



Sparse evolutionary deep learning with over one million artificial neurons on commodity hardware

Shiwei Liu¹ · Decebal Constantin Mocanu^{1,3} · Amarsagar Reddy Ramapuram Matavalam² · Yulong Pei¹ · Mykola Pechenizkiy¹

Received: 11 February 2020 / Accepted: 16 June 2020 / Published online: 6 July 2020
© The Author(s) 2020

Abstract

Artificial neural networks (ANNs) have emerged as hot topics in the research community. Despite the success of ANNs, it is challenging to train and deploy modern ANNs on commodity hardware due to the ever-increasing model size and the unprecedented growth in the data volumes. Particularly for microarray data, the very high dimensionality and the small number of samples make it difficult for machine learning techniques to handle. Furthermore, specialized hardware such as graphics processing unit (GPU) is expensive. Sparse neural networks are the leading approaches to address these challenges. However, off-the-shelf sparsity-inducing techniques either operate from a pretrained model or enforce the sparse structure via binary masks. The training efficiency of sparse neural networks cannot be obtained practically. In this paper, we introduce a technique allowing us to train truly sparse neural networks with fixed parameter count throughout training. Our experimental results demonstrate that our method can be applied directly to handle high-dimensional data, while achieving higher accuracy than the traditional two-phase approaches. Moreover, we have been able to create truly sparse multilayer perceptron models with over one million neurons and to train them on a typical laptop without GPU (<https://github.com/dcmocanu/sparse-evolutionary-artificial-neural-networks/tree/master/SET-MLP-Sparse-Python-Data-Structures>), this being way beyond what is possible with any state-of-the-art technique.

Keywords Truly sparse neural networks · Sparse evolutionary training (SET) · Microarray gene expression · Adaptive sparse connectivity

✉ Shiwei Liu
s.liu3@tue.nl

Decebal Constantin Mocanu
d.c.mocanu@utwente.nl

Amarsagar Reddy Ramapuram Matavalam
amar@iastate.edu

Yulong Pei
y.pei.1@tue.nl

Mykola Pechenizkiy
m.pechenizkiy@tue.nl

¹ Department of Mathematics and Computer Science, Eindhoven University of Technology, 5600 MB Eindhoven, The Netherlands

² Department of Electrical and Computer Engineering, Iowa State University, Ames, USA

³ Faculty of Electrical Engineering, Mathematics and Computer Science, University of Twente, Enschede 7522NB, The Netherlands

1 Introduction

In the past decades, artificial neural networks (ANNs) have become an active area of current research due to state-of-the-art performance they have achieved in a variety of domains, including image recognition, text classification, and speech recognition. The powerful hardware, like graphics processing unit (GPU), as well as the increasing growth of data volumes, accelerates the advances of ANNs significantly. Recently, some works [3, 21] show that increasing the model capacity beyond a particular threshold yields better generalization. However, GPU is expensive and the explosive increase of model size leads to prohibitive memory requirements. Thus, the required resources to train and employ the modern ANNs are at odds with commodity hardware where the resources are very limited.

Motivated by these challenges, sparse neural networks [9, 12] have been introduced to effectively reduce the

memory requirements to deploy ANN models. After that, various techniques have emerged to obtain sparse neural networks, including but not limited to pruning [27, 33, 48, 57], L_0 and L_1 regularization [45, 70], variational dropout [54], and soft weight-sharing [68]. While achieving a high level of sparsity and preserving competitive performance, these methods usually involve a pre-trained model and a retraining process, which makes the training process remain inefficient.

Recently, several works have developed techniques allowing to train sparse neural networks with fixed parameter budget throughout the training based on adaptive sparse connectivity, e.g., sparse evolutionary training (SET) [52], DEEP-R [4], dynamic sparse reparameterization (DSR) [55], sparse momentum [20], ST-RNNs [43], and rigged lottery (RigL) [24]. The sparse weights, initialized with a fixed sparsity (a fraction of model parameters with zero values), can be maintained throughout training. The heuristic behind these techniques is following a cycle of weight pruning and weight regrowing based on a certain criterion. Essentially, the whole process of sparse training can be treated as a combinatorial optimization problem (weights and sparse structures). As the number of parameters during training is strictly constrained, sparse training techniques based on adaptive sparse connectivity are able to achieve the training efficiency as well as the inference efficiency associated with the final compressed model. However, due to the limited support for sparse operations in GPU-accelerated libraries, the sparse structure is enforced with binary masks. Thus, the training efficiency is only demonstrated theoretically not practically.

Due to the above-mentioned problems, the memory requirements and computation capacity to directly train wide neural networks with hundreds of thousands of neurons to deal with high-dimensional non-spatial like data (e.g., tabular data) with over 20,000 dimensions (input features) and less than 100 samples, are usually beyond what is allowed on commodity hardware. This paper aims to process high-dimensional data with a truly sparse end-to-end model. More precisely, we focus on the original SET algorithm because it was shown that it is capable of reaching very high accuracy performance [52, 73], many times even higher than the dense counterparts [43], while being very versatile and suitable for many neural network models (e.g., restricted Boltzmann machines [50], multi-layer perceptrons [44], and convolutional neural networks) and non-grid-like data. However, due to the limitations of typical deep learning libraries (e.g., optimized operations just for fully connected layers and dense matrices), the largest number of neurons used in [52] is just 12,082 neurons—quite a low representational power. Practically, the original SET-MLP implementation uses the typical

approach from the literature to work with sparsely connected layers, i.e., fully connected layers with sparsity enforced by a binary mask over their weights—this approach, of course, is far from using the full advantage of sparsity. Instead of generating a mask to enforce sparsity, in this paper, we devise the first sparse implementation for adaptive sparse connectivity so that it is possible to design neural network models which are very large in terms of representational power, but small in terms of space complexity to fit onto memory-limited devices.

The first contribution of this paper is a truly sparse implementation of SET, which can create and train SET-MLP models with hundreds of thousands of neurons on a typical laptop without GPU to handle data with tens of thousands of dimensions, a situation which is over the capacity of traditional fully connected MLPs. Secondly, we show that our proposed approach can be a good replacement for the current methods which employ both feature reduction and classifiers to perform classification on high-dimensional non-image datasets. Thirdly, we show that our proposed solution is robust to the “curse of dimensionality,” avoiding overfitting and achieving very good performance in terms of classification accuracy on a dataset with over 20,000 dimensions (input features) and less than 100 samples.

2 Related work

In this section, we will introduce the advances of processing high-dimensional microarray data and techniques allowing training sparse neural networks from scratch.

2.1 Artificial neural networks on microarray gene expression

Data have become indispensable factors of the success of machine learning (ML). The performance of a ML application is primarily determined by the quality and the quantity of the training data [42]. Particularly, gene expression obtained from DNA microarray has emerged as a powerful solution to cancer detection and treatment [65]. However, most of the datasets in DNA microarray are high-dimensional and redundant, which would result in the unnecessary calculation, large memory requirement, and even the decrease of generalization ability due to the “curse of dimensionality” [19]. Moreover, the invisible relationships and non-standard structures among different features also make it very time-consuming to find the key features from tens of thousands of features.

To tackle this problem, various methods have been proposed by researchers. Among them, feature selection is undoubtedly a “de facto” standard as it is not only able to

remove the redundant features and to keep the important ones, but it also helps to improve the model performance [19]. Following the feature detection phase, standard machine learning classifiers can be used to perform classification on the selected features. Traditional feature selection methods can be roughly divided into three categories: filter methods [2, 7, 14, 15, 17, 26, 30, 39], wrapper methods [11, 16, 23, 38, 46, 61, 62], and embedded methods [41, 56, 63]. Independent of the classifier, filter methods are able to deal with large scale datasets efficiently as they have low computational costs due to the fact that they select variables using proxy measures (e.g., mutual information), not an error metric provided by the classifier [60]. Wrapper methods employ feedback classification accuracy to assess the different suboptimal subsets chosen by following search algorithms, which can have good results but also increases the computation cost [8]. The WrapperSubsetEval [32] is a general wrapper method which can be connected with various learning algorithms. Different from the previous two discussed categories, in embedded methods [47], the feature selection and the classifier are not separated from each other.

As more and more datasets with ultrahigh dimensions have emerged, these datasets also bring challenges to conventional algorithms running on normal computers due to the expensive computational costs. To address this problem, distributed computing has been proposed. A distributed decentralized algorithm for k-nearest neighbor (kNN) graph has been proposed in [59]. This framework is able to distribute the computation of the kNN graph with very big datasets by utilizing the sequential structure of kNN data. MapReduce [18] is an efficient programming model used by Google to compute different types of data and process large raw data. Moreover, a classifier framework combining MapReduce and proximal Support Vector Machine (mrPSVM) has been proposed in [40]. The results on several high-dimensional, low-sample benchmark datasets demonstrate that the ensemble of mrPSVM classifier with feature selection methods using statistical tests outperforms classical approaches.

Although the above-mentioned hierarchical algorithms can have a good performance on classification tasks, the proper performance heavily depends on the features meticulously selected by experts from different domains [49]. This means that at least a dimensionality reduction technique is needed before the classifier. As an emerging branch of machine learning, deep neural networks tackle this problem via the explosive increase of data and computation ability. Multilayer perceptron (MLP) is one of the most used architectures in deep neural networks, e.g., it represents 61% of a typical Google TPU (tensor processing unit) workload for production neural networks applications, while convolutional neural networks represent just

5% [37]. However, it is difficult to employ MLPs directly on high-dimensional data tasks due to the quadratic number of parameters in its fully connected layers. This limits MLPs size to several thousand neurons and a few thousand input features on commodity hardware, and implicitly their representational power.

Being a successful approach that has been widely used in image recognition, speech recognition, language translation, etc., deep neural networks have also been employed to deal with high-dimensional data. MLPs have been widely applied to solve gene expression regulation problems. Chen et al. [10] have presented MLPs for gene expression inference (D-GEX) to perform gene expression inference to the GEO microarray data and RNA-seq expression data. An autoencoder has been connected with principal component analysis (PCA) to learn the high-level features of 13 microarray data [25]. Convolutional neural networks (CNNs) are also used to solve biological sequence problems due to its outstanding capability to learn spatial information. Alipanahi et al. [1] have proposed a CNN-based approach, called DeepBind, to handles both microarray and sequencing data. By two downstream applications, DeepBind can automatically analyze sequencing data and alleviate the time-consuming human designing work.

2.2 Intrinsically sparse neural networks

Recently, there are some works attempting to train an intrinsically sparse neural network from scratch to obtain the efficiency for both the training and inference phases. Mocanu et al. [51] have trained sparse restricted Boltzmann machines that have fixed scale-free and small-world connectivity. After that, Mocanu et al. [52, 53] have introduced the sparse evolutionary training procedure and the concept of adaptive connectivity for intrinsically sparse networks to fit the data distribution. The Nest algorithm [13] gets rid of a fully connected network at the beginning by a grow-and-prune paradigm, that is, expanding a small randomly initialized sparse network to a large one and then shrink it down. A Bayesian posterior has been applied to sample the sparse network configurations, while providing a theoretical guarantee for connectivity rewiring [4]. Besides weights pruning and regrowth, cross-layer weights redistribution has been used to adjust network architectures for better performance [55]. Liu et al. [44] have further reduced the number of parameters by applying neurons pruning, while getting competitive performance. Dettmers et al. [20] have used the momentum information of momentum Stochastic gradient descent to tackle weights regrowth and redistribution problems, reaching dense performance levels with 35–50%, 5–10%, and 20–30% weights for AlexNet, VGG16, and Wide Residual Networks, respectively. Very

recently, by modifying the sparsity distribution of Erdős–Rényi introduced in [52], RigL [24] can match and sometimes exceed the performance of pruning-based approaches. On the other hand, the Lottery Ticket Hypothesis has been proposed to find the sparse networks that can reach better accuracy [27] than dense networks. However, a dense network trained at the beginning limits its efficiency only for inference, not the training process. While achieving proper performance, these methods demonstrated computational efficiency via applying a binary mask on the weights due to the lack of efficient sparse linear algebra support from processors like TPUs or GPUs.

2.3 Sparse evolutionary training

Inspired by the fact that biological neural networks are prone to be sparse, rather than dense [58, 66], there is an increasing interest in conceiving neural networks with a sparse topology [51, 71]. In [52], the authors proposed a novel concept, sparse neural networks with adaptive sparse connectivity to maintain sparsity during training. Given a dataset $\mathbf{D} = \{(x_i, y_i)\}_{i=1}^n$, let a network denoted by:

$$\hat{y} = f(x; \theta) \quad (1)$$

where the $f(x; \theta)$ is the neural network parameterized by θ . The parameters θ can be decomposed into dense matrix $\theta^l \in \mathbf{R}^{n^{l-1} \times n^l}$, where n^l and n^{l-1} represent the number of neurons of the layer l and $l-1$, respectively. We train the network to minimize the loss function $\sum L(f(x; \theta), y)$. The motivation of sparse neural networks is to reparameterize the dense network only with a fraction of parameters, θ_s . The parameters θ_s can be decomposed into sparse matrix $\theta_s^l \in \mathbf{R}^{n^{l-1} \times n^l}$, for each layer l . A sparse neural network can be demoted by:

$$\hat{y}_s = f_s(x; \theta_s) \quad (2)$$

Let us define the sparsity of the network as $S = 1 - \frac{\|\theta_s\|_0}{\|\theta\|_0}$, where $\|\theta\|_0$ refers to the l_0 norm of θ .

The sparse evolutionary training (SET) is a method that allows efficiently training sparse neural networks from scratch with a fixed number of parameters. The basic idea underlying SET is first initializing a network with a sparse topology and then optimizing the weight values and the sparse topology together during the training process, to fit the data distribution. Different from the conventional methods, e.g., weights pruning [12, 34] which creates sparse topologies during or after the training process, the network trained with SET is designed to be sparse before training. This quadratically reduces the number of connections during the whole training phase. The main parts of

SET are sparse initialization and the weight pruning–regrowing cycles, explained below.

2.3.1 Sparse initialization

The initial sparse topology proposed in SET is Erdős–Rényi random graph topology [22] where a sparse matrix $\theta_s^l \in \mathbf{R}^{n^{l-1} \times n^l}$ represents connections between two consecutive layers $l-1$ and l . More precisely, the network is initialized by:

$$\theta_s^l = \theta^l * M^l \quad (3)$$

where $*$ represents the Hadamard product and M^l is a binary matrix of the same size with θ^l , in which each element $M_{i,j}^l$ is given by the probability $P(M_{i,j}^l) = \min(\frac{\epsilon(n^l + n^{l-1})}{n^l \times n^{l-1}}, 1)$. $\epsilon \in \mathbf{R}^+$ is a hyperparameter to control the sparsity level S . Such initialization distributes higher sparsity to the layers where n^l is approximately in the same range with n^{l-1} , and lower sparsity to the layers where $n^l \gg n^{l-1}$ or vice versa.

2.3.2 Weight pruning–regrowing cycle

After each training epoch, unimportant connections (accounting for a certain fraction ζ of $\|\theta^l\|_0$) will be pruned in each layer. The remaining connections are given by:

$$\theta_s^l = \theta_s^l * (M^l - P^l) \quad (4)$$

where P^l is a binary matrix with the same size as M^l , $\|P^l\|_0 = \zeta \|\theta^l\|_0$, and the nonzero elements of P^l is a subset of the nonzero elements of M^l corresponding to largest negative weights and the smallest positive weights in θ_s^l . After that, an equal number of connections with $\zeta \|\theta^l\|_0$ are randomly added to each layer by:

$$\theta_s^l = \theta_s^l + \theta_r^l \quad (5)$$

where $\theta_r^l \in \mathbf{R}^{n^{l-1} \times n^l}$ has exactly $\zeta \|\theta^l\|_0$ nonzero values. The nonzero element locations from θ_r^l are picked using a random uniform distribution, and their values are set using a small Gaussian noise. Finally, M^l is updated as follows:

$$M_{i,j}^l = \begin{cases} 1, & \text{if } \theta_s^l(i,j) \neq 0 \\ 0, & \text{otherwise.} \end{cases} \quad \forall i,j \quad (6)$$

Roughly speaking, the removal of the connection in SET represents natural selection, whereas the emergence of new connections corresponds to the mutation phase in natural evolution inspiring computing.

However, the authors of SET have used Keras with TensorFlow backend to implement their SET-MLP models. This implementation choice, while having the significant

advantage of offering wide flexibility of architectural choices (e.g., various activation functions, optimizers, GPUs, and so on), which is very welcomed while conceiving new algorithms, does not offer proper support for sparse matrix operations. This limits the practical aspects of SET-MLP considerably with respect to its maximum possible number of neurons and implicitly to its representational power. Due to these reasons, the largest SET-MLP model reported in the original paper [52] only contains 12,082 neurons on NVIDIA Tesla M40. Note that it is possible to increase the size of such SET-MLP implementations with several thousands more neurons, but no chance to reach one million neurons.

Algorithm 1: Sparse evolutionary training (SET)

Data: A neural network M with L layers, Weight θ , sparsity S , pruning rate ζ , training epoch n

```

1 % Sparse initialization
2 for  $l \leftarrow 0$  to  $L$  do
3    $\theta_s^l = \theta^l * M^l$ 
4 end
5 % Weight pruning-regrowing cycle
6 for epoch  $\leftarrow 0$  to  $n$  do
7    $\theta_s \leftarrow \text{NormalTraining}(\theta_s)$ 
8   for  $l \leftarrow 0$  to  $L$  do
9      $\theta_s^l \leftarrow \theta_s^l * (M^l - P^l)$ 
10     $\theta_s^l \leftarrow \theta_s^l * (M^l + R^l)$ 
11   end
12 end
```

3 Proposed method

In this paper, we address the above limitations of the SET original implementation and we show how vanilla SET-MLP can be implemented from scratch using just pure Python, SciPy, and Cython. Our approach enables the construction of SET-MLPs with at least two orders of magnitude larger, i.e., over 1,000,000 neurons. What is more, such SET-MLPs do not need GPUs and can run perfectly fine on a standard laptop.

3.1 Sparse matrices operations

The key element of our very efficient implementation is to use sparse data structures from SciPy. It is important to use the right representation of a sparse matrix for different operations because different sparse matrix formats have different advantages and disadvantages. Below the SciPy sparse data structures used to implement SET-MLPs are briefly discussed, while the interested reader is referred to¹ for detailed information.

- Compressed sparse row (CSR) sparse matrix: The data are stored in three vectors. The first vector contains nonzero values, the second one stores the extents of rows, and the third one contains the column indices of

the nonzero values. This format is very fast for many arithmetic operations but slow for changes to the sparsity pattern.

- Linked list (LIL) sparse matrix: This format saves nonzero values in row-based linked lists. Items in the rows are also sorted. The format is fast and flexible in changing the sparsity patterns but inefficient for arithmetic matrix operations.
- Coordinate list (COO) sparse matrix: This format saves the nonzero elements and their coordinates (i.e., row and column). It is very fast in constructing new sparse matrices, but it does not support arithmetic matrix operations and slicing.
- Dictionary of keys (DOK) sparse matrix: This format has a dictionary that maps row and column pairs to the value of nonzero elements. It is very fast in incrementally constructing new sparse matrices, but cannot handle arithmetic matrix operations.

Note that, one format cannot handle all operations necessary for sparse weights matrices to implement a SET-MLP. Still, the conversions from one format to another are very fast and efficient. Thus, in our implementation which was done in pure Python 3, we have used for specific SET-MLP operations, specific sparse matrix formats, and their fast conversion capabilities, as follows.

Initialize sparsely connected layers The sparse matrices which store the sparsely connected layers are creating using the linked list (LIL) format and then are transformed into compressed sparse row (CSR) format.

Feed-forward phase During it, the sparse weights matrices are stored and used in the CSR format.

Backpropagation phase—computing gradients The only operations which cannot be implemented with SciPy sparse matrix operations are computing the gradients for backpropagation [64] due to the simple fact that by multiplying the vector of backpropagation errors from layer h^k with the vector of activation neurons from layer h^{k-1} will perform a considerable amount of unnecessary multiplications (for nonexistent connections) and will create a dense matrix for updates. This dense matrix, besides being very slow to process, will have a quadratically number of parameters with respect to its number of rows and columns and will fill a 16 GB RAM very fast (in practice, for less than 10,000 neurons per layer given all the other necessary information which have to be stored in the computer memory). To avoid this situation, we have implemented in Cython the computations necessary for the batch weight updates. In this way, we compute in a much faster manner than in pure Python the gradient updates just for the existing connections. For this step, the sparse weight matrices are stored and used in the Coordinate list (COO) format.

¹ <https://docs.scipy.org/doc/scipy/reference/sparse.html>. Last visit 3 June 2018.

Backpropagation phase—weights update For this, the sparse weights matrices are used in the CSR format.

3.2 Implementation of weight pruning–regrowing cycle

In this section, we introduce the implementation of weight pruning–regrowing cycle for Eqs. (4) and (5). The key aspect of the SET method that sets it apart from the conventional DNN training is the evolutionary scheme which modifies the connectivity of the layers at the end of every epoch. As the weight evolution routine is executed quite often, the routine needs to be implemented in an efficient manner to ensure that the SET-MLP training can be done as fast as possible. Furthermore, as the layer connections are extremely sparse in the SET scheme, the implementations should ensure that the sparsity level is maintained. Actually, it shall exploit the sparsity while removing and adding new weights. Two implementations of the weight evolution scheme were coded in native Python using Numpy sparse matrix routines.

3.2.1 Implementation I

The first implementation is readable and intuitive, but does not exploit the full capabilities of the Numpy library in its various operations. In this implementation, the sparse weight matrices in the CSR format are converted to three vectors representing the indices of the rows, columns of the nonzero elements along with the element values (either using the COO or LIL format). The values are then compared in a for-loop to the threshold to keep the weights or discard them, as per the user specified ζ values. To ensure that the total number of nonzeros in the weight matrix remains the same, random connections between neurons need to be created. Again a for-loop is used to create new random connections in an incremental manner and ensure that the total number of nonzeros is equal to the original number of nonzeros. Most of the processing time in the code occurs in the for-loops and the while loops, and this is confirmed by a code profiling tool in python.² Furthermore, as we are constantly accessing the weights by the row and column index, this method does not exploit the sparsity of the weight matrix. The code profile of the processing time demonstrated that the removal of weights of the weight matrix takes about 15% of total time in an epoch and adding new random connections takes about 50% of the total time during an epoch. The detailed algorithm is given in Algorithm 2.

² Line_profiler by Robert Kern, [Available Online] https://github.com/rkern/line_profiler.

Algorithm 2: Weight pruning–regrowing cycle - Implementation I

Input : Sparse Weight Matrix (W)
Output: Sparse Weight Matrix with random weights added

```

1 %Removal of small weights
2 Extract values ( $V$ ), row ( $R$ ) and column indices ( $C$ ) of the non-zeros from  $W$ 
3 Find maximum negative value ( $V_{neg}$ ) and minimum positive value ( $V_{pos}$ )
4 Initialize  $N = 0$ 
5 for  $i$  in  $R$  do
6   for  $c$  in  $C$  do
7     if  $V_{neg} < W_{r,c} < V_{pos}$  then
8        $W_{r,c} = 0$ 
9        $N = N + 1$ ;
10    end
11  end
12 end
13 %Addition of random weights
14 while  $N > 0$  do
15   Choose  $i$  randomly from 1 to rows of  $W$ 
16   Choose  $j$  randomly from 1 to columns of  $W$ 
17   if  $W_{i,j} == 0$  then
18     Add a random value to  $W_{i,j}$ 
19      $N = N - 1$ ;
20   end
21 end
```

Algorithm 3: Weight pruning–regrowing cycle - Implementation II

Input : Sparse Weight Matrix (W)
Output: Sparse Weight Matrix with random weights added

```

1 %Removal of small weights
2 Extract values ( $V$ ), row ( $R$ ) and column indices ( $C$ ) of the non-zeros from  $W$ 
3 Find the maximum negative value ( $V_{neg}$ ) and the minimum positive value ( $V_{pos}$ )
4 Find the index  $i_{zero}$  of the values ( $V$ ) which are bigger than  $V_{neg}$  and smaller than  $V_{pos}$ 
5 Delete the indices of  $V$ ,  $R$  and  $C$  corresponding to the index  $i_{zero}$ 
6  $N = \text{length}(i_{zero})$ 
7 %Addition of random weights
8 Create a list of arrays ( $L_{old}$ ) with the remaining elements after removing:
 $L_{old} = [R, C]$ 
9 while  $N > 0$  do
10    $I$  = array of  $N$  randomly chosen from 1 to rows of  $W$ 
11    $J$  = array of  $N$  randomly chosen from 1 to columns of  $W$ 
12   Create list ( $L_{new}$ ) of arrays with  $k$  elements:  $L_{new} = [I, J]$ 
13   Remove duplicate elements from  $L_{new}$ 
14   Remove elements from  $L_{new}$  in common with  $L_{old}$ 
15    $N = N - \text{length}(L_{new})$ 
16    $L_{old} = \text{append}(L_{old}, L_{new})$ 
17   Clear  $L_{new}$ ;
18 end
19 Append  $N$  random values to  $V$ :  $V_{new} = \text{append}(V, \text{rand}(\text{size} = (1, N)))$ 
20 Unzip 1st and 2nd elements of  $L_{old}$ :  $R_{new} = L_{old}(:, 1)$ ,  $C_{new} = L_{old}(:, 2)$ 
21 Use COO format to update the  $W$ :  $W = \text{COO}(V_{new}, (R_{new}, C_{new}))$ 
```

3.2.2 Implementation II

In order to make full use of advantages of different sparse matrix formats, we also propose fast weights evolution (FWE). In FWE, the sparse weight matrices in the CSR format are also converted to three vectors representing the indices of the rows, columns of the nonzero elements along with the element values using the COO format. The value vector is compared a single time with the minimum and maximum threshold values using the vectorized operations in Numpy. This enables the identification of the indices of small weights for fast deletion of the weights. Next, the remaining row and column indices are stored together into an array and a list of all the arrays of the nonzero elements is created. This is used directly to determine the random row and column indices of the additional weights to ensure that the number of connections between the neurons is constant. As the weights are sparse, the size of the list is much smaller than the full size of the weight matrix and performing all the computations with the list will be faster. The detailed algorithm is given in Algorithm 3. The

comparison of the running time of these two implementations is given in Table 1, which shows that Implementation II is more efficient than Implementation I. The computational complexity (Big O Notation) is the same for both implementations. The difference is given by running python code without optimized C++ routines (Algorithm 2) and with optimized C++ routines (Algorithm 3).

4 Experimental evaluation

For a good understanding of SET-MLP performance, we compare it against another sparse MLP model (implemented by us in the same manner) in which the bipartite layers are initialized with an Erdős–Rényi topology, but does not evolve over time and has a fixed sparsity pattern, dubbed $MLP_{FixProb}$ as in [52]. Note that it is impossible to report also the accuracy for FC-MLPs as they cannot run on a typical laptop due to their very high memory and computational requirements. Moreover, even if it would be possible to run FC-MLP, this comparison is outside the scope of this paper and it would be redundant as it has been shown in [6, 24, 44, 52, 53, 73] that SET-MLP typically outperforms its fully connected counterparts.

4.1 Datasets

We evaluate and discuss the performance of our efficient SET-MLP implementation on four publicly available microarray datasets, as detailed in Table 2. It is worth highlighting that both the training and testing sets are unbalanced for all datasets. We choose 2/3 of the data as training data and 1/3 of the data as testing data. Note that we do not set validation data, as the sample sizes of these datasets are extremely small.

Leukemia The Leukemia dataset is obtained from the NCBI GEO repository with the accession number GSE13159. It contains 2096 samples with 54,675 features each. The samples are divided into 18 classes. Among these 2096 samples, 1397 samples are selected as training data and 699 as testing data. Table 3 shows the number of test samples in each class.

CLL-SUB-111 The CLL-SUB-111 dataset is an unbalanced dataset contains gene expressions from high-density oligonucleotide arrays consist of both genetically and clinically distinct subgroups of B cell chronic lymphocytic leukemia (B-CLL). It has 11,340 features and 111 samples, out of which 74 samples are selected as the training set and 37 as the testing set.

SMK-CAN-187 The SMK-CAN-187 dataset is a RNA dataset obtained from the normal bronchial epithelium of

smokers with and without lung cancer. It has 19,993 features and 187 samples. Out of these 187 samples, 124 samples are chosen as training data and 63 as testing data.

GLI-85 The GLI-85 dataset is the Affymetrix HG U133 oligonucleotide arrays on 85 diffuse infiltrating gliomas of all histologic types. It has 22,283 features and 85 samples. Out of these 85 samples, 56 samples are training data and 29 are testing data.

4.2 Evaluation metrics

To evaluate the performance of the proposed method, we have used the accuracy metric and the confusion matrix to get detailed visual information. The confusion matrix (**M**) contains information per class about both model predictions and ground truth. These enable people to understand and diagnose the models better. The confusion matrix template and related performance measures for two-class classification problems are given in Table 4. In terms of multiclass classification, assuming that the number of classes is c , the performance measures of the i th class are given by the following equations:

$$\begin{aligned}\text{Recall}_i &= \frac{M_{ii}}{\sum_{j=1}^c M_{ji}} \\ \text{Precision}_i &= \frac{M_{ii}}{\sum_{j=1}^c M_{ij}} \\ \text{Accuracy} &= \frac{\sum_{i=1}^c M_{ii}}{\sum_{i=1}^c \sum_{j=1}^c M_{ij}}\end{aligned}\quad (7)$$

The rows of the confusion matrix represent the predicted classes, and the columns correspond to the true classes. The diagonal cells represent the numbers of samples that are correctly classified. The off-diagonal cells are the incorrectly classified number of samples. The row at the bottom of the confusion matrix gives the proportion of all examples belonging to each class that is correctly (green) and incorrectly (red) classified. The column on the far right of the confusion matrix represents the proportion of all the samples predicted to belong to each class that are correctly (green) and incorrectly (red) classified.

4.3 Experimental setup

For both models, SET-MLP and $MLP_{FixProb}$, the hyperparameters are the same to guarantee the fairness of comparison. The number of hidden layers for Leukemia, CLL-SUB-111, and SNK-CAN-187 is two but one for GLI-85, as overfitting occurs for GLI-85 with two hidden layers. The optimization method used in this paper is Stochastic Gradient Descent (SGD) with momentum. The numbers of neurons of each layer are given in Table 5. Note that, for

Table 1 Mean running time of evolution Implementation I and Implementation II

Matrix size	Implementation I (s)	Implementation II (s)
500*500	0.58	0.14
2000*2000	2.56	0.71
8000*8000	11.13	2.08
15,000*15,000	24.14	3.75

Leukemia dataset, the number of hidden neurons in each layer was set to 27,500, a value which is way above the usual number of neurons in fully connected MLP models. For all datasets, we get the mean accuracy by averaging the best test accuracy from 5 trials. Since the best accuracy is obtained at different epochs, there are some differences between accuracy in the figures and the mean accuracy reported in the text, especially for the CLL-SUB-111 and the SMK-CAN-187 datasets.

To demonstrate our algorithm's ability to significantly reduce the parameter count, we set the sparsity

Table 4 Confusion matrix for two-class classification

	Target class		
	Neg	Pos	
<i>Output Class</i>			
Classified as Neg	tn	fn	$npv = \frac{tn}{tn+fn}$
Classified as Pos	fp	tp	$Pre = \frac{tp}{tp+fp}$
	$Spe = \frac{tn}{tn+fp}$	$Rec = \frac{tp}{tp+fn}$	$Acc = \frac{tp+tn}{tp+fp+fn+tn}$

Spe specificity, *Rec* recall, *Pre* precision, *Acc* accuracy

Table 5 Number of neurons of SET-MLP on all datasets

Dataset	Input	1st hidden	2nd hidden	Output
Leukemia	54,675	27,500	27,500	18
CLL-SUB-111	11,340	9000	9000	3
SNK-CAN-187	19,993	16,000	16,000	2
GLI-85	22,283	20,000	None	2

Table 2 Microarray datasets used

Dataset	No. of samples	No. of features	No. of classes	Data size
Leukemia [31]	2096	54,675	18	1.93 GB
CLL-SUB-111 [35]	111	11,340	3	5.9 MB
SMK-CAN-187 [72]	187	19,993	2	11.9 MB
GLI-85 [28]	85	22,283	2	8.7 MB

Table 3 Leukemia class labels and their corresponding number of test samples

Leukemia	Class label	No. of samples
Mature B-ALL with t(8;14)	1	4
Pro-B-ALL with t(11q23)/MLL	2	23
C-ALL/Pre-B-ALL with t(9;22)	3	41
T-ALL	4	58
ALL with t(12;21)	5	19
ALL with t(1;19)	6	12
ALL with hyperdiploid karyotype	7	14
C-ALL/Pre-B-ALL without t(9;22)	8	79
AML with t(8;21)	9	14
AML with t(15;17)	10	12
AML with inv(16)/t(16;16)	11	9
AML with t(11q23)/MLL	12	13
AML with normal karyotype + other abnormalities	13	115
AML complex aberrant karyotype	14	18
CLL	15	149
CML	16	25
MDS	17	68
Non-leukemia and healthy bone marrow	18	26

hyperparameter $\epsilon = 10$ guaranteeing an extremely sparse network for all datasets. The corresponding sparsity for Leukemia, CLL-SUB-111, SNK-CAN-187, and GLI-85 is 99.93%, 99.78%, 99.88%, and 99.90%, respectively. The corresponding sparsity and parameter numbers are illustrated in Table 7. The rewiring rate ζ is set to 0.3. We train all models for 500 epochs by momentum SGD with a momentum of 0.9 and a weight decay of 0.0002. We choose the remaining hyperparameters based on a small random search. For Leukemia, we use a learning rate of 0.005 and a batch size of 5; for CLL-SUB-111, we choose a learning rate of 0.01 and a batch size of 5; for SNK-CAN-187, the learning rate is set as 0.005 and the batch size is set as 5; and for GLI-85, the learning rate is set as 0.005 and the batch size is set as 1.

All the experiments performed are executed on a typical laptop using *a single thread of the CPU*. The laptop configuration is as follows:

- Hardware configuration: CPU Intel Core i7-4700MQ, 2.40 GHz \times 8, RAM 16 GB, Hard disk 500 GB.
- Software used: Ubuntu 16.04, Python 3.5.2, Numpy 1.15.2, SciPy 1.1.0, and Cython 0.27.3.

4.4 Experimental results

Table 6 summarizes the performance of SET-MLP and $MLP_{FixProb}$ on all four datasets trained with extremely high sparsity levels. We can observe that SET-MLP consistently outperform $MLP_{FixProb}$ on all datasets, which means that the adaptive sparse connectivity associated with SET-MLP helps to find better sparse structures. This behavior suggests that the SET algorithm indeed solves successfully the combinatorial optimization problem. From the perspective of continuous optimization, the optimizer used in this experiment, momentum SGD, is used for optimizing model weights. Both SET and momentum SGD are crucial to the superior performance of sparse training [55].

For a better understanding of the learning process of our method, we show the learning curves of SET-MLP and $MLP_{FixProb}$ for all datasets in Fig. 1. It is shown that SET-MLP can reach a higher accuracy than $MLP_{FixProb}$ as the training epoch increases. More interestingly, the learning curves of SET-MLP oscillate more frequently than the

fixed sparse networks during the training process. This phenomenon makes sense since the weights rewiring (pruning and regrowing) cycle within adaptive sparse connectivity is triggered after each training epoch, changing 30% connections of the network. Furthermore, it is noteworthy that, on SMK-CAN-187 dataset, $MLP_{FixProb}$ seems to suffer from overfitting after around 350 epochs.

To provide deeper insights into the classification results of our method, we illustrate the confusion matrices of the best run of SET-MLP on Leukemia dataset in Fig. 2 and the rest of the datasets in Fig. 3. We can see that the test accuracy of the best run of Leukemia is 88.10%. Besides this, we can observe that SET-MLP has a perfect recall for class 1 (100.0%) of CLL-SUB-111, even though there are extremely unfavorable conditions, i.e., very few training samples. As shown in Fig. 3b, SET-MLP performs better for class 2 than class 1 on SNK-CAN-187. It is noteworthy that, for GLI-85, the best accuracy out of five runs of SET-MLP is 100%, which means on Gli-85 whose available data are extremely insufficient (85), SET-MLP can still model the dataset perfectly.

To further evaluate the effectiveness of our proposed method, we compare SET-MLP with the state-of-the-art conventional two-phase techniques on these datasets in terms of classification performance. To the best of our knowledge, the state-of-the-art performance for Leukemia is 81.11% reported in the literature [40]. Therein, an ensemble classifier is proposed to deal with microarray data by connecting several feature selection algorithms with MapReduce-based proximal support vector machine (mrPSVM). SET-MLP is able to achieve a higher accuracy of 87.60 ± 0.06 with exactly the same training and testing data splitting. Among the feature selection-based methods to CLL-SUB-111, an accuracy of 78.38% is obtained by using Incremental Wrapper-based Attribute Selection (IWSS) [5]. The state-of-the-art accuracy on SMK-CAN-187 is $(74.87 \pm 2.32\%)$ reported in [69], in which feature selection was performed by preserving class correlation. Reported in [67], an ensemble including three filter methods with a meta-heuristic algorithm is used to achieve an accuracy of 94%. We can observe that our method can outperform these traditional two-phase techniques via one efficient end-to-end model. It is noteworthy that although CLL-SUB-111, SMK-CAN-187, and GLI-85 seriously suffer from an extremely small number of

Table 6 Test accuracy of SET-MLP and $MLP_{FixProb}$ on the four datasets

Methods	Leukemia	CLL-SUB-111	SNK-CAN-187	GLI-85
SET-MLP	87.60 ± 0.06	81.62 ± 0.05	75.24 ± 0.04	94.48 ± 0.05
$MLP_{FixProb}$	82.74 ± 0.05	71.35 ± 0.04	68.57 ± 0.04	92.41 ± 0.06

Every number is averaged from 5 trials. The sparsity levels for Leukemia, CLL-SUB-111, SNK-CAN-187, and GLI-85 are 99.93%, 99.78%, 99.88%, and 99.90%, respectively

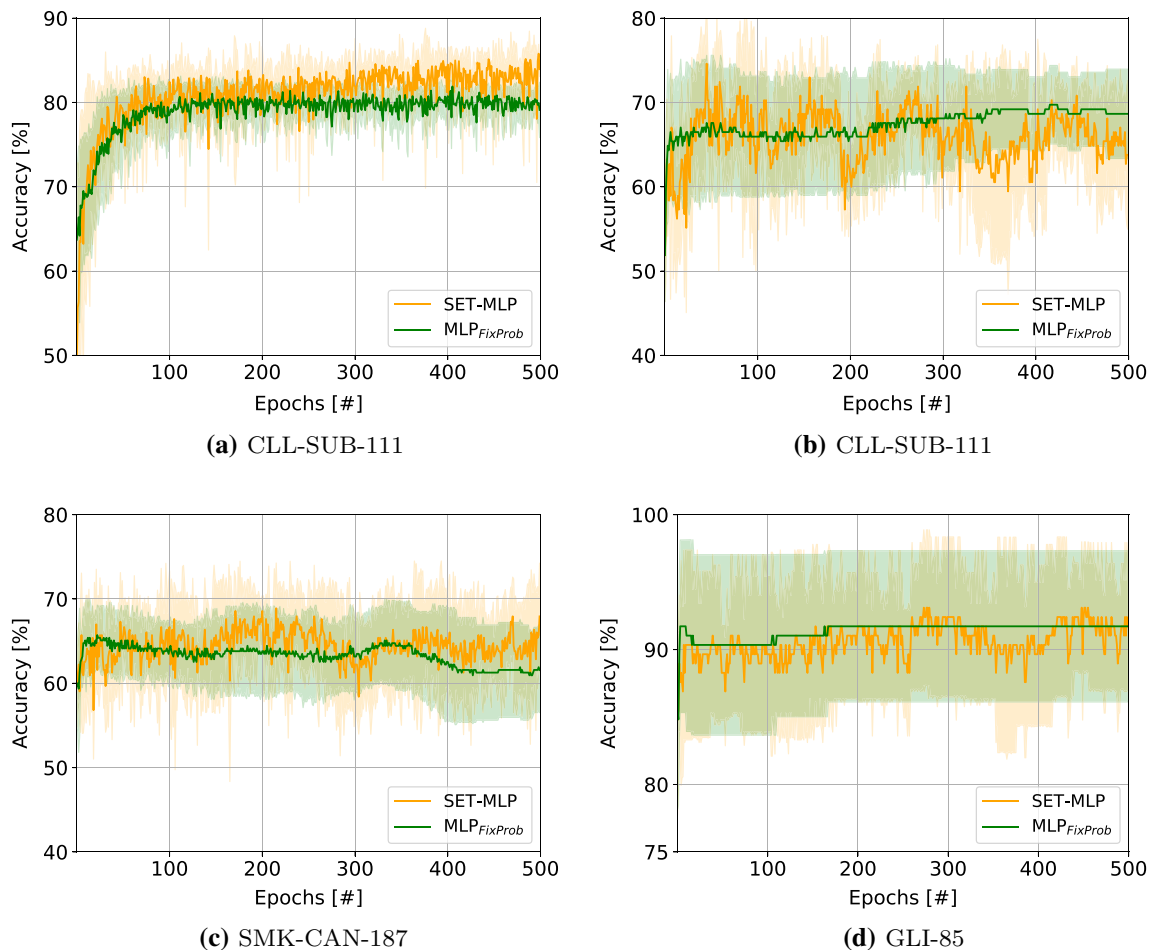


Fig. 1 Test accuracy of SET-MLP and $MLP_{FixProb}$ on CLL-SUB-111, SMK-CAN-187, and GLI-85. All the test accuracy at each epoch is averaged from 5 trials

samples, we are still able to obtain good performance with efficient sparse training.

4.5 Results analysis

To understand better the connections reduction made by the SET procedure in a SET-MLP model in comparison with a fully connected MLP (FC-MLP) which has the same amount of neurons, Fig. 4 and Table 7 provide the number of connections for the SET-MLP models discussed above and their FC-MLP counterparts on all four datasets. It is clear that SET has dramatically reduced the connection numbers in MLPs. For instance, a traditional FC-MLP on the Leukemia dataset would have 2,260,307,500 connections, while SET-MLP has just 1,582,376 connections. This quadratic reduction in the number of connections is the key factor in guaranteeing that SET-MLP can run fine on a standard laptop for datasets with tens (up to few hundreds) of thousands of input features.

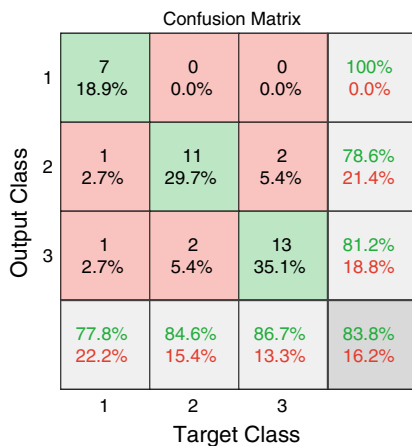
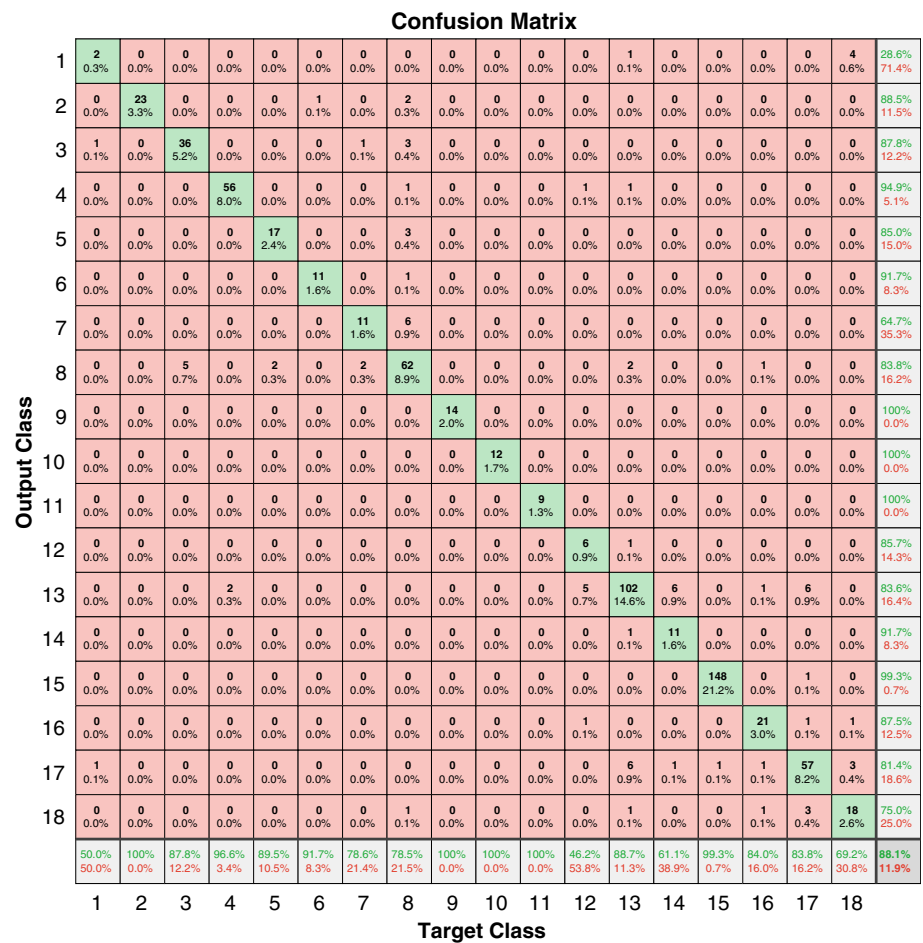
For a better understanding of SET computational requirements, Table 8 shows the average training and

testing time per epoch of the SET-MLPs used on the datasets. We can observe, as expected, that as the number of features and samples increases the training time is also increasing. Still, it is worth to highlight that although the average training time of Leukemia is relatively long (61.31s), it fulfills an almost impossible mission, that is, running such a large model on a commodity laptop.

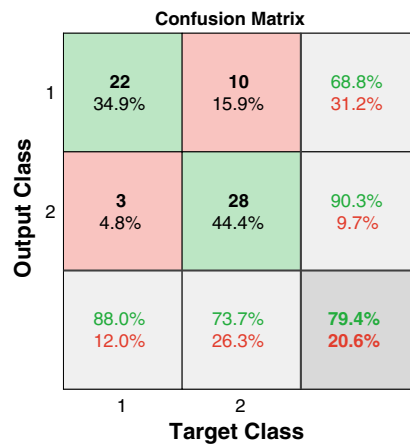
4.6 Extreme SET-MLP models on leukemia

While in the previous section, we have analyzed the qualitative performance of our proposed approach, in this section, we briefly discuss two extreme SET-MLP models on the largest dataset used in this paper, i.e., Leukemia. The goal is to assess how fast SET-MLP can achieve a good performance and to see how large a trainable SET-MLP model can be on a typical laptop. For each model, we used a SET-MLP with two hidden layers and a Softmax layer as output. For the small SET-MLP model, the number of hidden neurons per layer was set to 1000, while for the large SET-MLP model the number of hidden neurons per

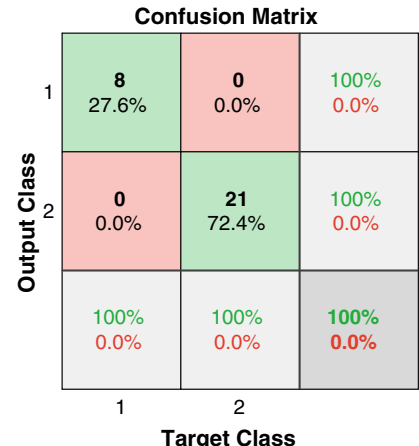
Fig. 2 Confusion matrix of the best run with SET-MLP on the Leukemia dataset



(a) CLL-SUB-111 dataset



(b) SMK-CAN-187 dataset



(c) GLI-85 dataset

Fig. 3 Confusion matrix of the best run with SET-MLP on CLL-SUB-111, SMK-CAN-187, and GLI-85

layer was set to 500,000. In both cases, we have used a very eager learning rate (0.05) and we trained the models for 5 epochs. On each hidden layer, we applied a dropout rate of 0.4. The other hyperparameters were set as in the

previous section for Leukemia and we have used the same training/testing data splitting.

Table 9 presents SET-MLP performance in comparison with the best state-of-the-art results of mrPSVM from [40]. We clarify that the goal of this experiment is not to obtain

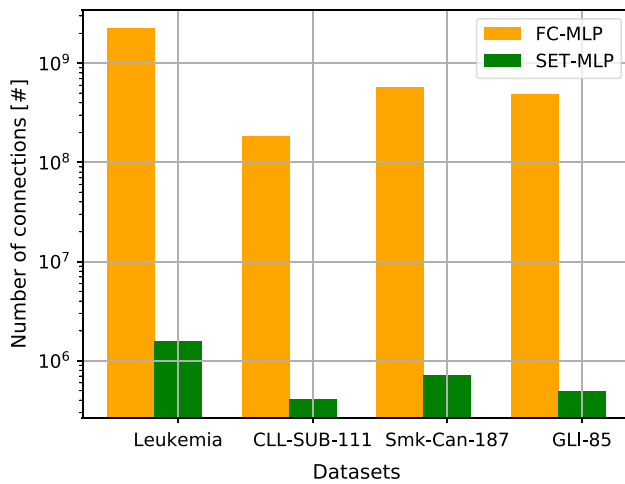


Fig. 4 The number of connections for the SET-MLP models with two hidden layers used on the Leukemia, CLL-SUB-111, and SMK-CAN-187 datasets and with one hidden layer used on the GLI-85 dataset, plotted against their FC-MLP counterparts

Table 7 Number of connections and sparsity levels for SET-MLP and FC-MLP on all datasets

Dataset	Number of connections (#)		Sparsity level (%)
	FC-MLP	SET-MLP	
Leukemia	2,260,307,500	1,582,376	99.93
CLL-SUB-111	183,087,000	409,033	99.78
SMK-CAN-187	575,920,000	711,305	99.88
GLI-85	490,270,000	486,350	99.90

the best accuracy possible with SET-MLP. Still, the small SET-MLP model, which has in total 56,693 neurons and 581,469 connections, has a total training and testing time of

Table 8 Running time in seconds (s) per epoch for SET-MLP

Dataset	Average training time (s) (per epoch)	Average testing time (s) (per epoch)
Leukemia	61.31	2.36
CLL-SUB-111	6.65	0.06
SMK-CAN-187	27.17	0.18
GLI-85	32.87	0.05

Table 9 Two extreme SET-MLP models on Leukemia against state-of-the-art (mrPSVM with ANOVA [40] for feature selection)

Model	Hardware	Density level (%)	Total time (s) (train + test)	Accuracy (%)
Small SET-MLP	1 CPU thread	1.04	65	82.88 ± 1.18
Large SET-MLP	1 CPU thread	0.007	4914	81.83 ± 1.11
mrPSVM [40]	Conventional	n/a	1265	81.1
mrPSVM [40]	Hadoop cluster	n/a	291	81.1

The numbers in brackets for SET-MLP reflect the number of neurons per layer from input to output. The accuracy of SET-MLP is reported as the mean and standard deviation of 5 runs. The density level represents the percentage of the number of existing connections in the SET-MLP model from the total number of connections in its corresponding FC-MLP

65 seconds. It is about 20 times faster than mrPSVM which runs on conventional hardware and about 4.5 times faster than mrPSVM which runs in a Hadoop cluster while reaching with 1.7% better accuracy. At the same time, its small standard deviation shows that the model is very stable. Furthermore, we highlight that the very large SET-MLP model which has in total 1,054,693 neurons with about 19,383,046 connections takes about 16 min per training epoch and in 5 epochs reaches a good accuracy, better than state-of-the-art. All of these happen on 1 CPU thread of a typical laptop. We highlight that this is the first time in the literature when a MLP variant with over 1 million neurons is trained on a laptop, while the usual MLP models trained on a laptop can have at maximum few thousand of neurons. In fact, it is hard to quantify, but according to [29], the size of the largest neural networks which run currently in the cloud is about 10–20 million neurons. Therefore, our results emphasize even more the capabilities of SET-MLPs and open the path for new research directions.

4.7 Sensitivity analysis of the number of hidden layers

Previously, we have discussed the performance of the SET-MLP models with two hidden layers on the Leukemia, CLL-SUB-111, and SMK-CAN-187 datasets and with one hidden layer on the GLI-85 datasets. We now explain our choices on the number of hidden layers by presenting the performance of SET-MLP models with one, two, and three hidden layers on all datasets comparatively and by discussing the beneficial effect of dropout [36] on SET-MLP. The number of neurons per hidden layer and the other

hyperparameters are set to be the same with the previous models. Figure 5 summarizes these experiments. From the first row, it can be inferred that SET-MLP with two hidden layers reaches the highest peak accuracy (88.12%) and has relatively the most robust performance on the Leukemia dataset. Similarly, SET-MLP with two hidden layers reaches outstanding accuracy (81.11%) on the CLL-SUB-111 dataset, while the accuracy cannot reach 80% with one or three hidden layers.

As expected, but at the same time having the most interesting results, due to the very small number of samples of GLI-85 (Fig. 5, third row), SET-MLP with one hidden layer avoids overfitting in exchange to quite an oscillating behavior. At the same time, SET-MLP with two or three hidden layers even if they are capable of also reaching perfect accuracy of 100%, after about 200 epochs, they have a dramatic drop in accuracy to about 80%. We hypothesize that this situation happens due to overfitting as the number of training samples is extremely insufficient. If this is the case, adding dropout regularization to SET-MLP is able to figure out this problem. We applied dropout with 0.5 dropout rate to both hidden layers. The performance is shown in Fig. 6a. It is clear that the accuracy of SET-MLP with dropout keeps the same trend as before, without any drop in accuracy after 200 epochs. Moreover, we conduct an extra experiment to test whether SET-MLP with no hidden layers can achieve higher accuracy or not. Since the number of input features is much higher than the number of classes, the connectivity is almost dense. As shown in Fig. 6b, it cannot reach 100% classification performance. This phenomenon highlights the fact that our proposed

method can guarantee efficient training while not compromising performance.

5 Conclusion

Processing microarray data have been treated in the literature as a difficult task due to their very high number of features but the little number of examples. Besides that, this type of data suffers from imbalance and data shift problems.

In this paper, an efficient implementation of SET-MLP, a sparse multilayer perceptron trained with the sparse evolutionary training procedure, is proposed to deal with high-dimensional microarray datasets. This implementation makes use just of Python 3, sparse data structures from SciPy, and Cython. With this implementation, we have created for the first time in literature sparse MLP models with over one million neurons which can be trained on a standard laptop using a single CPU thread and without GPU. This is with two orders of magnitude more than state-of-the-art MLP models trained on commodity hardware.

Besides, we demonstrated four microarray datasets with tens of thousands of input features and with up to just two thousand samples that our approach reduces the number of connections quadratically in large MLPs (about 99.9% sparsity) while outperforming the state-of-the-art methods on these datasets for the classification task. Moreover, our proposed SET-MLP models showed to be robust to overfitting, imbalanced and data shift problems, which is not so usual for fully connected MLPs. Additionally, the results

Fig. 5 Experiments with SET-MLPs on all four datasets to understand the effect of the number of hidden layers (n^h). For each dataset, three cases for the number of hidden layers are considered, i.e., $n^h = \{1, 2, 3\}$. Each row represents the test classification accuracy of SET-MLPs with one, two, or three hidden layers on the same dataset. Every model from each row has been trained with the same hyperparameters as in the paper, except for the number of hidden layers

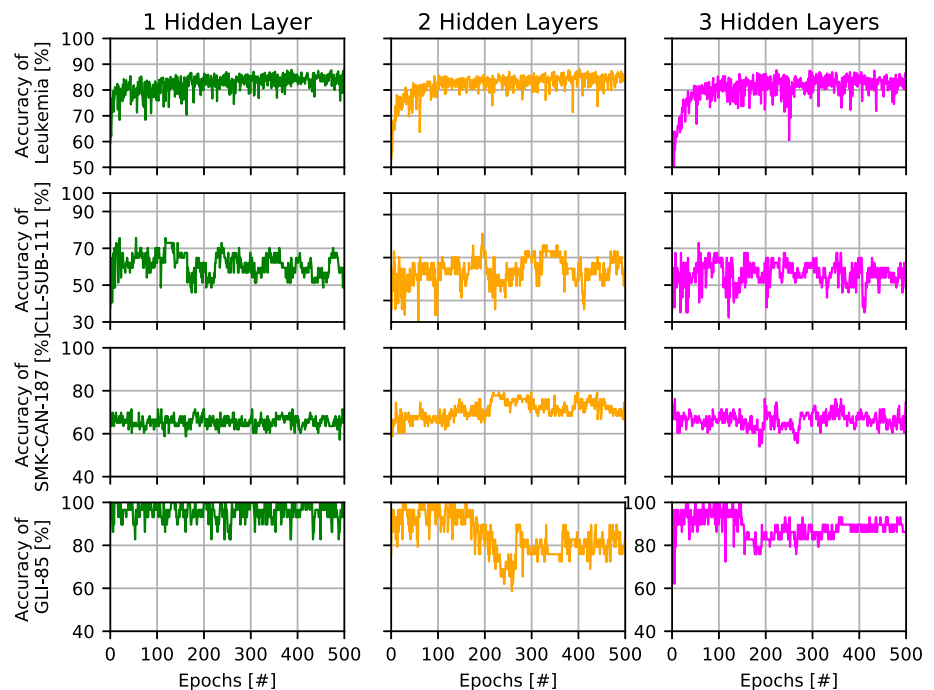
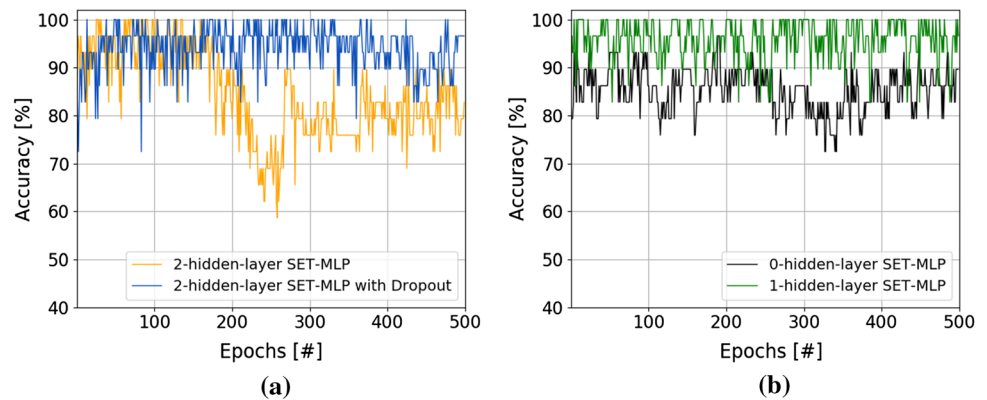


Fig. 6 Test accuracy of SET-MLP and MLP_{FixProb} on GLI-85. All the test accuracy at each epoch is averaged from 5 trials



suggest that our proposed approach can cope efficiently with the “curse of dimensionality,” being capable of learning from small amounts of labeled data, and outperforming the state-of-the-art methods (ensembles of classifiers and feature selection methods) which are currently employed on high-dimensional non-grid-like data (or tabular data).

In the future, we intend to put our emphasis on other types of neural layers, such as convolutional layers in CNN which have been widely used to deal with graphic data with grid-like topology. Furthermore, we intend to extend this work to address problems from other fields that suffer from the “curse of dimensionality” and which have ultrahigh-dimensional data (e.g., social networks, financial networks, semantic networks). The last but not the least future research direction would be to parallelize our implementation to use all CPU threads of a typical workstation efficiently and to incorporate it into usual Deep Learning frameworks, such as TensorFlow or PyTorch. This probably would allow us to scale with one order of magnitude more the SET-MLP models (up to the level of few tens of millions of neurons), while still using commodity hardware.

Acknowledgements We thank Ritchie Vink (<https://www.ritchievink.com/>, Last visit June 3, 2018) for providing on Github.com a vanilla fully connected MLP implementation and to Thomas Hagebols (<https://github.com/ThomasHagebols>, Last visit January 25, 2019) for analyzing the performance of SciPy sparse matrix operations.

Compliance with ethical standards

Conflict of interest The authors declare that they have no conflict of interest.

Open Access This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article’s Creative Commons licence, unless

indicated otherwise in a credit line to the material. If material is not included in the article’s Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit <http://creativecommons.org/licenses/by/4.0/>.

References

- Alipanahi B, Delong A, Weirauch MT, Frey BJ (2015) Predicting the sequence specificities of dna-and rna-binding proteins by deep learning. *Nat Biotechnol* 33(8):831
- Bekkerman R, El-Yaniv R, Tishby N, Winter Y (2003) Distributional word clusters vs. words for text categorization. *J Mach Learn Res* 3(Mar):1183–1208
- Belkin M, Hsu D, Ma S, Mandal S (2019) Reconciling modern machine-learning practice and the classical bias-variance trade-off. *Proc Natl Acad Sci* 116(32):15849–15854
- Bellec G, Kappel D, Maass W, Legenstein R (2017) Deep rewiring: training very sparse deep networks. *arXiv preprint arXiv:1711.05136*
- Bermejo P, de la Ossa L, Gámez JA, Puerta JM (2012) Fast wrapper feature subset selection in high-dimensional datasets by means of filter re-ranking. *Knowl Based Syst* 25(1):35–44
- Bourgin DD, Peterson JC, Reichman D, Griffiths TL, Russell SJ (2019) Cognitive model priors for predicting human decisions. *arXiv preprint arXiv:1905.09397*
- Caruana R, Sa VR (2003) Benefitting from the variables that variable selection discards. *J Mach Learn Res* 3(Mar):1245–1264
- Chandrashekar G, Sahin F (2014) A survey on feature selection methods. *Comput Electr Eng* 40(1):16–28
- Chauvin Y (1989) A back-propagation algorithm with optimal use of hidden units. In: *Advances in neural information processing systems*, pp 519–526
- Chen Y, Li Y, Narayan R, Subramanian A, Xie X (2016) Gene expression inference with deep learning. *Bioinformatics* 32(12):1832–1839
- Cordón O, Damas S, Santamaría J (2006) Feature-based image registration by means of the chc evolutionary algorithm. *Image Vis Comput* 24(5):525–533
- Cun YL, Denker JS, Solla SA (1990) Optimal brain damage. In: *Advances in neural information processing systems*. Morgan Kaufmann, pp 598–605
- Dai X, Yin H, Jha N (2019) Nest: a neural network synthesis tool based on a grow-and-prune paradigm. *IEEE Trans Comput* 68(10):1487–1497

14. Davidson JL, Jalan J (2010) Feature selection for steganalysis using the mahalanobis distance. In: Media forensics and security II, vol 7541. International Society for Optics and Photonics, p 754104
15. de Jesús Rubio J (2009) Sofmls: online self-organizing fuzzy modified least-squares network. *IEEE Trans Fuzzy Syst* 17(6):1296–1309
16. de Jesús Rubio J (2017) Usnfis: uniform stable neuro fuzzy inference system. *Neurocomputing* 262:57–66
17. de Jesus Rubio J, Pan Y, Lughofer E, Chen M-Y, Qiu J (2020) Fast learning of neural networks with application to big data processes. *Neurocomputing* 390:294–296. <https://doi.org/10.1016/j.neucom.2019.10.057>
18. Dean J, Ghemawat S (2008) Mapreduce: simplified data processing on large clusters. *Commun ACM* 51(1):107–113
19. Destrero A, Mosci S, De Mol C, Verri A, Odone F (2009) Feature selection for high-dimensional data. *Comput Manag Sci* 6(1):25–40
20. Dettmers T, Zettlemoyer L (2019) Sparse networks from scratch: faster training without losing performance. *arXiv preprint arXiv:1907.04840*
21. Elton DC (2020) Self-explainability as an alternative to interpretability for judging the trustworthiness of artificial intelligences. *arXiv preprint arXiv:2002.05149*
22. Erdős P, Rényi A (1959) On random graphs i. *Publ Math (Debrecen)* 6:290–297
23. Eshelman LJ (1991) The CHC adaptive search algorithm: How to have safe search when engaging in nontraditional genetic recombination. In: Rawlings GJE (ed) *Foundations of genetic algorithms*, vol 1. Elsevier, pp 265–283
24. Evci U, Elsen E, Castro P, Gale T (2020) Rigging the lottery: making all tickets winners. <https://openreview.net/forum?id=ryg7vA4tPB>
25. Fakoor R, Ladhak F, Nazi A, Huber M (2013) Using deep learning to enhance cancer diagnosis and classification. In: *Proceedings of the international conference on machine learning*, vol 28
26. Forman G (2003) An extensive empirical study of feature selection metrics for text classification. *J Mach Learn Res* 3(Mar):1289–1305
27. Frankle J, Carbin M (2018) The lottery ticket hypothesis: finding sparse, trainable neural networks. *arXiv preprint arXiv:1803.03635*
28. Freije WA, Castro-Vargas FE, Fang Z, Horvath S, Cloughesy T, Liau LM, Mischel PS, Nelson SF (2004) Gene expression profiling of gliomas strongly predicts survival. *Cancer Res* 64(18):6503–6510
29. Goodfellow I, Bengio Y, Courville A (2016) *Deep learning* (Subsection 1.2.3). The MIT Press, Cambridge
30. Guyon I, Elisseeff A (2003) An introduction to variable and feature selection. *J Mach Learn Res* 3(Mar):1157–1182
31. Haferlach T, Kohlmann A, Wiczeorek L, Basso G, Kronnie GT, Béné MC, De JV, Hernández JM, Hofmann WK, Mills KI et al (2010) Clinical utility of microarray-based gene expression profiling in the diagnosis and subclassification of leukemia: report from the international microarray innovations in leukemia study group. *J Clin Oncol* 28(15):2529–2537
32. Hall M, Frank E, Holmes G, Pfahringer B, Reutemann P, Witten IH (2009) The weka data mining software: an update. *ACM SIGKDD Explor Newsl* 11(1):10–18
33. Han S, Pool J, Tran J, Dally W (2015) Learning both weights and connections for efficient neural network. In: *Advances in neural information processing systems*, pp 1135–1143
34. Han S, Pool J, Tran J, Dally WJ (2015) Learning both weights and connections for efficient neural networks. In: *Proceedings of the 28th international conference on neural information processing systems*, vol 1. NIPS'15, MIT Press, Cambridge, pp 1135–1143. <http://dl.acm.org/citation.cfm?id=2969239.2969366>
35. Haslinger C, Schweifer N, Stilgenbauer S, Dohner H, Lichter P, Kraut N, Stratowa C, Abseher R (2004) Microarray gene expression profiling of b-cell chronic lymphocytic leukemia subgroups defined by genomic aberrations and vh mutation status. *J Clin Oncol* 22(19):3937–3949
36. Hinton GE, Srivastava N, Krizhevsky A, Sutskever I, Salakhutdinov R (2012) Improving neural networks by preventing co-adaptation of feature detectors. *CoRR arXiv:abs/1207.0580*
37. Jouppi NP, Young C, Patil N, Patterson D, Agrawal G, Bajwa R, Bates S, Bhatia S, Boden N, Borchers A, et al (2017) In-data-center performance analysis of a tensor processing unit. In: *2017 ACM/IEEE 44th annual international symposium on computer architecture (ISCA)*. IEEE, pp 1–12
38. Kohavi R, John GH (1997) Wrappers for feature subset selection. *Artif Intell* 97(1–2):273–324
39. Koller D, Sahami M (1996) Toward optimal feature selection. *Tech. rep, Stanford InfoLab*
40. Kumar M, Rath SK (2015) Classification of microarray using mapreduce based proximal support vector machine classifier. *Knowl Based Syst* 89:584–602
41. Langley P (1994) Selection of relevant features in machine learning. In: *Proceedings of the AAAI Fall symposium on relevance*. pp 1–5
42. LeCun Y, Bengio Y, Hinton G (2015) Deep learning. *Nature* 521(7553):436
43. Liu S, Mocanu DC, Pechenizkiy M (2019) Intrinsically sparse long short-term memory networks. *arXiv preprint arXiv:1901.09208*
44. Liu S, Mocanu DC, Pechenizkiy M (2019) On improving deep learning generalization with adaptive sparse connectivity. In: *ICML workshop on understanding and improving generalization in deep learning*
45. Louizos C, Welling M, Kingma DP (2017) Learning sparse neural networks through ℓ_1/ℓ_0 regularization. *arXiv preprint arXiv:1712.01312*
46. Meda-Campaña JA (2018) On the estimation and control of nonlinear systems with parametric uncertainties and noisy outputs. *IEEE Access* 6:31968–31973
47. Mejía-Lavalle M, Sucar E, Arroyo G (2006) Feature selection with a perceptron neural net. In: *Proceedings of the international workshop on feature selection for data mining*. pp 131–135
48. Michael H, Zhu SG (2018) To prune, or not to prune: exploring the efficacy of pruning for model compression. In: *International conference on learning representations workshop*
49. Min S, Lee B, Yoon S (2017) Deep learning in bioinformatics. *Brief Bioinform* 18(5):851–869
50. Mocanu DC, Ammar HB, Puig L, Eaton E, Liotta A (2017) Estimating 3d trajectories from 2d projections via disjunctive factored four-way conditional restricted boltzmann machines. *Pattern Recognit* 69:325–335
51. Mocanu DC, Mocanu E, Nguyen PH, Gibescu M, Liotta A (2016) A topological insight into restricted Boltzmann machines. *Mach Learn* 104(2–3):243–270
52. Mocanu DC, Mocanu E, Stone P, Nguyen PH, Gibescu M, Liotta A (2018) Scalable training of artificial neural networks with adaptive sparse connectivity inspired by network science. *Nat Commun* 9(1):2383
53. Mocanu DC, et al (2017) *Network computations in artificial intelligence*. Technische Universiteit Eindhoven
54. Molchanov D, Ashukha A, Vetrov, D (2017) Variational dropout sparsifies deep neural networks. In: *Proceedings of the 34th international conference on machine learning*, vol 70. JMLR. org, pp 2498–2507

55. Mostafa H, Wang X (2019) Parameter efficient training of deep convolutional neural networks by dynamic sparse reparameterization. *arXiv preprint arXiv:1902.05967*
56. Mundra PA, Rajapakse JC (2009) Svm-rfe with mrmr filter for gene selection. *IEEE Trans Nanobiosci* 9(1):31–37
57. Narang S, Elsen E, Diamos G, Sengupta S (2017) Exploring sparsity in recurrent neural networks. *arXiv preprint arXiv:1704.05119*
58. Pessoa L (2014) Understanding brain networks and brain organization. *Phys Life Rev* 11(3):400–435
59. Plaku E, Kavvaki LE (2007) Distributed computation of the knn graph for large high-dimensional point sets. *J Parallel Distrib Comput* 67(3):346–359
60. Pohjalainen J, Räsänen O, Kadioglu S (2015) Feature selection methods and their combinations in high-dimensional classification of speaker likability, intelligibility and personality traits. *Comput Speech Lang* 29(1):145–171
61. Pudil P, Novovičová J, Kittler J (1994) Floating search methods in feature selection. *Pattern Recognit Lett* 15(11):1119–1125
62. Reunanen J (2003) Overfitting in making comparisons between variable selection methods. *J Mach Learn Res* 3(Mar):1371–1382
63. Romero E, Sopena JM (2008) Performing feature selection with multilayer perceptrons. *IEEE Trans Neural Netw* 19(3):431–441
64. Rumelhart DE, Hinton GE, Williams RJ (1986) Learning representations by back-propagating errors. *Nature* 323(6088):533
65. Simon R, Radmacher MD, Dobbin K, McShane LM (2003) Pitfalls in the use of dna microarray data for diagnostic and prognostic classification. *J Natl Cancer Inst* 95(1):14–18
66. Strogatz SH (2001) Exploring complex networks. *Nature* 410(6825):268
67. Taheri N, Nezamabadi-pour H (2014) A hybrid feature selection method for high-dimensional data. In: 2014 4th international conference on computer and knowledge engineering (ICCKE). IEEE, pp 141–145
68. Ullrich K, Meeds E, Welling M (2017) Soft weight-sharing for neural network compression. *arXiv preprint arXiv:1702.04008*
69. Wang J, Wei J, Yang Z (2016) Supervised feature selection by preserving class correlation. In: Proceedings of the 25th ACM international on conference on information and knowledge management. ACM, pp 1613–1622
70. Wen W, He Y, Rajbhandari S, Zhang M, Wang W, Liu F, Hu B, Chen Y, Li H (2017) Learning intrinsic sparse structures within long short-term memory. *arXiv preprint arXiv:1709.05027*
71. Yoon J, Yang E, Lee J, Hwang S.J (2017) Lifelong learning with dynamically expandable networks. *arXiv preprint arXiv:1708.01547*
72. Zhao Z, Morstatter F, Sharma S, Alelyani S, Anand A, Liu H (2010) Advancing feature selection research. In: ASU feature selection repository. pp 1–28
73. Zhu H, Jin Y (2019) Multi-objective evolutionary federated learning. *IEEE Trans Neural Netw Learn Syst* 31(4):1310–1322

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.