Pruning via Iterative Ranking of Sensitivity Statistics

Verdenius, S.; Stol, M.; Forré, P.

Publication date
2020
Document Version
Submitted manuscript

Pruning via Iterative Ranking of Sensitivity Statistics

Stijn Verdenius
University of Amsterdam
stijn@verdenius.com

Maarten Stol
BrainCreators B.V.
maarten.stol@braincreators.com

Patrick Forré
University of Amsterdam
p.d.forre@uva.nl

Abstract

With the introduction of SNIP [40], it has been demonstrated that modern neural networks can effectively be pruned before training. Yet, its sensitivity criterion has since been criticized for not propagating training signal properly or even disconnecting layers. As a remedy, GraSP [70] was introduced, compromising on simplicity. However, in this work we show that by applying the sensitivity criterion iteratively in smaller steps - still before training - we can improve its performance without difficult implementation. As such, we introduce *'SNIP-it'*. We then demonstrate how it can be applied for both structured and unstructured pruning, before and/or during training, therewith achieving state-of-the-art sparsity-performance trade-offs. That is, while already providing the computational benefits of pruning in the training process from the start. Furthermore, we evaluate our methods on robustness to overfitting, disconnection and adversarial attacks as well.

1 Introduction

In the last decade, deep learning has undergone a period of rapid development. Since AlexNet's revolution [35], advances have been noticeable on many fronts; for instance in generative image modeling [33, 21] and translation [69]. Be that as it may, deep learning also has some practical drawbacks. Amongst others, it is time-consuming, computationally expensive and data-hungry.

Literature has since suggested that modern neural networks are heavily overparametrised [12, 10, 3, 25], hypothesising that a great deal of the parameter-space that these networks are equipped with, is actually redundant. As such, the field of *model compression* suggests networks can be pruned off their redundant model components, with promises of reduced storage and computational effort. This can be formalised in the following way; assume the existence of an uncompressed network $f(x;\theta)$, that has a compressed counterpart $f(x;\theta')$. Then, the latter is produced by some compression-algorithm, parametrised by hyperparameters and subjected to a sparsity constraint, such that:

$$\begin{aligned} f(x;\theta') &\approx f(x;\theta) \quad \forall_x \\ \|\theta'\|_0 &\ll \|\theta\|_0 \end{aligned}$$

Its sparsity κ is then the fraction of zeroes in θ'. Practically, this is often achieved by multiplying the parameters with a mask $M \in \{0,1\}^{|\theta|}$. The pair of networks also share some set of desirable properties, in which values of these properties are typically assumed to be more favourable, or at least as good, for a successfully compressed network. These properties minimally constitute some trade-off between a performance- and sparsity metric but can include many more. Modern pruning techniques are quite successful at finding these sparse solutions and prune over 95% of the weights in a network, whilst leaving raw performance intact [24, 43, 18, 15, 75]. They often operate, by the 'train-prune-finetune' pipeline [25, 43, 47], which proceeds as follows; (a) take a pre-trained network and (b) prune some model-components, after which (c) some fine-tuning is performed [25, 43]. Although these remarkable sparsity rates may give the impression that the problem is already solved, the following three reasons exhibit why they actually provide a skewed perspective:

Preprint. Under review.

1. The 'train-prune-finetune' routine still requires training a dense network first. On the upside, the weights are already close to their final value this way. However, pruning efforts have no contribution to the training phase. Some methods fight this by pruning during training [48, 18, 15, 75], which reduces the problem somewhat. Preferably, we would even prune *before training*, so that we can exploit the sparse setting from the start. Unfortunately, this is a harder problem to solve. Recently, [40, 70] started tackling this problem, yet nevertheless only aimed at pruning individual weights, which brings us to the next point;

2. Pruning methods frequently only target individual weights - i.e. unstructured pruning, which is more flexible and results in higher sparsity. However, down the line this leads to sparse weight matrices which require formatting such as 'Compressed Sparse Row' (CSR) to be efficiently stored [6, 60] and only speeds up computation with dedicated libraries [23, 47]. Fortunately, research concerning *structured pruning* exists as well - i.e. pruning entire rows, columns or filters of a weight-tensor. Be that as it may, those methods are limited to pruning *during or after training* in the literature, making the benefits available for inference only.

3. Evaluation of these compressed sub-networks is usually performed along the dimensions of classifier-accuracy, computational cost and sparsity. However, this only validates a pruned solution shallowly. One can imagine, more aspects get affected in the process. For example, robustness to overfitting and adversarial attacks. Of course, some works do actually discuss these themes [9, 61, 40, 2]. Even so, it is still uncommon to new methods in the literature.

In the recently proposed 'Lottery Ticket Hypothesis' [16], it is established that networks contain sub-networks that are trainable in isolation, to the same performance as the parent-network - thereby, demonstrating that it would be possible to compress networks early-on. Subsequently, in SNIP [40], it was first endeavoured to actually prune at initialisation, demonstrating notable sparsity-rates using a simple 'sensitivity criterion' - without resorting to difficult training schemes. Practically, they employ the gradients obtained from a single batch to rank weights in their sensitivity criterion, cut-off at a desired sparsity κ and then train without further disturbance. Their criterion was thereafter criticised, due to unfaithful gradient propagation. To mitigate, orthogonal initialisation [39] and 'GraSP' [70] were suggested. The latter replaces the gradient-magnitude product of SNIP with a hessian-gradient-magnitude product for a better gradient approximation [70], yet compromises on simplicity. Our work builds upon the aforementioned research, examining structured- and unstructured pruning, before and during training. Our contributions are four-fold and can be summarised as follows:

1. We will introduce *'SNIP-it'*, a method that uses the same criterion as SNIP [40] but applies its sensitivity statistics iteratively in smaller steps, showing that this greatly improves performance and robustness in the high-sparsity regime, without compromising on simplicity.

2. Then, we will demonstrate how this can be applied to structured pruning as well, yielding a method - *'SNAP-it'* - that doesn't require a pre-trained network or complex training schedule to obtain a competitive structured pruning algorithm. As such, to the best of our knowledge, we are introducing the first structured pruning method that operates *before training*.

3. Additionally, we will remark how the sensitivity signal is proportional to function elasticity [65, 78] and discuss the ranking and pruning of weights and nodes in a combined fashion.

4. Finally, we will rigorously test and analyse our methods, on more than one dimension, and argue why pruning iteratively before or during training is effective at maintaining robustness.

2 Related work

Magnitude pruning The concept of pruning connections is not new and started with Hessian-based criterions; 'Optimal Brain Damage' [38] and 'Optimal Brain Surgeon' [26]. Yet, it really began to gain traction in recent years, after [25] demonstrated high compression rates. Consequently, research was performed into magnitude pruning, where they use the ℓ_1-norm as pruning criterion [25, 24, 43, 16]. Next, came the aforementioned Lottery Ticket Hypothesis, which demonstrated the existence of sparse sub-networks inside the randomly initialised networks, that could be trained to the same performance as the parent-network - coined 'winning lottery tickets' [16]. This theory sparked a body of research examining the behaviour and obtainment of these winning tickets [18, 76, 54, 13, 15, 79, 17, 50], as well as some criticism [47, 19].

Bayesian approach Concurrently, using a Bayesian framework, some methods used uncertainty approximation to determine the redundancy of model components [71, 66, 32, 67, 52, 48]. Related to that, are methods employing a 'sparsity inducing prior', where using a distribution over model components as extra regularisation term in the loss pushes the model to a sparser solution [8, 48, 46, 73]. Notable, are relaxed ℓ_0-regularisation [48] where they approximate a differentiable ℓ_0-norm and use it as additional penalty and DeepHoyer [73], where they instead use the ratio between ℓ_2- and ℓ_1-norm. Moreover, work was done into the usage of dropout as a means to pruning [32, 52, 20].

Pruning sparsely initialised networks In addition, there are methods which train sparsely initialised networks. One way, is to initialise with a sparse distribution and then train with the periodical interchanging of which parameters are considered pruned and which are 'grown back ' [4, 14, 15]. Hence, they don't train a final sparse setting from initialisation, which was later introduced in SNIP [40] and quickly followed by works that improve it [39, 70, 27, 41]. The main criticism for SNIP has been, that its criterion is not scale-invariant for certain layers, such that it substantially influences the criterion's signal or sometimes even disconnects layers [39]. Solutions to solve these problems include using orthogonal initialisation [39], GraSP [70] and are also the focus of this work.

Structured pruning Likewise to unstructured pruning, the beginning of structured pruning was derivative-based [55] and only really started gaining traction after the work of a magnitude based criterion [43]. Subsequently, other research focused on reconstructing output feature-maps under the strain of structured pruning, with the assumption that this implies maintaining performance [29, 49]. In other works, group-sparsity is used as sparsity inducing prior, with subsequent channel-pruning [46, 72, 1, 36, 74]. Most notable of them, is Network-slimming [46], where they impose a ℓ_1-penalty on scaling factors in batch-norm layers and thereby push neurons to deactivate during training. More recently, in GateDecorators [75], this idea is combined with a sensitivity-based criterion, alike was employed earlier in [53, 40, 55] and now also in this work. Moreover, some works perform single-shot node pruning with pre-trained networks [44, 77, 42]. Finally, there have been multiple papers where ADMM training is used, in which iteratively different layers get optimised in isolation [45, 44].

3 Methodology

Revisiting the sensitivity criterion We will adopt the sensitivity criterion that is applied in SNIP [40], albeit practiced in a different fashion. It is worth mentioning that this criterion is not unique to SNIP [55, 53, 75, 39]. As such, multiple derivations coexist. In [55, 40], authors commence by defining auxiliary gates c over model parameters. They then initialise all $c = 1$ and don't update them anymore. At which point, the criterion $sc(\cdot)$ is defined as the derivative of the loss w.r.t. the gates:

$$sc(\theta_{ij}) \;=\; \frac{\partial L(\mathscr{D} \mid \theta \odot c)}{\partial c_{ij}}\bigg|_{c=1}$$

They then rank the model components by said criterion and cut-off at the desired sparsity κ. Later in [39], it is recognised how for the unstructured case, via the chain rule, we obtain the derivative-magnitude product, rendering the auxiliary variables to be notational convention only:

$$\frac{\partial L(\theta \odot c)}{\partial c} \;=\; \frac{\partial L}{\partial (\theta \odot c)} \odot \frac{\partial (\theta \odot c)}{\partial c} \;=\; \frac{\partial L}{\partial \theta} \odot \theta \tag{1}$$

On the other hand, in [53, 75, 70], it is concurrently shown how this is likewise found by a local Taylor-expansion. Presently, we will further demonstrate how this is also proportional to the 'function elasticity' of the loss w.r.t. the parameters. This is a term which constitutes a mathematical way to express relative change in a function as a result of relative change in an input variable [65, 78]:

$$\varepsilon_y[x] \;=\; \frac{\partial y(x)}{\partial x}\frac{x}{y(x)} \tag{2}$$

We now proceed to combine Formulas 1 & 2 in this work, yielding the following proportionality:

$$\frac{\partial L}{\partial \theta} \odot \theta \;\propto\; \frac{\partial L}{\partial \theta} \odot \frac{\theta}{L} = \varepsilon_L[\theta] \tag{3}$$

Incidentally, since the loss L is the same for each model component θ_{ij} anyway, it would not matter for the purpose of ranking them from most to least important. In SNIP [40], they instead divide by the sum of saliencies, which produces the same ranking too. However, by rather dividing by the loss, we gain some interpretability, as a weight is 'very inelastic' when $\varepsilon_L[\theta_{ij}] \ll 1$, meaning the loss will not change after pruning that weight. When inspecting the distribution of elasticities, we always empirically observe that the majority of weights fall in this 'very-inelastic' category and only a small percentage don't (see Supplementary Material A). This confirms the aforementioned finding of overparametrisation in literature [12, 10, 3, 25] and also explains the high sparsity rates that state-of-the-art algorithms obtain. Moreover, we can now compare the sensitivity of weights and nodes on the same footing, meaning we can rank them together in one pruning event from our algorithm - SNIP-it - and prune regardless of structure-type. This then yields a very flexible pruning technique, which we have explored but did not become the main focus of the paper due to limited applicational value (we report some explorations in Supplementary Material B). Additionally, it suggests something about a possible procedure of evaluating the importance of grouped model-components, as we can simply introduce a gate variable over any set of them and then employ elasticity to evaluate importance.

Iterative pruning as remedy to disconnection In the works of [39] and [70], it is argued that the pitfall of vanilla SNIP [40] is that it stops information flow. Here, a new solution to this problem is suggested; iteratively applying the saliency criterion - with the corresponding method that is coined *'SNIP-it'*. The intuition goes as follows; certain medium-ranking model components, that are not that important to the loss initially, will be more important after the initial pruning-event. Specifically, we hypothesise the ranking may change, making specific model components more important in the sub-network than they were in the parent-network - i.e. the parent-network does not effectively dictate how information flows through the sub-network in isolation. By pruning more conservatively, in multiple rounds, we grant the model components another chance, therefore making the process more robust to the pruning algorithm and lowering the chance of disconnecting the network. Note that we still use the criterion from Formula 3 and that it can be applied both before or during training.

In practice, this materialises as dissecting the single pruning-event into a few pruning steps (s), that will be performed in one sequence. Its details are discussed in Algorithm 1. As an easy rule of thumb, we start with pruning half the network's weights for the first event, since we observe that generally more than half are nearly completely inelastic [Supplementary Material A]. Thereafter we keep halving the remainder until the desired sparsity is reached with some ε-proximity - after which we prune that final bit ε towards final sparsity κ_{final}. These specifics of the steps are not essential for the algorithm though. Accordingly, we do not experiment with different schedules. Instead we aim at a method that does not introduce too much complexity - i.e. it is the concept of iterative pruning that is important. That is not to say, the method cannot be improved by tuning this as well.

Algorithm 1 SNIP-it

procedure SNIP-IT(f, κ_{final}, τ, s) ▷ i.e. network f, sparsity κ, interval τ, steps s
  $\theta_0 \leftarrow$ choose what to prune from $f_{\text{weights}} \cup f_{\text{nodes}}$ ▷ structured, unstructured or combined
  for i $\in [0, s]$ **do** ▷ we default $s = 5$
    train τ epochs ▷ $\tau = 0$ for *before-training*, $\tau > 0$ for *during training*
    $\kappa_i \leftarrow \kappa_{final} - (\kappa_{final} - \frac{1}{2}) \cdot \frac{1}{2}^i$ ▷ rule of thumb: 50% first, then half of the remainder each time after
    $\theta_i \leftarrow$ rank θ with $\varepsilon_L[\theta_i]$ and prune to κ_i ▷ See Formulas 3 & 4
  $\theta'_{final} \leftarrow$ rank θ with $\varepsilon_L[\theta_s]$ and prune to κ_{final}
  start training $f(\theta'_{final})$

Note, that just like SNIP, we use one batch for each gradient evaluation in $\varepsilon_L[\theta]$. Granted, with more iterations, we will thus cumulatively see more data. Yet, this is hardly computationally expensive and each batch just constitutes an independent unbiased estimator after all [5, 59]. In fact, in the SNIP paper it is argued that if the algorithm is performed with a batch of sufficient size, then the multiplication of that size with a constant $d > 1$, does not constitute any noticeable difference [40]. In all experiments we performed, algorithms are always provided with an accumulated batch of 2560 samples, which we deem large enough to minimise its influence sufficiently.

Structured pruning before training Additionally, SNIP-it is easily supplemented with a structural equivalent. Generally, structured pruning methods substantially reduce computational time and feature maps. Combined with the benefit of pruning *before training*, we unlock a potential to drastically

reduce the total required compute. At the time of writing, to the best of our knowledge, no methods endeavoured structured pruning before training yet. A such, it is introduced here. Where unstructured SNIP-it uses weight-elasticity $\varepsilon_L[\theta]$ as a criterion, we now employ node-elasticity. If h is the activation function of a node, we thus straightforwardly produce $\varepsilon_L[h]$. However, since $h(\cdot)$ itself is a function and not actually a model parameter, we adopt the same notational convention applied in Skeletonization [55] and define auxiliary gates $c_j = 1$ over each node's input, thus substituting the activations $h(\cdot)$ for each layer $\ell \in P$ (see Formula 4), subsequently leading to criterion $sc(h_i^{(\ell)})$.

$$\begin{aligned} h^{(\ell)}\left(\Theta^{(\ell)} \cdot \vec{x}^{(\ell)} + \vec{b}^{(\ell)}\right) &= h^{(\ell)}\left(diag(\vec{c}^{(\ell)}) \cdot (\Theta^{(\ell)} \cdot \vec{x}^{(\ell)} + \vec{b}^{(\ell)})\right) \Big|_{\vec{c}^{(\ell)}=\vec{1}} \\ \varepsilon_L[h_i^{(\ell)}] \approx sc(h_i^{(\ell)}) &= \frac{\partial L(\mathscr{D} \mid \{\Theta^{(\ell)}\}_{\ell=1}^{P})}{\partial c_i^{(\ell)}} \cdot \frac{1}{L} \end{aligned} \tag{4}$$

Thereafter, the same procedure as unstructured SNIP-it is followed. Additional measures are taken to not prune away entire layers, as well as not pruning away the input and output nodes.

4 Experimental setup

Datasets and networks Experiments are evaluated with a number of different networks and datasets. The datasets used are publicly available and consist of three common benchmarks; MNIST [37], CIFAR10 [34] and an ImageNet-subset; 'Imagenette' [11, 30]. Networks that are used are LeNet5 [37], MLP5 [L], Conv6 [16], ResNet18 [28], AlexNet [35] and VGG16 [63]. Implementation and more network, data and augmentation specifics can be found in the Supplementary Material.

Initialisation As mentioned earlier, [39] showed the significance of initialisation in SNIP. Specifically, it is argued that orthogonal initialisation - although needed to be scaled by the activation function - is the best way to keep the magnitude of the signal comparable over different layers. For this reason, we adopt this initialisation in our experiments - unless a baseline has its own preferred initialisation. Note that this concerns initialisation before the network enters the pruning algorithm.

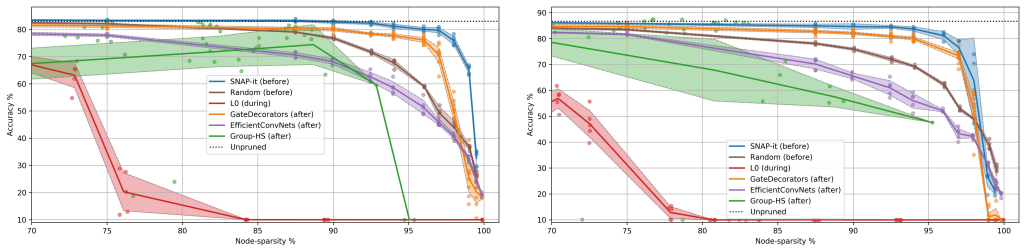
Baselines As baselines we use SNIP [40], GraSP [70], HoyerSquare [73] and global-IMP [16, 18] for unstructured pruning, whereas ℓ_0-regularisation [48], Group-HS [73], EfficientConvNets [43] and GateDecorators [75] for structured pruning. For all baselines hyperparameter tuning is performed - see Supplementary Material F for the tuning details - and implementation is sourced from authors' publications as much as possible, depending on availability and possibility of integration.

Optimisation For all experiments, the ADAM-optimiser [31] was used with a learning rate of $2 \cdot 10^{-3}$. No additional learning rate schedules were employed to avoid over-complication. Further, a weight decay of $5 \cdot 10^{-5}$ was used in all experiments. The batch-size in all CIFAR-10 and MNIST experiments was 512, whereas in Imagenette it is always 128. Finally, gradient clipping at a magnitude of 10.0 was applied as well. All methods have been given similar epoch budget, with an exception for ℓ_0-regularisation [48] and GateDecorators [75], which required twice that of the rest to converge.

Evaluation Evaluation is performed on sparsity, performance, connectivity and adversarial robustness. Additionally, in the structured domain; time, memory footprint and FLOPS are taken into account. The last is estimated with an implementation adopted from [47]. Furthermore, where applicable a confidence bound is displayed, which is calculated by: $1.96 \cdot \frac{\sigma}{\sqrt{n}}$, where n is the sample-size with a median value of 5, σ the standard deviation and 1.96 corresponds to a confidence interval of 95% - assuming the points are normally distributed. We evaluate in the structured and unstructured setting, as well as before and during training. To clearly distinguish SNIP-it's structured counterpart, we will refer to it as 'SNAP-it' from here forward. For metric details, see Supplementary Material H.

5 Results & analyses

Structured pruning before training We first evaluate SNAP-it in the domain with the most potential: structured pruning before training. We have assessed its during-training domain also, yet



(a) *AlexNet* (b) *VGG16*

Figure 1: *Structured sparsity-performance trade-off for CIFAR10 with confidence bound.*

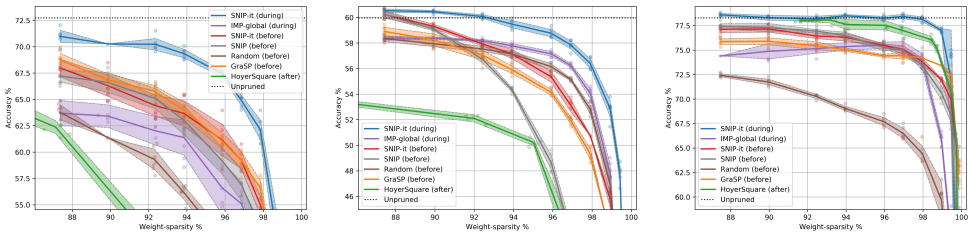
Table 1: *Structured results on Imagenette, showing optimal sparsity performance trade-off, as well as training-time, training-FLOPS and RAM-footprint for SNAP-it - compared to baselines. Points are picked using the highest harmonic mean between sparsity and accuracy.*

Method	Accuracy	Sparsity		FLOPS		Time		Storage	
		Weight	Node	Inference	Train	Inference	Train	Disk	RAM
VGG16									
Baseline	87%								
Prunes after training									
Effic.ConvN.	80% ± 2.5	92%	88%	49×	1 ×	**10**×	1 ×	6e1×	1 ×
GateDecorators	83% ± 1.8	96%	**93%**	47×	1 ×	5 ×	1 ×	1e2×	10×
Group-HS	84% ± 2.1	82%	68%	3 ×	1 ×	2 ×	1 ×	1e1×	1 ×
Prunes during training									
L0	28% ± 6.2	55%	58%	2 ×	1 ×	1 ×	1 ×	1e0×	1 ×
Prunes before training									
Random	82% ± 0.4	93%	90%	**76**×	**76**×	9 ×	**7** ×	1e2×	7 ×
SNAP-it (ours)	**85%** ± **1.4**	**97%**	**93%**	15×	15×	5 ×	4 ×	**1e3**×	**20**×
AlexNet									
Baseline	86%								
Prunes after training									
Effic.ConvN.	78% ± 1.9	92%	90%	**61**×	1 ×	**13**×	1 ×	4e1×	1 ×
GateDecorators	84% ± 2.5	93%	90%	14×	1 ×	5 ×	1 ×	2e1×	4 ×
Group-HS	84% ± 1.1	85%	78%	2 ×	1 ×	2 ×	1 ×	8e0×	1 ×
Prunes during training									
L0	31% ± 3.9	54%	58%	2 ×	1 ×	1 ×	1 ×	6e0×	1 ×
Prunes before training									
Random	83% ± 0.8	90%	88%	45×	**45**×	10×	**6** ×	2e1×	4 ×
SNAP-it (ours)	**85%** ± **0.9**	**98%**	**96%**	16×	16×	6 ×	5 ×	**8e2**×	**16**×

found it operates on-par with before training, so focused on the latter. It is illustrated in Figure 1, that SNAP-it outperforms baselines in terms of sparsity and performance, even though it prunes *before training*. Interestingly, Random forms a strong baseline. We speculate this is due to the nature of optimisation, where pruning *before training* gives the network enough chance to adapt, which again indicates how these networks are overparametrised. Additionally, we find that ℓ_0-regularisation [48] performs notably poor. To elaborate, we empirically observe that it performs better for the specific hyperparameter settings and architectures that were reported in their paper - as is portrayed in Figure 11 in Supplementary Material K.2. However, the method doesn't easily extend to new settings, even with the considerable tuning we performed, which we find defeats the purpose of an easy pruning method to begin with. Next, we witness that for any λ, Group-HS [73] is subject to high variance in both performance and sparsity, and collapses occasionally, which decreases its mean in Figure 1.

Moreover, we test SNAP-it on the Imagenette dataset, which is native to larger pictures. Results are found in Table 1. Again, we observe that SNAP-it reaches the best sparsity-performance trade-off and random forms a strong baseline. Granted, EfficientConvNets [43] and GateDecorators [75] still reach the best inference-FLOPS and inference-time reduction. However, when looking at training-FLOPS and training-time, SNAP-it has a significant advantage. Which, depending on the frequency a network needs to be retrained in practice, can make an extraordinary impact on the computational effort.

Equally promising, is the RAM-footprint reduction, which is also found in Table 1 as median reduction *during training*. However, when one is pruning *before training*, as SNAP-it does, we obtain this benefit immediately. Therefore, we can drastically lower the amount of feature maps



(a) *LeNet5* (b) *MLP5* (c) *ResNet18*

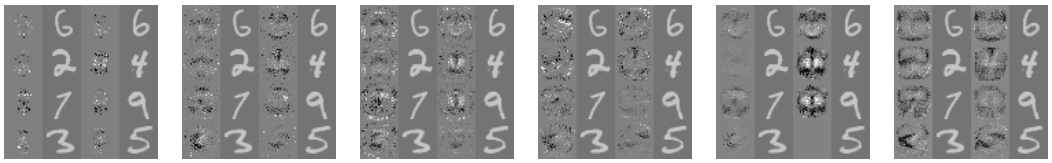
Figure 2: *Unstructured sparsity-performance trade-off for CIFAR10 with confidence bound.*

Table 2: *Unstructured sparsity-performance trade-offs, showing SNIP-it (during training) outperforms baselines. Points are picked using the highest harmonic mean (HM) between sparsity and accuracy.*

	CIFAR10											
	LeNet5			MLP5			Conv6			ResNet18		
	Accuracy	Sparsity	HM	Accuracy	Sparsity	HM	Accuracy	Sparsity	HM	Accuracy	Sparsity	HM
Baseline	73%			60%			87%			78%		
Prunes after training												
HoyerSquare	66% ± 1.4	82%	73	52% ± 0.3	92%	66	82% ± 0.6	89%	85	77% ± 0.5	97%	86
Prunes during training												
IMP-global	63% ± 1.3	90%	74	57% ± 0.2	**96%**	72	84% ± 0.8	92%	88	75% ± 0.9	97%	85
SNIP-it (ours)	**70% ± 0.7**	**92%**	**80**	**59% ± 0.4**	**96%**	**73**	**85% ± 0.4**	**97%**	**91**	**78% ± 0.3**	**98%**	**87**
Prunes before training												
GraSP	69% ± 0.8	87%	77	58% ± 0.6	90%	71	82% ± 0.5	96%	88	74% ± 0.7	**98%**	84
Random	64% ± 0.9	87%	74	57% ± 0.2	94%	71	82% ± 0.4	92%	87	70% ± 0.2	92%	80
SNIP	67% ± 1.2	90%	77	**59% ± 0.3**	87%	71	81% ± 0.9	94%	87	75% ± 1.0	97%	85
SNIP-it (ours)	68% ± 1.1	87%	76	**59% ± 0.2**	90%	71	83% ± 0.5	94%	88	75% ± 0.5	97%	85
	Imagenette											
	LeNet5			MLP5			Conv6			ResNet18		
	Accuracy	Sparsity	HM	Accuracy	Sparsity	HM	Accuracy	Sparsity	HM	Accuracy	Sparsity	HM
Baseline	83%			53%			87%			80%		
Prunes after training												
HoyerSquare	**83% ± 1.8**	92%	87	53% ± 0.9	90%	67	83% ± 1.2	97%	89	80% ± 0.7	95%	87
Prunes during training												
IMP-global	70% ± 1.7	**98%**	82	54% ± 0.7	97%	69	77% ± 3.0	97%	86	77% ± 1.4	98%	86
SNIP-it (ours)	**83% ± 2.0**	**98%**	**89**	56% ± 1.1	**99%**	**72**	**87% ± 1.1**	**99%**	**93**	**82% ± 0.8**	**99%**	**90**
Prunes before training												
GraSP	81% ± 0.8	**98%**	**89**	57% ± 0.9	94%	71	**87% ± 1.0**	98%	92	79% ± 0.9	**99%**	88
Random	78% ± 0.7	90%	84	54% ± 0.5	96%	69	85% ± 0.8	96%	90	79% ± 0.7	96%	87
SNIP	82% ± 1.1	**98%**	**89**	**58% ± 1.0**	97%	73	**87% ± 1.1**	98%	92	79% ± 1.0	98%	87
SNIP-it (ours)	81% ± 1.5	97%	88	**58% ± 0.9**	96%	**72**	86% ± 0.7	98%	92	80% ± 0.8	98%	88

we store, whereas the baselines require keeping all nodes with their corresponding feature maps in memory during training. Therefore, we gain the opportunity to proportionally increase the batch-size, which in turn allows us to increase the learning rate [22]. This observation is also reflected in Figure 8 in Supplementary Material C, where we see the trade-off between accuracy after training and RAM-footprint at the start of training. Note however, that RAM and time measurements can be influenced by processes playing at test-machines and should therefore be taken with some reservation.

Unstructured SNIP-it In addition, we evaluate SNIP-it in the unstructured domain, both before- and during-training. For this, we use $\tau = 0$ and $\tau = 4$ respectively (see Algorithm 1), where the latter was based on the hyperparameter tuning that was done for IMP [18], which we re-used to make SNIP-it apply the same schedule - see Supplementary Material F for details. In Table 2, results for various network-dataset combinations are displayed, with accuracy and sparsity as percentages. Adjacently, is the harmonic mean between the two (HM), which we designate as a proxy for the trade-off between them - just like the F1-score for information retrieval. Furthermore, sparsity-accuracy curves are shown in Figure 2. What follows from both, is that SNIP-it (during training) is significantly Pareto-optimal compared to all baselines on almost all setups. Moreover, SNIP-it (before training) significantly surpasses SNIP [40] and performs at least as good as GraSP [70], now and again outperforming it. Interestingly, in the case of an MLP5 random performs better than expected, which emphasizes the importance to test against a random baseline on different types of networks.



SNIP [40] *GraSP [70]* *SNIP-it (before)* *IMP-global [18]* *HoyerSquare [73]* *SNIP-it (during)*

Figure 3: *Saliency maps [62] (left) w.r.t. inputs (right), in MLP5 trained on MNIST at* $\kappa = 0.99$.

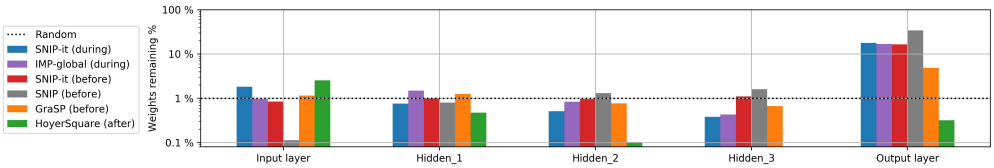
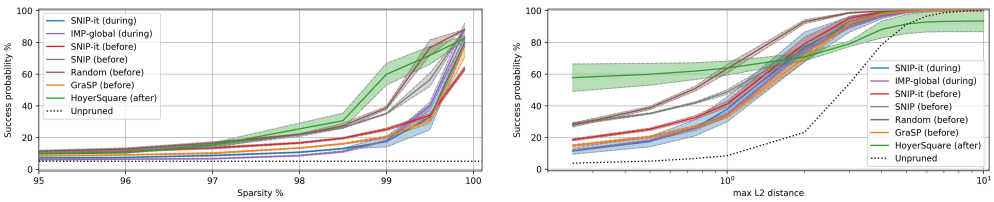


Figure 4: *Histogram of remaining weights in MLP5 trained on MNIST at* $\kappa = 0.99$.

Exploration on robustness If we zoom in on resulting architectures, findings correspond to our hypothesis, which said the ranking changes with each pruning event - introduced in Section 3. We display some saliency maps [62] with their respective inputs in Figure 3. Note that for the case of HoyerSquare [73] we had to interpolate to 99% sparsity from the closest point, due to its sparsity being harder to control. We observe that SNIP [40] overfits on only a few pixels, compared to GraSP [70] and SNIP-it (before training). Even better, are the saliency maps of HoyerSquare and SNIP-it *(during training)*, which hardly indicate any overfitting. These findings are again reflected in Figure 4, where we see that the different pruning methods favor different layers' weights as best candidates for pruning. As mentioned before, a common criticism against SNIP is its bias towards earlier layers [39, 70], which manifests itself again here. HoyerSquare [73] on the other hand, shows a bias towards pruning the output layer. If we assume the outermost two layers are the most important to maintain, we would expect to see a downward parabola-like shape in this figure, as that would prune heavier in the layers in the middle. This is the case for SNIP-it, in contrast to for instance; SNIP [40]. Thus, in accordance with our hypothesis, by applying the sensitivity criterion in iterations, weights in the earlier layers seem to be more important to intermediate pruning events, rather than SNIP's one-shot pruning of the full percentage, indicating the ranking changes between these intermediate events.

Consequently, we would also expect the methods that better conserve the input layer, to be more robust to adversarial attacks, since they don't over-fit on a few pixels [51]. Therefore, we subject the same setup to ℓ_2-CarliniWagner attacks [7] - using open-source library Foolbox [57]. Experiments have shown that these attacks, given unlimited ℓ_2-distance between the adversarial and real picture, approach a perfect success-rate [7, 58]. Therefore, we evaluate our setup at capped maximum distances. We discover that, for the low-distance regime where the ℓ_2-norm ≤ 1, the pruning methods which maintain input layer are indeed more robust in the face of these attacks. This is displayed on the left of Figure 5b. However, when we get to higher distances this does not hold. Generally, SNIP-it significantly improves in adversarial robustness from SNIP [40], yet performs on-par with some other baselines in the high sparsity regime (Figure 5a) and also in the low ℓ_2-distance regime (Figure 5b).



(a) *Success prob. vs. sparsity at* ℓ_2*-distance* ≤ 0.5. (b) *Success prob. vs. max* ℓ_2*-distance at* $\kappa = 0.99$.

Figure 5: *Robustness to* ℓ_2*-CarliniWagner attacks [7] as probability of success, for MLP5 on MNIST.*

6 Conclusion

In this work we demonstrated that applying the sensitivity criterion iteratively in a simple fashion creates a more robust and better performing algorithm in the high-sparsity regime, without adding supplemental complexity. With this, we achieved state-of-the-art sparsity-performance trade-offs. Within the unstructured domain, we further showed that our approach significantly improves overfitting and connectivity problems that were native to vanilla SNIP [40]. Moreover, we introduced a novel structured method, which operates *before training* and achieves significant speedups that can also be exploited for the training process itself. Furthermore, we related structured and unstructured pruning using the sensitivity criterion, showing we can now compare them on an equal footing. However, the latter we leave for future work to research.

Acknowledgments and Disclosure of Funding

This work was supported, funded and supervised by, *(a)* the University of Amsterdam, in cooperation with *(b)* the Amsterdam-based company BrainCreators B.V.

References

[1] J. M. Alvarez and M. Salzmann. Learning the number of neurons in deep networks. In D. D. Lee, M. Sugiyama, U. V. Luxburg, I. Guyon, and R. Garnett, editors, *Advances in Neural Information Processing Systems 29*, pages 2270–2278. Curran Associates, Inc., 2016.

[2] S. Arora, R. Ge, B. Neyshabur, and Y. Zhang. Stronger generalization bounds for deep nets via a compression approach. *arXiv e-prints*, page arXiv:1802.05296, Feb. 2018.

[3] J. Ba and R. Caruana. Do deep nets really need to be deep? In Z. Ghahramani, M. Welling, C. Cortes, N. D. Lawrence, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 27*, pages 2654–2662. Curran Associates, Inc., 2014.

[4] G. Bellec, D. Kappel, W. Maass, and R. Legenstein. Deep rewiring: Training very sparse deep networks. In *International Conference on Learning Representations*, 2018.

[5] L. Bottou, F. E. Curtis, and J. Nocedal. Optimization Methods for Large-Scale Machine Learning. *arXiv e-prints*, page arXiv:1606.04838, June 2016.

[6] A. Buluç, J. T. Fineman, M. Frigo, J. R. Gilbert, and C. E. Leiserson. Parallel sparse matrix-vector and matrix-transpose-vector multiplication using compressed sparse blocks. In *Proceedings of the twenty-first annual symposium on Parallelism in algorithms and architectures*, pages 233–244. ACM, 2009.

[7] N. Carlini and D. Wagner. Towards evaluating the robustness of neural networks. In *2017 IEEE symposium on security and privacy (sp)*, pages 39–57. IEEE, 2017.

[8] C. M. Carvalho, N. G. Polson, and J. G. Scott. Handling sparsity via the horseshoe. In *Artificial Intelligence and Statistics*, pages 73–80, 2009.

[9] J. Cosentino, F. Zaiter, D. Pei, and J. Zhu. The Search for Sparse, Robust Neural Networks. *arXiv e-prints*, page arXiv:1912.02386, Dec. 2019.

[10] Y. N. Dauphin and Y. Bengio. Big neural networks waste capacity. In *International Conference on Learning Representations*, 2013.

[11] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei. Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*, pages 248–255. Ieee, 2009.

[12] M. Denil, B. Shakibi, L. Dinh, M. A. Ranzato, and N. de Freitas. Predicting parameters in deep learning. In C. J. C. Burges, L. Bottou, M. Welling, Z. Ghahramani, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 26*, pages 2148–2156. Curran Associates, Inc., 2013.

[13] S. Desai, H. Zhan, and A. Aly. Evaluating lottery tickets under distributional shifts. In *Proceedings of the 2nd Workshop on Deep Learning Approaches for Low-Resource NLP (DeepLo 2019)*, pages 153–162, Hong Kong, China, Nov. 2019. Association for Computational Linguistics.

[14] T. Dettmers and L. Zettlemoyer. Sparse Networks from Scratch: Faster Training without Losing Performance. *arXiv e-prints*, page arXiv:1907.04840, July 2019.

[15] U. Evci, T. Gale, J. Menick, P. S. Castro, and E. Elsen. Rigging the Lottery: Making All Tickets Winners. *arXiv e-prints*, page arXiv:1911.11134, Nov. 2019.

[16] J. Frankle and M. Carbin. The lottery ticket hypothesis: Finding sparse, trainable neural networks. In *International Conference on Learning Representations*, 2019.

[17] J. Frankle, G. Karolina Dziugaite, D. M. Roy, and M. Carbin. Linear Mode Connectivity and the Lottery Ticket Hypothesis. *arXiv e-prints*, page arXiv:1912.05671, Dec. 2019.

[18] J. Frankle, G. Karolina Dziugaite, D. M. Roy, and M. Carbin. Stabilizing the Lottery Ticket Hypothesis. *arXiv e-prints*, page arXiv:1903.01611, Mar. 2019.

[19] T. Gale, E. Elsen, and S. Hooker. The State of Sparsity in Deep Neural Networks. *arXiv e-prints*, page arXiv:1902.09574, Feb. 2019.

[20] A. N. Gomez, I. Zhang, K. Swersky, Y. Gal, and G. E. Hinton. Learning sparse networks using targeted dropout. *CoRR*, abs/1905.13678, 2019.

[21] I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio. Generative adversarial nets. In Z. Ghahramani, M. Welling, C. Cortes, N. D. Lawrence, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 27*, pages 2672–2680. Curran Associates, Inc., 2014.

[22] P. Goyal, P. Dollár, R. Girshick, P. Noordhuis, L. Wesolowski, A. Kyrola, A. Tulloch, Y. Jia, and K. He. Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour. *arXiv e-prints*, page arXiv:1706.02677, June 2017.

[23] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, and W. J. Dally. Eie: efficient inference engine on compressed deep neural network. *ACM SIGARCH Computer Architecture News*, 44(3):243–254, 2016.

[24] S. Han, H. Mao, and W. J. Dally. Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding. *arXiv e-prints*, page arXiv:1510.00149, Oct. 2015.

[25] S. Han, J. Pool, J. Tran, and W. Dally. Learning both weights and connections for efficient neural network. In C. Cortes, N. D. Lawrence, D. D. Lee, M. Sugiyama, and R. Garnett, editors, *Advances in Neural Information Processing Systems 28*, pages 1135–1143. Curran Associates, Inc., 2015.

[26] B. Hassibi, D. G. Stork, and G. Wolff. Optimal brain surgeon: Extensions and performance comparisons. In J. D. Cowan, G. Tesauro, and J. Alspector, editors, *Advances in Neural Information Processing Systems 6*, pages 263–270. Morgan-Kaufmann, 1994.

[27] S. Hayou, J.-F. Ton, A. Doucet, and Y. W. Teh. Pruning untrained neural networks: Principles and analysis. *arXiv preprint arXiv:2002.08797*, 2020.

[28] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[29] Y. He, X. Zhang, and J. Sun. Channel pruning for accelerating very deep neural networks. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 1389–1397, 2017.

[30] J. Howard. Imagenette, 2019. Accessed April 1st, 2020 at `https://github.com/fastai/imagenette/`.

[31] D. P. Kingma and J. Ba. Adam: A Method for Stochastic Optimization. *arXiv e-prints*, page arXiv:1412.6980, Dec. 2014.

[32] D. P. Kingma, T. Salimans, and M. Welling. Variational dropout and the local reparameterization trick. In C. Cortes, N. D. Lawrence, D. D. Lee, M. Sugiyama, and R. Garnett, editors, *Advances in Neural Information Processing Systems 28*, pages 2575–2583. Curran Associates, Inc., 2015.

[33] D. P. Kingma and M. Welling. Auto-Encoding Variational Bayes. *arXiv e-prints*, page arXiv:1312.6114, Dec. 2013.

[34] A. Krizhevsky, G. Hinton, et al. Learning multiple layers of features from tiny images. 2009. CIFAR10 accessed May 26th, 2020 at `https://www.cs.toronto.edu/~kriz/cifar.html`.

[35] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 25*, pages 1097–1105. Curran Associates, Inc., 2012.

[36] V. Lebedev and V. Lempitsky. Fast convnets using group-wise brain damage. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 2554–2564, 2016.

[37] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998. MNIST accessed May 26th, 2020 at `http://yann.lecun.com/exdb/mnist/`.

[38] Y. LeCun, J. S. Denker, and S. A. Solla. Optimal brain damage. In D. S. Touretzky, editor, *Advances in Neural Information Processing Systems 2*, pages 598–605. Morgan-Kaufmann, 1990.

[39] N. Lee, T. Ajanthan, S. Gould, and P. H. S. Torr. A signal propagation perspective for pruning neural networks at initialization. In *International Conference on Learning Representations*, 2020.

[40] N. Lee, T. Ajanthan, and P. Torr. SNIP: Single-shot network pruning based on connection sensitivity. In *International Conference on Learning Representations*, 2019.

[41] N. Lee, P. H. Torr, and M. Jaggi. Data parallelism in training sparse neural networks. *arXiv preprint arXiv:2003.11316*, 2020.

[42] C. Li, Z. Wang, X. Wang, and H. Qi. Single-shot channel pruning based on alternating direction method of multipliers. *arXiv preprint arXiv:1902.06382*, 2019.

[43] H. Li, A. Kadav, I. Durdanovic, H. Samet, and H. P. Graf. Pruning Filters for Efficient ConvNets. *arXiv e-prints*, page arXiv:1608.08710, Aug. 2016.

[44] Z. Li, Y. Gong, X. Ma, S. Liu, M. Sun, Z. Zhan, Z. Kong, G. Yuan, and Y. Wang. Ss-auto: A single-shot, automatic structured weight pruning framework of dnns with ultra-high efficiency. *arXiv preprint arXiv:2001.08839*, 2020.

[45] N. Liu, X. Ma, Z. Xu, Y. Wang, J. Tang, and J. Ye. AutoCompress: An Automatic DNN Structured Pruning Framework for Ultra-High Compression Rates. *arXiv e-prints*, page arXiv:1907.03141, July 2019.

[46] Z. Liu, J. Li, Z. Shen, G. Huang, S. Yan, and C. Zhang. Learning efficient convolutional networks through network slimming. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 2736–2744, 2017.

[47] Z. Liu, M. Sun, T. Zhou, G. Huang, and T. Darrell. Rethinking the value of network pruning. In *International Conference on Learning Representations*, 2019.

[48] C. Louizos, M. Welling, and D. P. Kingma. Learning sparse neural networks through l_0 regularization. In *International Conference on Learning Representations*, 2018.

[49] J.-H. Luo, J. Wu, and W. Lin. Thinet: A filter level pruning method for deep neural network compression. In *Proceedings of the IEEE international conference on computer vision*, pages 5058–5066, 2017.

[50] E. Malach, G. Yehudai, S. Shalev-Shwartz, and O. Shamir. Proving the lottery ticket hypothesis: Pruning is all you need. *arXiv preprint arXiv:2002.00585*, 2020.

[51] A. Modas, S.-M. Moosavi-Dezfooli, and P. Frossard. Sparsefool: a few pixels make a big difference. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 9087–9096, 2019.

[52] D. Molchanov, A. Ashukha, and D. Vetrov. Variational dropout sparsifies deep neural networks. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pages 2498–2507. JMLR. org, 2017.

[53] P. Molchanov, S. Tyree, T. Karras, T. Aila, and J. Kautz. Pruning Convolutional Neural Networks for Resource Efficient Inference. *arXiv e-prints*, page arXiv:1611.06440, Nov. 2016.

[54] A. Morcos, H. Yu, M. Paganini, and Y. Tian. One ticket to win them all: generalizing lottery ticket initializations across datasets and optimizers. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d' Alche-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32*, pages 4932–4942. Curran Associates, Inc., 2019.

[55] M. C. Mozer and P. Smolensky. Skeletonization: A technique for trimming the fat from a network via relevance assessment. In D. S. Touretzky, editor, *Advances in Neural Information Processing Systems 1*, pages 107–115. Morgan-Kaufmann, 1989.

[56] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala. Pytorch: An imperative style, high-performance deep learning library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. dÁlché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32*, pages 8026–8037. Curran Associates, Inc., 2019. accessed May 29th, 2020 at `https://pytorch.org/docs/1.4.0/`.

[57] J. Rauber, W. Brendel, and M. Bethge. Foolbox: A python toolbox to benchmark the robustness of machine learning models. In *Reliable Machine Learning in the Wild Workshop, 34th International Conference on Machine Learning*, 2017.

[58] K. Ren, T. Zheng, Z. Qin, and X. Liu. Adversarial attacks and defenses in deep learning. *Engineering Journal Elsevier*, 6(3):346 – 360, 2020.

[59] H. Robbins and S. Monro. A stochastic approximation method. *The annals of mathematical statistics*, pages 400–407, 1951.

[60] Y. Saad. *Iterative Methods for Sparse Linear Systems*. Society for Industrial and Applied Mathematics, 2003.

[61] V. Sehwag, S. Wang, P. Mittal, and S. Jana. On pruning adversarially robust neural networks. *arXiv preprint arXiv:2002.10509*, 2020.

[62] K. Simonyan, A. Vedaldi, and A. Zisserman. Deep Inside Convolutional Networks: Visualising Image Classification Models and Saliency Maps. *arXiv e-prints*, page arXiv:1312.6034, Dec. 2013.

[63] K. Simonyan and A. Zisserman. Very Deep Convolutional Networks for Large-Scale Image Recognition. *arXiv e-prints*, page arXiv:1409.1556, Sept. 2014.

[64] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *The journal of machine learning research*, 15(1):1929–1958, 2014.

[65] K. Sydsaeter and P. J. Hammond. *Mathematics for economic analysis.* Pearson, 1995.

[66] M. E. Tipping. Sparse bayesian learning and the relevance vector machine. *Journal of machine learning research*, 1(Jun):211–244, 2001.

[67] K. Ullrich, E. Meeds, and M. Welling. Soft Weight-Sharing for Neural Network Compression. *arXiv e-prints*, page arXiv:1702.04008, Feb. 2017.

[68] G. Van Rossum and F. L. Drake. *Python 3 Reference Manual*. CreateSpace, Scotts Valley, CA, 2009.

[69] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. u. Kaiser, and I. Polosukhin. Attention is all you need. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems 30*, pages 5998–6008. Curran Associates, Inc., 2017.

[70] C. Wang, G. Zhang, and R. Grosse. Picking winning tickets before training by preserving gradient flow. In *International Conference on Learning Representations*, 2020.

[71] A. S. Weigend, D. E. Rumelhart, and B. A. Huberman. Generalization by weight-elimination with application to forecasting. In R. P. Lippmann, J. E. Moody, and D. S. Touretzky, editors, *Advances in Neural Information Processing Systems 3*, pages 875–882. Morgan-Kaufmann, 1991.

[72] W. Wen, C. Wu, Y. Wang, Y. Chen, and H. Li. Learning structured sparsity in deep neural networks. In D. D. Lee, M. Sugiyama, U. V. Luxburg, I. Guyon, and R. Garnett, editors, *Advances in Neural Information Processing Systems 29*, pages 2074–2082. Curran Associates, Inc., 2016.

[73] H. Yang, W. Wen, and H. Li. Deephoyer: Learning sparser neural network with differentiable scale-invariant sparsity measures. In *International Conference on Learning Representations*, 2020.

[74] J. Ye, X. Lu, Z. Lin, and J. Z. Wang. Rethinking the smaller-norm-less-informative assumption in channel pruning of convolution layers. In *International Conference on Learning Representations*, 2018.

[75] Z. You, K. Yan, J. Ye, M. Ma, and P. Wang. Gate decorator: Global filter pruning method for accelerating deep convolutional neural networks. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d' Alche-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32*, pages 2133–2144. Curran Associates, Inc., 2019.

[76] H. Yu, S. Edunov, Y. Tian, and A. S. Morcos. Playing the lottery with rewards and multiple languages: lottery tickets in rl and nlp. In *International Conference on Learning Representations*, 2020.

[77] J. Yu and T. Huang. AutoSlim: Towards One-Shot Architecture Search for Channel Numbers. *arXiv e-prints*, page arXiv:1903.11728, Mar. 2019.

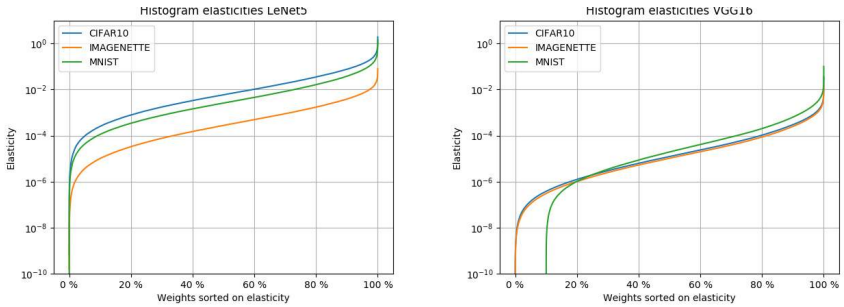
[78] V. Zelenyuk. A scale elasticity measure for directional distance function and its dual: Theory and dea estimation. *European Journal of Operational Research*, 228(3):592–600, 2013.

[79] H. Zhou, J. Lan, R. Liu, and J. Yosinski. Deconstructing lottery tickets: Zeros, signs, and the supermask. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d' Alche-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32*, pages 3597–3607. Curran Associates, Inc., 2019.

Supplementary Material

A Elasticity histograms

When examining the distribution of the elasticity signal that all parameters in a network receive, it is empirically observed that virtually all weights are in the inelastic category; a great deal of them are nearly completely inelastic, some are at most a little elastic and only very few approach unit-elasticity or higher $\varepsilon_L[\theta_{ij}] \geq 10^0$. Examples of these elasticities are displayed in log-scale in Figure 6. This is a pattern that is empirically found to re-emerge in every single network-dataset combination that has been evaluated in this research.

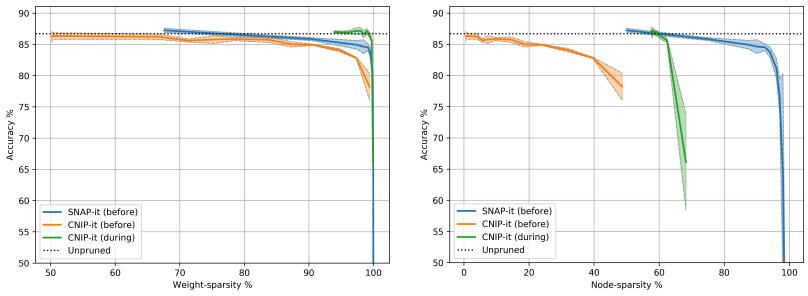


(a) *LeNet5* (b) *VGG16*

Figure 6: *Elasticities of loss function w.r.t. the network-weights at initialisation for different datasets. Elasticities are sorted like a Lorentz-curve (yet don't sum up to 1). Note that elasticities are displayed in log-scale.*

B Exploring combined structured and unstructured pruning

We explore a combined structured and unstructured approach. This would materialise as actually taking the union of both weights and nodes in Algorithm 1 on line 2 and subsequently evaluating and pruning them on an equal footing, with the elasticity framework we introduced. We will refer to this here as 'CNIP-it'.



(a) *Weight-sparsity* (b) *Node-sparsity*

Figure 7: *Weight-sparsity (left) and node-sparsity (right), versus performance of CNIP-it on VGG16 (CIFAR10) compared to our own method SNAP-it.*

We observe that CNIP-it reaches very high weight-sparsity levels without much loss in accuracy. However, the node sparsity is rather low. See Figure 7 for an illustration of this finding; CNIP-it has higher weight-sparsity than SNAP-it in Figure 7a, yet notable lower node-sparsity in Figure 7b. This is not at all unexpected as we presume most nodes are more important to the final loss than most individual weights. This finding may be desirable for some practical applications but it does not lead

to the same speedups as SNAP-it does or to the same weight-sparsity as SNIP-it does. Consequently, it is harder to compare to baselines, since combining the two structure-types was, to the best of our knowledge, not explicitly mentioned in literature. Therefore we concluded the limited applicational value was a ground to not make CNIP-it the main focus of the paper.

C Memory footprint prior to training

When one prunes structurally before training one gains memory and computational benefits from the start. This is reflected in Figure 8, where we unsurprisingly identify that the two methods that perform structured pruning before training, can substantially reduce their RAM-footprint before training, whereas the others still have the full network in memory at that point. As mentioned in the paper, this allows us to jointly increase the batch-size and learning rate in return [22].

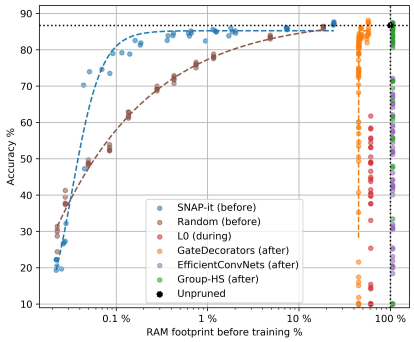


Figure 8: *Trade-off between RAM-footprint at the start of training and final performance for VGG16 on CIFAR10, with trend-line, showing the benefits on RAM for structured pruning before training.*

D Schematic illustration SNIP-it

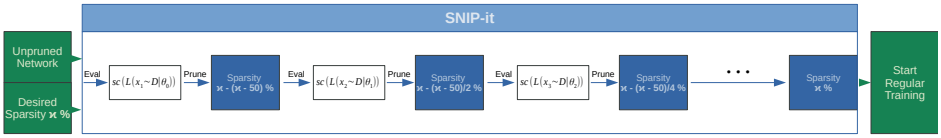


Figure 9: *Schematic illustration of SNIP-it, as discussed in Algorithm 1.*

E Documenting abandoned approaches

In order to prevent future research repeating work we already found to be ineffective, we will provide information on said findings. First of all, we pursued using GraSP's [70] criterion as a criterion for SNIP-it. This failed because when applied iteratively it tried to reactivate already pruned weights in a far greater proportion than it pruned the remaining weights. On the other hand, when the pruned weights were omitted from consideration, it resulted in disconnection. Moreover, we tried applying GraSP to a structured setting and found that, although it works, it doesn't yield favourable results. Additionally, we undertook experiments with SNAP-it (during training), but found it performed on-par with SNAP-it (before training), so we omitted the former as it delays the availability of the benefits of structured pruning to inference time, in contrast to the latter. Finally, we attempted to automatically estimate the optimal sparsity rate κ_{final} without tuning this parameter, so that it doesn't have to be set by a user and still have close to no drop in performance. For this we utilised a sorted histogram of the elasticity signal (see Figure 6). Although this was somewhat accurate, we couldn't find an exact enough point and had too many exceptions to justify discussing it in the paper. However, we encourage future research to examine this further.

F Hyperparameter tuning

All baselines were subject to considerable hyperparameter tuning. In its regard we will provide details in this section. All hyperparameters not discussed here are all kept on the recommended values from their native papers. For all tuning experiments we used a random held-out validation set, consisting of 20% of the training set. We don't perform tuning on non-pruning parameters that concern aspects like network or optimisation - e.g. learning rate or LeakyReLU-slope - but instead set these by hand.

- Firstly, IMP [18] has three tuneable parameters: the interval between pruning events τ, the rate of pruning κ_i and the rewind-epoch e_k. For the rate of pruning κ_i we employed the same rule of thumb as SNIP-it (Algorithm 1) to keep comparison as fair as possible. For the rewind-epoch e_k we used the suggested range of the paper: between 0.1% and 7% of convergence [18], which for us was an epoch in range $[0,6]$ and was tuned by grid-search. We found the optimum to be either epochs 5 or 6, with 6 being more frequent. Hence we settled on 6 and froze it for all experiments. The interval τ was jointly tuned by grid-search also. The original paper [16] claimed that rewinding should occur early in training, as 'winning tickets' can be found early in training. That, together with the desire to have the final sparsity as early as possible to allow for fine-tuning, led us to aim tuning for a lower value. We tuned in the range $[1,10]$ and found that it was optimal at 4 or 6. Hence, 4 was the lowest value where results were stable, which is the value we then also applied to SNIP-it (τ in Algorithm 1) to keep things equal as much as possible.
- Secondly, the baselines of ℓ_0-regularisation [48] and HoyerSquare [73] are both based on a regularisation parameter λ which usually requires tuning for the best sparsity-performance trade-off, instead of setting the preset sparsity rate κ like other methods - e.g. GraSP [70] or SNIP-it. However, we're actually exploring a range of sparsities for our experiments. Practically, we used 10 logarithmic intervals in the range $[10^{-6},5]$, which generated the data we worked with for these methods. So this is technically not tuning but part of the experiment.
- Thirdly, we manually tuned the threshold for HoyerSquare and Group-HS [73] with 7 logarithmic steps in the range $[10^{-5},10^{-2}]$ and finally settled on 10^{-3}. Even though the original paper recommended 10^{-4}, we found this worked better.
- Finally, for all methods that prune *after training* we train for 70 epochs, prune and thereafter perform fine-tuning for another 10 epochs. This was tuned by hand. Methods that are not in this category are trained for the same number of total epochs (80), except ℓ_0-regularisation [48] and GateDecorators [75], which required twice that to converge. Again tuned by hand.

G Implementation

All work was implemented in Python [68], using automatic differentiation library PyTorch [56]. Code is freely available at `https://github.com/StijnVerdenius/SNIP-it`

H Metrics

This section is dedicated to how the specifics metrics are calculated with the aim of reproducibility. They consist of two metrics that are directly measurable and two that are estimations. Said metrics are always only reported in the paper as a reduction w.r.t. the unpruned baseline and are calculated as follows:

- **FLOPS** are a proxy for computational effort. We have not directly measured these from the machines but instead estimated them. In literature multiple estimation methods coexist. We used the estimation technique from the implementation of [47], yet slightly altered for integration in our implementation.
- **Time** is a metric that can be measured directly. However, its measurements can depend on the host machine's capabilities, external processes running on the same machine, temperature and/or other unknown factors. We measured inference time on the GPU with PyTorch's CUDA time measurement tool [56] and training time as the total time of a run in milliseconds.

- **RAM-footprint** is how much memory the process takes up on the GPU. This effectively dictates the maximum combined model-size and batch-size that can be run at the same time. However, its measurements can depend on external processes as well. Furthermore, the epoch that measuring takes place can play a role. Hence, we document both point-measurements and median values over longer periods of time. We measured RAM on the GPU directly with PyTorch's CUDA memory measurement tool [56].
- **Disk storage** was estimated as the possible size when storing by CSR-format [6, 60]. Here, a sparse matrix is stored in three vectors: one with all nonzero entries and two with its indices. By considering only the nonzero entries and storing said indices in a lower precision we obtain compression. When we estimated the disk size we used a ratio of 16:1 for precision of the nonzero floats and the indices respectively. This is somewhat on the high-end, for more conservative estimations one could consider the ratios 8:1 or 4:1 as well.

I Infrastructure

All experiments were run on a distributed scheduling system with an array of different computers with different GPUs. The GPUs available to this system were GeForce RTX 2080 Ti, TITAN X, GeForce GTX 1080 Ti and Tesla V100-DGXS-16GB. Furthermore, the operating system for all was Ubuntu 18.04. All run-configurations were equally likely to end up at each machine. We are aware that this may cause higher variance in raw time measurement and therefore assert those are taken with reservation. However, we empirically observed it depended disproportionately more so on the network architecture and dataset used. Overall, experiments' run-times were observed to be between roughly a few minutes and five hours - depending on its setup.

J Datasets

Three different datasets were adopted to demonstrate the effectiveness of the methods described. A short description will follow of the dataset and how it was handled.

- **MNIST** [37] is employed as a baseline dataset for pruning algorithms, for exploratory parts of the research. It contains 60k training images of hand-written digits as well as 10k test images, all of 10 different classes. Images are of input dimensions $1 \times 28 \times 28$. The images were normalised, yet no additional data augmentation was performed.
- **CIFAR10** [34] is a classic dataset for pruning algorithms and was chosen as a default dataset to evaluate the algorithms on. It contains 50k images of different object-classes, as well as 10k test images. Images are of input dimensions $3 \times 32 \times 32$. The images were normalised. Moreover, data augmentation was applied to the training set, in the form of random horizontal flips with a probability of 20%.
- **Imagenette** [30] is a subset of images from the Imagenet challenge [11], freely available online [30]. It was chosen to additionally evaluate on a more realistic benchmark with larger pictures and less available data. It contains 10 classes with 13k training images and 500 test images. Images are preprocessed to have 3 valid channels and its shortest size resized to 320 and aspect ratio maintained. Thereafter, we scale them down to $3 \times 128 \times 128$ for computational reasons. The images were normalised. Moreover, training data augmentation was applied, again in the form of random horizontal flips with a probability of 20%.

K Supplementary results

Here we will report and display results that did not make it in the main paper due to spatial constraints. Sections are organised by the dataset or metric that was used to obtained the results that are displayed.

K.1 MNIST

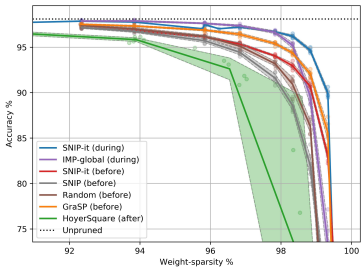


Figure 10: *Unstructured sparsity-performance trade-off with confidence bound (MLP5). This figure reports points corresponding to Figures 3, 4 & 5 in the main paper.*

Table 3: *Unstructured sparsity-performance trade-off. Points are picked using the highest harmonic mean (HM) between sparsity and accuracy. This table reports points corresponding to Figures 3, 4 & 5 in the main paper.*

	MNIST		
	MLP5		
	Accuracy	Sparsity	HM
Baseline	98%		
	Prunes after training		
HoyerSquare	96% ± 0.3	94%	95
	Prunes during training		
IMP-global	**97% ± 0.2**	**98%**	**97**
SNIP-it (ours)	**97% ± 0.1**	**98%**	**97**
	Prunes before training		
GraSP	96% ± 0.2	97%	96
Random	96% ± 0.2	96%	96
SNIP	96% ± 0.3	96%	96
SNIP-it (ours)	95% ± 0.2	97%	96

K.2 CIFAR10

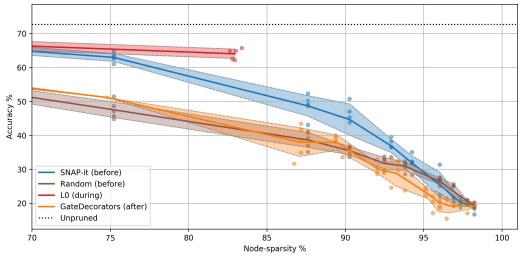


Figure 11: *Structured sparsity-performance trade-off with confidence bound (LeNet5), showing ℓ_0-regularisation [48] works well with recommended setup from their paper, unlike is observed in other setup-instances.*

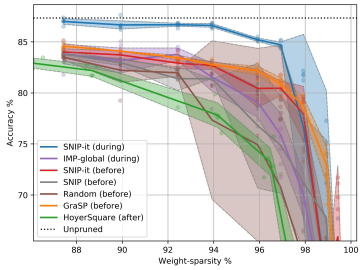


Figure 12: *Unstructured sparsity-performance trade-off with confidence bound (Conv6). This figure reports points corresponding to Table 2 in the main paper.*

Table 4: *Structured results on CIFAR10. Points are picked using the highest harmonic mean between sparsity and accuracy. This table reports points corresponding to Figure 1 in the main paper.*

Method	Accuracy	Sparsity		FLOPS		Time		Storage	
		Weight	Node	Inference	Train	Inference	Train	Disk	RAM
VGG16									
Baseline	87%								
Prunes after training									
Effic.ConvN.	82 %± 0.3	94%	75%	15×	1 ×	3 ×	1 ×	6e+02 ×	1 ×
GateDecorators	83 %± 1.0	**99%**	88%	**30**×	1 ×	5 ×	1 ×	3e+04 ×	2 ×
Group-HS	**85 %± 0.3**	96%	82%	3 ×	1 ×	3 ×	1 ×	2e+03 ×	1 ×
Prunes during training									
L0	57 %± 4.2	86%	70%	2 ×	1 ×	1 ×	1 ×	9e+01 ×	2 ×
Prunes before training									
Random	83 %± 0.4	94%	75%	15×	**15**×	**6** ×	**2** ×	6e+02 ×	21×
SNAP-it (ours)	**85 %± 0.6**	**99%**	**93%**	11×	11×	5 ×	**2** ×	**6e+04** ×	**156**×
AlexNet									
Baseline	83%								
Prunes after training									
Effic.ConvN.	78 %± 0.6	94%	75%	14×	1 ×	3 ×	1 ×	6e+02 ×	1 ×
GateDecorators	80 %± 0.9	**99%**	**90%**	25×	1 ×	6 ×	1 ×	**6e+04** ×	**205**×
Group-HS	80 %± 1.7	95%	87%	2 ×	1 ×	2 ×	1 ×	1e+03 ×	1 ×
Prunes during training									
L0	72 %± 1.2	88%	67%	2 ×	1 ×	1 ×	1 ×	1e+02 ×	2 ×
Prunes before training									
Random	79 %± 0.2	98%	88%	**50**×	**50**×	**8** ×	**2** ×	1e+04 ×	109×
SNAP-it (ours)	**83 %± 0.6**	**99%**	**90%**	9 ×	9 ×	5 ×	**2** ×	2e+04 ×	121×

K.3 Imagenette

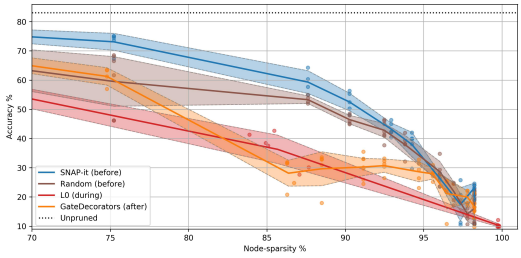
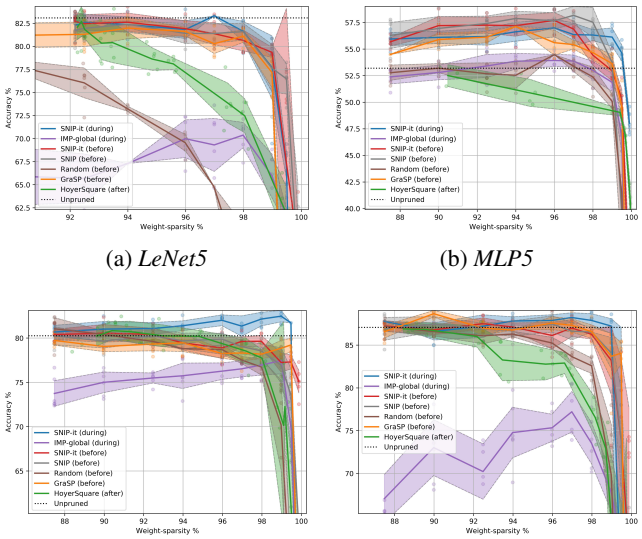


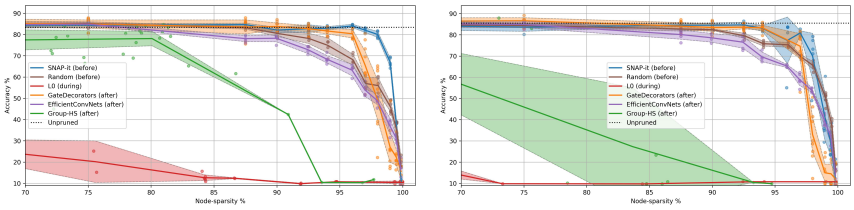
Figure 13: *Structured sparsity-performance trade-off with confidence bound (LeNet5).*



(a) *LeNet5* (b) *MLP5*

(c) *ResNet18* (d) *Conv6*

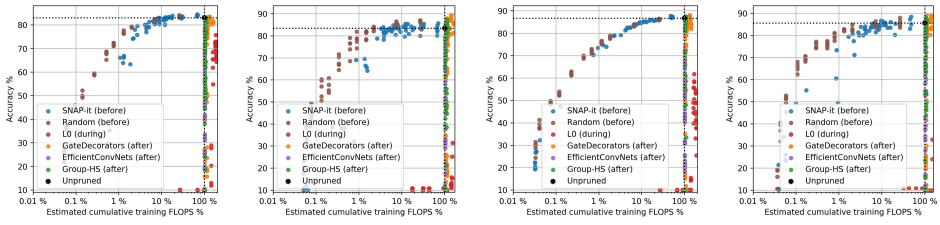
Figure 14: *Unstructured sparsity-performance trade-off with confidence bound. This figure reports points corresponding to Table 2 in the main paper.*



(a) *AlexNet* (b) *VGG16*

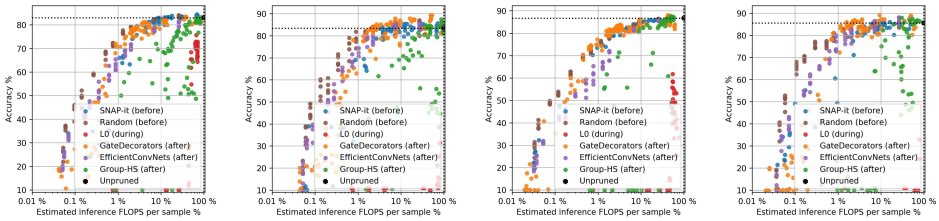
Figure 15: *Structured sparsity-performance trade-off with confidence bound. This figure reports points corresponding to Table 1 in the main paper.*

K.4 FLOPS



(a) *AlexNet-CIFAR10* (b) *AlexNet-Imagnette* (c) *VGG16-CIFAR10* (d) *VGG16-Imagnette*

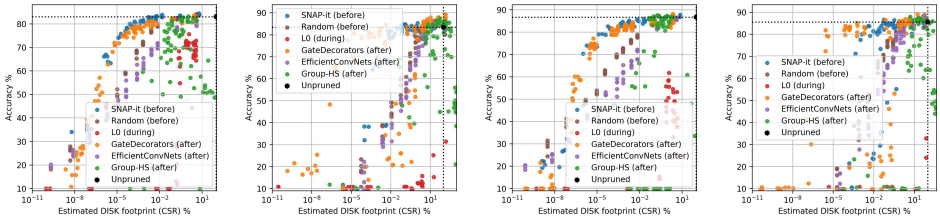
Figure 16: *Estimations of total cumulative FLOPS of the training process for a set number of epochs for multiple network-dataset combinations w.r.t. unpruned baseline. GateDecorators [75] and ℓ_0-regularisation [48] required a higher number of epochs and naturally have a higher total because of it.*



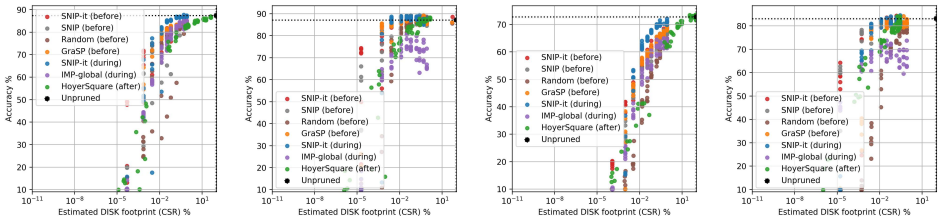
(a) *AlexNet-CIFAR10* (b) *AlexNet-Imagnette* (c) *VGG16-CIFAR10* (d) *VGG16-Imagnette*

Figure 17: *Estimation of inference FLOPS after training for multiple network-dataset combinations w.r.t. unpruned baseline.*

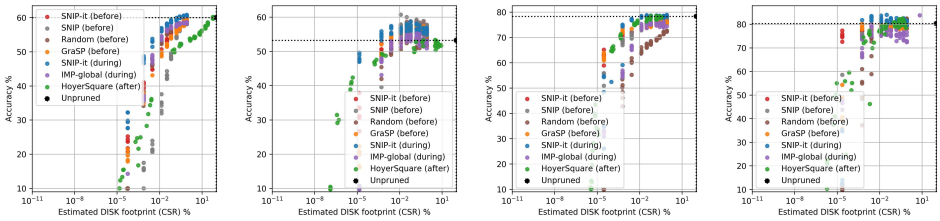
K.5 Disk



(a) *AlexNet-CIFAR10* (b) *AlexNet-Imagnette* (c) *VGG16-CIFAR10* (d) *VGG16-Imagnette*



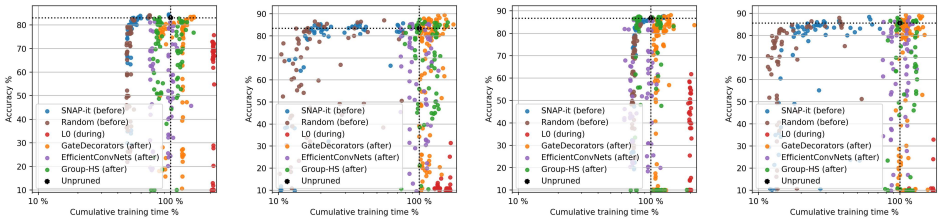
(e) *Conv6-CIFAR10* (f) *Conv6-Imagenette* (g) *LeNet5-CIFAR10* (h) *LeNet5-Imagenette*



(i) *MLP5-CIFAR10* (j) *MLP5-Imagenette* (k) *ResNet18-CIFAR10* (l) *ResNet18-Imagenette*

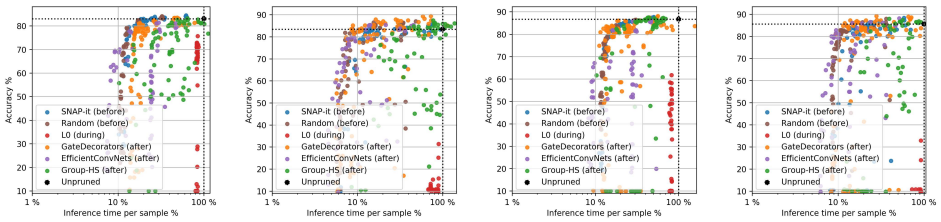
Figure 18: *Estimations of Disk storage for multiple network-dataset combinations as reduction in CSR-format [6, 60] w.r.t. unpruned baseline.*

K.6 Time



(a) *AlexNet-CIFAR10* (b) *AlexNet-Imagnette* (c) *VGG16-CIFAR10* (d) *VGG16-Imagnette*

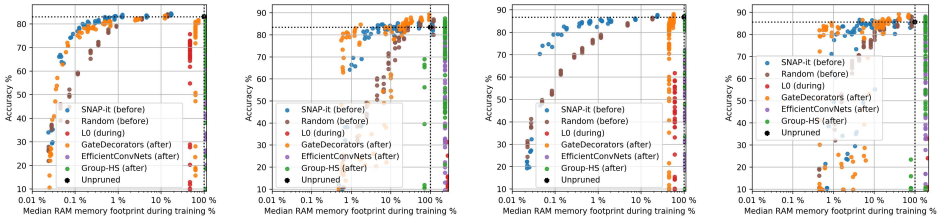
Figure 19: *Measurements of total run-time for a set number of epochs for multiple network-dataset combinations w.r.t. unpruned baseline. GateDecorators [75] and ℓ_0-regularisation [48] required a higher number of epochs and naturally have higher measurements because of it.*



(a) *AlexNet-CIFAR10* (b) *AlexNet-Imagnette* (c) *VGG16-CIFAR10* (d) *VGG16-Imagnette*

Figure 20: *Measurements of inference time after training for multiple network-dataset combinations w.r.t. unpruned baseline.*

K.7 RAM-footprint



(a) *AlexNet-CIFAR10* (b) *AlexNet-Imagnette* (c) *VGG16-CIFAR10* (d) *VGG16-Imagnette*

Figure 21: *Median measurements of RAM-footprint during training. Note that for GateDecorators [75], since it still trains rather long after the pre-train phase, sometimes a sparser measurement ended up being the median and sometimes one from the the pre-train phase, causing two separate patterns to emerge.*

L Network architectures

Here different network architecture details are documented. I will denote the input dimensions and K the number of classes. We alter most networks to use the LeakyReLU activation function with a slope of 5%, dropout [64] with a probability of 30%, batch normalisation and pooling layers.

- **MLP5** [1]
  Linear(in features=I, out features=512, bias=True)
  BatchNorm1d(512, eps=1e-05, momentum=0.1, affine=True, track running stats=True)

[1] Own creation

LeakyReLU(negative slope=0.05)
Dropout(p=0.3, inplace=False)
Linear(in features=512, out features=512, bias=True)
BatchNorm1d(512, eps=1e-05, momentum=0.1, affine=True, track running stats=True)
LeakyReLU(negative slope=0.05)
Dropout(p=0.3, inplace=False)
Linear(in features=512, out features=512, bias=True)
BatchNorm1d(512, eps=1e-05, momentum=0.1, affine=True, track running stats=True)
LeakyReLU(negative slope=0.05)
Dropout(p=0.3, inplace=False)
Linear(in features=512, out features=512, bias=True)
BatchNorm1d(512, eps=1e-05, momentum=0.1, affine=True, track running stats=True)
LeakyReLU(negative slope=0.05)
Dropout(p=0.3, inplace=False)
Linear(in features=512, out features=K, bias=True)

- **LeNet5**
  Conv2d(I, 6, kernel size=(5, 5), stride=(1, 1), padding=(2, 2))
  BatchNorm2d(6, eps=1e-05, momentum=0.1, affine=True, track running stats=True)
  LeakyReLU(negative slope=0.05)
  MaxPool2d(kernel size=2, stride=2, padding=0, dilation=1, ceil mode=False)
  Conv2d(6, 16, kernel size=(5, 5), stride=(1, 1))
  BatchNorm2d(16, eps=1e-05, momentum=0.1, affine=True, track running stats=True)
  LeakyReLU(negative slope=0.05)
  MaxPool2d(kernel size=2, stride=2, padding=0, dilation=1, ceil mode=False)
  Conv2d(16, 120, kernel size=(5, 5), stride=(1, 1))
  BatchNorm2d(120, eps=1e-05, momentum=0.1, affine=True, track running stats=True)
  LeakyReLU(negative slope=0.05)
  AdaptiveAvgPool2d(output size=(3, 3))
  Dropout(p=0.3, inplace=False)
  Linear(in features=1080, out features=84, bias=True)
  BatchNorm1d(84, eps=1e-05, momentum=0.1, affine=True, track running stats=True)
  LeakyReLU(negative slope=0.05)
  Dropout(p=0.3, inplace=False)
  Linear(in features=84, out features=K, bias=True)

- **Conv6**
  Conv2d(I, 64, kernel size=(3, 3), stride=(1, 1), padding=(1, 1))
  BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track running stats=True)
  LeakyReLU(negative slope=0.05)
  Conv2d(64, 64, kernel size=(3, 3), stride=(1, 1), padding=(1, 1))
  BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track running stats=True)
  LeakyReLU(negative slope=0.05)
  MaxPool2d(kernel size=2, stride=2, padding=0, dilation=1, ceil mode=False)
  Conv2d(64, 128, kernel size=(3, 3), stride=(1, 1), padding=(1, 1))
  BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track running stats=True)
  LeakyReLU(negative slope=0.05)
  Conv2d(128, 128, kernel size=(3, 3), stride=(1, 1), padding=(1, 1))
  BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track running stats=True)
  LeakyReLU(negative slope=0.05)
  MaxPool2d(kernel size=2, stride=2, padding=0, dilation=1, ceil mode=False)
  Conv2d(128, 256, kernel size=(3, 3), stride=(1, 1), padding=(1, 1))
  BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track running stats=True)
  LeakyReLU(negative slope=0.05)
  Conv2d(256, 256, kernel size=(3, 3), stride=(1, 1), padding=(1, 1))
  BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track running stats=True)
  LeakyReLU(negative slope=0.05)
  AdaptiveAvgPool2d(output size=(3, 3))
  Dropout(p=0.3, inplace=False)
  Linear(in features=2304, out features=256, bias=True)
  BatchNorm1d(256, eps=1e-05, momentum=0.1, affine=True, track running stats=True)
  LeakyReLU(negative slope=0.05)
  Dropout(p=0.3, inplace=False)
  Linear(in features=256, out features=256, bias=True)
  BatchNorm1d(256, eps=1e-05, momentum=0.1, affine=True, track running stats=True)

LeakyReLU(negative slope=0.05)
Dropout(p=0.3, inplace=False)
Linear(in features=256, out features=K, bias=True)

- **ResNet18**
  Conv2d(I, 64, kernel size=(7, 7), stride=(2, 2), padding=(3, 3), bias=False)
  BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track running stats=True)
  LeakyReLU(negative slope=0.05, inplace=True)
  MaxPool2d(kernel size=3, stride=2, padding=1, dilation=1, ceil mode=False)
  BasicBlock(in featurs=64, out features=64, downsampling=False)
  BasicBlock(in featurs=64, out features=64, downsampling=False)
  BasicBlock(in featurs=64, out features=128, downsampling=True)
  BasicBlock(in featurs=128, out features=128, downsampling=False)
  BasicBlock(in featurs=128, out features=256, downsampling=True)
  BasicBlock(in featurs=256, out features=256, downsampling=False)
  BasicBlock(in featurs=256, out features=512, downsampling=True)
  BasicBlock(in featurs=512, out features=512, downsampling=False)
  AdaptiveAvgPool2d(output size=(2, 2))
  Sequential(
  Dropout(p=0.3, inplace=False)
  Linear(in features=2048, out features=256, bias=True)
  BatchNorm1d(256, eps=1e-05, momentum=0.1, affine=True, track running stats=True)
  LeakyReLU(negative slope=0.05)
  Dropout(p=0.3, inplace=False)
  Linear(in features=256, out features=K, bias=True)

- **VGG16**
  Conv2d(I, 64, kernel size=(3, 3), stride=(1, 1), padding=(1, 1))
  BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track running stats=True)
  LeakyReLU(negative slope=0.05, inplace=True)
  Conv2d(64, 64, kernel size=(3, 3), stride=(1, 1), padding=(1, 1))
  BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track running stats=True)
  LeakyReLU(negative slope=0.05, inplace=True)
  MaxPool2d(kernel size=2, stride=2, padding=0, dilation=1, ceil mode=False)
  Conv2d(64, 128, kernel size=(3, 3), stride=(1, 1), padding=(1, 1))
  BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track running stats=True)
  LeakyReLU(negative slope=0.05, inplace=True)
  Conv2d(128, 128, kernel size=(3, 3), stride=(1, 1), padding=(1, 1))
  BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track running stats=True)
  LeakyReLU(negative slope=0.05, inplace=True)
  MaxPool2d(kernel size=2, stride=2, padding=0, dilation=1, ceil mode=False)
  Conv2d(128, 256, kernel size=(3, 3), stride=(1, 1), padding=(1, 1))
  BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track running stats=True)
  LeakyReLU(negative slope=0.05, inplace=True)
  Conv2d(256, 256, kernel size=(3, 3), stride=(1, 1), padding=(1, 1))
  BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track running stats=True)
  LeakyReLU(negative slope=0.05, inplace=True)
  Conv2d(256, 256, kernel size=(3, 3), stride=(1, 1), padding=(1, 1))
  BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track running stats=True)
  LeakyReLU(negative slope=0.05, inplace=True)
  MaxPool2d(kernel size=2, stride=2, padding=0, dilation=1, ceil mode=False)
  Conv2d(256, 512, kernel size=(3, 3), stride=(1, 1), padding=(1, 1))
  BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track running stats=True)
  LeakyReLU(negative slope=0.05, inplace=True)
  Conv2d(512, 512, kernel size=(3, 3), stride=(1, 1), padding=(1, 1))
  BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track running stats=True)
  LeakyReLU(negative slope=0.05, inplace=True)
  Conv2d(512, 512, kernel size=(3, 3), stride=(1, 1), padding=(1, 1))
  BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track running stats=True)
  LeakyReLU(negative slope=0.05, inplace=True)
  MaxPool2d(kernel size=2, stride=2, padding=0, dilation=1, ceil mode=False)
  Conv2d(512, 512, kernel size=(3, 3), stride=(1, 1), padding=(1, 1))
  BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track running stats=True)
  LeakyReLU(negative slope=0.05, inplace=True)
  Conv2d(512, 512, kernel size=(3, 3), stride=(1, 1), padding=(1, 1))

BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track running stats=True)
LeakyReLU(negative slope=0.05, inplace=True)
Conv2d(512, 1024, kernel size=(3, 3), stride=(1, 1), padding=(1, 1))
BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track running stats=True)
LeakyReLU(negative slope=0.05, inplace=True)
AdaptiveAvgPool2d(output size=(2, 2))
Dropout(p=0.3, inplace=False)
Linear(in features=4096, out features=4096, bias=True)
BatchNorm1d(4096, eps=1e-05, momentum=0.1, affine=True, track running stats=True)
LeakyReLU(negative slope=0.05, inplace=True)
Dropout(p=0.3, inplace=False)
Linear(in features=4096, out features=4096, bias=True)
BatchNorm1d(4096, eps=1e-05, momentum=0.1, affine=True, track running stats=True)
LeakyReLU(negative slope=0.05, inplace=True)
Dropout(p=0.3, inplace=False)
Linear(in features=4096, out features=K, bias=True)

- **AlexNet**
  Conv2d(I, 64, kernel size=(5, 5), stride=(1, 1), padding=(2, 2))
  BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track running stats=True)
  LeakyReLU(negative slope=0.05)
  MaxPool2d(kernel size=3, stride=2, padding=0, dilation=1, ceil mode=False)
  Conv2d(64, 192, kernel size=(5, 5), stride=(1, 1), padding=(2, 2))
  BatchNorm2d(192, eps=1e-05, momentum=0.1, affine=True, track running stats=True)
  LeakyReLU(negative slope=0.05)
  MaxPool2d(kernel size=3, stride=2, padding=0, dilation=1, ceil mode=False)
  Conv2d(192, 384, kernel size=(3, 3), stride=(1, 1), padding=(1, 1))
  BatchNorm2d(384, eps=1e-05, momentum=0.1, affine=True, track running stats=True)
  LeakyReLU(negative slope=0.05)
  Conv2d(384, 256, kernel size=(3, 3), stride=(1, 1), padding=(1, 1))
  BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track running stats=True)
  LeakyReLU(negative slope=0.05)
  Conv2d(256, 512, kernel size=(3, 3), stride=(1, 1), padding=(1, 1))
  BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track running stats=True)
  LeakyReLU(negative slope=0.05)
  AdaptiveAvgPool2d(output size=(2, 2))
  Dropout(p=0.3, inplace=False)
  Linear(in features=2048, out features=4096, bias=True)
  BatchNorm1d(4096, eps=1e-05, momentum=0.1, affine=True, track running stats=True)
  LeakyReLU(negative slope=0.05)
  Dropout(p=0.3, inplace=False)
  Linear(in features=4096, out features=4096, bias=True)
  BatchNorm1d(4096, eps=1e-05, momentum=0.1, affine=True, track running stats=True)
  LeakyReLU(negative slope=0.05)
  Dropout(p=0.3, inplace=False)
  Linear(in features=4096, out features=K, bias=True)