphblas. *2017 IEEE High Performance Extreme Computing Conference (HPEC)*, pp. 1–10, 2017.

Jeremy Kepner and John Gilbert. *Graph Algorithms in the Language of Linear Algebra*. Society for Industrial and Applied Mathematics, 2011. doi: 10.1137/1.9780898719918. URL https://epubs.siam.org/doi/abs/10.1137/1.9780898719918.

Jeremy Kepner and Ryan Robinett. Radix-net: Structured sparse matrices for deep neural networks. *2019 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, May 2019. doi: 10.1109/ipdpsw.2019.00051. URL http://dx.doi.org/10.1109/IPDPSW.2019.00051.

Jeremy Kepner, Vikalo Gadepally, Hayden Jananthan, Lauren Milechin, and Sid Samsi. Sparse deep neural network exact solutions. *2018 IEEE High Performance extreme Computing Conference (HPEC)*, Sep 2018. doi: 10.1109/hpec.2018.8547742. URL http://dx.doi.org/10.1109/HPEC.2018.8547742.

Jeremy Kepner, Simon Alford, Vijay N. Gadepally, Michael Jones, Lauren Milechin, Ryan A. Robinett, and Siddharth Samsi. Sparse deep neural network graph challenge. *2019 IEEE High Performance Extreme Computing Conference (HPEC)*, pp. 1–7, 2019.

A. Krizhevsky. Learning multiple layers of features from tiny images. 2009.

M. Kumar, W. Horn, J. Kepner, J. Moreira, and P. Pattnaik. Ibm power9 and cognitive computing. *IBM J. Res. Dev.*, 62:10, 2018.

Aditya Kusupati, V. Ramanujan, Raghav Somani, Mitchell Wortsman, Prateek Jain, Sham M. Kakade, and Ali Farhadi. Soft threshold weight reparameterization for learnable sparsity. *ArXiv*, abs/2002.03231, 2020.

Siu Kwan Lam, Antoine Pitrou, and Stanley Seibert. Numba: A llvm-based python jit compiler. In *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*, LLVM '15, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450340052. doi: 10.1145/2833157.2833162. URL https://doi.org/10.1145/2833157.2833162.

G. Lan and Yi Zhou. Asynchronous decentralized accelerated stochastic gradient descent. *ArXiv*, abs/1809.09258, 2018.

Y. LeCun, J. Denker, and S. Solla. Optimal brain damage. In *NIPS*, 1989.

Yann LeCun, Y. Bengio, and Geoffrey Hinton. Deep learning. *Nature*, 521:436–44, 05 2015. doi: 10.1038/nature14539.

Namhoon Lee, Thalaiyasingam Ajanthan, and Philip H. S. Torr. Snip: Single-shot network pruning based on connection sensitivity, 2019.

Namhoon Lee, Thalaiyasingam Ajanthan, Stephen Gould, and Philip H. S. Torr. A signal propagation perspective for pruning neural networks at initialization, 2020.

X. Li, G. Zhang, K. Li, and W. Zheng. Chapter 4 - deep learning and its parallelization. In Rajkumar Buyya, Rodrigo N. Calheiros, and Amir Vahid Dastjerdi (eds.), *Big Data*, pp. 95 – 118. Morgan Kaufmann, 2016. ISBN 978-0-12-805394-2. doi: https://doi.org/10.1016/B978-0-12-805394-2.00004-0. URL http://www.sciencedirect.com/science/article/pii/B9780128053942000040.

Xiangru Lian, Yijun Huang, Y. Li, and J. Liu. Asynchronous parallel stochastic gradient for nonconvex optimization. In *NIPS*, 2015.

Tao Lin, S. Stich, and M. Jaggi. Don't use large mini-batches, use local sgd, 2020.

B. Liu, M. Wang, H. Foroosh, M. Tappen, and Marianna Pensky. Sparse convolutional neural networks. *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 806–814, 2015.

Junjie Liu, Zhe Xu, Runbin Shi, Ray C. C. Cheung, and Hayden K. H. So. Dynamic sparse training: Find efficient sparse network from scratch with trainable masked layers, 2020a.

S. Liu, D. Mocanu, Amarsagar Reddy Ramapuram Matavalam, Y. Pei, and M. Pechenizkiy. Sparse evolutionary deep learning with over one million artificial neurons on commodity hardware. *Neural Computing and Applications*, pp. 1–16, 2020b.

S. Liu, Tim van der Lee, A. Yaman, Zahra Atashgahi, Davide Ferraro, Ghada Sokar, M. Pechenizkiy, and D. Mocanu. Topological insights into sparse neural networks. volume abs/2006.14085, 2020c.

S. Liu, D. Mocanu, Yulong Pei, and Mykola Pechenizkiy. Selfish sparse rnn training. *ArXiv*, abs/2101.09048, 2021.

Christos Louizos, Max Welling, and Diederik P. Kingma. Learning sparse neural networks through l_0 regularization, 2018.

Andrew L. Maas. Rectifier nonlinearities improve neural network acoustic models. 2013.

H. McMahan, Eider Moore, D. Ramage, S. Hampson, and Blaise Agüera y Arcas. Communication-efficient learning of deep networks from decentralized data. In *AISTATS*, 2017.

Risto Miikkulainen. *Topology of a Neural Network*, pp. 988–989. Springer US, Boston, MA, 2010. ISBN 978-0-387-30164-8. doi: 10.1007/978-0-387-30164-8_837. URL https://doi.org/10.1007/978-0-387-30164-8_837.

Ioannis Mitliagkas, Ce Zhang, Stefan Hadjis, and C. Ré. Asynchrony begets momentum, with an application to deep learning. *2016 54th Annual Allerton Conference on Communication, Control, and Computing (Allerton)*, pp. 997–1004, 2016.

D.C. Mocanu. *Network computations in artificial intelligence*. PhD thesis, Department of Electrical Engineering, June 2017. Proefschrift.

Decebal Constantin Mocanu, Elena Mocanu, Phuong H. Nguyen, Madeleine Gibescu, and Antonio Liotta. A topological insight into restricted boltzmann machines. *Machine Learning*, 104(2):243–270, Sep 2016.

Decebal Constantin Mocanu, Elena Mocanu, Peter Stone, Phuong H. Nguyen, Madeleine Gibescu, and Antonio Liotta. Scalable training of artificial neural networks with adaptive sparse connectivity inspired by network science. *Nature Communications*, 9, Jun 2018. ISSN 2041-1723.

Yousuf Ahmad Mohammad Hasanzadeh Mofrad, Rami Melhem and Mohammad Hammoud. Accelerating distributed inference of sparse deep neural networks via mitigating the straggler effect. In *In proceedings of IEEE High Performance Extreme Computing (HPEC), Waltham, MA USA*, 2021. URL http://people.cs.pitt.edu/~moh18/files/papers/PID6571125.pdf. under review.

Hesham Mostafa and Xin Wang. Parameter efficient training of deep convolutional neural networks by dynamic sparse reparameterization. *ArXiv*, abs/1902.05967, 2019a.

Hesham Mostafa and Xin Wang. Parameter efficient training of deep convolutional neural networks by dynamic sparse reparameterization, 2019b.

Michael C. Mozer and Paul Smolensky. Using relevance to reduce network size automatically. *Connection Science*, 1(1):3–16, 1989. doi: 10.1080/09540098908915626. URL https://doi.org/10.1080/09540098908915626.

Vinod Nair and Geoffrey E. Hinton. Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th International Conference on International Conference on Machine Learning*, ICML'10, pp. 807–814, Madison, WI, USA, 2010. Omnipress. ISBN 9781605589077.

Sharan Narang, Erich Elsen, Gregory Diamos, and Shubho Sengupta. Exploring sparsity in recurrent neural networks, 2017.

F. Pawłowski, R. Bisseling, B. Uçar, and A. Yzelman. Combinatorial tiling for sparse neural networks. *2020 IEEE High Performance Extreme Computing Conference (HPEC)*, pp. 1–7, 2020.

Ameya Prabhu, G. Varma, and A. Namboodiri. Deep expander networks: Efficient deep networks from graph theory. In *ECCV*, 2018.

Md Aamir Raihan and Tor Aamodt. Sparse weight activation training. In H. Larochelle, M. Ranzato, R. Hadsell, M.F. Balcan, and H. Lin (eds.), *Advances in Neural Information Processing Systems*, volume 33, pp. 15625–15638. Curran Associates, Inc., 2020. URL https://proceedings.neurips.cc/paper/2020/file/b44182379bf9fae976e6ae5996e13cd8-Paper.pdf.

B. Recht, Christopher Ré, Stephen J. Wright, and Feng Niu. Hogwild: A lock-free approach to parallelizing stochastic gradient descent. In *NIPS*, 2011.

Herbert Robbins and Sutton Monro. A stochastic approximation method. *The Annals of Mathematical Statistics*, 22(3):400–407, 1951. ISSN 00034851. URL http://www.jstor.org/stable/2236626.

N. S. Sattar and Shaikh Anfuzzaman. Data parallel large sparse deep neural network on gpu. *2020 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pp. 1–9, 2020.

Suraj Srinivas and R. Venkatesh Babu. Data-free parameter pruning for deep neural networks. In *BMVC*, 2015.

Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *J. Mach. Learn. Res.*, 15(1):1929–1958, January 2014. ISSN 1532-4435.

S. Stich, Jean-Baptiste Cordonnier, and M. Jaggi. Sparsified sgd with memory. In *NeurIPS*, 2018.

Sebastian U. Stich. Local sgd converges fast and communicates little, 2019.

Martijn P. van den Heuvel and Olaf Sporns. Network hubs in the human brain. pp. 683–696, 2013. URL https://doi.org/10.1016/j.tics.2013.09.012.

Chaoqi Wang, Guodong Zhang, and Roger Grosse. Picking winning tickets before training by preserving gradient flow, 2020.

Jianqiao Wangni, Jialei Wang, J. Liu, and Tong Zhang. Gradient sparsification for communication-efficient distributed optimization, 2018.

Wei Wen, Yuxiong He, Samyam Rajbhandari, Minjia Zhang, Wenhan Wang, Fang Liu, Bin Hu, Yiran Chen, and Hai Li. Learning intrinsic sparse structures within long short-term memory, 2018.

Han Xiao, Kashif Rasul, and Roland Vollgraf. Fashion-mnist: a novel image dataset for benchmarking machine learning algorithms, 2017.

Yang You, Igor Gitman, and Boris Ginsburg. Scaling sgd batch size to 32k for imagenet training. 08 2017.

Hao Yu, S. Yang, and Shenghuo Zhu. Parallel restarted sgd with faster convergence and less communication: Demystifying why model averaging works for deep learning. In *AAAI*, 2019.

Xin Yuan, Pedro H. P. Savarese, and Michael Maire. Growing efficient deep networks by structured continuous sparsification. *ArXiv*, abs/2007.15353, 2021.

Sergey Zagoruyko and Nikos Komodakis. Paying more attention to attention: Improving the performance of convolutional neural networks via attention transfer, 2017.

C. Zhang, S. Bengio, M. Hardt, B. Recht, and Oriol Vinyals. Understanding deep learning requires rethinking generalization, 2017.

Jian Zhang, Christopher De Sa, Ioannis Mitliagkas, and Christopher Ré. Parallel sgd: When does averaging help?, 2016.

Shuxin Zheng, Qi Meng, Taifeng Wang, Wei Chen, Nenghai Yu, Zhi-Ming Ma, and Tie-Yan Liu. Asynchronous stochastic gradient descent with delay compensation, 2020.

Hattie Zhou, Janice Lan, Rosanne Liu, and J. Yosinski. Deconstructing lottery tickets: Zeros, signs, and the supermask. In *NeurIPS*, 2019.

Michael Zhu and Suyog Gupta. To prune, or not to prune: exploring the efficacy of pruning for model compression, 2017.

5 Supplementary Information

5.1 From SReLU to All-ReLU

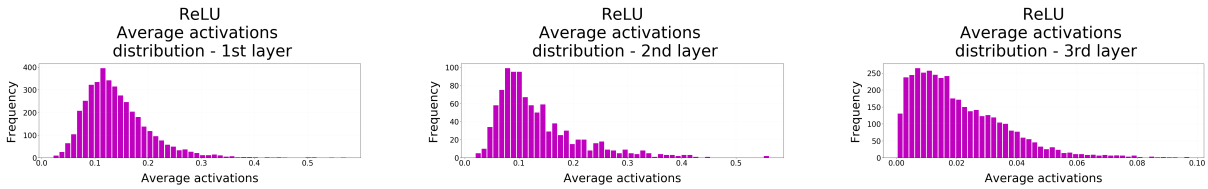


Figure 9. Average activations distribution for a 3-layers sparse MLP on CIFAR10 trained with ReLU and *momentum* SGD after 1000 epochs.

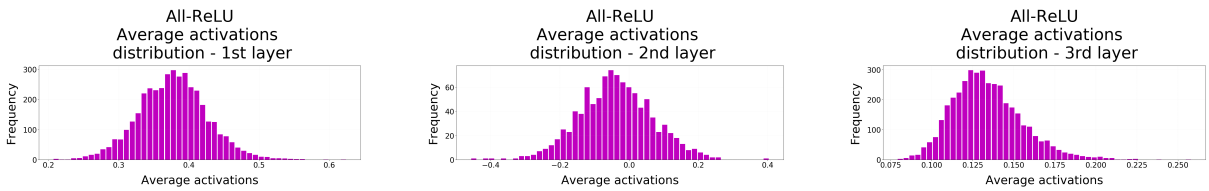


Figure 10. Average activations distribution for a 3-layers sparse MLP on CIFAR10 trained with All-ReLU and *momentum* SGD after 1000 epochs.

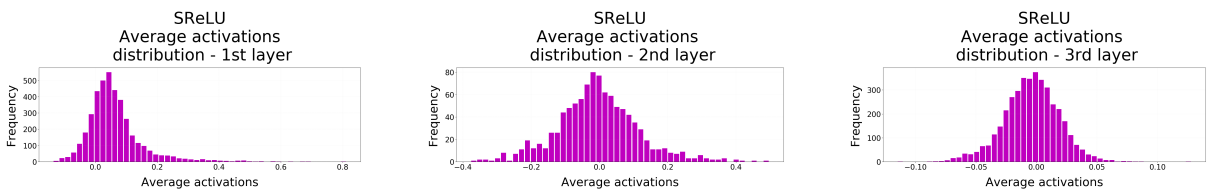


Figure 11. Average activations distribution for a 3-layers sparse MLP on CIFAR10 trained with SReLU and *momentum* SGD after 1000 epochs.

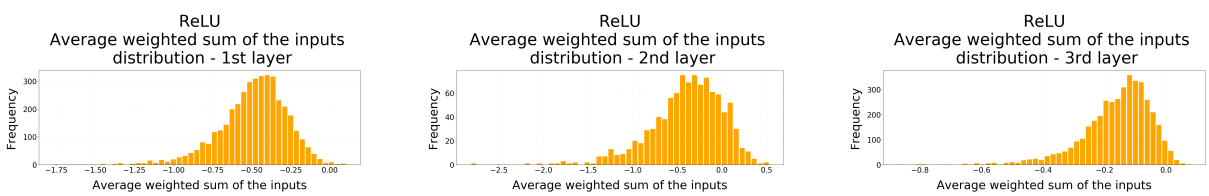


Figure 12. Average weighted sum of the inputs distribution for a 3-layers sparse MLP on CIFAR10 trained with ReLU and *momentum* SGD after 1000 epochs.

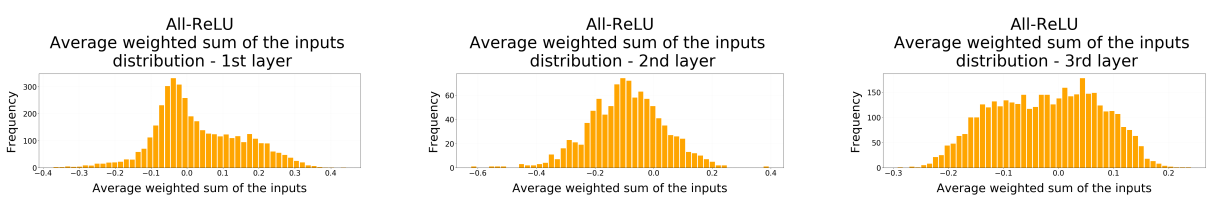


Figure 13. Average weighted sum of the inputs distribution for a 3-layers sparse MLP on CIFAR10 trained with All-ReLU and *momentum* SGD after 1000 epochs.

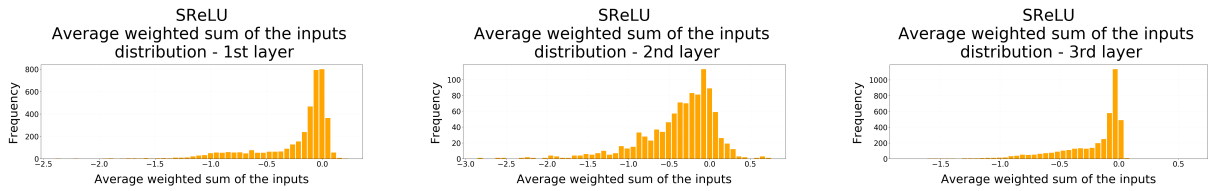


Figure 14. Average weighted sum of the inputs distribution for a 3-layers sparse MLP on CIFAR10 trained with SReLU and *momentum* SGD after 1000 epochs.

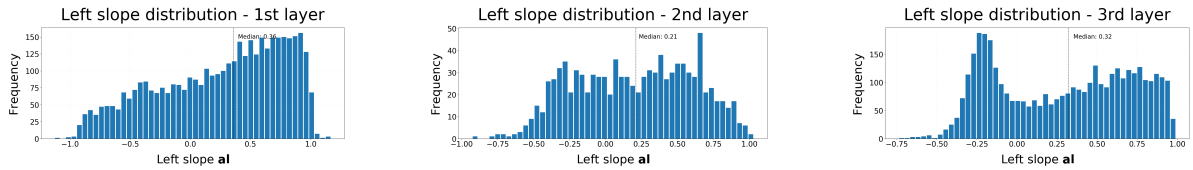


Figure 15. Left slopw a_l distribution for a 3-layers sparse MLP on CIFAR10 trained with SReLU and *momentum* SGD after 1000 epochs.

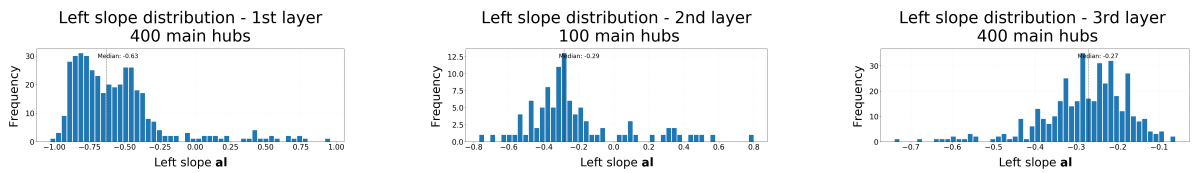


Figure 16. Left slopw a_l distribution for a 3-layers sparse MLP on CIFAR10 trained with SReLU and *momentum* SGD after 1000 epochs.

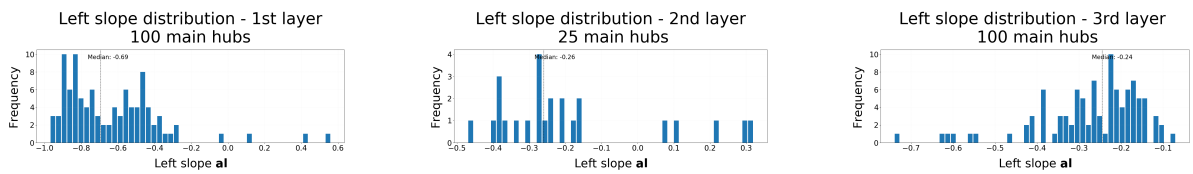


Figure 17. Left slopw a_l distribution for a 3-layers sparse MLP on CIFAR10 trained with SReLU and *momentum* SGD after 1000 epochs.

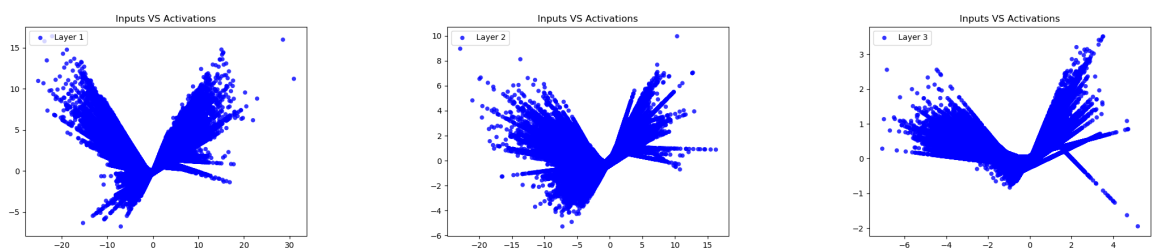


Figure 18. Input vs Activations for a 3-layers sparse MLP on CIFAR10 trained with SReLU and *momentum* SGD after 1000 epochs.

5.2 Finding the Best Slope for All-ReLU

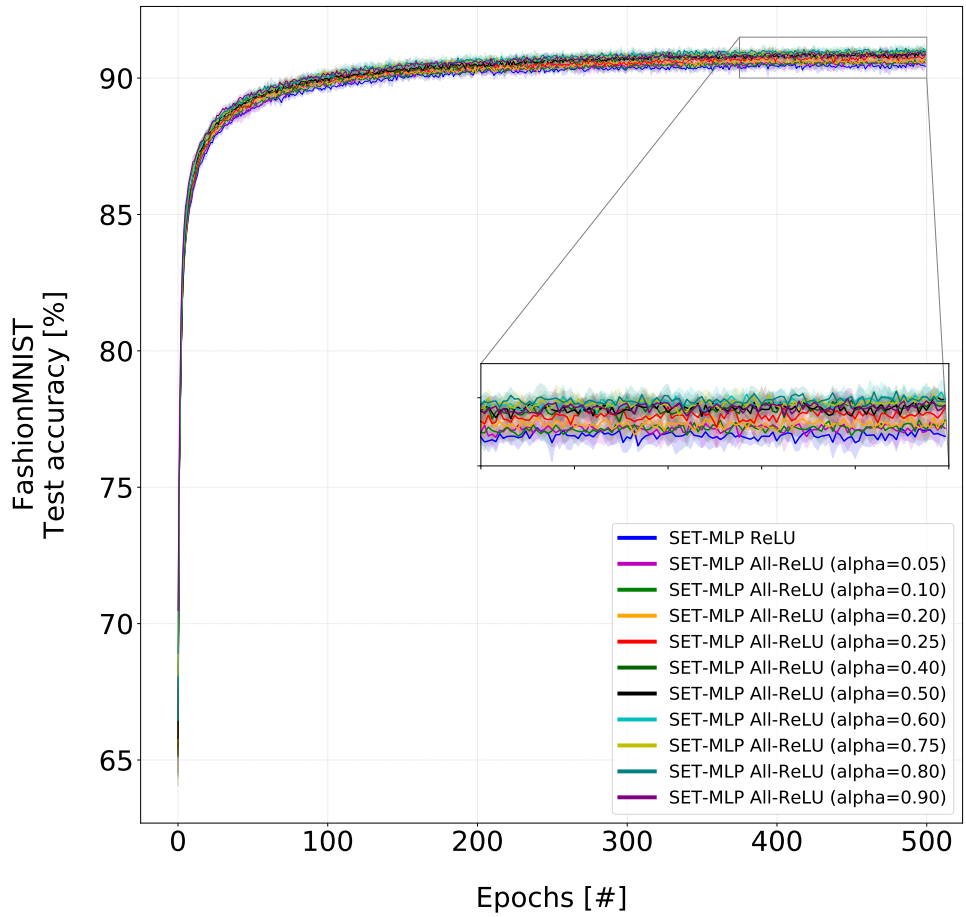


Figure 19. Grid Search for slope α on FashionMNIST for the proposed All-ReLU activation function. The plot reports the accuracy on the test dataset when training the various SET MLPs models for 500 epochs. Each run is repeated five times and averaged to visualize the learning curves.

In this appendix, we illustrate the tuning of the hyperparameter α required by the proposed All-ReLU function when considering the FashionMNIST dataset. In Figure 19, we report the learning curve for different values of the slope, averaged over five runs, while in Table 5 are shown the best results in term of accuracy. The best value of α for training a SET model on FashionMNIST is 0.6. To avoid the expensive grid search, we want to propose a practical method to narrow down the choices of this parameter. Based on our intuition, we propose to pick the slope α based on the observed skewness of the input data distribution. When the input is skewed on the left side, the chosen slope should reflect the level of distortion or asymmetry in the data distribution.

	$\alpha = 0$	$\alpha = 0.5$	$\alpha = 0.1$	$\alpha = 0.2$	$\alpha = 0.25$	$\alpha = 0.5$	$\alpha = 0.6$	$\alpha = 0.75$	$\alpha = 0.8$	$\alpha = 0.9$
Best accuracy [%]	90.48	90.86	90.87	90.75	91	91.08	91.38	91.09	91.16	91.02

Table 5. The table reports the best accuracy obtained by the sparse models on the test dataset when adopting All-ReLU for different values of α.

It is essential to notice that any alpha level greater than 0.05 leads to better results than ReLU. Hence, even with suboptimal values is possible to attain satisfactory results.

5.3 Importance Pruning Post-Training

The proposed method *Importance Pruning* based on our neuron importance metric can be easily applied one time only, once the sparse training procedure is concluded. With the experiments reported in Table 6, we empirically demonstrate that the **Importance Pruning** technique should be integrated during training to gain the best results in terms of the memory footprint, running time and accuracy.

Dataset	Model Accuracy [%]	Parameters [#]	Pruning threshold t	Results	
				Accuracy [%]	end_n^W [#]
Leukemia	86.42	1684944	**5**th percentile	**86.89**	1074251
			10th percentile	86.69	1034977
			15th percentile	85.84	992711
			20th percentile	85.84	947981
			25th percentile	85.69	901289
Higgs	73.67	50224	**5**th percentile	73.00	48508
			10th percentile	**73.16**	46693
			15th percentile	72.61	44897
			20th percentile	72.27	42601
			25th percentile	71.51	40491
Madelon	71.33	19006	**5**th percentile	71.33	18385
			10th percentile	**72.00**	17661
			15th percentile	70.33	16883
			20th percentile	69.16	16106
			25th percentile	65.33	15297
FashionMNIST	91.38	125901	**5**th percentile	**90.66**	120894
			10th percentile	90.12	115486
			15th percentile	89.37	109923
			20th percentile	88.7	104158
			25th percentile	85.97	98360
CIFAR10	69.83	381758	**5**th percentile	**69.53**	368062
			10th percentile	68.61	352791
			15th percentile	68.38	336968
			20th percentile	67.35	320316
			25th percentile	66.57	276349

Table 6. On each dataset, we report the classification accuracy obtained by each model on the test data when the pruning method based on neuron importance is applied solely once at the end of the training procedure. The models employed for this evaluation are the resulting sparse MLPs from Table 2 trained with All-ReLU and no pruning. *Parameters [#]* and *Model Accuracy [%]* represent, respectively, the number of weights in the model and the final accuracy at the end of the training, while end_n^W represents the number of parameters in the pruned model.

Here, the pruning threshold t is used to remove all neurons and related connection which have an importance value lower than t. The post-pruning is carried out multiple times for different values of t (pruning threshold) to observe the loss in accuracy as the threshold increases. If we compare the results with the one obtained in Table 2 when importance pruning is employed while training, we can notice that we are not able to eliminate a significant number of parameters without reflecting a remarkable drop in accuracy. Hence, the *Importance Pruning* technique has shown to have more advantages if integrated during training.

5.4 Experiments setup

Datasets

We evaluate and discuss the performance of our proposed methods on sparse MLP models by considering five publicly available datasets listed in Table 1. We report both the number of data points used as training data and as testing data. The *Leukemia* dataset is obtained from the NCBI GEO repository (Edgar & Lash (2002)) with the accession number GSE13159. In (Liu et al. (2020b)), the authors provide a table with class labels (unbalanced) and their corresponding number of test samples. *HIGGS* (Baldi et al. (2014)) dataset is a classification problem to distinguish between a signal process which produces Higgs bosons and a background process which does not. The data has been produced using Monte Carlo simulations. Due to the limited resources available, we merely considered a fraction of the full dataset. *Madelon* (Guyon et al. (2005)) is an artificial dataset that has five informative features and 15 linear combinations of those features. The other 480 features are probes that provide no information about the class label. Lastly, FashionMNIST (Xiao et al. (2017)) and CIFAR10 (Krizhevsky (2009)) are both image datasets.

Implementation details

The following environment has been selected to implement the parallel algorithm WASAP-SGD, based on the implementation provided by Anderson et al. (2017) and Liu et al. (2020b):

Language: Pure Python 3.7 for quick prototyping, where *SciPy* and *Numpy* are employed for sparse matrix operations while *Numba* accelerates some critical part of the code such as backpropagation.

Framework: Message Passing Interface (MPI) standard.

Library: `mpi4py`, the library is constructed on top of the MPI-1/2 specifications and provides an object-oriented interface which directly follows MPI-2 C++ bindings.

All the experiments performed in this chapter are executed on a typical laptop with the following configuration:

- Hardware configuration: CPU Intel Core i7-9750H, 2.60 GHz $\times$ 6, RAM 32 GB, Hard disk 1000 GB, NVIDIA GeForce GTX 1650 4GB.
- Software used: Windows 10, Python 3.7, Numpy 1.19.1, SciPy 1.4.1, and Numba 0.48.0.

Experiment	**Dataset**	**Hyper-parameters**				
		ϵ	η	$\mathcal{B}$	**Weight initialization**	α
MLPs	Leukemia	10	0.005	5	*normal*	0.75
	Higgs	10	0.01	128	*xavier*	0.05
	Madelon	10	0.01	32	*normal*	0.5
	FashionMNIST	20	0.01	128	*he uniform*	0.6
	CIFAR10	20	0.01	128	*he uniform*	0.75

Table 7. List of hyperparameters used for the experiments.

Experiments hyperparameters

We primarily used the same configuration of parameters as in the original SET paper (Mocanu et al. (2018)). Like SET, our methods were also tested on a multilayer perceptron, in which the fully connected layers have been replaced with sparse layers. The MLP models in subsection 2.2 are trained with sequential momentum SGD (momentum is set to 0.9), weight decay, a dropout rate of 0.3 and fixed learning schedule. In Table 7 we provide an overview of the primary hyperparameters. The slope α for All-ReLU has been mostly identified via grid search. Similarly, the epoch τ, that determines the starting point of Importance Pruning is determined based on a local search within a limited set of value, and it is set to 200 for all models. The fraction ζ of the smallest positive weights and the largest negative weights to be removed is always set to 0.3. Here, ϵ controls the sparsity level as discussed in Mocanu et al. (2018), η is the learning rate and $\mathcal{B}$ the batch size. Furthermore, to improve the

learning process of our networks, we standardise the features of our datasets such that each attribute has zero mean and unit variance. The models in subsection 2.3 employ the same hyperparameters configurations, but the training procedure is parallelised with WASAP-SGD.