Consiglio Nazionale delle Ricerche

**Dynamic Hard Pruning of Neural Networks
at the Edge of the Internet**

L. Valerio, F. Nardini, A. Passarella, R. Perego

IIT TR-21/2020

Technical Report

Dicembre 2020

Istituto di Informatica e Telematica

Dynamic Hard Pruning of Neural Networks at the Edge of the Internet

Lorenzo Valerio[b,*], Franco Maria Nardini[a], Andrea Passarella[b], Raffaele Perego[a]

[a]*ISTI-CNR, via Moruzzi 1, 56124 Pisa, Italy*
[b]*IIT-CNR, via Moruzzi 1, 56124 Pisa, Italy*

Abstract

Neural Networks (NN), although successfully applied to several Artificial Intelligence tasks, are often unnecessarily over-parametrized. In fog/edge computing, this might make their training prohibitive on resource-constrained devices, contrasting with the current trend of decentralising intelligence from remote data-centres to local constrained devices. Therefore, we investigate the problem of training effective NN models on constrained devices having a fixed, potentially small, memory budget. We target techniques that are both resource-efficient and performance effective, while enabling significant network compression. Our technique, called Dynamic Hard Pruning (DynHP), incrementally prunes the network during training, identifying neurons that marginally contribute to the model accuracy. DynHP enables a tunable size reduction of the final neural network and reduces the NN memory occupancy during training. Freed memory is reused by a *dynamic batch sizing* approach to counterbalance the accuracy degradation caused by the hard pruning strategy, improving its convergence and effectiveness. We assess the performance of DynHP through reproducible experiments on two public datasets, comparing them against reference competitors. Results show that DynHP compresses a NN up to 10 times without significant performance drops (up to 5% relative error w.r.t. competitors), reducing up to 80% the training memory occupancy.

Keywords: Artificial neural networks, pruning, compression, resource-constrained devices.

1. Introduction

In the last years, AI solutions have been successful adopted in a variety of different tasks. Neural networks (NN) are one of the most successful technologies achieving state-of-the-art performance in several application fields including image recognition, computer vision, natural language processing and speech recognition. The main ingredients of NNs success are the increased availability of huge training datasets and the possibility of scaling their models to millions of parameters while allowing for a tractable optimization with mini-batch stochastic gradient descent (SGD), graphical processing units (GPUs) and parallel computing. Nevertheless,

---

*Corresponding author

Email addresses: `lorenzo.valerio@iit.cnr.it` (Lorenzo Valerio), `francomaria.nardini@isti.cnr.it` (Franco Maria Nardini), `andrea.passarella@iit.cnr.it` (Andrea Passarella), `raffaele.perego@isti.cnr.it` (Raffaele Perego)

NNs are characterized by several drawbacks and many research challenges are still open. Recently, it has been proven that NNs may suffer over-parametrization [1, 2, 3] so that they can be pruned significantly without any loss in accuracy. Moreover, they can easily over-fit and even memorize random patterns in the data [4] if not properly regularized.

NN solutions are typically designed having in mind large data centres with plenty of storage, computation and energy resources, where data are collected for training, and input data are also collected at inference time. This scenario might not fit emerging application areas enabled by the widespread diffusion of IoT devices, commonly referred to as *fog computing environments*. Typical application areas are smart cities, autonomous vehicular networks, Industry 4.0, just to name a few. IoT devices in fog environments generate huge amounts of data that, for several reasons, might be impossible or impractical to move to remote data centers both for training and for inference. Typically, real-time or privacy/ownership constraints on data make such an approach unfeasible. Therefore, knowledge extraction needs to leverage distributed data collection and computing paradigms, whereby NNs are "used" in locations closer to where data are generated, such as fog gateways or even individual devices such as tablets or Raspberry PIs. Unfortunately, with respect to data centers, these devices have much more limited capabilities in terms of computational power, memory, network and energy. In these contexts, the exploitation of models trained "offline" asks for a significant amount of memory – from hundreds of MBytes to GBytes – and their use, i.e., inference, requires GFLOPs of computation. As an example, the inference step using the AlexNet network [5] costs 729 MFLOPs (FLoating-point OPerations) and the model requires $\sim 250 MBytes$ of memory to be stored. Furthermore, resource-constrained devices should deal with limited energy availability (e.g., some of those devices might be battery powered). Interestingly, the energy consumption of such devices is dominated by memory access - a DRAM access costs two orders of magnitude more than a SRAM access and three orders of magnitude more than a CMOS add operation [6]. These limitations jeopardize the exploitation of large models trained "offline" in resource-constrained devices characterizing the edge/fog computing paradigms. Training DNNs on such devices is even more challenging, as the training phase is typically much more resource hungry than the inference phase.

Nowadays, several researchers are investigating the use of neural networks on resource-constrained devices. The effort is focused on enabling their use at both training and inference time on this kind of devices by limiting – or possibly totally avoiding – the loss in performance introduced by the reduced memory and computational capacity of the devices targeted. These approaches can be broadly classified in two research lines. On the one hand we have methods that, given an already trained neural network, try to reduce its size, or to distribute it on several devices collaborating during the inference phase. On the other hand, there are a few proposals that apply network compression techniques at training time. It is important to note that techniques following the latter approach typically measure the compression efficacy in terms of the average number of neurons (or weights) simultaneously active during the training. This kind of compression,

2

during an epoch, switches-off some neurons according to a given criterion. However, we have *soft* pruning techniques that allow neurons to be switched on again after they have been switched off. Conversely, this does not happen when *hard* pruning techniques are adopted: neurons that are switched off are lost. While soft pruning guarantees more flexibility during training, it does not reduce the memory occupation of the model during the training process, as information about switched-off neurons has to be always kept from epoch to epoch.

In this paper, we aim to design new hard pruning techniques for learning compressed neural networks on resource-constrained devices typical of edge/fog environments. We want to target techniques that are both resource-efficient and allow comparable levels of performance with respect to conventional neural network training algorithms while enabling significant levels of compression of the network. We target the goal by assuming that the training of a neural network in edge/fog devices relies on a fixed – and often small – budget of memory that can be used to perform the process. Our assumption comes from observing that the edge/fog paradigm is characterized by moving the computation close to the data source [7, 8]. In this scenario, the devices employed can perform many other operations in parallel w.r.t. training a neural network, e.g., operations related with data gathering/indexing/storing [9]. The edge/fog scenario is different from a standard cloud environment, where we can assume to have servers totally available to learn the neural network. We thus propose a new technique based on an effective compression of the network during the training process. Our novel technique, called *Dynamic Hard Pruning* (DynHP), prunes incrementally but permanently the network by identifying neurons that contribute only marginally to the model accuracy. By doing so, DynHP enables a significant and controllable reduction of the size of the final neural network learned. Moreover, by hard pruning the neural network we progressively reduce the memory occupied by the model as the training process progresses.

However, hard pruning neurons also brings a side effect related to the convergence of the training process to accurate solutions. Precisely, it slows down the convergence and makes stochastic gradient descent more susceptible of being stuck in poor local minima. To avoid this limitation, we capitalize on the increasing amount of memory saved during training to introduce a dynamic sizing of the mini-batches used to train the network. Our dynamic batch sizing technique enables a direct control on the convergence and the final effectiveness of the network by varying the amount of data seen in a batch. DynHP tunes the size of the batch dynamically, epoch by epoch, by computing its optimal size as a function of the variance of the gradients observed during the training. Moreover, we dynamically adjust the amount of data seen by considering the constraint on the total memory available for the training process. Our proposal thus effectively reuses the memory progressively saved with hard pruning to increase the size of batches used to estimate the gradient and improves convergence speed and quality.

The result achieved by DynHP is three-fold. First, the training of compressed neural networks can be done directly on resource-constrained edge/fog devices. Second, our technique minimizes the performance

3

loss due to network compression by reusing the memory saved for the network to increase the batch size and improve convergence and accuracy. Third, we explicitly target the training of neural networks under hard memory constraints by dynamically optimizing the learning process on the basis of the amount of memory available.

We assess our DynHP on two public datasets, MNIST and Fashion-MNIST, against several state-of-the-art competitors. Results of our reproducible experiments show that DynHP is able to effectively compress from 2.5 to 10 fold a DNN without any significant drops in effectiveness, i.e., the maximum relative misclassification error w.r.t. competitors is lower than 5%. Moreover, beyond obtaining highly accurate compressed models, our solutions prove to dramatically reduce the overall memory occupation during the entire training process, i.e., we use from $\sim 50\%$ to $\sim 80\%$ less memory to complete the training.

The paper is structured as follows: Section 2 discusses related work, while Section 3 presents the background of our work. Moreover, Section 4.1 introduces the hard pruning of neural network, while Section 4.2 presents the dynamic hard pruning technique. Section 5 presents and discusses the experimental results on two public datasets. Finally, Section 6 concludes the article.

2. Related Work

It is a known fact that neural networks models are most of the times over-parametrized, i.e., the number of the model's parameters that must be trained and the consequent complexity of the model overshoot the one actually needed by the problem at hand. Due to this fact, several methods have been proposed to reduce the size of a neural network model (i.e., lower the number of parameters). The challenge of all such methods is to find a way to obtain a final network model that has only the very necessary number of parameters needed to achieve the same accuracy of the "over-parametrized model". In the literature, three main directions have emerged along with we can categorize the algorithmic solutions that cope with such a problem.

The first category includes solutions that perform the compression *after* the training phase. Once the over-parametrized model is trained, these approaches exploit and/or manipulate it to obtain a smaller one with the same generalisation capabilities. One of the seminal work of this category is the so called *Knowledge Distillation method* [10, 11] that uses an already trained huge and complex Deep Neural Network (called teacher network) to guide the training of a smaller one (called student network) whose task is to learn to mimic the behaviour of the teacher network. Other approaches propose to prune useless or redundant connections of a Deep Network. Han *et al.* propose a pruning methodology in three steps: first they train the network, then they prune redundant weights, and finally they re-train the pruned network [1]. Similarly, Guo *et al.* propose a compression method made of two phases: pruning and splicing [12]. Briefly, in the pruning phase the connections deemed as useless are removed. However, to recover from possibly aggressive

pruning that cause sheer drop in the accuracy performance of the model, the authors propose a way to reactivate some of the connections that might be necessary to the generalisation capabilities of the network. Such an idea has been explored also in [13, 14]. All these techniques, despite the very impressing results both in terms of compression and accuracy, require several training phases through which the compression is incrementally done. Therefore, in terms of memory usage, the advantages becomes significant only when the final model is deployed and used, while the training phase is still quite resource demanding.

Alternatively, instead of pruning the connections of an already over-parametrized model, other approaches propose to act on the numerical representation of the weights. Note that these solutions can be applied to the network either *before* or *after* the training phase. Han *et al.* propose a three stage pipeline - *pruning, trained quantization and Huffman coding* - that work together to reduce the storage requirement of neural networks by 35x to 49x without affecting their accuracy [1]. Hubara *et al.* propose a method to train a Deep Network with binary weights and activation functions [15]. They are able to run the network 7x faster achieving *nearly* state of the art accuracy on relatively small datasets like MNIST and CIFAR-10. However, with more challenging datasets, like ImageNet, this approach does not obtain significant accuracies. Another work that follows the same principles proposes to use a fixed point quantization for representing the weights [16]. The main drawback is that it can be applied only to an already trained model. Similarly, [17] propose to use binary weights and a custom forward and backward pass through which they reach nearly state of the art accuracies on relative simple datasets (e.g. MNIST, CIFAR-10). Summarizing, most of these quantization/sparsification solutions are orthogonal to the one proposed in this paper because they can be applied, at least in principle, to our method to further reduce the memory occupation of the model at hand.

Finally, the third category includes solutions that compress the model *during* the training phase. According to the literature, the compression at training time is mainly based on weights pruning, which can be accomplished in two different ways. On the one hand there are works that prune single weights, obtaining a sparsification of the model. On the other hand, there are solutions that prune neurons by removing an entire row of the weights matrix. In the first line of research, Bellec *et al.* train directly a sparsely connected neural network and replace weights that reach zero with new random connections [18]. They experimentally show that the technique can be effective in learning sparse networks with a negligible loss in performance. In the same line, Narang *et al.* integrate magnitude-based pruning into training showing that the model size can be reduced by 90% with a speed-up from 2x to 7x [19]. Srinivas *et al.* and Louizos *et al.* learn gating variables with the aim at minimizing the number of nonzero parameters of the network [20, 21]. Recently, Lin *et al.* propose a model compression method that generates a sparse trained model without additional overhead [22]. The proposed technique works i) by allowing a dynamic allocation of the sparsity pattern and ii) by incorporating feedback signals to reactivate prematurely pruned weights. Results on CIFAR-10 show the effectiveness of the technique in learning effective sparse networks.

In the second line of research, Srinivas *et al.* rely on the fact that similar neurons are redundant. They compute the similarity of each pair of neurons and remove, one neuron at a time the most similar ones. [23]. Another structured approach is the one by Molchanov *et al.*, where authors propose a structured dropout technique to prune convolutional kernels in neural networks to make them more efficient [24]. On the same line, Neklyudov *et al.* propose a structured Bayesian dropout that allows to sparsify the network at training time [25]. Finally, Frankle *et al.* introduce the concept of *winning tickets*, discussing the existance of one or more small networks inside a bigger one that are sufficient to reach good accuracies with less parameters. The authors propose a method to discover such subnetworks. All these method have the advantage to produce a compressed and accurate network at the end of the training phase.

In the techniques above, only the work by Louizos *et al.* allows to reduce the memory size of a network at training time [21]. In fact, pruning the weights in a scattered way - as most methods do - is not useful for memory saving purposes at training time because of the way matrices are represented in memory. Conversely, pruning the neurons is much more effective since we can actually reduce the size of the weight matrix. In this paper, we build on top of the gating mechanism proposed by Louizos *et al.* [21]. Specifically, our novel technique, called Dynamic Hard Pruning (DynHP), prunes incrementally but permanently the network by identifying neurons that contribute only marginally to the model accuracy. By doing so, DynHP enables a significant reduction of the size of the final neural network learned. Moreover, by hard pruning the neural network we enable a significant reduction of the memory occupied by the model during the training process. The available memory is thus reused to minimizes the accuracy degradation caused by the hard pruning strategy by adaptively adjusting the size of the data provided to the neural network so to improve its convergence and final effectiveness.

3. Background

In this section we first introduce some useful notation used in the rest of the paper. We then discuss the state of the art technique for soft pruning neural networks during training that inspired our solution. For the sake of clarity, we report only the details that are necessary to make this paper self-contained.

3.1. Preliminaries

Let us now introduce the notation used hereafter and some important assumptions. We assume to have a dataset $\mathcal{D} = \{(x_i, y_i)\}_{i=1}^N \in \mathbb{R}^{N \times d+k}$ containing N i.i.d. d-dimensional observations $x_i \in \mathcal{X} \subseteq \mathbb{R}^d$, each one accompanied by a label $y_i \in \mathcal{Y} \subseteq \mathbb{R}^k$. Note that we target supervised learning problems with one or more labels per observation. The neural network model with weights $\boldsymbol{\omega}$ is denoted by the function $h : \mathbb{R}^d \to \mathbb{R}^k$. Additionally, in the following we refer to the set of neurons with the symbol $\boldsymbol{\theta}$. Let $\ell : \mathbb{R}^k \times \mathbb{R}^k \to \mathbb{R}$ be the loss function used to evaluate the prediction accuracy of the model h. The operator $\odot$ is the Hadamard

product, i.e., the element-wise product between vectors or matrices. Finally, let us denote with $\|\cdot\|_p$ the generic p-norm.

3.2. Soft Pruning of Neural Networks

Our solution is inspired by the technique proposed in [21], from now on referred to as *Soft Pruning* (SP). SP is based on the following *Regularized Empirical Risk minimization problem*:

$$\mathcal{L}(\boldsymbol{\omega}) = \frac{1}{N} \sum_{i=1}^{N} \ell(h(x_i; \boldsymbol{\omega}), y_i) + \lambda \|\boldsymbol{\omega}\|_0 \tag{1}$$

$$\boldsymbol{\omega}^* = \underset{\boldsymbol{\omega}}{\operatorname{argmin}} \, \mathcal{L}(\tilde{\boldsymbol{\omega}})$$

where the first component is the average loss of model h over the dataset and the second component is the norm L_0 acting as a regularizer tuned with the $0 < \lambda < 1$ parameter. Since the L_0-norm counts the number of non-zero parameters of the neural network, using it as a regularization term drives the learning algorithm towards solutions having a small number of connections different from zero. We note that as soon as *all* the weights of a neuron become equal to zero, the neuron itself could be pruned and the size of the network reduced. However, the above formulation does not provide any fine-grained control over the turning-off of all the weights belonging to individual neurons and the resulting pruning strategy is not particularly effective. A second drawback is that the L_0-norm of the weights is not a differentiable function, which prevents the straightforward usage of the any gradient-based optimization method to train the neural network.

To overcome the above problems, Louizos *et al.* propose to approximate the L_0-norm with an equivalent and differentiable function [21]. They propose a re-parametrization of $\boldsymbol{\omega}$ such that each parameter ω_j is defined as follows:

$$\omega_j = \tilde{\omega}_j z_j$$

where $z_j \in \{0, 1\}$ is a binary gate that controls the activation of the j-th parameter. Therefore the L_0-norm becomes:

$$\|\boldsymbol{\omega}\|_0 = \sum_{j=1}^{|\boldsymbol{\omega}|} z_j \tag{2}$$

The main intuition in [21] is to model the gates as Bernoulli random variables:

$$q(z_j|\pi_j) = \text{Bern}(\pi_j)$$

Precisely, each binary gate z_j has a probability π_j of being active. Adopting a probabilistic representation for the gates means that weights $\boldsymbol{\omega}$ and the loss values $\ell(,)$ become random variables. Therefore, the objective function in Equation 1 needs to become an average, as shown in Equation 3[1]:

$$\mathcal{L}(\tilde{\boldsymbol{\omega}}, \boldsymbol{\pi}) = \mathbb{E}_{q(\boldsymbol{z}|\boldsymbol{\omega})} \left[\frac{1}{N} \sum_{i=1}^{N} \ell(h(x_i; \tilde{\boldsymbol{\omega}} \odot \boldsymbol{z}), y_i) \right] + \lambda \sum_{j=1}^{|\boldsymbol{\omega}|} \pi_j \tag{3}$$

---

[1]For simplicity we omit most of the technicalities needed to obtain such formulation. The interested reader can find more information in [21] and [26].

$$\tilde{\boldsymbol{\omega}}^*, \boldsymbol{\pi}^* = \underset{\tilde{\boldsymbol{\omega}}, \boldsymbol{\pi}}{\operatorname{argmin}} \, \mathcal{L}(\tilde{\boldsymbol{\omega}}, \boldsymbol{\pi})$$

In this way the learning process is affected by the number of active binary gates and in particular, we are minimizing at the same time both the generalization error and the sum of the probabilities of the gates. However, in this form the function $\mathcal{L}(\tilde{\boldsymbol{\omega}}, \boldsymbol{\pi})$ is not yet suitable for efficient gradient computation due to the fact that the Bernoulli distribution is a discrete function and, consequently, it prevents the smoothness of $\mathcal{L}(\tilde{\boldsymbol{\omega}}, \boldsymbol{\pi})$. The solution proposed in [21] is to substitute the Bernoulli distribution used for modeling the gates, with the Hard Concrete Distribution [27] which is its continuous and differentiable approximation. In the following we denote its random variable as $\mathbf{s}$.

Briefly, let us define $\boldsymbol{z}$ as

$$\boldsymbol{z} = \min(1, \max(0, \boldsymbol{s}))$$

where $s \sim q(\boldsymbol{s}|\boldsymbol{\phi})$ and $q(\mathbf{s})$ is the distribution function of $\mathbf{s}$ with parameter ϕ. $\boldsymbol{z}$ is a random variable equal to zero whenever $\boldsymbol{s}$ is less than 0, equal to $\boldsymbol{s}$ when $0 < \boldsymbol{s} < 1$ and equal to 1 when $\boldsymbol{s}$ is larger than 1. In this way we can, on the one hand have gates that are exactly zero (this happens with the probability that $\boldsymbol{s}$ is less than 0) and on the other hand still compute the probability of a gate being active, as the probability that $\boldsymbol{s}$ is greater than 0:

$$P(\boldsymbol{z} > 0) = 1 - Q_{\mathbf{s}}(0|\boldsymbol{\phi}) \tag{4}$$

where $Q_{\mathbf{s}}()$ is the CDF of the Hard Concrete distribution. Combining Equations 3 and 4 we obtain the differentiable version of $\mathcal{L}(\tilde{\boldsymbol{\omega}}, \boldsymbol{\pi})$, denoted as $\mathcal{R}(\tilde{\boldsymbol{\omega}}, \boldsymbol{\phi})$:

$$\mathcal{R}(\tilde{\boldsymbol{\omega}}, \boldsymbol{\phi}) = \frac{1}{T} \sum_{l=1}^{T} \left(\frac{1}{N} \left(\sum_{i=1}^{N} \ell(h(x_i; \tilde{\boldsymbol{\omega}} \odot \boldsymbol{z}^{(t)}), y_i) \right) + \lambda \sum_{j=1}^{|\boldsymbol{\omega}|} \left(1 - Q_{s_j}(0|\phi_j) \right) \right) \tag{5}$$

Equation 5 assumes that training occurs over T epochs, and in each epoch a random draw for the activation gates is used. The first part of the equation is thus the average over the sample of the average losses obtained at each round, which is a standard estimator for the average value of the average loss. The second part is the average value of the number of non-zero gates. We minimize the objective function with respect to both $\tilde{\boldsymbol{\omega}}$ and $\boldsymbol{\phi}$, meaning that we learn, at the same time, the parameters ($\tilde{\boldsymbol{\omega}}$) and how many of them are, on average, really useful for the good predictions ($\boldsymbol{\phi}$).

The training of a neural network that implements this gating mechanism is performed as follows. At each input pattern in the mini-batch, a random set of neurons is switched off and a regular forward pass and a backward pass are performed. During the backward pass, the weights of the network are updated. This process continues until the end of the training phase. The effect of this double learning process is that during the training process the number of neurons actively contributing to the learning process is, on average, progressively decreasing due to the gates actively muting neurons epoch by epoch. This means that

at the end of the training, only a subset of neurons are actually responsible of the generalisation capabilities of the network. However, although the use of this technique allows to obtain a compressed version of the initial network maintaining almost the same generalisation performance, it is fundamental to point out that, during the training, such compression is just virtual. In fact, according to the gating mechanism described so far, the number of active neurons decreases only on average, meaning that in practice the neurons switched off in one epoch, might return active during the next one and they are not removed from the network.

This characteristic represents a major shortcoming for our purposes of devising a learning solution suitable for training models with limited memory footprints.

4. Dynamic Hard Pruning of Neural Networks

In this section we introduce and discuss our *Dynamic Hard Pruning* technique for training and compressing a neural network at a fixed memory budget. For the sake of clarity, we split the presentation of our solution in two parts. In the following, we first present the *hard pruning* technique that we use to *ablate* parts of the neural network during the training process. We then introduce our mechanism of *dynamic batch-sizing* that we use to drive the overall training process to achieve a significant reduction of the neural network with no loss in accuracy. The main idea is thus to remove less useful neurons at each epoch to save memory. As this may reduce accuracy, we exploit the freed memory within a dynamic batch size to optimize the performance of the pruned network.

4.1. Hard Pruning

Our *Hard Pruning* mechanism (HP) is based on a "one-way-only strategy" aimed at identifying and removing the less useful neurons together with all their inward and outward connections. Thanks to HP, the size of the network monotonically decreases as training progresses and we achieve a corresponding incremental reduction of the network's memory footprint. Differently from SP, the ablation of neurons performed in one epoch cannot be subsequently reverted. However, deleting permanently part of the network can be highly detrimental for its performance. Thus, to avoid disrupting the learning capabilities of the network, we have to identify very carefully the neurons whose removal does not harm, or harms only minimally, the quality of the training process. To this end, we exploit the weights gating mechanism discussed above and collect detailed statistics on gates activation during a fixed observation time window, e.g., a training epoch. At the end of each epoch we compute the average activation rate $a_j \in [0, 1]$ for each gate j, and we use such values to identify the least active neurons to be pruned. More formally, let $\boldsymbol{Z} = \{0, 1\}_{j=1}^{|\boldsymbol{\theta}|}$ be a vector that stores the binary information regarding the status (i.e., active/inactive) of each neuron of a layer l^2. Vector $\boldsymbol{Z}$ is updated at the end of each epoch to always record the active neurons

---

in the layer. Moreover, let $\boldsymbol{a}$ be a vector recording the activation rate for each neuron in the layer during an epoch. $\boldsymbol{a}$ is computed as follows:

$$\boldsymbol{a} = \frac{1}{E} \sum_{e=1}^{E} \boldsymbol{z}_e \tag{6}$$

where $\boldsymbol{z}$ is the number of times the random gates were active during an epoch and E is the total number of gate's state observations (i.e. active or inactive) during a training epoch. To identify the less active neurons in the layer we use a hard thresholding function $g(\cdot, \gamma)$ with fixed threshold γ:

$$g(x, \gamma) = \begin{cases} 1 & \text{if } x \geq \gamma \\ 0 & \text{otherwise} \end{cases} \tag{7}$$

By applying element-wise function $g(\cdot, \gamma)$ to vector $\boldsymbol{a}$ we obtain a binary vector

$$\hat{\boldsymbol{z}} = g(\boldsymbol{a}, \gamma)$$

that identifies the most active neurons in the layer. At the end of the epoch we use this information to update the status of the neurons stored in $\boldsymbol{Z}$:

$$\boldsymbol{Z} = \boldsymbol{Z} \odot \hat{\boldsymbol{z}}$$

In addition, vector $\boldsymbol{Z}$ is used to create a binary matrix M,

$$M = \boldsymbol{Z} \cdot \mathbb{1}^{T}$$

where $\mathbb{1}$ is a vector with all elements equal to 1 and the same dimension of the successive layer $l + 1$. M is finally used to set to zero all the weights corresponding to the deactivated neurons. This HP step is repeated at the end of each epoch for all the layers of the network.

4.2. Dynamic Hard Pruning

We now present our *Dynamic Hard Pruning* technique. As one might expect, our HP technique results in an effective reduction of the size of the neural network as training progresses but it has a significant impact on the convergence of the training process, possibly leading to a worse accuracy with respect to the SP technique, as we will show in Section 5.2.

The worse performance is mostly due to the interplay between the hard pruning and the training processes: while the SGD algorithm tries to reduce error, the HP step performed at each epoch may have the opposite effect of increasing the error due to the ablation of parts of the network. Moreover, an additional difficulty regards the relative speed of these two contrasting processes: if pruning is too fast due to an aggressive setting of the parameters involved, the network loses most of its expressiveness with the

10

consequence of converging to a poor model. Conversely, if the pruning is not incisive enough, the resulting model is still unnecessarily over-parametrised. Therefore, it is crucial to find a trade-off between these two different dimensions involved.

Several hyper-parameters drive the dynamics of the pruning and the learning process thus actively controlling the trade-off above. On the one hand, the pruning is controlled by the parameters λ and γ, i.e., λ regulates the aggressiveness of the pruning, γ defines the threshold used to identify the less active neurons of the network (Eq. 7). On the other hand, the learning dynamics are controlled by several parameters but two among all, i.e., learning rate and batch size, play a very important role. They affect directly the updates of the model during back-propagation actively driving the convergence of the learning and determining its final performance. To preserve the possibility of converging to good solutions with a network that progressively looses some of its parts, we adaptively reduce the learning rate to counterbalance the destabilizing effects of the hard pruning process. Alternatively, as it is shown in [28], it is possible to obtain the same generalisation performance by adaptively adjusting the size of the mini-batches during the epoch-by-epoch training.

In this paper, we exploit the equivalence above. We propose to *dynamically update* the size of the mini-batches during the training while keeping fixed the learning rate. By doing so, we obtain a double benefit because with a single parameter to tune (i.e., the growth rate of the mini batches, as we will explain in the following) we control both the speed of convergence and the total amount of memory used by the learning process. We dynamically regulate the batch size according to the *relative variance* (or Variance-To-Mean Ratio) of the gradients. Precisely, we link the size of the mini-batch to the amount of variability (or noise) contained in the gradients. The effect of using this index is to incrementally increase the size of the mini-batch, i.e., to allow for more precise estimations of the gradient, according to the amount of variance, i.e., instability, of the gradients. To compute this index, we use the algorithm proposed in [29] which gives a simple and effective way to estimate the variance of the gradient after each gradient computation.

Formally, let S be the variance estimation of the gradients for the current mini-batch and F the value of the loss function for the current mini-batch. At the end of each gradient computation, the new size b of the mini-batch is computed as:

$$b \quad = \quad b + \left\lfloor (1 - \alpha_{bs}) \frac{\|S\|_1}{F} \right\rfloor \tag{8}$$

where $\alpha_{bs}, \in [0, 1]$ is a smoothing parameter used to drive the batch size according to the observed variance on the gradient. Therefore, the parameter α_{bs} provided in Equation 8 is the only parameter employed in our method to drive the learning process by varying the batch size according to the stability of the gradient estimation. As mentioned before, Equation 8 shows that our batch size can only increase epoch by epoch by a quantity that is proportional to the variance of the gradient. The rationale is that in this way, we exploit the higher variance of the gradients to accelerate the convergence of SGD to better solutions, contrasting the

11

effect of hard pruning. Moreover, as the training evolves, the increasing mini-batches contribute to stabilize the training and to fine tune the network.

A clear effect of the introduction of Equation 8 is the fact that the batch size can indefinitely grow and, especially in resource-constrained devices, where memory is limited, this effect may harm the training of the neural network. To avoid this effect while keeping this approach exploitable in our fixed memory budget settings, we guarantee that DynHP does not exceed a given memory budget as follows. At the end of each training epoch, after having pruned the network, we compute the memory available for the growth of the i-th mini-batch size as the difference between the memory budget ($\mathcal{C}$) and the current memory occupation of the network $\mathcal{N}_i$.

$$\mathcal{B}_i = \mathcal{C} - \mathcal{N}_i \tag{9}$$

We impose the maximum increase of the mini-batch to be

$$\Delta\mathcal{B}_i = b_i + (\mathcal{C} - \mathcal{N}_i)$$

Denoting with $\tilde{b}_{i+1}$ the "candidate" size of the mini batch, computed through Equation 8, and taking into account the above maximum size limitation, we obtain that the mini-batch size at epoch $i+1$ is:

$$b_{i+1} = \min(\Delta\mathcal{B}_i, \tilde{b}_{i+1}). \tag{10}$$

Note that the network pruning is performed at the end of the epoch, therefore, at the beginning of the next epoch additional free memory might be available to grow the size of mini-batches, i.e., $\mathcal{B}_{i+1} \geq \mathcal{B}_i$ always holds.

5. Experiments

The experiments conducted aims to comprehensively evaluate our proposal and compare it with the state-of-the-art SP competitor [21]. More precisely, we are interested in answering the following research questions:

RQ1: To what extent our HP technique compresses the network and reduces over-parametrization? How much HP impacts on the quality of the learned model?

RQ2: Can we reduce the loss in accuracy introduced by HP by dynamically adjusting the size of mini-batches during the learning process? To what extent DynHP impacts on convergence speed and memory occupation?

5.1. Datasets and Competitors

We validate the performance of our method on the well-known task of multi-label image classification on two public datasets. In detail, we perform i) handwritten digit recognition on the MNIST dataset [30] and ii) fashion product classification on the Fashion-MNIST dataset [31]. Table 1 reports some salient statistics of the two datasets employed in this study. For the sake of comparison, in both tasks, we employ the same

Table 1: Properties of the two datasets employed.

Dataset	# Images		Image size (# pixels)	# channels
	Training Set	Test Set		
MNIST	60,000	10,000	28×28	1
Fashion-MNIST	60,000	10,000	28×28	1

network architecture, i.e., a Multi-Layer Perceptron (MLP) firstly employed by Louizos *et al.* [21]. The network consists of two hidden layers of size 300 and 100 and an output layer of 10 neurons to deal with multi-label classification[3]. Moreover, the input layer is composed of 784 neurons as each neuron encodes a pixel of an input image. We compare the performance of both HP and DynHP presented in Sections 4.1 and 4.2 against SP.

In both HP and DynHP, we set the parameter λ that controls the aggressiveness of the pruning process to 0.01, and we train each network for 2000 epochs. The value of λ is the one that best fits (found after a parameter search) for the chosen benchmarks for all the solutions considered in the paper: SP, HP and DynHP. Regarding the parameter α_{bs} we perform a sensitivity analysis, and we present the results in Sections 5.2 and 5.3. As far as the parameter γ is concerned, we set it to 0.50 as it is the outcome of the sensitivity analysis performed on it.

In the experimental results, we show the capabilities of HP and DynHP in saving memory compared to SP, and to what extent the pruning process affects the final model's accuracy.

Evaluation metrics. We measure the generalisation capabilities of the network models learned by using the misclassification error rate

$$ERR = \frac{1}{N_{te}} \sum_{i=1}^{N_{te}} I_{\hat{y}_i \neq y_i}$$

where N_{te} is the cardinality of the test set. Moreover, to understand the impact of each technique on Memory Occupation (MO), we analyse the size of the memory occupation during training. Precisely, we

---

[3]Our MLP network achieves comparable results with the publicly available ones on the Fashion-MNIST dataset. See `https://github.com/zalandoresearch/fashion-mnist`.

measure MO in Kilobytes (KB), computed as the sum of the of non-zero (Z) floating-point numbers of the weight matrices of each layer of the neural network plus the ones used to store a mini-batch of the data. In our implementations, we adopt a 32 bits representation for floating-point numbers.

5.2. Impact of Hard Pruning (RQ1)

We start the analysis of our hard pruning method by evaluating the impact that it has on the entire learning process. In this first set of results, we use both datasets to compare Hard Pruning and Soft Pruning on two aspects: i) the effectiveness of the pruning process and ii) the accuracy of the final solution.

We evaluate the effectiveness of the pruning process performing a layer-by-layer analysis of both methods, reporting the different behaviour of SP and HP in reducing the number of active neurons in Figure 1. Each plot shows the number of active neurons in each layer of the network along with the training. Precisely, the first row of plots (Figures 1a-c) refers to the MNIST dataset while the plots in Figures 1d-f refer to Fashion-MNIST. From left to right we show the first, second and third layer of the network, respectively. For this comparison, we report only the curves of the best operating configurations in terms of accuracy, as it will be clear from the results we show later on. Precisely, for SP the best configurations are those with batch size 512 for MNIST and 128 for Fashion-MNIST, for HP we set them to 100 and 128, respectively. Moreover, while in HP the number of active neurons corresponds to the actual size of the layers, in the case of SP no neurons are actually pruned, so the size of the network (corresponding to this memory occupation) is (much) higher than the number of active neurons, i.e. in SP the memory occupation during training remains constant. However, to better understand the mechanisms of both SP and HP, we plot for SP the average number of active neurons during the training.

For both datasets, SP starts with a smaller number of active neurons w.r.t. the ones available, e.g., 250 out of 300 in hidden layer 1 (HL-1) and $\sim$ 80 out of 100 in hidden layer 2 (HL-2) and during the initial training epochs this number increases. In this phase, SP explores several configurations reactivating previously muted neurons. Note that SP at some point reactivates almost all the neurons in HL-2. Then, it starts decreasing again until the end of the training epochs. This behaviour is due to how SP works: if the initial pruning is too aggressive, it adds back neurons to the network. It is worth noting that such a behaviour greatly differs between layers and datasets.

Conversely, this consideration does not hold for HP where the number of active neurons monotonically decreases in all the layers of the network. Depending on the specific dataset, the rate at which the pruning removes the unnecessary neurons changes but, still, we can recognize a clear pattern. Specifically, in HP once a neuron is removed, it cannot be added back to the network. Moreover, as it will be clear in Section 5.3, the pruning is, to some extent, connected to the convergence rate of the learning process. The monotonicity of the HP's pruning process let us shed light on an important difference between the two methods: while SP always uses all the memory for training the network, HP allows to reduce memory usage by *removing*
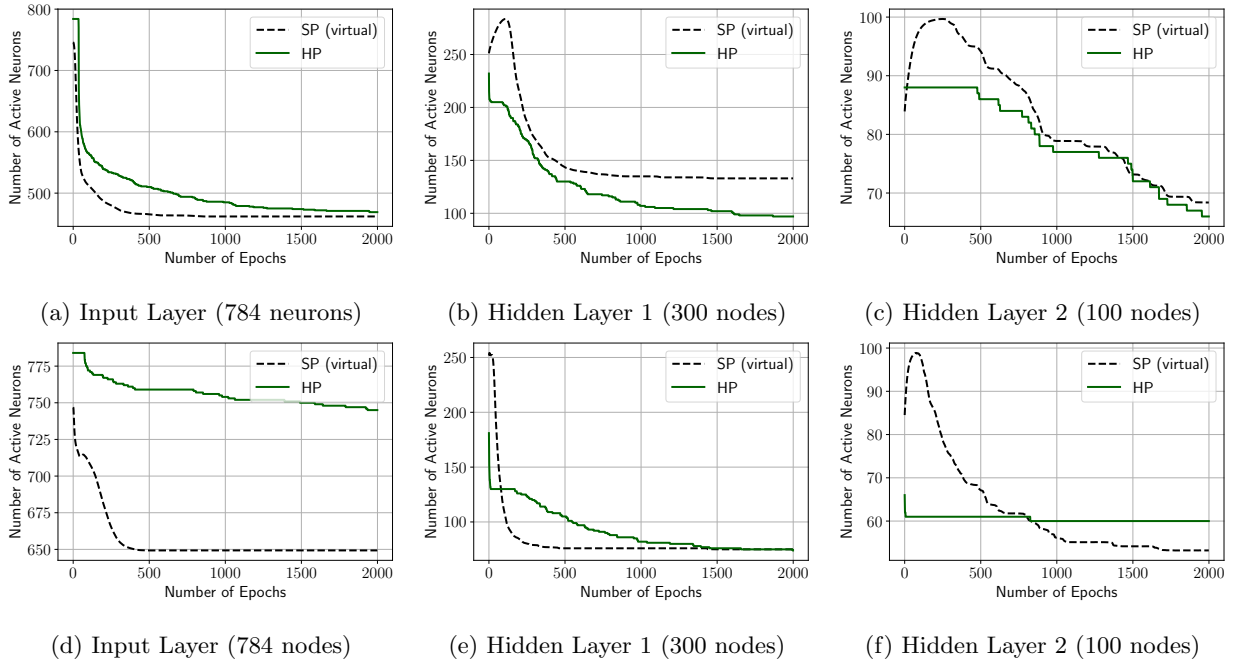
14

Figure 1: First row refers to MNIST, second row refers to Fashion-MNIST. The curves show the number of active neurons for SP and HP for each layer of the network, i.e., input and two hidden layers, by increasing the number of epochs of the training.

rows/columns of the weights matrix. We experimentally assess the effect above by analyzing the memory occupation of SP and HP during training. Figure 2 shows the total memory used during the 2000 training epochs, for both datasets.

Experiments show that HP, from the very beginning, uses 25% and 45% less memory than SP (solid black line) for MNIST and Fashion-MNIST, respectively. Conversely, SP always needs more than 1 MBytes of RAM to keep the network status. The overall memory savings at the end of the training is 80% and 78% for MNIST and Fashion-MNIST, respectively. It is worth noting that with SP we can obtain a network of the same size of the one obtained with HP only once the training is completed. Conversely, with HP we can progressively reduce the size of the network as training progresses. Interestingly, comparing SP (virtual) and HP on MNIST we notice that the final sizes of network trained with SP (virtual) and HP are comparable. Coupled with the accuracy results we discuss next, this proves that our method, epoch by epoch, is able to identify the unnecessary neurons correctly, possibly without harming the quality of the training process.

Let us now evaluate the accuracy of our method. We recall that although the HP mechanism leads to a remarkable saving of memory, it also forbids any possibility to rollback on the pruned neurons that might be needed to regain the learning power of the network "damaged" by the ablation. Hence, the following analysis compares HP and SP in terms of misclassification error to show the impact of HP on the quality of the final model learned. Figures 3 and 4 plot the test error as a function of the number of epochs for

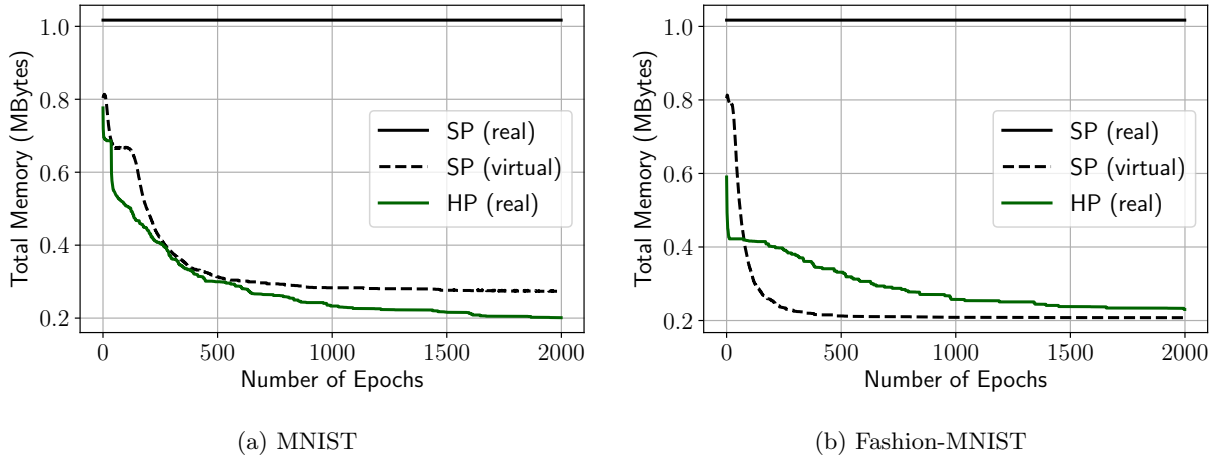|	
(a) MNIST	(b) Fashion-MNIST

Figure 2: Total memory (MBytes) used by SP and HP during the 2000 epochs of the training phase.

sizes of the mini-batches varying in the set $\{16, 32, 64, 100, 128, 256, 512\}$. First, we show in Figures 3a,4a, for both datasets, the performance of SP at different values of the batch size. As we can see - and as expected - the size of the mini-batch affects the performance of the network trained with SP. The reason is the interplay between the pruning process and the amount of information controlled by the size of the mini-batch (the larger the batch size, the more information are usable for a single update). SP obtains the best performance with batch size 512 for MNIST and 128 for Fashion-MNIST. The corresponding average misclassification errors are 1.40 and 9.96, respectively. As anticipated, we use those configurations as a reference benchmark for the rest of the comparisons. Interestingly, even if SP and HP use the same loss function, the two approaches react differently to the batch size. Looking at Figures 3b and 4b, we notice that the minimum error achieved by HP are 1.40% and 10.20% with a batch size of 100 and 128 for MNIST and Fashion-MNIST, respectively. The results show that SP performs better than HP in the presence of larger mini-batches, e.g., 512. This is an expected result: while SP works by only temporarily switching off some of the neurons during learning, i.e., one muted neuron can be reactivated if its contribution is important for the final prediction, HP switches off neurons by removing them permanently from the network[4]. Conversely, in HP the process of hard pruning provides some instability to the learning in a way that SGD performs better with less precise gradient updates (i.e. with more variance) to keep improving the solution. In fact, in terms of convergence speed, we see that in general, HP is slower than SP. Nevertheless, the solution achieved by HP in both cases are quite accurate and HP results to be only 4% (on average) less precise than SP despite the remarkable lower memory footprint.

---

[4]The curve for HP with batch size 512 is so high with respect to the others that it does not enter in the plot
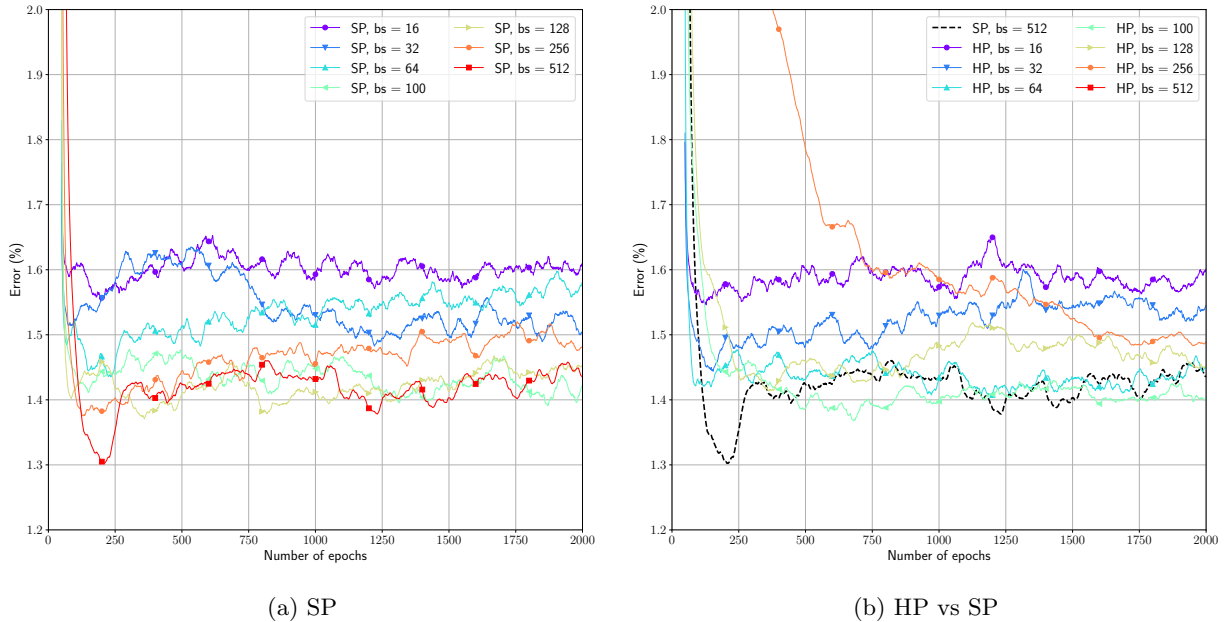
(a) SP	(b) HP vs SP

Figure 3: MNIST dataset. Misclassification error (%) of SP and HP for different sizes of mini-batches as a function of the number of epochs of the training.

5.3. Analysis of Dynamic Hard Pruning (RQ2)

From the results shown in Section 5.2, it appears clear that the size of the mini-batch is a crucial parameter which must be fine-tuned. However, in a resource-constrained scenario, where computational resources are to be spared, performing the fine-tuning of the batch size might be prohibitive since they would impose to train the network multiple times. In the following, we show the results obtained by the introduction of our mechanism for dynamically controlling the size of the mini-batches during training. In particular, we show that by dynamically adjusting the batch size, we can improve the effectiveness and convergence speed on HP. Moreover, we also evaluate the effect of such a dynamic mechanism on the amount of memory required to train the network.

To make a fair comparison between our DynHP and the other competitors, i.e., SP and HP, in which the size of the mini-batches is fixed a priori (tuned to their best performance) we adopt the following procedure. In Section 4.2, we described a mechanism that allows the training process of DynHP to not exceed a fixed memory budget. To this end, we instrument our code to monitor during network training the memory occupation of the network and the one of the mini-batch in the current epoch[5]. To fairly compare DynHP with SP, we set the maximum available memory budget of DynHP to the quantity of memory required by the best configuration of SP. In this way, we are sure that DynHP uses, at most, the same amount of memory resources used by SP.

---

[5] These quantities dominate the whole memory occupation of the training process.
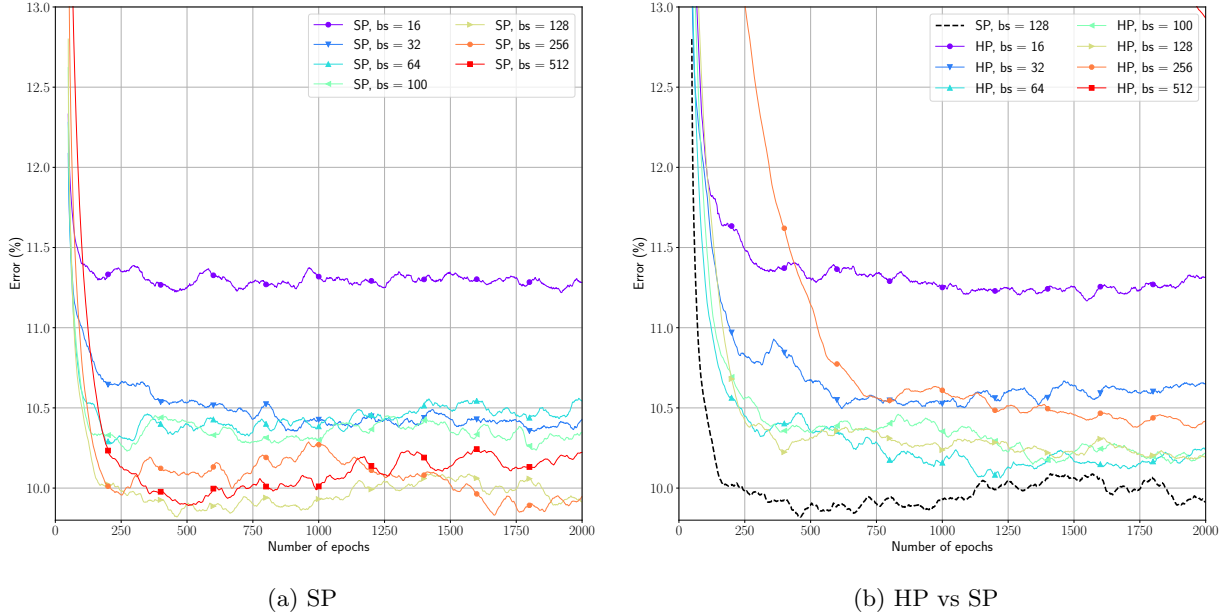
(a) SP

(b) HP vs SP

Figure 4: Fashion-MNIST dataset. Misclassification error (%) of SP and HP for different sizes of mini-batches as a function of the number of epochs of the training.

Tables 2 and 3 report for each method and configuration (i.e., batch size or α_{bs}) on the two datasets: i) the final misclassification error with, in parenthesis, the difference (%) of misclassification error with respect to SP: a negative difference of error indicates an accuracy improvement, while a positive one indicates an accuracy degradation, ii) the final size of the model with, in parenthesis, the percentage of space saved with respect to SP, and iii) the total memory occupation during the training computed as the area under the training-memory curve shown in Figures 7 with, in parenthesis, the percentage of memory savings w.r.t. SP based on the whole training epochs. Table 2 reveals that, with a proper tuning (batch size = 100), on MNIST data HP reaches a 88% space saving on the model and a reduction of memory used during training of up to 77% of memory while showing an improvement of performance of 1% w.r.t. the SP performance. Conversely, in the case of Fashion-MNIST (Table 3), we see that HP achieves a final 77% model reduction by using, during training, 52% less memory than SP. Also in this case, the relative accuracy degradation with respect SP is just 2%. This results can be considered an additional confirmation that neural networks are often over-parametrized and, thus, it is possible to significantly reduce their size sometimes without at a minimal cost in terms of accuracy. Let us focus on the results of DynHP where the size of the mini-batches is adapted according to the evolution of the training process. On the MNIST dataset (Table 2), it appears clear the existence of a *trade-off* between the final misclassification error and the final network size. In detail, we notice that incrementing α_{bs} the relative error shows a non-linear trend (i.e., it first decreases and then increases) that has its minimum at $\alpha_{bs} = 0.972$ where DynHP is 4% and 5% more accurate than HP and SP,

18

Table 2: Performance and impact on memory occupation for SP, HP and DynHP on the MNIST dataset. In parenthesis, we report the difference (%) in terms of misclassification error achieved by HP/DynHP with respect to SP. We also report the space saved (%) in terms of final model size and memory used for the training achieved by HP/DynHP with respect to SP.

Method	α_{bs}	Misclassification Error (%)	Model Size (MBytes)	Training Memory (MBytes)
SP ($bs = 512$)	–	1.42 (–)	1.041 (–)	5.219 (–)
HP ($bs = 100$)	–	1.41 (-1%)	0.206 (-88%)	1.185 (-77%)
DynHP	0.971	1.50 ($+5\%$)	0.572 (-45%)	5.165 (-1%)
DynHP	0.972	1.35 (-5%)	0.398 (-62%)	2.736 (-48%)
DynHP	0.973	1.40 (-1%)	0.205 (-80%)	1.279 (-75%)
DynHP	0.974	1.45 ($+2\%$)	0.217 (-79%)	1.361 (-74%)
DynHP	0.975	1.43 ($+1\%$)	0.178 (-83%)	0.980 (-81%)

respectively. Regarding the relation between α_{bs} and the memory savings, there is a positive correlation, i.e., the higher is α_{bs}, the more memory we can save. This holds for both the compression of the model (Models Size column), which ranges from 45% ($\alpha_{bs} = 0.971$) to 83% ($\alpha_{bs} = 0.975$), and for the training memory occupation (Training Memory column) where DynHP saves up to 81% w.r.t. SP. Interestingly, the highest compression (85% at $\alpha_{bs} = 0.975$) does not lead to the best accuracy (1.35 at $\alpha_{bs} = 0.972$) which is obtained with a sensibly less compressed model (62%). On the Fashion-MNIST dataset (Table 3), we notice a different trend. First, DynHP with $\alpha_{bs} \in [0.979, 0.982]$ performs equally or even better than SP showing up to 3% of accuracy improvement with a model that is 60% smaller than SP ($\alpha_{bs} = 0.980$). However, the memory occupation during training (Training Memory column) is equivalent to SP. The reason of this behaviour relies in the fact that DynHP, to counterbalance the loss of the network expressiveness, increases the size of the mini-batches. Second, results for $\alpha_{bs} \in [0.987, 0.989]$ reveal that under the condition of accepting a small degradation of relative accuracy (from 1% to 5% w.r.t. SP), DynHP produces highly compressed models, i.e., up to 88% space saving with up to 79% of memory occupation during training.

5.3.1. Convergence speed and accuracy

We now go deeper into the analysis by evaluating the convergence speed and the accuracy of SP, HP and DynHP. Figure 5 shows the performance on the test set for SP, HP and DynHP. For the sake of clarity and comparison, for SP and HP we report in the plot only the settings resulting in the best performance, achieved by employing a batch size of 512 and 100, respectively. For DynHP, we report several lines for

Table 3: Accuracy performance and impact on memory occupation for SP, DynHP and HP on the Fashion-MNIST dataset. In parenthesis, we report the difference (%) in terms of misclassification error achieved by HP/DynHP with respect to SP. We also report the space saved (%) in terms of final model size and memory used for the training achieved by HP/DynHP with respect to SP.

Method	α_{bs}	Misclassification Error (%)	Model Size (%)	Training Memory (MBytes)
SP ($bs = 128$)	–	9.96 (–)	1.041 (–)	2.866 (–)
HP ($bs = 128$)	–	10.20 (+2%)	0.236 (−77%)	1.377 (−52%)
DynHP	0.979	9.97 (0%)	0.430 (−59%)	2.874 (0%)
DynHP	0.980	9.64 (−3%)	0.415 (−60%)	2.874 (0%)
DynHP	0.981	9.96 (0%)	0.418 (−60%)	2.874 (0%)
DynHP	0.982	9.77 (−2%)	0.399 (−62%)	2.874 (0%)
DynHP	0.983	10.21 (+2%)	0.407 (−61%)	2.872 (0%)
DynHP	0.984	10.12 (+2%)	0.405 (−61%)	2.872 (0%)
DynHP	0.985	10.06 (+1%)	0.410 (−61%)	2.872 (0%)
DynHP	0.986	9.93 (0%)	0.386 (−63%)	2.858 (0%)
DynHP	0.987	10.08 (+1%)	0.252 (−76%)	2.573 (−10%)
DynHP	0.988	10.23 (+3%)	0.135 (−87%)	0.829 (−71%)
DynHP	0.989	10.50 (+5%)	0.124 (−88%)	0.611 (−79%)

different values of the parameter α_{bs}, responsible for the dynamic increment of the batch size. According to the strategy explained in Section 4.2, we start training with a very small batch size of 16.

Looking at Figure 5 we notice that α_{bs} controls the trade-off between training convergence speed and model accuracy. In fact, looking at Figure 5a, low values of α_{bs} (e.g., $\alpha_{bs} = 0.971$) induce a quite slow convergence to medium quality solutions while with higher values of α_{bs} (e.g., $\alpha_{bs} = 0.975$) the results are comparable with SP and HP. Interestingly, for values in between (e.g., $\alpha_{bs} = 0.972$), DynHP converges to a higher quality solution, outperforming both SP and HP in terms of average misclassification error. As far as Fashion-MNIST is concerned, we notice that the converge speed of DynHP is in general slower that both SP and HP. However, though slower, DynHP shows that for some configurations it is able to out perform both HP and SP.

These results represent a first insight on the benefits obtained by the combination of dynamic mini-batch size and the hard-pruning. The take-home message is that by fine-tuning the growth rate for the size of mini-batches, we can not only counterbalance the loss of expressive power of the network due to the irreversible hard-pruning process but also to converge to better solutions than the ones obtained with SP and HP.
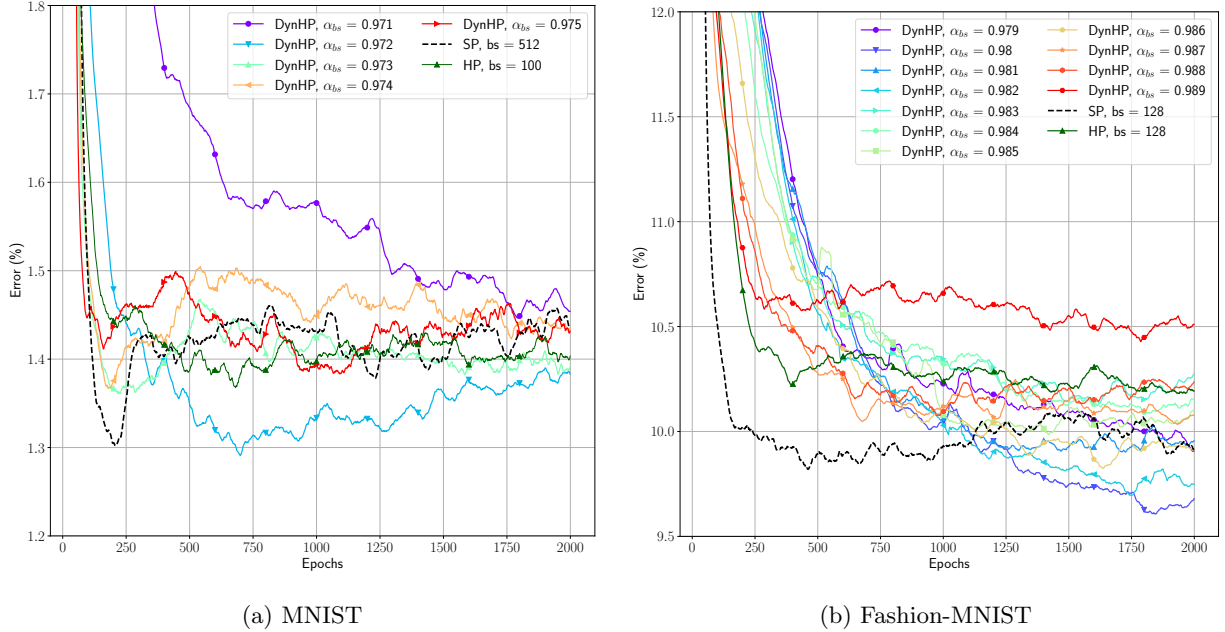

(a) MNIST

(b) Fashion-MNIST

Figure 5: Misclassification error (%) of DynHP, SP, HP as a function of the number of training epochs by varying α_{bs}.

5.3.2. Memory occupation

Let us now have a closer look at the dynamic batch sizing behaviour and its effect on the memory occupation of the entire learning process. In our configuration, we always start from a small initial batch size (i.e., $bs = 16$) and we increase it according to the amount of relative variance in the gradients. In Figure 6, we show the size of mini-batches as a function of the training epochs for different values of α_{bs}. The first interesting observation regards the fact that the growth of the mini-batches is a convergent process. In fact, for both datasets, we see that during training, almost all the curves stop increasing at different levels. Clearly, for each configuration of α_{bs}, there is a different size at which the mini-batch stabilizes. It is worth noting that there are two main reasons for which a curve stabilizes. On the one hand there is the effect induced by Equation 10 that forces the size of the mini-batches to not exceed the maximum memory budget. This happens for example for $\alpha_{bs} = 0.971$ in Figure 6a and $\alpha_{bs} = 0.987$ in Figure 6b. On the other hand, the curve stabilizes when the relative variance of the gradients computed in Equation (8) becomes negligible, e.g., as for $\alpha_{bs} = 0.972$ in Figure 6a and $\alpha_{bs} = 0.988$ in Figure 6b.
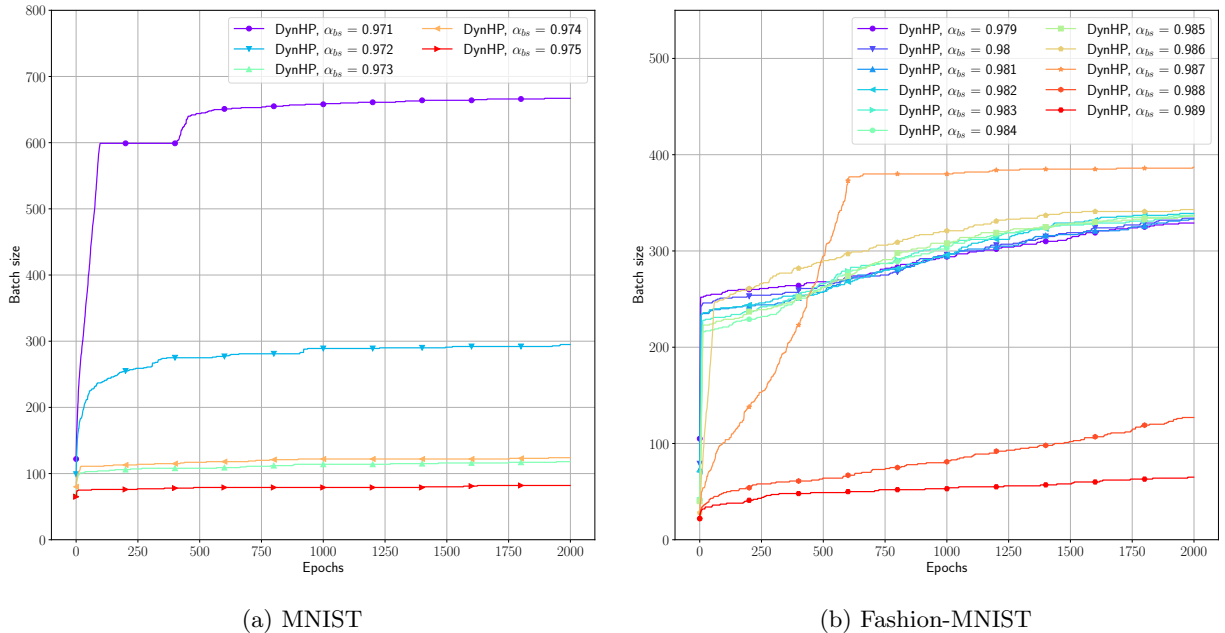
21

(a) MNIST

(b) Fashion-MNIST

Figure 6: Size of mini-batches as a function of training epochs for different values of the smoothing parameter α_{bs}.

The second interesting observation is that due to the definition of α_{bs} in Equation 8, the lower the value of α_{bs} the steeper the growth of the batch size. The effect of such steeper growth is compatible with the convergence performance shown in Figure 5 because, the larger the mini-batches, the slower the convergence[6]. However, there is no a priori notion of which is the correct steepness that drives the learning process to the best possible result. Our results suggest that the increment of the updates should be small.

Finally, in Figure 7 we plot the total memory occupation for DynHP, SP and HP that we measured during training epochs. For SP we show two curves: one is the real memory occupation, which is constant (solid black) and the second one is the virtual memory occupation (dashed black) corresponding to the average number of active neurons during a training epoch.

To have a clear understanding of the hard-pruning behaviour without the influence of the dynamic batch sizing, we can observe the HP curve, where all the memory reduction is due to the hard-pruning procedure. As we can see, the pruning process is stepwise, i.e., after a pruning session, there is a plateau during which the network recovers the loss of expressive power, followed by another pruning session and another plateau.

Regarding DynHP, we see that, with proper values of α_{bs}, it uses less memory than SP since higher values of α_{bs} limit the increment of the mini-batches. Moreover, we can appreciate how the dynamic sizing of the mini-batches combines with the HP procedure. Specifically, it is possible to distinguish when one of

---

[6]Recall that an update is based on the average of the gradients computed on a mini-batch. Therefore, averaging too many different gradients, as they are in during first training epochs, might drive SGD far away from good solutions, increasing the number of iterations needed to converge to better ones.

the two mechanisms dominates over the other one. In the first trait of each curve, the dynamic sizing of the mini-batches is predominant over the hard pruning because of the total memory occupation increases. Afterwards, we see that the HP mechanism causes a sheer drop of the total memory occupation, e.g., for MNIST around epoch 50 for DynHP with $\alpha_{bs} = 0.972$ and for Fashion-MNIST at epoch 100 for $\alpha_{bs} = 0.988$, followed by a modest but still constant pruning. Note that, the moment of the training at which this happens varies depending on the specific settings.

(a) MNIST

(b) Fashion-MNIST
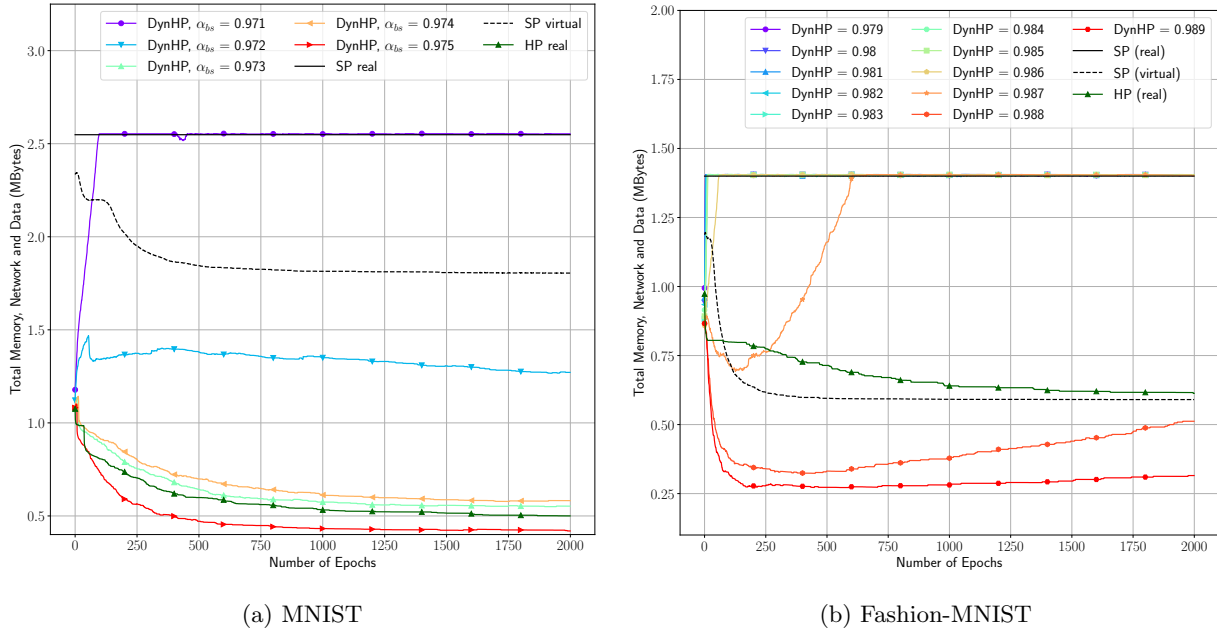
Figure 7: Total memory occupation for DynHP with different configurations, HP (best) and SP (best).

6. Conclusion

We investigated the problem of learning compressed NNs with a fixed - and potentially small - budget of memory on edge/fog devices. We proposed DynHP, a new resource-efficient NN learning technique achieving performances comparable to conventional neural network training algorithms while enabling significant levels of compression of the network. DynHP prunes incrementally but permanently the network as the training process progresses by identifying neurons that contribute only marginally to the accuracy of the model. The memory saved during training is effectively reused by a *dynamic batch sizing* technique that minimizes the accuracy degradation caused by the hard pruning strategy by adaptively adjusting the size of the data provided to the neural network so to improve its convergence and final effectiveness. The careful combination in DynHP of these two ingredients – hard pruning and dynamic batch sizing – allow us to train high-quality NN models by respecting the memory foortprint constraints introduced by the use of resource-constrained

edge/fog devices. The extensive experiments conducted on two public datasets show that DynHP is able to compress a NN from 2 to 10 times without significant performance drops – up to 5% of relative error w.r.t. state-of-the-art competitors – by reducing up to 80% the overall memory occupation during training.

CRediT author statement

Lorenzo Valerio and **Franco Maria Nardini**: Conceptualization, Methodology, Software, Validation, Writing - Original Draft. **Andrea Passarella** and **Raffaele Perego**: Conceptualization, Writing - Review & Editing.

References

[1] S. Han, H. Mao, W. J. Dally, Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding (2015) 1–14arXiv:1510.00149, doi:abs/1510.00149/1510.00149.
   URL http://arxiv.org/abs/1510.00149

[2] K. Ullrich, E. Meeds, M. Welling, Soft weight-sharing for neural network compression, arXiv preprint arXiv:1702.04008.

[3] D. Molchanov, A. Ashukha, D. Vetrov, Variational dropout sparsifies deep neural networks, in: Proceedings of the 34th International Conference on Machine Learning-Volume 70, JMLR.org, 2017, pp. 2498–2507.

[4] C. Zhang, S. Bengio, M. Hardt, B. Recht, O. Vinyals, Understanding deep learning requires rethinking generalization, arXiv preprint arXiv:1611.03530.

[5] A. Krizhevsky, I. Sutskever, G. E. Hinton, Imagenet classification with deep convolutional neural networks, in: Advances in neural information processing systems, 2012, pp. 1097–1105.

[6] M. Modarressi, H. Sarbazi-Azad, Chapter six - topology specialization for networks-on-chip in the dark silicon era, in: A. R. Hurson, H. Sarbazi-Azad (Eds.), Dark Silicon and Future On-chip Systems, Vol. 110 of Advances in Computers, Elsevier, 2018, pp. 217 – 258. doi:https://doi.org/10.1016/bs.adcom.2018.03.009.
   URL http://www.sciencedirect.com/science/article/pii/S0065245818300226

[7] P. Garcia Lopez, A. Montresor, D. Epema, A. Datta, T. Higashino, A. Iamnitchi, M. Barcellos, P. Felber, E. Riviere, Edge-centric computing: Vision and challenges, SIGCOMM Comput. Commun. Rev. 45 (5) (2015) 37–42. doi:10.1145/2831347.2831354.
   URL https://doi.org/10.1145/2831347.2831354

[8] M. Conti, A. Passarella, S. K. Das, The internet of people (iop): A new wave in pervasive mobile computing, Pervasive and Mobile Computing 41 (2017) 1 – 27. doi:https://doi.org/10.1016/j.pmcj.2017.07.009.
   URL http://www.sciencedirect.com/science/article/pii/S1574119217303723

[9] A. Barbalace, M. L. Karaoui, W. Wang, T. Xing, P. Olivier, B. Ravindran, Edge computing: The case for heterogeneous-isa container migration, in: Proceedings of the 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments, VEE '20, Association for Computing Machinery, New York, NY, USA, 2020, pp. 73–87. `doi:10.1145/3381052.3381321`.
URL `https://doi.org/10.1145/3381052.3381321`

[10] G. Hinton, O. Vinyals, J. Dean, Distilling the Knowledge in a Neural Network (2015) 1–9`arXiv:1503.02531`.
URL `http://arxiv.org/abs/1503.02531`

[11] L. J. Ba, R. Caruana, Do Deep Nets Really Need to be Deep? (2013) 1–9`arXiv:1312.6184`.
URL `http://arxiv.org/abs/1312.6184`

[12] Y. Guo, A. Yao, Y. Chen, Dynamic Network Surgery for Efficient DNNs, in: Intl. Conf. Neural Inf. Process., 2016. `arXiv:1608.04493`.
URL `http://arxiv.org/abs/1608.04493`

[13] S. Han, J. Pool, S. Narang, H. Mao, E. Gong, S. Tang, E. Elsen, P. Vajda, M. Paluri, J. Tran, B. Catanzaro, W. J. Dally, DSD: Dense-Sparse-Dense Training for Deep Neural Networks, in: Proc. ICLR, 2017. `arXiv:1607.04381`.
URL `http://arxiv.org/abs/1607.04381`

[14] X. Jin, X. Yuan, J. Feng, S. Yan, Training Skinny Deep Neural Networks with Iterative Hard Thresholding Methods (2016) 1–11`arXiv:1607.05423`.
URL `http://arxiv.org/abs/1607.05423`

[15] I. Hubara, D. Soudry, R. E. Yaniv, Binarized Neural Networks (Nips) (2016) 1–9. `arXiv:1602.02505`.
URL `http://arxiv.org/abs/1602.02505`

[16] D. D. Lin, S. S. Talathi, V. S. Annapureddy, F IXED P OINT Q UANTIZATION OF D EEP C ONVOLU -, in: ICLR, 2016.

[17] M. Courbariaux, Y. Bengio, J.-P. David, BinaryConnect: Training Deep Neural Networks with binary weights during propagations (2015) 1–9`arXiv:1511.00363`, `doi:arXiv:1412.7024`.
URL `http://arxiv.org/abs/1511.00363`

[18] G. Bellec, D. Kappel, W. Maass, R. Legenstein, Deep Rewiring: Training very sparse deep networks, 2017, pp. 1–24. `arXiv:1711.05136`.
URL `http://arxiv.org/abs/1711.05136`

[19] S. Narang, E. Elsen, G. Diamos, S. Sengupta, Exploring Sparsity in Recurrent Neural Networks, in: Proc. ICLR, 2017, pp. 1–10. `arXiv:1704.05119`.
URL `http://arxiv.org/abs/1704.05119`

[20] S. Srinivas, A. Subramanya, R. V. Babu, Training Sparse Neural Networks, IEEE Comput. Soc. Conf. Comput. Vis. Pattern Recognit. Work. 2017-July (2017) 455–462. `doi:10.1109/CVPRW.2017.61`.

[21] C. Louizos, M. Welling, D. P. Kingma, Learning Sparse Neural Networks through L_0 Regularization (2017) 1–13`arXiv:1712.01312`.
URL `http://arxiv.org/abs/1712.01312`

[22] T. Lin, S. U. Stich, L. Barba, D. Dmitriev, M. Jaggi, Dynamic model pruning with feedback, in: ICLR, OpenReview.net, 2020.

[23] S. Srinivas, R. V. Babu, Data-free parameter pruning for Deep Neural Networks (2015) 1–12`arXiv:1507.06149`.
URL `http://arxiv.org/abs/1507.06149`

[24] P. Molchanov, S. Tyree, T. Karras, T. Aila, J. Kautz, Pruning Convolutional Neural Networks for Resource Efficient Inference, Proc. ICLR (2017) 1–17`arXiv:1611.06440`.
URL `http://arxiv.org/abs/1611.06440`

[25] K. Neklyudov, D. Molchanov, A. Ashukha, D. Vetrov, Structured Bayesian Pruning via Log-Normal Multiplicative Noise, Intl. Conf. Neural Inf. Process. (Nips) (2017) 1–10. `arXiv:1705.07283`.
URL `http://arxiv.org/abs/1705.07283`

[26] D. P. Kingma, T. Salimans, M. Welling, Variational dropout and the local reparameterization trick, in: Nips '15, 2015, pp. 1–9. `arXiv:1506.02557`.

[27] C. J. Maddison, A. Mnih, Y. W. Teh, U. Kingdom, U. Kingdom, THe Concrete Distribution : a Continuous Relaxation of Discrete random variables, in: ICLR, 2017, pp. 1–17.

[28] S. L. Smith, P.-J. Kindermans, Q. V. Le, Don't decay the learning rate, increase the batch size, in: International Conference on Learning Representations, 2018.
URL `https://openreview.net/forum?id=B1Yy1BxCZ`

[29] L. Balles, J. Romero, P. Hennig, Coupling adaptive batch sizes with learning rates, in: Proceedings of the Thirty-Third Conference on Uncertainty in Artificial Intelligence, UAI 2017, Sydney, Australia, August 11-15, 2017, 2017.

[30] Y. LeCun, The mnist database of handwritten digits, http://yann. lecun. com/exdb/mnist/.

[31] H. Xiao, K. Rasul, R. Vollgraf, Fashion-mnist: a novel image dataset for benchmarking machine learning algorithms (2017). `arXiv:cs.LG/1708.07747`.