Doctoral Dissertation

Doctoral Program in Computer Science (35thcycle)

Exploring the Design and Implementation of Pruning Techniques for Deep Neural Networks

Andrea Bragagnolo

Supervisor
Prof. Marco Grangetto

Committee

Università degli Studi di Torino

2023

Declaration

I hereby declare that, the contents and organization of this dissertation constitute my own original work and does not compromise in any way the rights of third parties, including those relating to the security of personal data.

I would like to dedicate this thesis to my loving parents

Acknowledgements

And I would like to acknowledge ...

Abstract

Deep neural networks have become one of the go-to tools to achieve state-of-the-art performance for various tasks, such as computer vision, speech recognition, and many more. Unfortunately, most modern architectures owe their generalization capabilities to the sheer amount of parameters they possess, with many reaching the order of millions or even billions, resulting in a significant increase in the resources required to use such models. This, in turn, impacts the deployability of neural networks on resource-constrained devices such as smartphones or FPGAs. In this thesis, we will cover the products of my research on neural network pruning, i.e., removing less essential elements of the network (single parameters or entire neurons) to reduce the storage, memory, and computation requirements needed to run the model. We will explore the design of pruning procedures, the effect of pruning on the features extracted by the network model, and the practical application of pruned networks on low-power devices. In this thesis, we present and subsequently employ two pruning techniques: LOBSTER, an unstructured approach that leverages the sensitivity of the parameters as a regularizer, and SeReNe, a structured procedure that evaluates the contribution of the neurons to the output of the network. Neural networks pruned with LOBSTER and SeReNe have been used to assess the advantages of pruning when deploying the networks to embedded devices. To this end, we implemented the Simplify library that removes the zeroed neurons from the architecture. Finally, we focus on reducing the computational resources required to train a neural network. Since backpropagation is the more computation-heavy part of the training process, we defined a technique to disable the computation of the gradients of neurons that reached equilibrium, effectively pruning the backpropagation graph and decreasing the number of operations performed during the procedure.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In the last years, network architectures have become even more powerful and "expensive": performing simple inferences requires a lot of computational resources and training the models even more. For example, cornerstone architectures such as AlexNet and VGG [1] have a complexity in the order of 60 and 130 million parameters, soaring to more than 100 billion for models like GPT-3 [2]. This massive number of parameters poses a significant challenge when working in contexts where memory, computational power, or storage are limited (e.g., smartphones or FPGAs) and the provided resources are insufficient to use the network satisfactorily.

For these reasons, many applications opt for a client-server system. Here, the trained network resides in a server with plenty of resources to satisfy the client's requests. While this setup works well overall, it may pose some concerns in some use cases. For example, transferring private data from the client to the server requires multiple precautions and safety features, also, the internet connection may not be available, or the latency between the request and the response may render the service unusable. Many of these problems could be solved by deploying the neural network directly on the consumer device, reducing privacy concerns, internet dependency, and maintenance costs. Unfortunately, as mentioned, most neural networks require resources that most consumer-grade devices can hardly provide.

Multiple (complementary) approaches are being developed to cope with neural networks' memory requirements, inference time, and energy consumption:

- **Re-designing the network topology**. Moving from one architecture to another, possibly forcing precise neuronal connectivity or weight sharing, can reduce the number of parameters or the complexity of the network [3–5].
- **Quantization**. Representing the parameters (and activation functions) as fixed-point digits reduces the memory footprint and speeds up computations [6].
- **Pruning**. It is well known that many deep architectures are typically over-parametrized [7, 8], but such redundant parameters can be removed [9–13], leading to smaller topologies.

This thesis will focus on the topic of neural network pruning, and it will be structured as follows:

- In **Chapter 2**, we will present the notation used throughout the thesis.
- In **Chapter 3**, we will expand on the concept of neural network pruning, providing an overview of the state-of-the-art and the main differences between classes of pruning techniques.
- **Chapter 4** will present the concept of *parameter sensitivity* and describe *LOBSTER* (LOss-Based SensiTivity rEgulaRization), a pruning technique we developed that exploits this sensitivity value. This technique enforces sparse topology during the network training and does not require a pre-trained model.
- In **Chapter 5** we will move on to *SeReNe* (Sensitivity-based Regularization of Neurons), a structured pruning procedure that leverages on the concept of *neural sensitivity*.
- **Chapter 6** will present *Simplify*, a PyTorch compatible library for achieving effective model simplification. This library tries to bridge the gap between research and the actual deployment of pruned neural networks by removing zeroed neurons from the topology. Simplified models benefit from a smaller memory footprint and a lower inference time, making their deployment to embedded or mobile devices much more efficient.
- In **Chapter 7**, we will use Simplify to give practical examples of deployed pruned neural networks. To this end, we deployed some simplified models to

both mobile devices and FPGA circuits, validating that pruning can reduce the resource requirements of modern deep neural networks.

- In **Chapter 8**, we will analyze the effect of one-shot and gradual pruning on the resulting model, showing the higher effectiveness of the latter. To this end, we will introduce and use *PSP-entropy* to show that gradual pruning allows access to narrow, well-generalizing minima, typically ignored when using one-shot approaches.
- In **Chapter 9**, we will shift the focus from reducing the size and inference time of the networks to improving the training time. To this end, we propose *NEq*, a technique that can prevent the update (and the gradient computation) for neurons that reached equilibrium.
- Finally, **Chapter 10** will draw the conclusions.

Chapter 2

Notation

Let us define a neural network as the function $\mathcal{N}(X;\Theta)$, where X represents the network's input, and Θ is the set of all the parameters which uniquely identify the network. The cost function optimized during training is $\mathcal{L}\Theta)$. As the network model is composed of N hidden layers, we identify with $n = 0$ the input layer and $n = N$ the output layer; other n values indicate the hidden layers. We denote with $\boldsymbol{y}_n$ the output vector of layer n. Given that each layer is composed of a set of neurons, for the i-th neuron of the n-th layer ($x_{n,i}$), we define:

- $y_{n,i}$ as its output,
- $\boldsymbol{y}_{n-1}$ as its input vector ($\boldsymbol{y}_0 = X$),
- $\theta_{n,i}$ as its own parameters: $\boldsymbol{w}_{n,i}$ the weights and $b_{n,i}$ the bias,

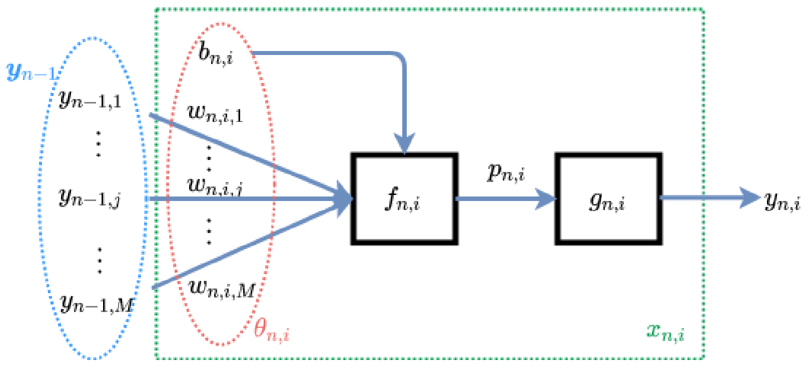


Fig. 2.1 Representation of the neuron $x_{n,i}$ with activation function $g_{n,i}$.

To each neuron, we can associate an affine function $f_{n,i}(\cdot)$ (e.g., a convolution or the dot product) and an activation function $g_{n,i}(\cdot)$ (e.g., the ReLU function). The output of a neuron is given by

$$y_{n,i} = g_{n,i}(p_{n,i}), \tag{2.1}$$

where $p_{n,i}$ is the post-ynaptic potential [14] of $x_{n,i}$ defined as:

$$p_{n,i} = f_{n,i}(\boldsymbol{\theta}_{n,i}, \boldsymbol{y}_{n-1}). \tag{2.2}$$

This notation is summarized in Figure 2.1.

Chapter 3

Neural Networks Pruning

Given the neural network $\mathcal{N}(X;\Theta)$, a pruning procedure that operates on such a model will result in a new model $\mathcal{N}(X;\Theta')$ where $\Theta' = M \odot \Theta$ with $M \in \{0,1\}^{|\Theta|}$ representing a binary mask responsible for setting some of the parameters to 0 and $\odot$ representing the element-wise product operation. Pruning procedures aim to solve

$$\underset{||\Theta'||_0}{\arg\min}[|\mathcal{L}(\Theta) - \mathcal{L}(\Theta')| \leq \varepsilon] \tag{3.1}$$

where $||\Theta'||_0$ represents the number of non-zero parameters and ε is a positive and arbitrarily small number. Figure 3.1 gives a visual representation of the effect of pruning procedures.

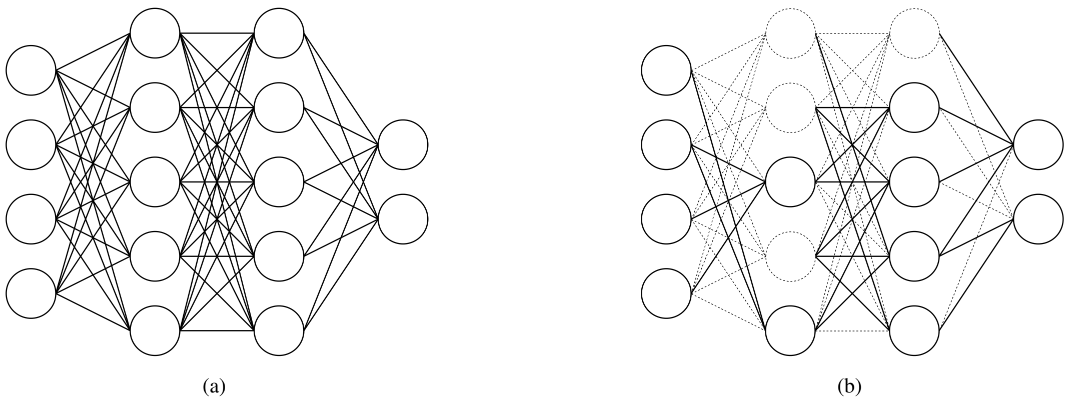


Fig. 3.1 Representation of the effect of a pruning procedure: (a) dense neural network with all its original parameters intact, (b) pruned network (dashed lines represent pruned neurons and connections).

3.1 State-of-the-art

Algorithm 1 Pseudocode of the procedure proposed by Han *et al.* [15]

```
    Input: Neural network $\mathcal{N}$, dataset D, pruning iterations $N$
    Output: Pruned network $\mathcal{N}^\star$
 1: procedure PRUNE AND FINE-TUNE($\mathcal{N}, D, N$)
 2:     $i \leftarrow 0$
 3:     $\mathcal{N}^\star \leftarrow \mathcal{N}$
 4:     $\mathcal{N}^\star \leftarrow train(\mathcal{N}^\star, D)$
 5:     while $i < N$ do
 6:         $\mathcal{N}^\star \leftarrow prune(\mathcal{N}^\star)$
 7:         $\mathcal{N}^\star \leftarrow finetune(\mathcal{N}^\star, D)$
 8:         $i \leftarrow i + 1$
        return $\mathcal{N}^\star$
```

Different attempts to reduce the number of parameters have been proposed, with some even dating to the late 1980s. For example, in 1989, Mozer and Smolensky proposed *skeletonization* [16], a technique to identify and remove less relevant neurons in a trained model by comparing the error of the pruned model to the original network. Around the same time, LeCun *et al.* also proposed a work where they leveraged the information from the second-order derivative of the error function to rank the parameters of the trained model on a saliency-like basis [9].

In the last few years, thanks to the broad availability of computational resources and the spread of deeper network models, pruning approaches gained-back popularity and have become a hot topic in deep learning, with tens of thousands of articles published each year. In particular, in 2015, Han *et al.* proposed a procedure that has become a cornerstone in the pruning literature [15]. The process, summarized in Algorithm 1, works as follows: first, the network is trained to convergence. Then each parameter is issued a score and pruned accordingly (for example, prune all weights with a magnitude lower than some threshold). Lastly, some fine-tuning iterations are performed to recover the performance lost due to pruning. These last two steps are often iterated several times to reach the desired sparsity.

Many works that followed proposed variations of this approach. Some employ a regularization technique designed to push many parameters toward zero: Wen *et al.* [17] use group lasso regularization, Louizos *et al.* [18] employ a proxy for the L_0 regularization, and Tartaglione *et al.* [19] propose to evaluate the sensitivity of the parameters to the output of the network. Other works add and train auxiliary parameters to promote sparsity and use them to decide which weights to prune [20–22].

Recent works by Frankle *et al.* [13, 23] show that, given a randomly initialized neural network, it is possible to extract a subnetwork that, when trained in isolation, can achieve the accuracy of the original model. These findings fit perfectly with recent concerns regarding the financial and environmental cost of training modern neural networks [24]: uncovering these subnetworks in the early stages could allow us to reduce the cost of training the model. Towards this goal, many strategies have been proposed; for example, Lee *et al.* [25] prune weights that less affect the loss using sensitivity-based saliency metrics, and Wang *et al.* [26] evaluates the score of the weights based on the gradient flow. Tanaka *et al.* [27] propose to use "synaptic strength" to prune the network iteratively.

3.2 Differences between pruning procedures

All pruning procedures can be categorized based on two key characteristics: the sparsified layers' resulting structure and the pruning schedule.

Unstructured vs. structured

Techniques that prune individual parameters without any constraint on the resulting topology are known as *unstructured*. These techniques can result in many neurons with very few non-zero parameters, leading to irregular memory access and impacting the practical speed-up. This sparsity could be exploited using sparse matrix-vector product implementation [28, 29]; unfortunately, these still need to be commonly included in consumer-grade devices. Therefore the (high) sparsity levels achievable are hardly exploitable. On the other hand, *structured* pruning removes entire neurons (or channels).

Figure 3.2 shows the difference between unstructured and structured pruning procedures. Figure 3.2c and Figure 3.2d provide the matrix representation of the two hidden layers of the pruned network in Figure 3.2a and Figure 3.2b. Each row of the matrices correlates with a neuron, and each element represents an inbound connection to the corresponding neuron; the red squares represent the pruned parameters. We can see that unstructured procedures lead to sparse matrices without any particular

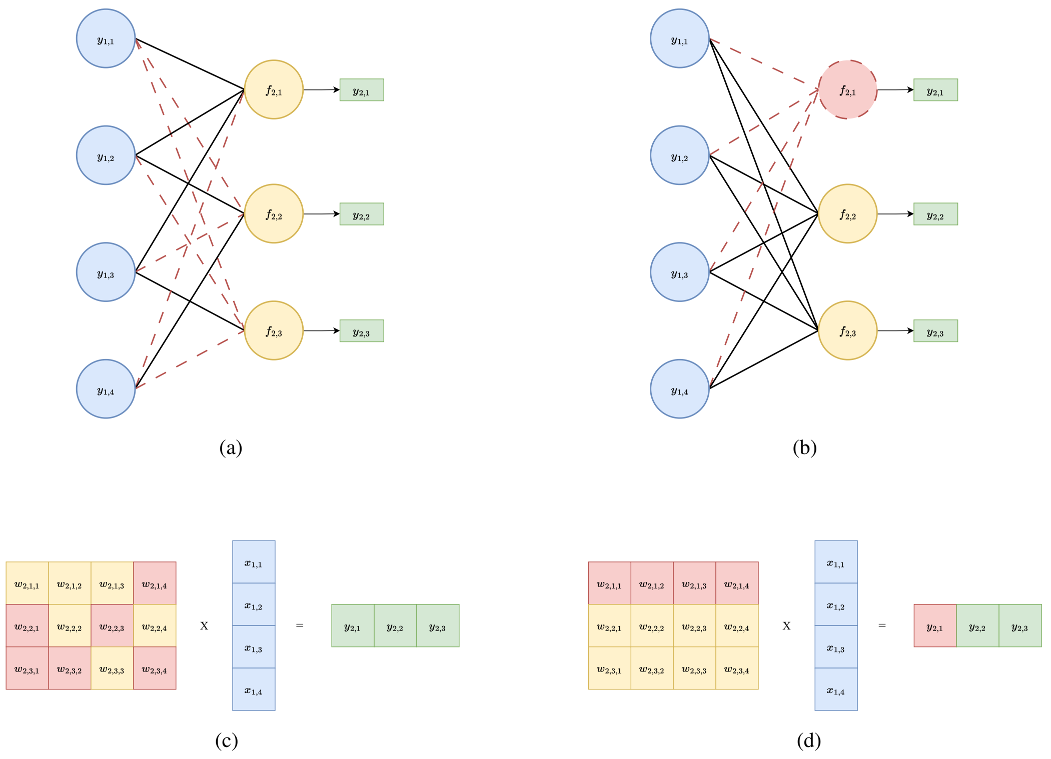
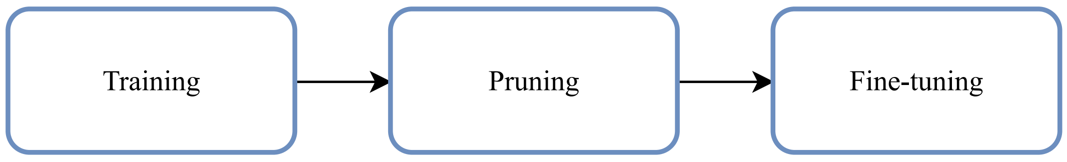


Fig. 3.2 Comparison between unstructured and structured pruning procedures. (a) and (b) show a possible result of unstructured and structured pruning. (c) and (d) show the matrix representation of the hidden layer (yellow) for each pruned network (the activation function $g_{n,i}$ is omitted); each row represents a neuron, and each matrix element is a weight of the neurons. Red elements represent pruned parameters.

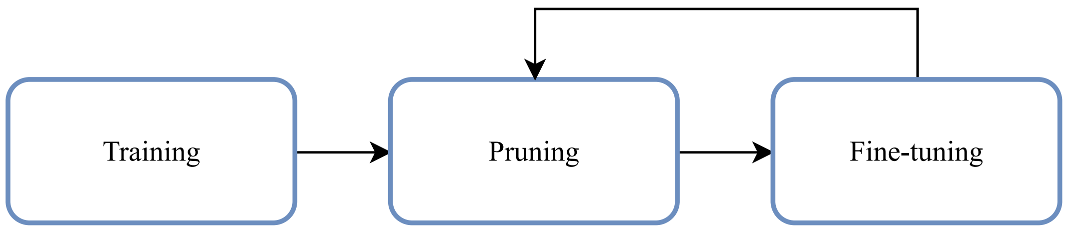
pattern and every element is used to define the output vector. On the other hand, structured approaches result in clear patterns (i.e., where the neuron is pruned, an entire row is removed from the matrix). This structure in the pruned topology could be exploited by removing the zeroed neurons from the architecture, resulting in smaller, dense matrices and a reduction in the resource required to perform inference on the pruned network.

One-shot vs. gradual

Another difference between pruning procedures is how many steps are used to reach the desired sparsity level (i.e., how the pruning is scheduled). Some techniques use a single pruning step to remove all the intended weights at once, sometimes followed by a few fine-tuning iterations, and take the name of *one-shot*. Others, instead, prune some parameters iteratively (either a fixed or a variable amount),



(a)



(b)

Fig. 3.3 Comparison between the two pruning schedules: (a) one-shot pruning, (b) gradual pruning.

alternating the pruning step with some training iterations; this can be classified as *gradual* approaches. Overall, one-shot methodologies can reach the desired sparsity faster than gradual procedures, but the latter can achieve higher sparsity and a minor performance loss. A simple schematic of the two pruning schedules is shown in Figure 3.3.

Chapter 4

LOBSTER

Many popular pruning procedures, such as those presented in Chapter 3, require that the model to be pruned has been preliminary trained via standard gradient descent or introduce some rewinding steps in the training process, which increases the total learning time. This chapter will tackle this issue by proposing LOBSTER (LOss-Based SensiTivity rEgulaRization). LOBSTER is an unstructured, gradual approach that employs local pruning. The method extends the framework presented in Algorithm 1 of Chapter 3 and uses a novel sensitivity-based regularization to promote sparsity in the architecture.

In this context, we define the sensitivity of the parameter of a network model as the derivative of the loss function with respect to the target parameter. Intuitively, low-sensitivity parameters have a negligible impact on the loss function when perturbed and are fit to be shrunk without compromising the network performance. In practice, LOBSTER pushes toward zero parameters with low sensitivity, using a regularize-and-prune approach, resulting in a sparse network topology. Contrary to other approaches that require a pre-trained model, this procedure, thanks to its loss-based sensitivity formulation, allows training a network from scratch. Moreover, unlike different sensitivity-based approaches, LOBSTER computes the sensitivity by exploiting the already available gradient of the loss function, avoiding additional derivative computations [16, 19] or second-order derivatives [9].

We performed multiple experiments over different network topologies and datasets, showing that LOBSTER outperforms several competitors in various tasks.

4.1 Proposed Regularization

This section will define the sensitivity measure and illustrate the proposed network update rule that uses this measure as a regularization term to promote sparsity.

4.1.1 Loss-based Sensitivity

Neural networks are typically trained via gradient-descent-based optimization, i.e., minimizing the loss function. Methods based on mini-batches of samples have gained popularity as they allow better generalization than stochastic learning while also being memory and time efficient. In such a framework, a network parameter $w_{n,i,j}$ is updated towards the averaged direction, which minimizes the averaged loss for the minibatch, i.e., using the well-known stochastic gradient descent or its variations.

Our ultimate goal is to assess to which extent a variation of the value of $w_{n,i,j}$ would affect the error on the network output $\boldsymbol{y}_N$: the parameters not affecting the network output can be set to zero, i.e., pruned away. We make the first attempt towards this end by introducing a small perturbation $\Delta w_{n,i,j}$ over $w_{n,i,j}$ and measuring the variation of $\boldsymbol{y}_N$ as

$$\Delta \boldsymbol{y}_N = \sum_k \left|\Delta y_{N,k}\right| \approx \Delta w_{n,i,j} \sum_k \left|\frac{\partial y_{N,k}}{\partial w_{n,i,j}}\right|. \tag{4.1}$$

where $y_{N,k}$ indicates the k-th output for the output layer.

Unfortunately, evaluating equation 4.1 is computationally expensive, as it would require a complexity that grows linearly with the number of the output classes [19]. However, we can directly estimate the error function's variations using some differentiable proxy function, i.e., the loss function $\mathcal{L}(\Theta)$ (for the sake of simplicity, we will drop the argument of the loss function from here on out), shifting the focus from the output to the error of the network

$$\Delta \mathcal{L} \approx \Delta w_{n,i,j} \left|\frac{\partial \mathcal{L}}{\partial \boldsymbol{y}_N} \cdot \frac{\partial \boldsymbol{y}_N}{\partial w_{n,i,j}}\right| = \Delta w_{n,i,j} \left|\frac{\partial \mathcal{L}}{\partial w_{n,i,j}}\right|. \tag{4.2}$$

We can now properly define the *sensitivity* S for a given parameter $w_{n,i,j}$ as

$$S(\mathcal{L}, w_{n,i,j}) = \left| \frac{\partial \mathcal{L}}{\partial w_{n,i,j}} \right|. \tag{4.3}$$

Large values of S correspond to a significant variation of the loss function for small perturbations of $w_{n,i,j}$.

4.1.2 Update Rule

Given the sensitivity definition in equation 4.3, we can promote sparse topologies by pruning parameters with both low sensitivity S (i.e., in a flat region of the loss function gradient, where a small perturbation of the parameter has a negligible effect on the loss) and low magnitude. Toward this end, we propose the following parameter update rule to promote sparsity:

$$w_{n,i,j}^{t+1} := w_{n,i,j}^{t} - \eta \frac{\partial \mathcal{L}}{\partial w_{n,i,j}^{t}} - \lambda w_{n,i,j}^{t} \left[1 - S(\mathcal{L}, w_{n,i,j}^{t})\right] P\left[S(\mathcal{L}, w_{n,i,j}^{t})\right], \tag{4.4}$$

where

$$P(x) = \Psi\left[1 - |x|\right], \tag{4.5}$$

$\Psi(\cdot)$ is the one-step function, and η and λ are two positive hyper-parameters.

Plugging equation 4.3 in equation 4.4 we can rewrite the update rule as:

$$w_{n,i,j}^{t+1} = w_{n,i,j}^{t} - \eta \frac{\partial L}{\partial w_{n,i,j}^{t}} - \lambda \Gamma\left(\mathcal{L}, w_{n,i,j}^{t}\right) \left[1 - \left| \frac{\partial \mathcal{L}}{\partial w_{n,i,j}^{t}} \right| \right], \tag{4.6}$$

where

$$\Gamma(y, x) = x \cdot P\left(\frac{\partial y}{\partial x}\right). \tag{4.7}$$

After some algebraic manipulations, we can rewrite equation 4.6 as

$$w_{n,i,j}^{t+1} = w_{n,i,j}^{t} - \lambda \Gamma\left(\mathcal{L}, w_{n,i,j}^{t}\right) - \frac{\partial \mathcal{L}}{\partial w_{n,i,j}^{t}} \left[\eta - \text{sign}\left(\frac{\partial \mathcal{L}}{\partial w_{n,i,j}^{t}}\right) \lambda \Gamma\left(\mathcal{L}, w_{n,i,j}^{t}\right)\right]. \tag{4.8}$$

In equation 4.8, we observe two different components of the proposed regularization term:

- a weight decay-like term $\Gamma\left(\mathcal{L}, w_{n,i,j}\right)$ which is enabled/disabled by the magnitude of the gradient on the parameter;
- a correction term for the learning rate. In particular, the full learning process follows an *equivalent* learning rate

$$\tilde{\eta} = \eta - \text{sign}\left(\frac{\partial \mathcal{L}}{\partial w_{n,i,j}}\right) \lambda \Gamma\left(\mathcal{L}, w_{n,i,j}\right). \tag{4.9}$$

Let us analyze the corrections in the learning rate. If $\left|\frac{\partial \mathcal{L}}{\partial w_{n,i,j}}\right| \geq 1$ ($w_{n,i,j}$ has large sensitivity), it follows that $P\left(\frac{\partial \mathcal{L}}{\partial w_{n,i,j}}\right) = 0$ and $\Gamma\left(\mathcal{L}, w_{n,i,j}\right) = 0$ and the dominant contribution comes from the gradient. In this case our update rule reduces to the classical GD:

$$w_{n,i,j}^{t+1} = w_{n,i,j}^{t} - \eta \frac{\partial \mathcal{L}}{\partial w_{n,i,j}^{t}}. \tag{4.10}$$

When we consider less sensitive $w_{n,i,j}$ with $\left|\frac{\partial \mathcal{L}}{\partial w_{n,i,j}}\right| < 1$, we get $\Gamma\left(\mathcal{L}, w_{n,i,j}\right) = w_{n,i,j}$ (weight decay term) and we can distinguish two sub-cases for the learning rate:

- if $\text{sign}\left(\frac{\partial \mathcal{L}}{\partial w_{n,i,j}}\right) = \text{sign}\left(w_{n,i,j}\right)$, then $\tilde{\eta} \leq \eta$ (Figure 4.1a and Figure 4.1d),
- if $\text{sign}\left(\frac{\partial \mathcal{L}}{\partial w_{n,i,j}}\right) \neq \text{sign}\left(w_{n,i,j}\right)$, then $\tilde{\eta} \geq \eta$ (Figure 4.1b and Figure 4.1c).

Finally, let us consider the corner case where the network has "fully-converged" over the training set, i.e. $\left|\frac{\partial \mathcal{L}}{\partial w_{n,i,j}}\right| = 0 \; \forall w_{n,i,j}$. In this case, the update rule in equation 4.4 reduces to

$$w_{n,i,j}^{t+1} := (1 - \lambda) w_{n,i,j}^{t} \tag{4.11}$$

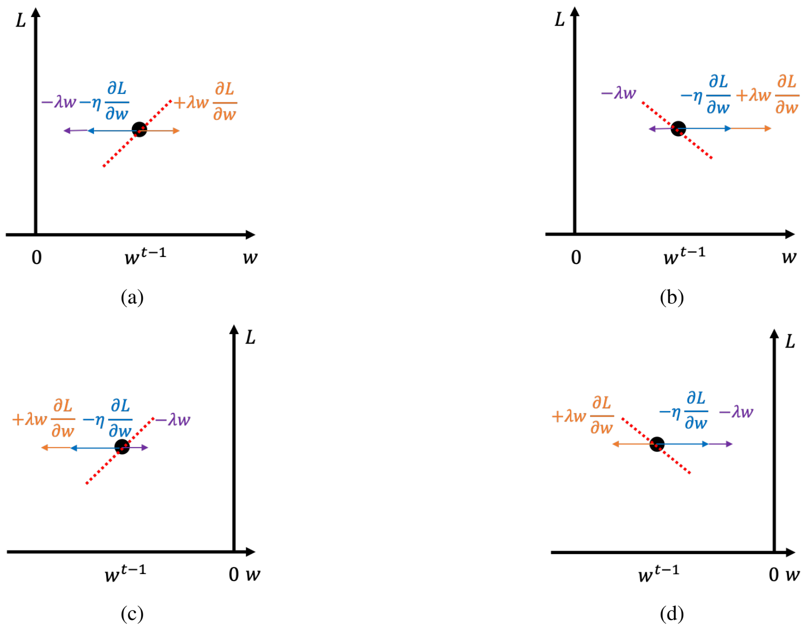

L
$-\lambda w$
$-\eta\frac{\partial L}{\partial w}$
$+\lambda w\frac{\partial L}{\partial w}$
0
w^{t-1}
w
(a)
(b)
(c)
(d)

Fig. 4.1 Update rule effect on the parameters. The red dashed line is the tangent to the loss function in the black dot, in blue the standard SGD contribution, in purple the weight decay while in orange the LOBSTER contribution. Here we assume $P(L, w_{n,i,j}) = 1$.

as $P[S(\mathcal{L}, w^t_{n,i,j})] = 1$. The only term remaining here is a weight decay-like term, which greedily tends to push the parameters toward zero.

Table 4.1 reports the schematics of all these cases, and Figure 4.1. shows the possible effects. The contribution from $\Gamma\left(\mathcal{L}, w_{n,i,j}\right)$ aims to minimize the parameter magnitude, disregarding the loss minimization. If the loss minimization tends to minimize the magnitude, then the equivalent learning rate is reduced. On the contrary, when the gradient descent tends to increase the magnitude, the learning rate is increased to compensate for the contribution coming from $\Gamma\left(\mathcal{L}, w_{n,i,j}\right)$. This mechanism allows us to succeed in the learning task while introducing sparsity.

4.1.3 Regularization function minimized

Let us now investigate the objective function we are minimizing more precisely by imposing the update rule equation equation 4.6.

To this end, we can follow the approach as in [19], and we can compute the regularization function R_i for the single parameter $w_{n,i,j}$ by solving

$P\left(\frac{\partial \mathcal{L}}{\partial w_{n,i,j}}\right)$	$\text{sign}\left(\frac{\partial \mathcal{L}}{\partial w_{n,i,j}}\right)$	$\text{sign}(w)$	$\frac{\tilde{\eta}}{\eta}$
0	any	any	1
1	0	any	1
1	+	+	≤ 1
1	+	-	≥ 1
1	-	+	≥ 1
1	-	-	≤ 1

Table 4.1 Behavior of $\tilde{\eta}$ compared to η ($\eta > 0$).

$$R_i = \int \left(w_{n,i,j} - w_{n,i,j} \left| \frac{\partial \mathcal{L}}{\partial w_{n,i,j}} \right| \right) \Psi \left(1 - \left| \frac{\partial \mathcal{L}}{\partial w_{n,i,j}} \right| \right) dw_{n,i,j}. \tag{4.12}$$

We rewrite R_i as the L_2 regularization followed by a correction term as

$$R_i = \Psi \left(1 - \left| \frac{\partial \mathcal{L}}{\partial w_{n,i,j}} \right| \right) \left(\frac{w_{n,i,j}^2}{2} + \tilde{R}_i \right), \tag{4.13}$$

where

$$\tilde{R}_i = -\int w_{n,i,j} \frac{\partial \mathcal{L}}{\partial w_{n,i,j}} \text{sign} \left(\frac{\partial \mathcal{L}}{\partial w_{n,i,j}} \right) dw_{n,i,j}. \tag{4.14}$$

Let us integrate equation 4.14 by parts:

$$\tilde{R}_i = -\frac{w_{n,i,j}^2}{2} \frac{\partial \mathcal{L}}{\partial w_{n,i,j}} \text{sign} \left(\frac{\partial \mathcal{L}}{\partial w_{n,i,j}} \right) + \int \frac{w_{n,i,j}^2}{2} \frac{\partial^2 \mathcal{L}}{\partial w_{n,i,j}^2} \text{sign} \left(\frac{\partial \mathcal{L}}{\partial w_{n,i,j}} \right) dw_{n,i,j}. \tag{4.15}$$

If we integrate a further step, we obtain:

$$\begin{aligned} \tilde{R}_i = &-\frac{w_{n,i,j}^2}{2} \frac{\partial \mathcal{L}}{\partial w_{n,i,j}} \text{sign} \left(\frac{\partial \mathcal{L}}{\partial w_{n,i,j}} \right) + \frac{w_{n,i,j}^3}{6} \frac{\partial^2 \mathcal{L}}{\partial w_{n,i,j}^2} \text{sign} \left(\frac{\partial \mathcal{L}}{\partial w_{n,i,j}} \right) + \\ &- \int \frac{w_{n,i,j}^3}{6} \frac{\partial^3 \mathcal{L}}{\partial w_{n,i,j}^3} \text{sign} \left(\frac{\partial \mathcal{L}}{\partial w_{n,i,j}} \right) dw_{n,i,j}. \end{aligned} \tag{4.16}$$

Applying infinite steps of integration by parts we have

$$\tilde{R}_i = \text{sign}\left(\frac{\partial \mathcal{L}}{\partial w_{n,i,j}}\right)\sum_{k=1}^{\infty}(-1)^k \frac{\partial^k \mathcal{L}}{\partial w_{n,i,j}^k}\frac{w_{n,i,j}^{k+1}}{(k+1)!}. \tag{4.17}$$

Overall, the regularization function to minimize at training time, over all the $w_{n,i,j}$, is

$$\begin{aligned} R = &\sum_i \Psi\left(1 - \left|\frac{\partial \mathcal{L}}{\partial w_{n,i,j}}\right|\right)\cdot \\ &\cdot\left[\frac{w_{n,i,j}^2}{2} + \text{sign}\left(\frac{\partial \mathcal{L}}{\partial w_{n,i,j}}\right)\sum_{k=1}^{\infty}(-1)^k \frac{\partial^k \mathcal{L}}{\partial w_{n,i,j}^k}\frac{w_{n,i,j}^{k+1}}{(k+1)!}\right]. \end{aligned} \tag{4.18}$$

According to equation 4.13 and, for instance, equation 4.18, we observe that overall the regularization function we are minimizing is the standard L_2 regularization, corrected by a loss-dependent term, defined within our proposed LOBSTER framework.

In the next section, we provide a practical procedure to learn a sparse neural network topology exploiting the above regularization function at training time, followed by a pruning stage.

4.2 Training Procedure

This section describes a procedure to train a sparse neural network $\mathcal{N}$ leveraging the sensitivity-based rule above to update the network parameters. We assume that the parameters have been randomly initialized, albeit the procedure holds if the network has been pre-trained. The process is illustrated in Figure 4.2 and iterates over two stages.

4.2.1 Learning Stage

During the learning stage, the network is iteratively trained according to the update rule equation 4.4 on some training set.

Let j indicate the current learning stage iteration (i.e., epoch) and $\mathcal{N}^j$ represent the network (i.e., the set of learnable parameters) at the end of the j-th iteration. Also,

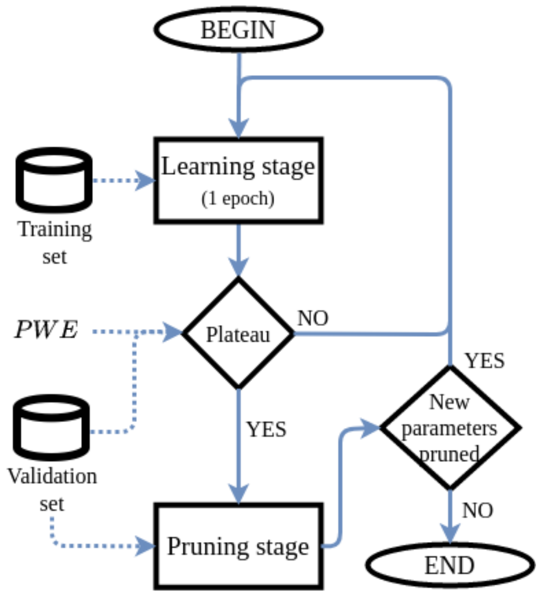


Fig. 4.2 The complete training procedure of LOBSTER.

let $\mathcal{L}^j$ be the loss measured on some validation set at the end of the j-th iteration and $\widehat{\mathcal{L}}$ be the best (lowest) loss measured so far on $\widehat{\mathcal{N}}$ (network with lowest validation loss so far). As initial condition, we assume, $\widehat{\mathcal{N}} = \mathcal{N}^0$. If $\mathcal{L}^j < \widehat{\mathcal{L}}$, the reference to the best network is updated as $\widehat{\mathcal{N}} = \mathcal{N}^j$, $\widehat{\mathcal{L}} = L^j$. We iterate the learning stage until the best validation loss $\widehat{\mathcal{L}}$ has not decreased for PWE iterations of the learning stage in a row (we say the regularizer has reached a performance *plateau*). At such point, we move to the pruning stage.

4.2.2 Pruning Stage

During the pruning stage, parameters with magnitude below a threshold value T are pinpointed to zero, eventually sparsifying the network topology, as shown in Figure 4.3. Namely, we look for the largest T that worsens the classification loss $\widehat{\mathcal{L}}$ at most by a relative quantity TWT:

$$\mathcal{L}_b = (1 + TWT)\,\widehat{\mathcal{L}}, \tag{4.19}$$

where L_b is called *loss boundary*.

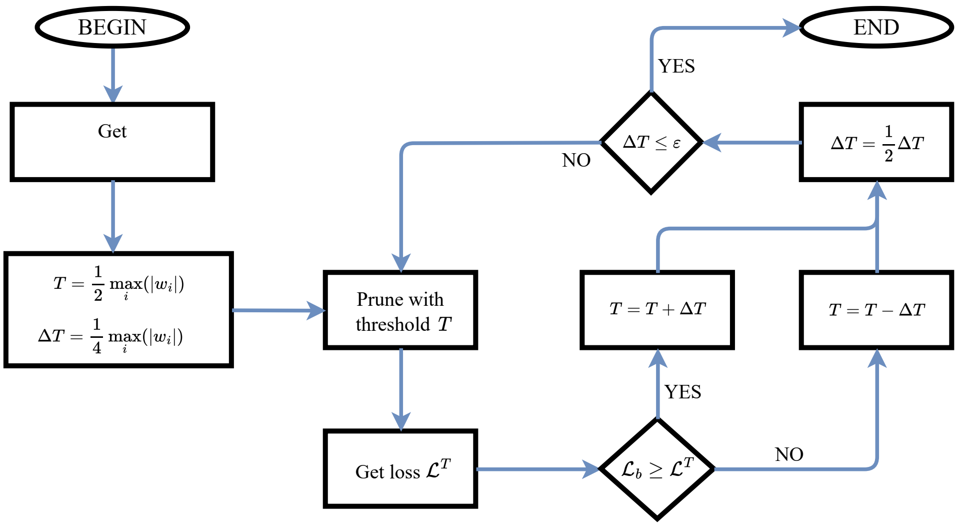

BEGIN
Get
$T = \frac{1}{2}\max_i(|w_i|)$
$\Delta T = \frac{1}{4}\max_i(|w_i|)$
Prune with threshold T
Get loss $\mathcal{L}^T$
$\mathcal{L}_b \geq \mathcal{L}^T$
YES
NO
$T = T + \Delta T$
$T = T - \Delta T$
$\Delta T = \frac{1}{2}\Delta T$
$\Delta T \leq \varepsilon$
YES
NO
END

Fig. 4.3 The pruning stage.

Before the pruning procedure begins, we initialize the threshold T to half of the maximum magnitude for the parameters in $\widehat{\mathcal{N}}$. We also initialize ΔT to $\frac{T}{2}$. Then, we proceed in the research of T as follows:

1. We prune $\widehat{\mathcal{N}}$ with threshold T, obtaining $\mathcal{N}^T$;
2. We compute the loss $\mathcal{L}^T$ on the validation set:
   - if $\mathcal{L}_b \geq \mathcal{L}^T$ the network tolerates a larger amount of pruned parameters, so T is increased by ΔT;
   - if $\mathcal{L}_b < \mathcal{L}^T$, then too many parameters have been pruned. This means we have to restore the parameters pruned at the previous step. Then, we decrease T by ΔT.
3. We update ΔT, dividing its value by half;
4. We test over ΔT value:
   - if $\Delta T \leq \varepsilon$, where ε is some small value for which $\mathcal{L}^T = \mathcal{L}^{T+\varepsilon}$ (typically 10^{-10}), we end our pruning stage;
   - otherwise, we go back to point 1.

Once T is found, all the parameters whose magnitude is below T are permanently set to zero, i.e., they are pruned for good. If at least one parameter has been pruned during the last iteration of the pruning stage, a new iteration of the regularization stage follows; otherwise, the procedure ends by returning the trained, sparse network.

4.3 Results

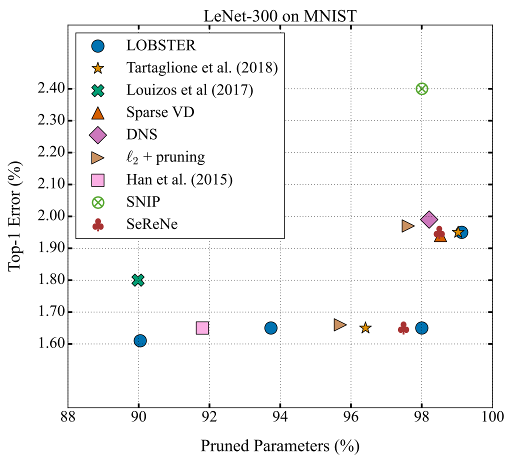
In this section, we experimentally evaluate LOBSTER over multiple architectures and datasets commonly used as a benchmark in the literature:

- LeNet-300 on MNIST (Figure 4.4a),
- LeNet-5 on MNIST (Figure 4.4b),
- LeNet-5 on Fashion-MNIST (Figure 4.4c),
- ResNet-32 on CIFAR-10 (Figure 4.4d),
- ResNet-18 on ImageNet (Figure 4.4e),
- ResNet-101 on ImageNet (Figure 4.4f),
- U-Net on ISIC skin lesion segmentation (Table 4.2).

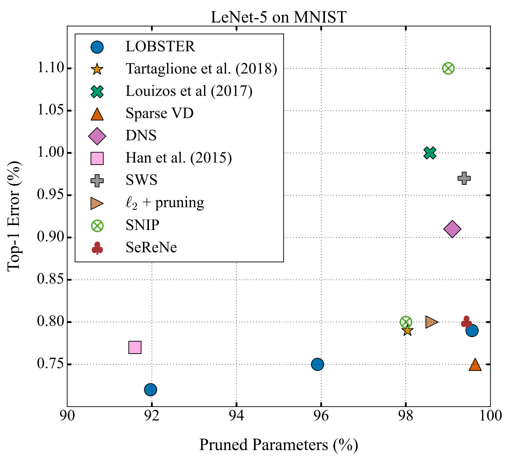
We compare with other state-of-the-art approaches wherever numbers are publicly available. Besides these, we also perform an ablation study with an L2-based regularizer and our proposed pruning strategy (as discussed in Section 4.2. We measure the performance as the achieved model sparsity versus classification error (Top-1 or Top-5 error). The network sparsity is defined here as the percentage of pruned parameters in the network model.

Our algorithms are implemented in Python[1], using PyTorch 1.2, and we used an RTX2080 Ti NVIDIA GPU to run the simulations. The validation set size for all the experiments is 5k images. All the hyper-parameters have been tuned via grid search. For all datasets, the learning and pruning stages take place on a random split of the training set, whereas the numbers reported below are related to the test set.

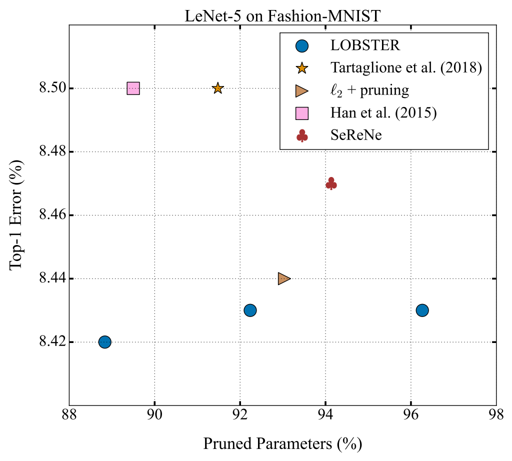
[1]the source code is available at https://github.com/EIDOSlab/LOBSTER.git



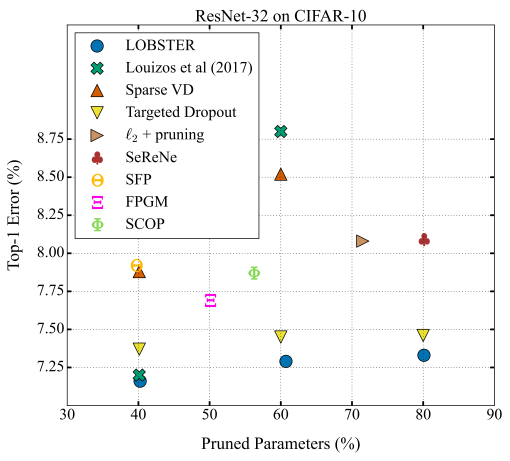
(a)



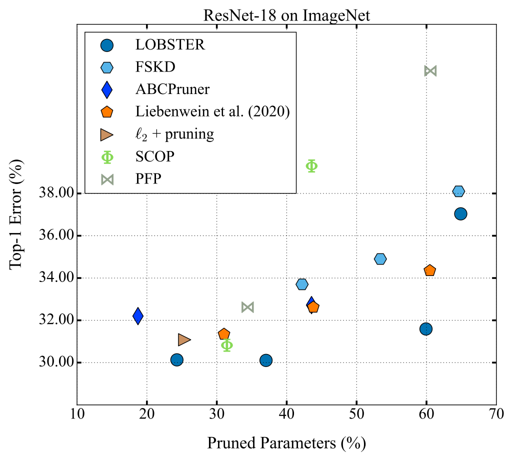
(b)



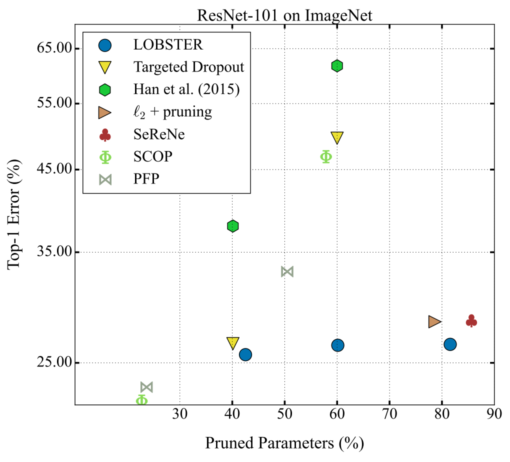
(c)



(d)



(e)



(f)

Fig. 4.4 Performance (Top-1 error) vs. ratio of pruned parameters for LOBSTER and other state-of-the-art methods over different architectures and datasets.

4.3.1 LeNet-300 on MNIST

As a first experiment, we trained a sparse LeNet-300 [30] architecture consisting of three fully-connected layers with 300, 100, and 10 neurons, respectively. We trained the network on the MNIST dataset, composed of 60k training and 10k test gray-scale 28x28 pixels large images depicting handwritten digits. Starting from a randomly initialized network, we trained LeNet-300 via SGD with learning rate $\eta = 0.1$, $\lambda = 10^{-4}$, $PWE = 20$ epochs and $TWT = 0.05$.

The related literature reports several compression results that can be clustered into two groups corresponding to classification error rates of about 1.65% and 1.95%, respectively. Figure 4.4a provides results for the proposed procedure. Our method reaches higher sparsity than the approaches found in the literature; this is particularly noticeable around the 1.65% classification error (low left in Figure 4.4a), where we achieve almost twice the sparsity of the second-best method.

LOBSTER also achieves the highest sparsity for the higher error range (right side of the graph), gaining especially in regards to the number of parameters removed from the first fully-connected layer (the largest, consisting of 235k parameters), in which we observe that just the 0.59% of the parameters survives.

4.3.2 LeNet-5 on MNIST and Fashion-MNIST

Next, we experiment with the Caffe version of the LeNet-5 architecture, consisting of two convolutional and two fully-connected layers. Again, we use a randomly-initialized network, trained via SGD with learning rate $\eta = 0.1$, $\lambda = 10^{-4}$, $PWE =$ 20 epochs, and $TWT = 0.05$. The results are shown in Figure 4.4b. As seen from the graph, even with a convolutional architecture, we obtain a competitively small network with a sparsity of 99.57%. At higher compression rates, Sparse VD slightly outperforms all other methods in the LeNet5-MNIST experiment.

We observe that LOBSTER, in this experiment, sparsifies the first convolutional layer (22% sparsity) more than the Sparse VD solution (33%). In particular, LOBSTER prunes 14 filters out of the 20 original filters in the first layer (in other words, just 6 filters survive and contain all the un-pruned parameters). However, since we are above 99% of sparsity, the difference between the two techniques is minimal. We hypothesize that, in the case of Sparse VD and for this particular dataset, extracting a wider variety of features at the first convolutional layer both eases the classification task (hence the lower Top-1 error) and allows to drop more parameters in the following layers (a slightly improved sparsity).

To scale up the difficulty of the training task, we experimented on the classification of the Fashion-MNIST dataset [31], using LeNet5 again. This dataset has the same size and image format as the MNIST dataset. However, it contains images of clothing items, resulting in a non-sparse distribution of the pixel intensity value. Since the images are not as sparse, such a dataset is notoriously more challenging to classify than MNIST. For this experiment, we trained the network from scratch using SGD with $\eta = 0.1$, $\lambda = 10^{-4}$, $PWE = 20$ epochs, and $TWT = 0.1$. The results are shown in Figure 4.4c. Fashion-MNIST is an inherently more challenging dataset than MNIST, so the possible sparsity is lower. Nevertheless, the proposed method still reaches higher sparsity than other approaches, removing a higher percentage of parameters, especially in the fully connected layers, while maintaining good generalization. In this case, we observe that the first layer is the least sparsified: this is an effect of the higher complexity of the classification task, which requires more features to be extracted.

4.3.3 ResNet-32 on CIFAR-10

To evaluate how our method scales to deeper, modern architectures, we applied it on a PyTorch implementation of the ResNet-32 network [32] that classifies the CIFAR-10 dataset.[2] This dataset consists of 60k 32×32 RGB images divided in 10 classes (50k training images and 10k test images). We trained the network using SGD with momentum $\beta = 0.9$, $\lambda = 10^{-6}$, $PWE = 10$ and $TWT = 0$. The full training is performed for 11k epochs.

Our method performs well on this task and outperforms other state-of-the-art techniques. Furthermore, LOBSTER improves the network generalization ability reducing the baseline Top-1 error from 7.37% to 7.33% of the sparsified network while removing 80.11% of the parameters. This effect is likely due to the LOBSTER technique, which self-tunes the regularization on the parameters as explained in Section 4.1.2.

[2]the source code is available at https://github.com/akamaster/pytorch_resnet_cifar10

Table 4.2 Results on the ISIC 2018 Skin Lesion Segmentation using U-Net architecture.

Method	Dice score	Intersection over Union	Sparsity (%)
Baseline	**0.8282**	**0.7073**	0
Sparse VD [33]	0.8245	0.7030	32.14
L_2+pruning	0.8273	0.7062	79.43
LOBSTER	0.8269	0.7057	**82.13**

4.3.4 ResNet on ImageNet

Finally, we further scale up the classification problem's output and complexity, testing the proposed method on the network over the well-known ImageNet dataset (ILSVRC-2012), composed of more than 1.2 million train images, for a total of 1k classes. For this test, we used SGD with momentum $\beta = 0.9$, $\lambda = 10^{-6}$, and $TWT = 0$.

The full training lasts 95 epochs. Due to time constraints, we decided to use the pre-trained network offered by the torchvision library.[3] Figure 4.4e shows the results for ResNet-18 while Figure 4.4f shows the results for ResNet-101.

4.3.5 U-Net on ISIC skin lesion segmentation

Besides classification tasks, we want to show how LOBSTER behaves for different tasks. Towards this end, we have trained the U-Net architecture [34] to segment skin lesions [35, 36]. The ISIC skin lesion segmentation dataset consists of 2594 training images and 100 test images having resolution 1024×768 pixels, in RGB format. Models are trained with weight decay = 10^{-4}, momentum = 0.9 starting learning rate $\eta = 0.1$. LOBSTER and L_2+pruning models were obtained with $PWE = 10$ and $TWT = 0$. For LOBSTER we used $\lambda = 10^{-4}$. All the models are trained to minimize a Jaccard loss function. Results are shown in Table 4.2. Even in segmentation tasks, LOBSTER can remove many parameters, namely the 82.13%. We observe, however, that in this specific scenario, the main contribution is given by the pruning algorithm we propose in Section 4.2, as the sparsity achieved with plain L_2 regularization is not distant though lower than LOBSTER (82.13%) when other techniques which perform well for classification tasks, like Sparse VD, in this case, can only prune

[3]https://pytorch.org/docs/stable/torchvision/models.html

Dataset	Architecture	L_2+pruning Top-1 (%)	L_2+pruning Sparsity (%)	L_2+pruning FLOPs	LOBSTER Top-1 (%)	LOBSTER Sparsity (%)	LOBSTER FLOPs
MNIST	LeNet-300	1.97	97.62	22.31k	1.95	99.13	10.63k
	LeNet-5	0.80	98.62	589.75k	0.79	99.57	207.38k
F-MNIST	LeNet-5	8.44	93.04	1628.39k	8.43	96.27	643.22k
CIFAR-10	ResNet-32	8.08	71.51	44.29M	7.33	80.11	32.90M
ImageNet	ResNet-18	31.08	25.40	2.85G	30.10	37.04	2.57G
	ResNet-101	28.33	78.67	3.44G	26.44	81.58	3.00G

Table 4.3 Comparing LOBSTER against standard L_2+pruning as in Figure 4.4 (best sparsity results are reported).The sensitivity-based regularization term allows higher sparsification rates for improved accuracy.

32.14% of the parameters. For these cases, proper tuning of the threshold T results determinant towards achieving high performance with a little performance drop.

4.3.6 Ablation study

As a final ablation study, we replace our sensitivity-based regularizer with a simpler L_2 regularizer in our learning scheme.

Such scheme "L_2+pruning" uniformly applies an L_2 penalty to all the parameters regardless of their contribution to the loss. This scheme is comparable with [15], yet enhanced with the same pruning strategy with adaptive thresholding shown in Figure 4.3. A comparison between LOBSTER and L_2+pruning is reported in Table 4.3.

In all the experiments, we observed that dropping the sensitivity-based regularizer impairs performance. This experiment verifies the role of sensitivity-based regularization in the performance of our scheme. Finally, Table 4.3 also reports the corresponding inference complexity in FLOPs. For the same or lower Top-1 error, LOBSTER yields benefits as fewer operations at inference time and suggests the presence of some structure in the sparsity achieved by LOBSTER.

4.4 Summary

We presented LOBSTER, a regularization method suitable to train neural networks with a sparse topology without preliminary training.

Unlike L_2 regularization, LOBSTER is aware of the global contribution of the parameters on the loss function and self-tunes the regularization effect depending on factors like the network's architecture or the training problem itself. Moreover, tuning its hyper-parameters is easy, and the optimal threshold for parameter pruning is self-determined by the proposed approach employing a validation set.

LOBSTER achieves competitive results from shallow architectures like LeNet-300 and LeNet-5 to deeper topologies like ResNet over ImageNet. In these scenarios, we have observed the boost provided by the proposed regularization approach towards less-unaware approaches like L_2 regularization in terms of achieved sparsity.

Chapter 5

SERENE

In the previous chapter, we presented LOBSTER, a technique that can reach high sparsity levels. Still, as mentioned in Chapter 3, the unstructured nature of the introduced sparsity can limit the practical applications of the pruned networks. To confront this issue, in this chapter, we propose a structured approach.

SeReNe (Sensitivity-based Regularization of Neurons) is a method for learning sparse topologies with a structure by exploiting the concept of neural sensitivity as a regularizer. We define the sensitivity of a neuron as the variation of the network output with respect to the variation of the neuron's activity (i.e., the post-synaptic potential of the neuron), and the lower the sensitivity of a neuron, the less the network output is perturbed if the neuron output changes. Thanks to this sensitivity formulation, this procedure can drive all the neuron's parameters to zero, allowing learning network topologies that are sparse and have fewer neurons (fewer filters for convolutional layers). As a side benefit, smaller and denser architectures may also speed up network execution thanks to better use of cache locality and memory access pattern.

We experimentally show that SeReNe outperforms state-of-the-art references over multiple learning tasks and network architectures. We observe the benefit of structured sparsity when storing the neural network topology and parameters using the Open Neural Network eXchange (ONNX) format [37], with a reduction of the memory footprint.

5.1 Sensitivity-based Regularization for Neurons

In this section, we first formulate the network's sensitivity with respect to a neuron's post-synaptic potential. Then, we derive a general parameter update rule which relies on the proposed sensitivity term. As a reference scenario, we considered a multi-class classification problem with C labels; however, our strategy can be extended to other learning tasks, e.g., regression, in a straightforward way.

5.1.1 Neuron Sensitivity

Let us assume that our method is applied to a pre-trained network. To estimate the relevance of neuron $x_{n,i}$ for the task upon which the network was trained, we evaluate the neuron contribution to the network output $\boldsymbol{y}_N$. To this end, we first provide intuition on how slight variations of the post-synaptic potential $p_{n,i}$ of neuron $x_{n,i}$ affect the k-th output of the network $y_{N,k}$. By a Taylor series expansion, for minor variations of $p_{n,i}$, let us express the variation of $y_{N,k}$ as

$$\Delta y_{N,k} \approx \Delta p_{n,i} \frac{\partial y_{N,k}}{\partial p_{n,i}} \tag{5.1}$$

where $y_{N,k}$ indicates the k-th output for the output layer. In the case $\Delta y_{N,k} \to 0, \forall k$, for small variations of $p_{n,i}$, $y_{N,k}$ does not change. Such a condition allows driving the post-synaptic potential $p_{n,i}$ to zero without affecting the network output $y_{N,k}$ (and, for instance, its performance). Otherwise, if $\Delta y_{N,k} \neq 0$, any variation of $p_{n,i}$ might alter the network output, possibly impairing its performance. We can now properly quantify small changes' effect on the network output by defining the *neuron sensitivity*.

We define the sensitivity of the network output $\boldsymbol{y}_N$ with respect to the post-synaptic potential $p_{n,i}$ of neuron $x_{n,i}$ as

$$S_{n,i}(\boldsymbol{y}_N, p_{n,i}) = \frac{1}{C} \sum_{k=1}^{C} \left| \frac{\partial y_{N,k}}{\partial p_{n,i}} \right| \tag{5.2}$$

where $\boldsymbol{y}_N \in R^C$ and $S_{n,i} \in [0; +\infty)$. Intuitively, the higher $S_{n,i}$, the higher the fluctuation of $\boldsymbol{y}_N$ for small variations of $p_{n,i}$.

Before moving on, we would like to clarify our choice of leveraging the post-synaptic potential $p_{n,i}$ rather than the neuron output $y_{n,i}$ in the equation above. In order to understand our choice, we re-write equation 5.2 using the chain rule:

$$S_{n,i}(\boldsymbol{y}_N, p_{n,i}) = \frac{1}{C}\sum_{k=1}^{C}\left|\frac{\partial y_{N,k}}{\partial y_{n,i}} \cdot \frac{\partial y_{n,i}}{\partial p_{n,i}}\right|. \tag{5.3}$$

Without loss of generality, let us assume $\frac{\partial y_{N,k}}{\partial y_{n,i}} \neq 0$ and $g_{n,i}(\cdot)$ corresponds to the well known ReLU activation function. Under the hypothesis that $p_{n,i} < 0$, $\frac{\partial y_{n,i}}{\partial p_{n,i}} = 0$ for the considered ReLU activation. Had we written equation 5.2 as a function of the neuron output $y_{n,i}$, the vanishing gradient $\frac{\partial y_{n,i}}{\partial p_{n,i}} = 0$ would have prevented us from estimating the neuron sensitivity. This consideration applies beyond ReLU to any activation function except for the identity function, for which $y_{n,i} = p_{n,i}$.

5.1.2 Bounds on Neuron Sensitivity

Here we provide two computationally-efficient bounds to the sensitivity function. Popular neural network training frameworks rely on differentiation frameworks, such as *autograd*, for automatic variable differentiation along computational graphs. Such frameworks take as input some objective function J and automatically compute all the gradients along the computational graph. In order to get $S_{n,i}$ as an outcome from the differentiation engine, we define

$$S_{n,i}(\boldsymbol{y}_N, p_{n,i}) = \frac{\partial J}{\partial p_{n,i}} \tag{5.4}$$

where J is a proper function. Such function turns out to be:

$$J = \frac{1}{C}\sum_{k=1}^{C}\int\left|\frac{\partial y_{N,k}}{\partial p_{n,i}}\right| dp_{n,i} \tag{5.5}$$

therefore, computing the sensitivity in equation 5.2 requires C calls to the differentiation engine. In the following, with some little algebra, we derive a lower and upper bound to equation 5.2 that we show to be particularly useful from a computational perspective.

Let the objective function to differentiate be

$$J^l = \frac{1}{C}\sum_{k=1}^{C} y_{N,k}. \tag{5.6}$$

The automatic differentiation engine called on S^l will return

$$\frac{\partial J^l}{\partial p_{n,i}} = \frac{1}{C}\sum_{k=1}^{C} \frac{\partial y_{N,k}}{\partial p_{n,i}}. \tag{5.7}$$

According to the triangle inequality, a lower bound to the sensitivity in equation 5.2 can be computed as

$$S^l_{n,i} = \frac{1}{C}\left|\sum_{k=1}^{C} \frac{\partial y_{N,k}}{\partial p_{n,i}}\right| \leq \frac{1}{C}\sum_{k=1}^{C}\left|\frac{\partial y_{N,k}}{\partial p_{n,i}}\right| \tag{5.8}$$

$S^l_{n,i}$ can be conveniently evaluated differentiating over equation 5.6 (and taking the absolute value) with a single call to the differentiation engine. As shown in equation 5.8, this gives us a lower bound estimation over the neuron sensitivity.

In order to estimate an upper bound to $S_{n,i}$, we rewrite equation 5.2 as

$$S_{n,i} = \frac{1}{C}\sum_{k=1}^{C}\left|\frac{\partial y_{N,k}}{\partial \boldsymbol{y}_{N-1}} \cdot \prod_{l=n+1}^{N-1} \frac{\partial \boldsymbol{y}_l}{\partial \boldsymbol{y}_{l-1}} \cdot \boldsymbol{\delta}_{n,i}\frac{\partial y_{n,i}}{\partial p_{n,i}}\right| \tag{5.9}$$

However, $\forall k$ we have in common the term

$$\boldsymbol{\Gamma}_{n,i} = \prod_{l=n+1}^{N-1} \frac{\partial \boldsymbol{y}_l}{\partial \boldsymbol{y}_{l-1}} \cdot \boldsymbol{\delta}_{n,i}\frac{\partial y_{n,i}}{\partial p_{n,i}} \leq \prod_{l=n+1}^{N-1}\left|\frac{\partial \boldsymbol{y}_l}{\partial \boldsymbol{y}_{l-1}}\right| \cdot \boldsymbol{\delta}_{n,i}\left|\frac{\partial y_{n,i}}{\partial p_{n,i}}\right| = \boldsymbol{\Gamma}^u_{n,i}$$

where $\boldsymbol{\delta}_{n,i}$ is a one-hot vector selecting the i-th neuron at the n-th layer and $|\cdot|$ is an element-wise operator. Hence, we rewrite equation 5.9 as

$$S^u_{n,i} = \frac{1}{C}\left(\sum_{k=1}^{C}\left|\frac{\partial y_{N,k}}{\partial \boldsymbol{y}_{N-1}}\right|\right) \cdot \boldsymbol{\Gamma}^u_{n,i} \geq S_{n,i}. \tag{5.10}$$

Thus, we have shown that $S^u_{n,i}$ is an upper bound to the sensitivity in equation 5.2.

Upper and lower bounds are obtained here for two main reasons: computational efficiency and relaxing/tightening conditions on the sensitivity itself. We will see in Section 5.3.1 a typical population distribution of the sensitivities on a pre-trained network, comparing equation 5.2, equation 5.8 and equation 5.10. In the following sections, we exploit the formulation of the sensitivity function equation 5.2 and its two bounds equation 5.8, equation 5.10 to define a parameter update rule.

5.1.3 Parameters Update Rule

As hinted before, if the sensitivity $S_{n,i}$ of neuron $x_{n,i}$ is small, i.e., $S_{n,i} \rightarrow 0$, then neuron $x_{n,i}$ yields a small contribution to the i-th network output $y_{N,i}$; its parameters can be moved towards zero with little perturbation to the network's output. To this end, we define the *insensitivity* function $\overline{S}_{n,i}$ as

$$\overline{S}_{n,i} = \max\{0, 1 - S_{n,i}\} = (1 - S_{n,i}) \cdot \Psi(1 - S_{n,i}) \tag{5.11}$$

where $\Psi(\cdot)$ is the one-step function. The higher the insensitivity of neuron $x_{n,i}$ (i.e., $\overline{S}_{n,i} \rightarrow 1$ or equivalently $S_{n,i} \rightarrow 0$), the less the neuron affects the network output. Therefore, if $\overline{S}_{n,i} \rightarrow 1$, then neuron $x_{n,i}$ contributes little to the network output and its parameters $w_{n,i,j}$ can be driven towards zero without significantly perturbing the network output. Using the insensitivity definition in equation 5.11, we propose the following update rule:

$$w^{t+1}_{n,i,j} = w^t_{n,i,j} - \eta \frac{\partial \mathcal{L}}{w^t_{n,i,j}} - \lambda w^t_{n,i,j} \overline{S}_{n,i} \tag{5.12}$$

where

- the first contribution term is the classical minimization of a loss function $\mathcal{L}$, ensuring that the network still solves the target task, e.g., classification;
- the second one represents a penalty applied to the parameter $w_{n,i,j}$ belonging to the neuron $x_{n,i}$, which is proportional to the insensitivity of the output to its variations.

Finally, since

$$\frac{\partial p_{n,i}}{\partial y_{n-1,j}} = w_{n,i,j} \tag{5.13}$$

we rewrite equation 5.12 as

$$w_{n,i,j}^{t+1} = w_{n,i,j}^{t} - \eta \frac{\partial \mathcal{L}}{w_{n,i,j}^{t}} - \lambda \dot{S}_{n,i,j} \tag{5.14}$$

where

$$\dot{S}_{n,i,j} = \left[w_{n,i,j} - \frac{\text{sign}(w_{n,i,j})}{C} \sum_{k=1}^{C} \left| \frac{\partial y_{N,k}}{\partial y_{n-1,j}} \right| \right] \cdot \Psi(1 - S_{n,i}) \tag{5.15}$$

From equation 5.15 we can better understand the effect of the proposed penalty term: as expected by our discussion above, $\dot{S}_{n,i,j}$ is inversely proportional to the impact on the output for variations of the input for the neuron $x_{n,i}$.

5.1.4 Local neuron sensitivity-based regularization

We propose now an approximate formulation of the sensitivity function in equation 5.2 based only on the post-synaptic potential and output of a neuron that we will refer to as the *local* sensitivity. Let us recall that, for each neuron $x_{n,i}$, the sensitivity provided by equation equation 5.2 measures the overall impact of a given neuron $x_{n,i}$ on the network output while taking into account all the following neurons involved in the computation.

The local neuron sensitivity of the output $y_{n,i}$ with respect to the post-synaptic potential $p_{n,i}$ of the neuron $x_{n,i}$ is defined as

$$\tilde{S}_{n,i} = \left| \frac{\partial y_{n,i}}{\partial p_{n,i}} \right|. \tag{5.16}$$

In the case of ReLU-activated networks, it simply reads

$$\tilde{S}_{n,i} = \Psi(p_{n,i}). \tag{5.17}$$

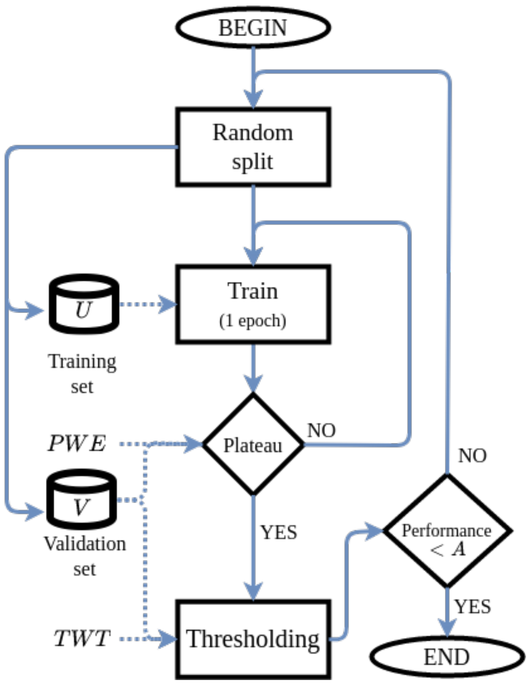


Fig. 5.1 High-level view of the SeReNe procedure.

Under this setting, the update rule equation 5.14 simplifies to

$$w_{n,i,j}^{t+1} = w_{n,i,j}^{t} - \eta \frac{\partial \mathcal{L}}{w_{n,i,j}^{t}} - \lambda w_{n,i,j}^{t} \Psi(-p_{n,i}), \tag{5.18}$$

i.e., the penalty is applied only in case the neuron stays off. While local sensitivity is a looser approximation of equation 5.2, it is far less complex to compute, especially for ReLU-activated neurons.

5.2 The SeReNe procedure

This section introduces a practical procedure to prune neurons from a neural network $\mathcal{N}$ leveraging the sensitivity-based regularizer presented above.

Let us assume $\mathcal{N}$ has been preliminary trained at some task over the dataset D, achieving performance (e.g., classification accuracy) A. We do not put any constraint over the actual training method, training set, or network architecture. Algorithm 2 summarizes the procedure in pseudo-code. In a nutshell, the procedure consists in iteratively looping over the *Regularization* and *Thresholding* procedures.

At the beginning of the loop, dataset D is split into disjoint subset V (used for validation purposes) and U (to update the network). At line 5, the regularization procedure (summarized in Algorithm 3) trains $\mathcal{N}$ over D according to equation 5.12 driving towards zero parameters of neurons with low sensitivity. The loop ends if the performance of the regularized network falls below threshold A. Otherwise, the thresholding procedure sets to zero parameters below threshold T and prunes neurons such that all parameters are equal to zero. The procedure's output is the pruned network, i.e., with fewer neurons, $\mathcal{N}^\star$. The Regularization and Thresholding procedures are detailed in the following. A high-level graphical representation of SeReNe is also displayed in Figure 5.1.

Algorithm 2 The SeReNe procedure

```
    Input: Trained network $\mathcal{N}$, dataset D,
    Target performance A, PWE, TWT
    Output: Pruned network $\mathcal{N}^\star$
1:  procedure SeReNe($\mathcal{N}, D, A, PWE, TWT$)
2:      $\mathcal{N}^\star \leftarrow \mathcal{N}$
3:      while true do
4:          $U, V \leftarrow$ RandomSplit($D$)
5:          $\mathcal{N} \leftarrow$ Regularization($\mathcal{N}, U, V, PWE$)
6:          if Performance($\mathcal{N}, V$) $< A$ then
7:              break
8:          $\mathcal{N}^\star \leftarrow \mathcal{N}$
9:          $\mathcal{N} \leftarrow$ Thresholding($\mathcal{N}, V, TWT$)
        return $\mathcal{N}^\star$
```

5.2.1 Regularization

This procedure takes in input a network $\mathcal{N}$ and returns a regularized network according to the update rule equation 5.12. Namely, the procedure iteratively trains $\mathcal{N}$ on U and validates it on V for multiple epochs. Let $\mathcal{N}^r$ represent the *best* regularized network found at a given time according to the loss function. For each iteration, the procedure operates as follows. First (line 5), $\mathcal{N}$ is trained for one epoch over U: the result is a regularized network according to equation 5.12. Second (line 6), this network is validated on V. If the loss is lower than the loss of $\mathcal{N}^r$ over V, then $\mathcal{N}$ takes the place of $\mathcal{N}^r$ (line 7). If $\mathcal{N}^r$ is not updated for PWE (*Plateau Waiting*

Epochs) epochs, we assume we have reached a performance plateau. In this case, the procedure ends and returns the sensitivity-regularized network $\mathcal{N}^r$.

Algorithm 3 The regularization procedure

```
    Input: Model $\mathcal{N}$, data sets V and U, PWE
    Output: The sensitivity-regularized network $\mathcal{N}^r$
 1: procedure REGULARIZATION($\mathcal{N}, U, V, PWE$)
 2:     $\mathcal{N}^r \leftarrow \mathcal{N}$        ▷ $\mathcal{N}^r$ is best regularized network on $V$
 3:     $epochs \leftarrow 0$
 4:     while $epochs < PWE$ do
 5:         $\mathcal{N} \leftarrow$ TRAIN($\mathcal{N}, U$)        ▷ 1 train epoch on $U$
 6:         $epochs++$
 7:         if LOSS($\mathcal{N}, V$) < LOSS($\mathcal{N}^r, V$) then
 8:             $\mathcal{N}^r \leftarrow \mathcal{N}$
 9:             $epochs \leftarrow 0$
        return $\mathcal{N}^r$
```

5.2.2 Thresholding

The *thresholding* procedure is where the parameters of neurons with low sensitivity are thresholded to zero. Namely, parameters whose absolute value is below threshold T are pruned as

$$w_{n,i,j} = \begin{cases} w_{n,i,j} & |w_{n,i,j}| > T \\ 0 & otherwise. \end{cases} \tag{5.19}$$

The pruning threshold T is selected so that the performance (or, in other words, the loss on V) worsens at most of a relative value we call *thresholding worsening tolerance* (TWT) we provide as hyper-parameter. We expect the loss function to be locally a smooth, monotone function of T for small values of T. The threshold T can be found using linear search-based heuristics. We can, however, reduce this using a bisection approach, converging to the optimal T value in log-time steps. Because of the stochasticity introduced by mini-batch-based optimizers, parameters pruned during a thresholding iteration may be reintroduced by the following regularization iteration. To overcome this effect, we enforce that pruned parameters can no longer be updated during the following regularizations (we term this behavior as *parameter pinning*). To this end, the update rule equation 5.12 is modified as follows:

$$w_{n,i,j}^{t+1} = \begin{cases} w_{n,i,j}^t - \eta \frac{\partial \mathcal{L}}{w_{n,i,j}^t} - \lambda w_{n,i,j}^t \overline{S}_{n,i} & w_{n,i,j}^t \neq 0 \\ 0 & w_{n,i,j}^t = 0 \end{cases} \tag{5.20}$$

We have noticed that without parameter pinning, the compression of the network may remain low because the noisy gradient estimates in a mini-batch keep reintroducing previously pruned parameters. On the contrary, by adding equation 5.20, a lower number of epochs is sufficient to achieve much higher compression.

5.3 Results

In this section, we experiment with our proposed neuron pruning method, comparing the four sensitivity formulations we introduced in the previous section:

– SeReNe (exact) - the exact formulation in equation 5.2;

– SeReNe (LB) - the lower bound in equation 5.8;

– SeReNe (UB) - the upper bound in equation 5.10;

– SeReNe (local) - the local version in equation 5.16;

– L_2 + pruning - is a baseline reference where we replace our sensitivity-based regularization term with a standard L_2 term (all the rest of the framework is identical).

We experiment with different combinations of architectures and datasets commonly used as benchmarks in the relevant literature:

– LeNet-300 on MNIST (Table 5.1 and Table 5.2),

– LeNet-5 on MNIST (Table 5.3),

– LeNet-5 on Fashion-MNIST (Table 5.4),

– VGG-16 on CIFAR-10 (Table 5.5 and Table 5.6),

– ResNet-32 on CIFAR-10 (Table 5.7),

– AlexNet on CIFAR-100 (Table 5.8),

– ResNet-101 on ImageNet (Table 5.9).

Notice that the VGG-16, AlexNet and ResNet-32 architectures are modified to fit the target classification task (CIFAR-10 and CIFAR-100). The validation set (V) size for all experiments is 10% of the training set. The pruning performance is evaluated according to multiple metrics.

– The *compression ratio*, i.e., the ratio between the number of parameters in the original network and the number of remaining parameters after pruning (the higher, the better).

– The number of remaining neurons (or filters for convolutional layers) after pruning.

– The size of the networks when stored on disk in the popular ONNX format [37] (`.onnx` column). ONNX files are then losslessly compressed using the Lempel–Ziv–Markov algorithm (LZMA) [38] (`.7z` column).

In our experiments, we compare all available references for each combination of architecture and dataset. For this reason, the reference set may vary from experiment to experiment. Our algorithms are implemented in Python, using PyTorch 1.5, and simulations are run on an RTX2080 NVIDIA GPU with 8GB of memory.[1]

5.3.1 Preliminary experiment

First, we plot the sensitivity distribution for a LeNet-5 network trained on the MNIST dataset (SGD with learning rate $\eta = 0.1$ weight decay 10^{-4}). This network will also be used as the baseline in Sec. 5.3.3. Figure 5.2 shows SeReNe (exact) (red), SeReNe (LB) (green), and SeReNe (UB) (blue); the vertical bars represent the mean values. As expected, SeReNe (LB) and SeReNe (UB) under-estimate and over-estimate SeReNe (exact), respectively. Interestingly, SeReNe (UB) sensitivity values lie in the range $[10^{-4}; 10^{-2}]$ while both for SeReNe (exact) and SeReNE (LB) show a longer trail towards smaller figures, whereas all distributions look similar.

[1]the source code is available at https://github.com/EIDOSlab/SeReNe.git

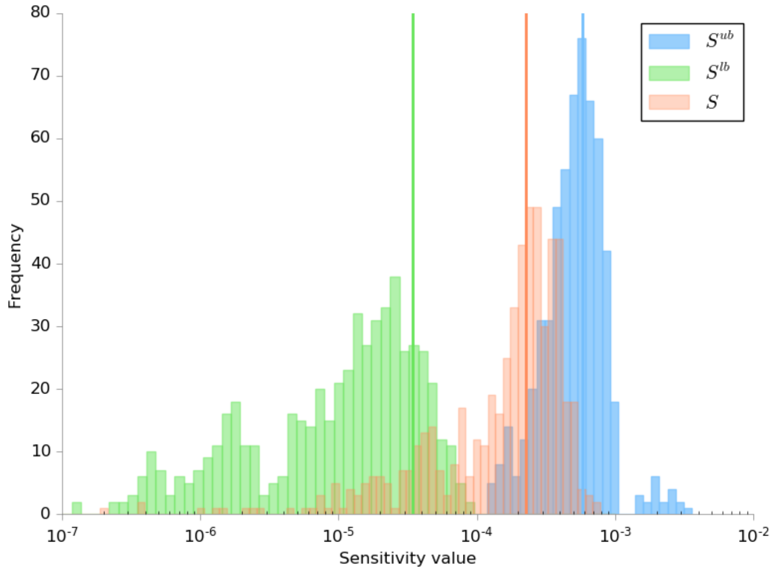


Fig. 5.2 Population of sensitivities S and relative lower S^l and upper S^u bounds for a LeNet-5 architecture pre-trained on MNIST. Vertical bars indicate relative mean values.

5.3.2 LeNet-300 on MNIST

As a first experiment, we prune a LeNet-300 architecture consisting of three fully-connected layers with 300, 100, and 10 neurons, respectively, trained over the MNIST dataset. We pre-trained LeNet-300 via SGD with learning rate $\eta = 0.1$ and $PWE = 20$ epochs with $\lambda = 10^{-5}$, $TWT = 0.3$ for SeReNe (exact), SeReNe (LB) SeReNe (UB) and $\lambda = 10^{-5}$, $TWT = 1$ for SeReNe (local). The related literature reports mainly i) results for classification errors around 1.65% (Table 5.1) and ii) results for errors in the order of 1.95% (Table 5.2). For this reason, we trained for about 1k epochs to achieve a 1.95% error rate and other 2k epochs to score a 1.65%

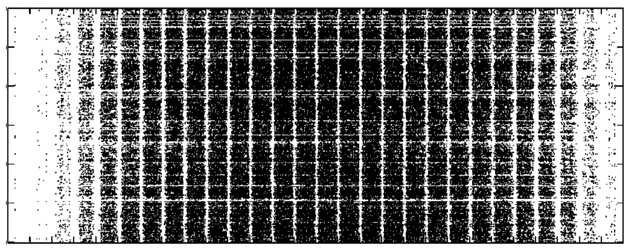
Table 5.1 LeNet-300 trained on MNIST (1.65% error rate).

Approach	Remaining parameters (%)			Compr.	Remaining	Network size [kB]			Training time	Top-1
	FC1	FC2	FC3	ratio	neurons	.onnx		.7z	(s/epoch)	(%)
Baseline	100	100	100	1x	[300]-[100]-[10]	1043	$\rightarrow$	933	**3.65**	**1.44**
Han *et al.* [12]	8	9	26	12.2x	-		-		-	1.6
Tartaglione *et al.* [11]	2.25	11.93	69.3	27.87x	[251]-[88]-[10]		-		-	1.65
ℓ_2+pruning	2.44	15.76	68.50	23.26x	[212]-[82]-[10]	723	$\rightarrow$	64	3.65	1.66
SeReNe (exact)	**1.42**	**9.54**	60.9	**42.55x**	**[159]-[75]**-[10]	**538**	$\rightarrow$	**46**	13.25	1.64
SeReNe (UB)	22.45	60.81	87.75	3.71x	[295]-[92]-[10]	1016	$\rightarrow$	324	5.13	1.67
SeReNe (LB)	1.51	10.05	**60.53**	39.79x	[164]-[78]-[10]	557	$\rightarrow$	55	4.88	1.65
SeReNe (local)	3.85	32.53	73.49	13.81x	[251]-[86]-[10]	859	$\rightarrow$	119	3.83	1.64

Table 5.2 LeNet-300 trained on MNIST (1.95% error rate).

Approach	Remaining parameters (%)			Compr.	Remaining	Network size [kB]			Training time	Top-1
	FC1	FC2	FC3	ratio	neurons	`.onnx`		`.7z`	(s/epoch)	(%)
Baseline	100	100	100	1x	[300]-[100]-[10]	1043	$\rightarrow$	933	**3.65**	**1.44**
Sparse VD [20]	-	-	-	68x	-		-		-	1.92
SWS [39]	-	-	-	23x	-		-		-	1.94
Tartaglione *et al.* [11]	0.93	**1.12**	5.9	**103x**	[179]-[88]-[10]		-		-	1.95
DNS [40]	1.8	1.8	**5.5**	56x	-		-		-	1.99
ℓ_2+pruning	1.22	8.77	61.10	41.95x	[167]-[76]-[10]	566	$\rightarrow$	42	3.65	1.97
SeReNe (exact)	0.76	5.85	49.77	66.28x	[148]-**[70]**-[10]	498	$\rightarrow$	38	13.25	1.93
SeReNe (UB)	13.67	50.76	84.47	5.47x	[293]-[91]-[10]	1008	$\rightarrow$	240	5.13	1.95
SeReNe (LB)	**0.75**	5.79	49.3	66.41x	**[146]**-**[70]**-[10]	**492**	$\rightarrow$	**37**	4.88	1.95
SeReNe (local)	1.7	19.94	63.59	25.07x	[192]-[83]-[10]	656	$\rightarrow$	70	3.83	1.93

error rate. SeReNe outperforms the other methods in terms of compression ratio and the number of pruned neurons. SeReNe (exact) achieves a compression ratio of 42.55$\times$, and the number of remaining neurons in the hidden layers drops from 300 to 159 and from 100 to 75, respectively. SeReNe (LB) enjoys comparable performance to SeReNe (exact) despite the lower computational cost.



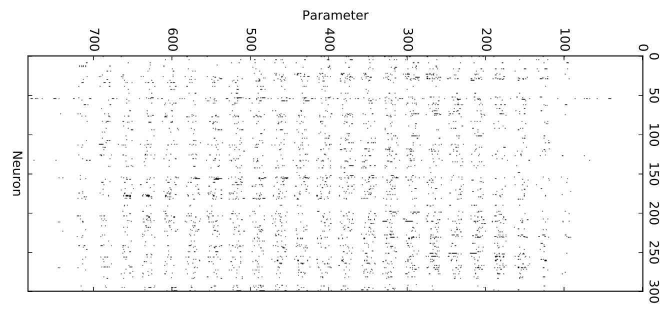


Fig. 5.3 Parameters distribution in FC1 of LeNet-300 trained on MNIST from Han *et al.* (top), and he proposed SeReNe (bottom). In black are the remaining parameters.

For the 1.95% error band (Tab. 5.2), SeReNe (LB) performs more effectively at pruning parameters than SeReNe (exact), allowing lower error. In this case, SeReNe (LB) achieves comparable compression to other state-of-the-art approaches, except for Tartaglione *et al.* [11]. However, when we compare the final architecture, we see that [11] prunes fewer neurons than SeReNe, despite a higher compression ratio. Evidently, [11] enhances unstructured sparsity, while SeReNe exploits structured sparsity, resulting in more entirely-removed neurons. Serene (LB) prunes more parameters than SeReNe (UB), we hypothesize because equation 5.10 overestimates the sensitivity of the parameters and prevents them from being pruned. On the other side, SeReNe (LB) underestimates the sensitivity; however, small λ values set this off. SeReNe (local) prunes fewer parameters than the other SeReNe formulations as it relies on a locally computed sensitivity formulation despite lower complexity. Concerning training time (second column from the right), SeReNe (local) is the fastest and introduces very little computational overhead, SeReNe (UB) and SeReNe (LB) have comparable training times, and the slowest is the SeReNe (exact), approximately 2.7x slower than its boundaries. In light of the excellent trade-off between the ability to prune neurons, error rate, and training time of SeReNe (LB), we will restrict our experiments to this sensitivity formulation in the following.

Figure 5.3 (bottom) shows the location of the parameters not pruned by SeReNe (exact) in LeNet-300's first fully-connected layer (black dots). For comparison, we report the equivalent image from Figure 4 of [12] (top). Our method yields blank columns in the matrix that can be represented in memory as uninterrupted sequences of zeroes. When stored on disk, LZMA compression (`.7z` column) is particularly effective at encoding long sequences of the same symbol, which explains the 10x compression rate it achieves (from 538 to 46 kB) over the `.onnx` file.

Finally, we perform an ablation study to assess the impact of a simpler L_2-only regularization, i.e., classical weight decay, in place of our sensitivity-based regularizer. Towards this end, we retrain LeNet-300 with $\lambda = 0$ and a weight decay set to 10^{-4} in its place (line L_2+pruning in the tables above). We point out in equation 5.12 that the sensitivity can be interpreted as a weighting factor for the L_2-regularization. Using weight decay is equivalent to assuming all the parameters have the same sensitivity. For this experiment, we used $\eta = 0.1$, $PWE = 5$ and $TWT = 0$ ($TWT > 0$ significantly and uncontrollably worsens the performance). Table 5.1 shows that such a method is less effective at pruning neurons than SeReNe (LB), which removes 15% more neurons. Similar conclusions can be drawn if a

Table 5.3 LeNet-5 trained on MNIST.

Approach	Remaining parameters (%)				Compr.	Neurons	Network size [kB]			Top-1
	Conv1	Conv2	FC1	FC2	ratio		`.onnx`		`.7z`	(%)
Baseline	100	100	100	100	1x	[20]-[50]-[500]-[10]	1686	$\rightarrow$	1510	**0.68**
Sparse VD [20]	33	**2**	**0.2**	5	**280x**	-		-		0.75
Han *et al.* [12]	66	12	8	19	11.9x	-		-		0.77
SWS [39]	-	-	-	-	162x	-		-		0.97
Tartaglione *et al.* [11]	67.6	11.8	0.9	31.0	51.1x	[20]-[48]-[344]-[10]		-		0.78
DNS [40]	**14**	3	0.7	**4**	111x	-		-		0.91
ℓ_2+pruning	60.20	7.37	0.61	22.14	72.3	[19]-[37]-[214]-[10]	577	$\rightarrow$	46	0.8
SeReNe (LB)	33.75	3.25	0.27	10.22	177.05x	**[11]-[26]-[113]-[10]**	**208**	$\rightarrow$	**19**	0.8

Table 5.4 LeNet-5 trained on Fashion-MNIST.

Approach	Remaining parameters (%)				Compr.	Neurons	Network size [kB]			Top-1
	Conv1	Conv2	FC1	FC2	ratio		`.onnx`		`.7z`	(%)
Baseline	100	100	100	100	1x	[20]-[50]-[500]-[10]	1686	$\rightarrow$	1510	**8.1**
Sparse VD [20]	-	-	-	-	6.98x	-		-		8.53
Tartaglione *et al.* [11]	**76.2**	32.56	6.5	**44.02**	11.74x	[20]-**[47]**-[470]-[10]		-		8.5
ℓ_2+pruning	85.80	34.13	4.57	55.24	14.36x	[20]-[50]-[500]-[10]	1496	$\rightarrow$	197	8.44
SeReNe (LB)	85.71	**32.14**	**3.63**	52.03	**17.04x**	[20]-[49]-**[449]**-[10]	**1494**	$\rightarrow$	**46**	8.47

higher error is tolerated, as in Table 5.2. The L_2+pruning has been performed for comparison in all following experiments in the paper, yielding the same results.

5.3.3 LeNet5 on MNIST and Fashion-MNIST

Next, we repeat the previous experiment over the LeNet-5 [41] architecture, preliminarily trained as for the LeNet-300 above, yet with SGD with learning rate $\eta = 0.1$ and $PWE = 20$ epochs. We experiment with SeReNe (LB) with parameters ($\lambda = 10^{-4}$, $TWT = 1.45$). Our method requires about 500 epochs for this architecture to achieve the same error range as other state-of-the-art references. According to Table 5.3, SeReNe (LB) approaches the classification accuracy of its competitors outperforms the considered references in terms of compression ratio and pruned neurons. In this case, the benefits of the structured sparsity are evident: the uncompressed network storage footprint decreases from 1686 kB to 208 kB (-90%), which after lossless compression further decreases to 19 kB with a 0.12% performance drop only.

Then, we experiment with the same LeNet-5 architecture on the Fashion-MNIST [42] dataset. Fashion-MNIST has the same size as the MNIST dataset, yet it contains

Table 5.5 VGG-like architecture with 1 fully connected layer (*VGG-1*) trained on CIFAR-10.

Approach	Remaining parameters (%) [neurons]						Compr.	Network size [MB]			Top-1
	Conv1	Conv2	Conv3	Conv4	Conv5	FC1	ratio	.onnx		.7z	(%)
Baseline	100	100	100	100	100	-	1x	57.57	$\rightarrow$	51.51	**7.36**
	[64]	[128]	[256]	[512]	[512]	[10]					
	[64]	[128]	[256]	[512]	[512]						
			[256]	[512]	[512]						
ℓ_2+pruning	11.86	15.07	6.59	0.36	0.11	66.70	88.84x	13.58	$\rightarrow$	1.14	7.79
	[23]	[126]	[250]	[406]	**[60]**	[10]					
	[64]	[123]	[251]	[108]	[81]						
			[250]	**[128]**	[398]						
SeReNe (LB)	10.18	11.68	**4.73**	**0.20**	**0.05**	**61.11**	**124.82x**	**11.56**	$\rightarrow$	**0.97**	7.8
	[23]	[126]	[250]	**[382]**	[65]	[10]					
	[64]	[123]	[251]	**[93]**	**[76]**						
			[250]	[136]	**[373]**						

natural images of dresses, shoes, etc., so it is harder to classify than MNIST since the images are not as sparse as MNIST digits. In this experiment, we used SGD with learning rate $\eta = 0.1$ and $PWE = 20$ epochs. For SeReNe (LB) we used $\lambda = 10^{-5}$ and $TWT = 1$ for about 2k epochs. Unsurprisingly, the average compression ratio is lower than for MNIST: since the classification problem is much harder than MNIST (Sec. 5.3.3), more complexity is required and SeReNe, in order not to degrade the Top-1 performance, is not pruning as much as it did for the MNIST experiment. Most importantly, the SeReNe (LB) compressed network is 46 kB only, despite the higher number of pruned parameters.

5.3.4 VGG on CIFAR-10.

Next, we experiment with two popular implementations of the VGG architecture [43]. We recall that VGG consists of 13 convolutional layers arranged in 5 groups of 2, 2, 3, 3, and 3 layers, with 64, 128, 256, 512, and 512 filters per layer, respectively. *VGG-1* is a VGG implementation widespread in CIFAR-10 experiments that includes only one fully-connected layer as the output layer and is pre-trained on ImageNet.[2] *VGG-2* [20] is similar to VGG-1 but includes one hidden fully connected layer with 512 neurons before the output layer. We experiment over the CIFAR-10 dataset, which consists of 50k 32×32, RGB images for training, and 10k for testing,

[2]https://github.com/kuangliu/pytorch-cifar

Table 5.6 VGG-like architecture with 2 fully connected layers (*VGG-2*) trained on CIFAR-10.

Approach	Remaining parameters (%) [neurons]							Compr.	Network size [MB]			Top-1
	Conv1	Conv2	Conv3	Conv4	Conv5	FC1	FC2	ratio	`.onnx`		`.7z`	(%)
Baseline	100	100	100	100	100	100	100	1x	58.61	$\rightarrow$	52.44	**6.16**
	[64]	[128]	[256]	[512]	[512]	[512]	[10]					
	[64]	[128]	[256]	[512]	[512]							
			[256]	[512]	[512]							
Sparse-VD [20]	-	-	-	-	-	-	-	48x		-		7.3
ℓ_2+pruning	27.62	30.74	13.67	0.88	0.24	1.88	70.78	40.96x	34.42	$\rightarrow$	2.86	7.21
	[44]	[126]	[247]	[498]	[409]	[367]	[10]					
	[60]	[120]	[247]	[463]	[417]							
			[243]	[79]	[461]							
SeReNe (LB)	**25.9**	**26.38**	**9.75**	**0.48**	**0.15**	**1.24**	**70**	**57.99x**	**29.41**	$\rightarrow$	**2.47**	7.25
	[44]	[126]	[247]	[498]	**[354]**	[367]	[10]					
	[60]	[120]	[247]	**[433]**	**[366]**							
			[243]	**[65]**	**[459]**							

distributed in 10 classes. For both VGG-1 and VGG-2, we have used SGD with learning rate $\eta = 0.01$ and $PWE = 20$ epochs. For the SeReNe (LB), we used $\lambda = 10^{-6}$ and $TWT = 1.5$. Both architectures were pruned for approximately 1k epochs, and Tables 5.5 and 5.6 detail the pruned topologies. For each architecture, we detail the number of surviving filters (convolutional layers) or neurons (fully connected layers) for each layer within square brackets. The tables show that SeReNe introduces a significantly structured sparsity for both VGG-1 and VGG-2 and outperforms Sparse-VD [20] in terms of compression ratio. We can prune a significant number of filters also in the convolutional layers; as an example, the three layers in block Conv4 are reduced to [382]-[93]-[136] for VGG-1 and [498]-[433]-[65] for VGG-2. This positively impacts the network's footprint: VGG-1 memory footprint drops from 57.57 MB to 11.56 MB for the pruned network, while the *7zip* compressed representation is 0.97 MB only. VGG-2's memory footprint decreases from 58.61 MB to 29.41 MB, while the compressed file representation amounts to 2.47 MB.

5.3.5 ResNet-32 on CIFAR-10

We then evaluate SeReNe over the ResNet-32 architecture [3] trained on the CIFAR-10 dataset using SGD with learning rate $\eta = 0.001$, momentum 0.9, $\lambda = 10^{-5}$, $TWT = 0$ and $PWE = 10$. Table 5.7 shows the resulting architecture. Due to the

Table 5.7 ResNet-32 trained on CIFAR-10.

Approach	Remaining parameters (%) [neurons]					Compr.	Network size [MB]			Top-1
	Conv1	Block1	Block2	Block3	FC1	ratio	`.onnx`		`.7z`	(%)
Baseline	100	100	100	100	100	1x	1.84	$\rightarrow$	1.63	**7.36**
	[64]	[160]	[320]	[640]	[10]					
Sparse VD [20]	-	-	-	-	-	2.5x		-		8.16
ℓ_2+pruning	65.97	33.30	33.41	26.32	88.75	3.51x	1.82	$\rightarrow$	0.54	8.08
	[14]	[157]	[319]	[633]	[10]					
SeReNe (LB)	**60.19**	**24.52**	**24.14**	**17.84**	**81.88**	**5.03x**	**0.87**	$\rightarrow$	**0.37**	8.09
	[**12**]	[**93**]	[**203**]	[**364**]	[10]					

Table 5.8 AlexNet trained on CIFAR-100.

Approach	Remaining parameters (%) [neurons]								Compr.	Network size [MB]			Top-1	Top-5
	Conv1	Conv2	Conv3	Conv4	Conv5	FC1	FC2	FC3	ratio	`.onnx`		`.7z`	(%)	(%)
Baseline	100	100	100	100	100	100	100	100	1x	92.31	$\rightarrow$	79.27	45.58	20.09
	[64]	[192]	[384]	[256]	[256]	[4096]	[4096]	[100]						
Sparse VD [20]	-	-	-	-	-	-	-	-	26.45x		-		49.62	20.93
ℓ_2+pruning	75.00	21.95	5.21	3.65	5.59	0.62	0.17	6.44	114.45x	60.88	$\rightarrow$	3.56	46.43	19.91
	[64]	[192]	[384]	[256]	[256]	[4094]	[2180]	[100]						
SeReNe (LB)	**79.05**	**20.33**	**5.72**	**3.33**	**2.23**	**0.18**	**0.04**	**2.77**	**179.52x**	**43.80**	$\rightarrow$	**2.47**	**44.99**	**17.88**
	[64]	[**191**]	[384]	[256]	[256]	[**3322**]	[**1310**]	[100]						

number of layers, we represent the network architecture in five different blocks: the first corresponds to the first convolutional layer that takes in input the original input image, and the last represents the fully-connected output layer. The other three blocks in the middle represent the rest of the network, based on the number of output channels of each layer: *block1* contains all the layers with an output of 16 channels, *block2* contains all the layers with an output of 32 channels and *block3* collects the layers with an output of 64 channels. ResNet is an already optimized architecture, so it is more challenging to prune compared to, e.g., VGG. Nevertheless, SeReNe can still prune about 40% of the neurons and 70% of the parameters over the original ResNet-32. This is reflected in the size of the network, which drops from 1.84 MB (1.63 MB compressed) to 0.87 MB (0.57MB compressed).

5.3.6 AlexNet on CIFAR-100

Next, we upscale the output dimensionality of the learning problem, i.e., in the number of classes C, testing the proposed method on an AlexNet-like network over the CIFAR-100 dataset. Such a dataset consists of 32×32 RGB images divided

Table 5.9 ResNet-101 trained on ImageNet.

Approach	Remaining parameters (%) [neurons]						Compr.	Network size [kB]			Top-1	Top-5
	Conv1	Block1	Block2	Block3	Block4	FC1	ratio	`.onnx`		`.7z`	(%)	(%)
Baseline	100	100	100	100	100	100	1x	174.49	$\rightarrow$	156.67	**22.63**	**6.44**
	[64]	[1408]	[3584]	[36352]	[11264]	[1000]						
Sparse VD [20]	-	-	-	-	-	-	2.48x		-		35.76	13.45
ℓ_2+pruning	**53.12**	25.42	25.57	13.71	17.74	51.94	5.75x	172.94	$\rightarrow$	32.93	28.33	9.18
	[49]	[1241]	[3280]	[33278]	[11250]	[1000]						
SeReNe (LB)	55.36	**24.27**	**23.79**	**11.24**	**14.81**	**40.82**	**6.94**x	**172.15**	$\rightarrow$	**27.84**	28.41	9.45
	[49]	[**1197**]	[**3142**]	[**31948**]	[**11249**]	[1000]						

into 100 classes (50k training images, 10k test images). In this experiment, we use SGD with learning rate $\eta = 0.1$ and $PWE = 20$ epochs. Concerning SeReNe (LB), we used $\lambda = 10^{-5}$ and $TWT = 1.5$ and the pruning process lasted 300 epochs. Table 5.8 shows compression ratios over 179x, whereas the network size drops from 92.31 MB to 43.80 MB and further to 2.47 MB after compression. We hypothesize that the larger number of target classes to discriminate prevents pruning neurons in the convolutional layers. Yet, it allows the pruning of a significant number of neurons from the hidden, fully connected layers. Contrarily from the previous experiments, the top-5 and the top-1 errors *improve* with respect to the baseline.

5.3.7 ResNet-101 on ImageNet

As a last experiment, we test SeReNe on ResNet-101 trained over ImageNet (ILSVRC-2012), using the pre-trained network provided by the torchvision library.[3] Due to the long training time, we employed a batch-wise heuristic such that, instead of waiting for a performance plateau, the pruning step is taken every time a fifth of the train set (around 7.9k iterations) has been processed. We trained the network using SGD with a learning rate $\eta = 0.001$ and momentum 0.9; for SeReNe (LB), we used $\lambda = 10^{-6}$ and $TWT = 0$. Table 5.9 shows the result of the pruning procedure with the layers grouped in blocks similarly as for the ResNet-32 experiment. Despite the complexity of the classification problem (1000 classes) that make it challenging to prune entire neurons, we prune around 86% of the parameters and obtain a smaller network, especially when compressed, going from 156.67 MB to only 27.84 MB. Comparing SeReNe (LB) to L_2+pruning, we observe a boost in the performance when using SeReNe, but not as wide as in previous results. If the classification task to solve is

[3]https://pytorch.org/docs/stable/torchvision/models.html

complex (as for ImageNet), the sensitivity of many neurons will be high (because we need more neurons to solve the task). If many neurons have a comparable sensitivity, then SeReNe's regularization naturally reduces to L_2 regularization. This is one of the major benefits of SeReNe, which self-tunes the penalty to the neurons according to the model's complexity and the complexity of solving the target task.

5.4 Summary

In this work, we have proposed a sensitivity-driven neural regularization technique. The effect of this regularizer is to penalize all the parameters belonging to a neuron whose output is not influential in the output of the network. We have learned that evaluating the sensitivity at the neuron level (SeReNe) is extremely important to promote a structured sparsity, obtaining a smaller network with minimal performance loss. Our experiments show that the SeReNe strikes a favorable trade-off between the ability to prune neurons and computational cost while controlling the impairment in classification performance. Our sensitivity-based approach introduced a structured sparsity while achieving state-of-the-art compression ratios for all the tested architectures and datasets. Furthermore, using cross-validation, the designed sparsifying algorithm guarantees minimal (or no) performance loss, which the user can tune via a hyper-parameter (TWT).

Chapter 6

Simplify

As seen throughout this thesis, modern pruning techniques allow for an impressive theoretical reduction in both memory requirements and inference time for state-of-the-art neural network architectures. However, these procedures are usually limited to only identifying which portion of the weights can be set to zero, offering little to no practical advantages when the model is deployed to resource-constrained devices such as mobile phones or embedded systems. While most of the pruning-related works report some form of theoretical speedup, either in terms of Floating Point Operations (FLOPs) or inference speed [44], this does not always reflect the achievable performance gain, and it is usually overestimated.

We try to fill this gap and propose *Simplify*[1]. This PyTorch-compatible simplification library allows obtaining a smaller model in which the pruned neurons are removed and do not weigh on the size and inference time of the network. This technique can be used to correctly evaluate the actual impact of a pruning procedure when applied to a given network architecture. Moreover, Simplify allows the application of the simplification process, even at training time, in conjunction with pruning techniques, thus reducing the required time for pruning and fine-tuning neural networks. Since our proposed library removes the pruned neurons from the network, we will focus on models pruned using structured techniques. A high-level representation of the pruning and simplification pipeline is given in Figure 6.1.

Various hardware and software accelerators for sparse neural networks have been proposed [45–48]. The main downside of this kind of solution is the requirement

[1]the source code is available at https://github.com/EIDOSlab/simplify

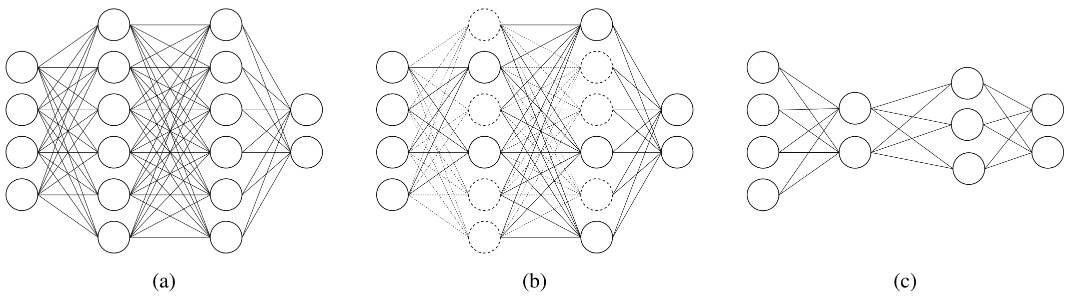


Fig. 6.1 Overview of the simplification procedure: *(a)* dense network *(b)* pruned network (dotted lines represent pruned neurons and connections) *(c)* simplified network in which the pruned neurons are removed from the architecture.

for specific hardware or software that can hardly be applied to standard consumer devices. Also, they are designed to apply inference-time acceleration using the zero-filled model instead of building an optimized structure, thus precluding the ability to train a pruned neural network. Simplify solves these issues by extracting the remaining structure from a pruned model and removing all the zeroed-out neurons from the network. This allows for obtaining a model that can be saved, shared, and used without special hardware or software. While, at first glance, this may seem a straightforward procedure, the removal of zeroed neurons poses some hidden challenges, like the presence of bias in said neurons or some constraints in the output's dimensions due to skip or residual connections. Even though the interest of the deep learning community on the matter seems to be quite strong [49–52], very few approaches and libraries for simplifying pruned models have been proposed. Moreover, they are usually limited to simpler architectures such as VGG [1], and their usage is restricted to deploying an already pruned model. On the other hand, with Simplify, we provide a way to:

1. Optimize more complex network architectures (e.g., ResNet [3], DenseNet [53] and so on), and, in general, custom architectures, without constraints given by the connectivity patterns (i.e., residual connections);
2. Optimize models during training: this allows for speedups in the time required for training a model and reduces the memory occupation when applied together with an iterative pruning technique.

6.1 Challenges of neural network simplification

In this section, we will highlight the main challenges we faced while developing Simplify; mainly, we will cover: Batch Normalization fusion, bias propagation, and channel removal.

6.1.1 Batch Normalization fusion

A vast amount of modern neural networks use Batch Normalization (from here on out, BatchNorm) as a way to improve generalization. Given an input x, we can define the output of BatchNorm as:

$$y = \gamma \frac{x-\mu}{\sqrt{\sigma^2+\varepsilon}} + \beta \tag{6.1}$$

where γ and β represent the weights and bias of the layer and are learned using standard backpropagation procedures; μ and σ^2 represent the mean and variance computed over the batch. During training, this layer keeps running estimates of its computed mean and variance, which are then used for normalization during evaluation. Let us denote this approximations as $\widehat{\mu}$ and $\widehat{\sigma}^2$. Notice that each parameter is defined for each channel of the input feature map; we will denote them as γ_c, β_c, $\widehat{\mu}_c$ and $\widehat{\sigma}_c^2$ for a given channel c. Once a neural network is trained to completion, all the parameters of its layer can be considered frozen, i.e., no longer updated from further training. Also, in standard network architectures, it is possible to identify pairs of Convolutional and BatchNorm layers whose output is the same size. In such conditions is possible to reduce the network complexity by fusing these two layers into a single one. Note that this operation is only applicable if there is no non-linearity between the two layers.

Let us consider a generic BatchNorm's output

$$y = \gamma_c \frac{x-\widehat{\mu}_c}{\sqrt{\widehat{\sigma}_c{}^2+\varepsilon}} + \beta_c \tag{6.2}$$

this can be rewritten as

$$y = \frac{\gamma_c}{\sqrt{\widehat{\sigma_c}^2 + \varepsilon}} x - \frac{\gamma_c}{\sqrt{\widehat{\sigma}^2 + \varepsilon}} \widehat{\mu_c} + \beta_c \tag{6.3}$$

since this BatchNorm layer is preceded by a Convolutional layer, x_c can be defined as

$$x = W_c \cdot z + b_c \tag{6.4}$$

where z is the input of the Convolutional layer, W_c are its weights and b_c its bias.

We can now express the BatchNorm output as a function of the Convolutional layer, substituting equation 6.4 in equation 6.3.

$$y = \frac{\gamma_c}{\sqrt{\widehat{\sigma_c}^2 + \varepsilon}} (W_c \cdot z + b_c) - \frac{\gamma_c}{\sqrt{\widehat{\sigma_c}^2 + \varepsilon}} \widehat{\mu_c} + \beta_c \tag{6.5}$$

Leveraging on equation 6.5, we can finally fuse the Convolutional and the BatchNorm layer in a single Convolutional layer whose weights and bias are defined as

$$W_{fuse} = \gamma_c \frac{W_c}{\sqrt{\widehat{\sigma_c}^2 + \varepsilon}} \tag{6.6}$$

$$b_{fuse} = \gamma_c \frac{b_c - \widehat{\mu_c}}{\sqrt{\widehat{\sigma_c}^2 + \varepsilon}} + \beta_c \tag{6.7}$$

and the output y is therefore

$$y = W_{fuse} \cdot z + b_{fuse} \tag{6.8}$$

6.1.2 Bias propagation

This step is necessary if biases are present in the model's hidden layers or are introduced by the fusion of batch normalization layers. Neurons with zeroed-out channels might have non-zero bias so that they will fire a constant output value. Hence, a neuron cannot immediately be removed if the corresponding bias is non-zero. These values, however, can be propagated and accumulated into the biases of the next layer. This operation can be repeated until all biases have been propagated to the final layer of the network. After a bias has been propagated, it can then be set to zero in the original neuron, allowing the removal of the whole weight channel.

Linear layers

We denote as $\ell_1 = \langle A, a \rangle$ and $\ell_2 = \langle B, b \rangle$ two sequential linear layers. A and a denote the weight matrix and bias vector of ℓ_1, of size $N \times M$ and N respectively. B and b denote the weight matrix and bias vector of ℓ_2 of size $T \times N$ and T respectively. We also denote as f the activation function (e.g., ReLU). A forward pass for ℓ_1 consists in:

$$y = f(xA^T + a) \tag{6.9}$$

where x represents an input vector of size M and for ℓ_2:

$$z = yB^T + b. \tag{6.10}$$

Focusing on equation 6.9, we can visualize the vector-matrix product:

$$y = \begin{bmatrix} x_0 \\ x_1 \\ \dots \\ x_M \end{bmatrix}^T \cdot \begin{bmatrix} A_{0,0} & A_{1,0} & \dots & A_{N,0} \\ A_{0,1} & A_{1,1} & \dots & A_{N,1} \\ \vdots & \ddots & \vdots & \\ A_{0,M} & A_{1,M} & \dots & A_{N,M} \end{bmatrix} + \begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_N \end{bmatrix}^T = \begin{bmatrix} xA_0^T + a_0 \\ xA_1^T + a_1 \\ \vdots \\ xA_N^T + a_N \end{bmatrix}^T$$

We suppose that some output channel of A has been zeroed-out after applying some pruning criterion, e.g., every element of the column A_1 is zero. The multiplication becomes:

$$y = \begin{bmatrix} x_0 \\ x_1 \\ \dots \\ x_M \end{bmatrix}^T \cdot \begin{bmatrix} A_{0,0} & 0 & \dots & A_{N,0} \\ A_{0,1} & 0 & \dots & A_{N,1} \\ \vdots & \vdots & \ddots & \vdots \\ A_{0,M} & 0 & \dots & A_{N,M} \end{bmatrix} + \begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_N \end{bmatrix}^T = \begin{bmatrix} xA_0^T + a_0 \\ a_1 \\ \vdots \\ xA_N^T + a_N \end{bmatrix}^T$$

We now focus on the forward pass of ℓ_2. As an example, we analyze what happens with the first neuron B_0. If we rewrite equation 6.10 focusing on B_0 we obtain:

$$z_0 = f(xA_0^T + a_0)B_{0,0} + \boldsymbol{f(a_1)B}_{0,1} + \dots + f(xA_N^T + a_N)B_{0,N} + b_0. \tag{6.11}$$

The term $f(a_1)B_{0,1}$ is a constant which can be accumulated into b_0. The same reasoning can be extended to all neurons in ℓ_2 by adding $f(a_1)$ multiplied by the respective incoming weight to the neuron bias. The new set of biases $\widehat{b}$ for the layer can be written as:

$$\widehat{b} = \begin{bmatrix} b_0 + f(a_1)B_{0,1} \\ b_1 + f(a_1)B_{1,1} \\ \vdots \\ b_T + f(a_1)B_{T,1} \end{bmatrix}^T$$

and the original bias a_1 can be set to zero in ℓ_1, resulting in $\widehat{a} = [a_0, 0, a_1, \dots, a_N]$. This procedure can be applied when multiple neurons are pruned in ℓ_1, and the general rule to obtain the updated biases $\widehat{b}$ is as follows:

$$\widehat{b} = \begin{bmatrix} b_0 + \sum_i f(a_i)B_{0,i} \\ b_1 + \sum_i f(a_i)B_{1,i} \\ \vdots \\ b_T + \sum_i f(a_i)B_{T,i} \end{bmatrix}^T$$

where i represents the indices of zeroed channels in ℓ_1. After the bias propagation procedure, the layers ℓ_1 and ℓ_2 can be replaced by $\widehat{\ell_1} = \langle A, \widehat{a} \rangle$ and $\widehat{\ell_2} = \langle B, \widehat{b} \rangle$ respectively.

Convolutional layers

Similar reasoning can be applied to Convolutional layers. However, the propagation process must consider whether the convolution employs zero padding on the input tensor.

For the sake of simplicity, using the same notation of Section 6.1.2, let us consider two sequential convolutional layers $\ell_1 = \langle A, a \rangle$ and $\ell_2 = \langle B, b \rangle$. We also assume that ℓ_1 has one input channel and two output channels (A has shape $2 \times 1 \times H_1 \times W_1$ and a is a vector of length 2), while ℓ_2 has two input channels and one output channels (B has shape $1 \times 2 \times H_2 \times W_2$, and b is a vector of length 1). The forward pass for ℓ_1 is:

$$y = \left(\begin{bmatrix} x * A_0 \\ x * A_1 \end{bmatrix} + \begin{bmatrix} a_0 \\ a_1 \end{bmatrix} \right)^T = \left(\begin{bmatrix} F^0 \\ F^1 \end{bmatrix} + \begin{bmatrix} a_0 \\ a_1 \end{bmatrix} \right)^T = \begin{bmatrix} F^0 + a_0 \\ F^1 + a_1 \end{bmatrix}^T$$

where $*$ represents the convolution operation and x is a properly sized input. In this context, the addition operation $+$ between the resulting feature map $F^i = x * A_i$ and the corresponding bias value a_i will perform a shape expansion of a_i to match the feature map shape, for example:

$$F^i + a_i = \begin{bmatrix} F^i_{0,0} & \dots & F^i_{0,H_{out}} \\ \vdots & \ddots & \vdots \\ F^i_{W_{out},0} & \dots & F^i_{W_{out},H_{out}} \end{bmatrix} + \begin{bmatrix} a_i & \dots & a_i \\ \vdots & \ddots & \vdots \\ a_i & \dots & a_i \end{bmatrix}$$

We assume that the second channel A_1 of ℓ_1 has been zeroed out after the application of some pruning criterion, hence if we consider $F^1 + a_1$, we obtain:

$$F^1 + a_1 = \begin{bmatrix} 0 & \dots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \dots & 0 \end{bmatrix} + \begin{bmatrix} a_1 & \dots & a_1 \\ \vdots & \ddots & \vdots \\ a_1 & \dots & a_1 \end{bmatrix} = \begin{bmatrix} a_1 & \dots & a_1 \\ \vdots & \ddots & \vdots \\ a_1 & \dots & a_1 \end{bmatrix}$$

Thus, y becomes:

$$\begin{bmatrix} F^0 + a_0 \\ \overline{a_1} \end{bmatrix}^T$$

where $\bar{\cdot}$ denotes that the element shape has been expanded. We can now analyze what happens with ℓ_2. For the sake of simplicity, we assume that $W_{out} = H_{out} = 3$, that $W_2 = H_2 = 2$, and that every value of B is equal to 1. We also consider a stride value of 1 for ℓ_2.

Convolution without padding (or "same" padding): This is the more straightforward case, and it is similar to the linear layers (Section 6.1.2). The forward pass of ℓ_2 can be expressed as follows:

$$z = f(F^0 + a_0) * B_{0,0} + \boldsymbol{f(\overline{a_1}) * B}_{0,1} + b_0. \tag{6.12}$$

The factor $f(\overline{a_1}) * B_{0,1}$ is constant and can be accumulated into b_0. Visualizing it, we obtain:

$$\widehat{b_0} = \begin{bmatrix} f(a_1) & f(a_1) & f(a_1) \\ f(a_1) & f(a_1) & f(a_1) \\ f(a_1) & f(a_1) & f(a_1) \end{bmatrix} * \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix} + b_0 = \begin{bmatrix} 4f(a_1) & 4f(a_1) \\ 4f(a_1) & 4f(a_1) \end{bmatrix} + b_0. \tag{6.13}$$

In this case, the updated bias can be converted as a scalar replacing the original value b_0. Given that the resulting matrix is constant, we can directly factor out $4f(a_1)$ and set a_1 to 0 in ℓ_1, obtaining a new bias $\widehat{b_0} = 4f(a_1) + b_0$ which will be used from now on in ℓ_2[2]. The same reasoning can be extended to the case of multiple neurons in the convolution layer and multiple pruned channels in the preceding layer: each bias value will be updated according to the rule in equation 6.13. The general rule to obtain the new bias vector $\widehat{b}$ can be expressed as follows:

$$\widehat{b} = \begin{bmatrix} b_0 + \sum_i f(\overline{a_i}) * B_{0,i} \\ b_1 + \sum_i f(\overline{a_i}) * B_{1,i} \\ \vdots \\ b_{C_{out}} + \sum_i f(\overline{a_i}) * B_{C_{out},i} \end{bmatrix} \tag{6.14}$$

where i represents the indices of the zeroed output channels in ℓ_1.

[2]Notice that this can be applied for every choice of values for B. Of course, the resulting bias factor will change accordingly

Convolution with zero padding If the Convolution applies zero padding to the input values, then the bias cannot be accumulated into a scalar, as the resulting matrix will not be constant. To show this, we rewrite equation 6.13, applying a zero padding of size one along each dimension of the input tensor:

$$\widehat{b_0} = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & a' & a' & a' & 0 \\ 0 & a' & a' & a' & 0 \\ 0 & a' & a' & a' & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix} * \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix} + b_0 = \begin{bmatrix} a' & 2a' & 2a' & a' \\ 2a' & 4a' & 4a' & 2a' \\ 2a' & 4a' & 4a' & 2a' \\ a' & 2a' & 2a' & a' \end{bmatrix} + b_0 \qquad (6.15)$$

where $a' = f(a_1)$ for brevity. In this case, the new bias value needs to be maintained in a matrix form, i.e.:

$$\widehat{b_0} = \begin{bmatrix} a'+b_0 & 2a'+b_0 & 2a'+b_0 & a'+b_0 \\ 2a'+b_0 & 4a'+b_0 & 4a'+b_0 & 2a'+b_0 \\ 2a'+b_0 & 4a'+b_0 & 4a'+b_0 & 2a'+b_0 \\ a'+b_0 & 2a'+b_0 & 2a'+b_0 & a'+b_0 \end{bmatrix}$$

To obtain the updated biases in case of multiple neurons and multiple channels, the same rule of equation 6.14 can be applied, keeping in mind that in this case, it will result in a tensor of shape $C_{out} \times H_{out} \times W_{out}$ instead of a vector. This introduces a constraint on the feature map size; hence the model can only be used at a fixed input size. However, given that the whole simplification procedure is executed on an already trained model before deploying to production, it should not represent a significant issue.

Residual connections

While the above process works fine for simple feed-forward models, special care must be taken to handle residual connections. As an example, let us consider the case of two linear layers $\ell_1 = \langle A, a \rangle$, and $\ell_2 = \langle C, c \rangle$, whose outputs y and t are summed together in a residual connection, followed by another layer $\ell_3 = \langle B, b \rangle$:

$$y+t=\left(\begin{bmatrix} xA_0^T & + & a_0 \\ 0 & + & a_1 \\ \vdots & & \\ 0 & + & a_{N-1} \\ xA_N^T & + & a_N \end{bmatrix} + \begin{bmatrix} 0 & + & c_0 \\ \hat{x}C_1^T & + & c_1 \\ \vdots & & \\ 0 & + & c_{N-1} \\ \hat{x}C_N^T & + & c_N \end{bmatrix}\right)^T \tag{6.16}$$

where 0 denotes that a channel was pruned. The residual (sum) operation introduces a new constraint: only biases corresponding to matching pruned channels in ℓ_1 and ℓ_2 can be propagated to the next layer. To see why, we can rewrite equation 6.16 as equation 6.11 and obtain:

$$\begin{aligned} z_0 = & f(xA_0^T + a_0 + c_0)B_{0,0} + f(a_1 + \hat{x}C_1^T + c_1)B_{0,1} + \cdots + \\ & + \boldsymbol{f(a_{N-1} + c_{N-1})B_{0,N-1}} + \\ & + f(xA_N^T + a_N + \hat{x}C_N^T + c_N)B_{0,N} + b_0 \end{aligned} \tag{6.17}$$

It is clear that even if multiple channels are pruned from ℓ_1 and ℓ_2, only the factor $f(a_{N-1} + c_{N-1})B_{0,N-1}$ becomes a constant. In this case, we opt not to propagate any bias and employ an expansion scheme (Section 6.1.3) to speed up the convolution operations.

6.1.3 Channels removal

Once the biases have been propagated and removed from the hidden layers, the weight matrices corresponding to zeroed channels can be removed. The process, which we call simplification, is quite simple. For each layer L, we denote with W^L the corresponding weight tensor, with shape $N \times I \times W \times H$ for convolutional layers and $N \times I$ for linear layers. The simplification consists of two steps:

1. Remove all the input channels corresponding to zeroed channels in the previous layer (none if it is the input layer):

$$\widehat{W}^L = \left[W_{0,i}^{\ell}, W_{1,i}^{\ell}, \dots, W_{N,i}^{\ell}\right] \tag{6.18}$$

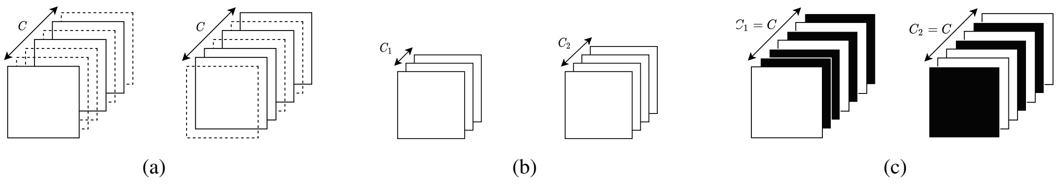


Fig. 6.2 Pruned weight matrices: a dotted line indicates a zeroed channel *(a)* simplified weight matrices *(b)* expanded weight matrices: black slices mean that the channel is a zero matrix *(c)*.

where i is the indices of the remaining output channels in W^{L-1}. The resulting weight tensor will be of shape $N \times I_s \times W \times H$ for convolutional layers and $N \times I_s$ for linear layers, where $I_s \leq I$ is the number of remaining output channels in the previous layer $L-1$.

2. Remove all the output channels corresponding to zeroed neurons:

$$\widetilde{W}^L = \left[\widehat{W}^{\ell}_j\right]_{\forall j \in J} \tag{6.19}$$

where J is the set of indices corresponding to the remaining output channels, the resulting tensor will be of shape $N_s \times I_s \times W \times H$ for convolutional layers and $N_s \times I_s$ for linear layers, where $N_s \leq N$ is the number of remaining (non-zero) output channels of L.

Residual connections

Residual or skip connections introduce a constraint on the output size of a layer. A residual connection consists of the sum of the output of two layers ℓ_1 and ℓ_2. As an example, we assume ℓ_1 and ℓ_2 to be convolutional layers, with their respective output Z_1 and Z_2 being of size $C_1 \times H \times W$ and $C_2 \times H \times W$ (ignoring the batch size). To compute $Z_1 + Z_2$, C_1 must be equal to C_2. However, after the simplification step, the output sizes may differ, depending on whether some output channels in ℓ_1 and/or ℓ_2 were removed. Many works that propose a solution to channel removal face issues when applied to residual connections. For example, He *et al.* [54], or Kruglov [55] use some approximation and need to resort to finetuning to recover the lost performance. Others, like Hu *et al.* [56], ignore residual networks without proposing any solution.

We perform an expansion operation on the output tensors to address this issue. Assuming the original size (before the simplification) was C, then Z_1 and Z_2 are expanded to the original number of channels before performing the addition. The process is illustrated in Figure 6.2. After the expansion step, we sum the layer's biases (which were not propagated as explained in Section 6.1.2). While it is true that the expansion operations introduce a computational overhead in the model inference, the speedups achieved by the simplified convolutions compensate for it when using the model for inference. However, given that the time required by the indexing operations employed in the expansion scheme depends on the given batch size, we opt not to adopt this scheme when using *Simplify* in training mode. In this case, we do not remove any output channel in the weight tensor.

6.2 Software Description

The *Simplify* library leverages on the main PyTorch packages and is composed of three main modules that, even if designed to function in a predefined order, can be used independently based on the user requirements. We now provide a brief overview of each module functionalities and purpose. A more detailed explanation of the maths involved in each module is provided in the Appendix.

Fuse First, we have the *fuse* module. Here we perform a non-mandatory optimization of the model by merging, in a single Convolutional layer, pairs of consecutive Convolutional and Batch Normalization layers. This process is known as Batch Normalization fusion or folding. This step can be ignored if the presence of Batch Normalization layers in the network is required, i.e. for further training of the simplified model. This step is not needed to define the simplified model, but provides inference-time and memory usage advantages, especially when deploying a trained model to production, thanks to an optimization of the model architecture.

Propagate The second module is called *propagate*. With this module we solve the problem of non-zero bias in zeroed neurons mentioned in Sec. **??**. It is possible that some pruned neuron retain non-zero bias; in such situation it would be impossible to remove the neuron without losing the bias contribution. To solve this problem, in the *propagate* module, we essentially treat such neurons as a constant signal that can then be absorbed by the next layer, making the zeroed neuron removable.

Table 6.1 Inference time for different dense, pruned, and simplified torchvision models.

Architecture	Inference Time (ms) Dense	Pruned	Simplified
AlexNet [57]	7.58 ± 0.29	7.55 ± 0.28	2.95 ± 0.02
DenseNet-121 [53]	36.41 ± 4.88	34.31 ± 3.85	21.87 ± 1.45
GoogLeNet [58]	15.44 ± 3.19	13.68 ± 0.09	10.31 ± 0.82
InceptionV3 [59]	25.29 ± 7.31	21.68 ± 2.90	13.22 ± 2.23
MNASNet [60]	17.66 ± 0.57	13.64 ± 0.13	11.59 ± 0.07
MobileNetV3 [4]	13.74 ± 0.67	12.18 ± 0.46	11.95 ± 0.21
ResNet-50 [3]	24.39 ± 4.48	26.19 ± 5.84	18.21 ± 1.98
ResNeXt-101 [61]	76.11 ± 15.79	77.35 ± 20.04	65.68 ± 16.41
ShuffleNetV2 [62]	18.07 ± 2.23	14.32 ± 0.21	13.06 ± 0.08
SqueezeNet [63]	4.50 ± 0.06	4.39 ± 0.05	4.09 ± 0.50
VGG-19 [1]	40.41 ± 12.13	38.56 ± 10.72	12.39 ± 0.19
WideResNet-101 [64]	79.40 ± 25.57	82.86 ± 22.47	60.16 ± 10.77

Remove Lastly, with the *remove* module we perform the actual simplification of the model, removing the zeroed-out neurons. Here we make sure that the output and input dimensions of adjacent layers correspond, while also taking into account architecture constraints such as the presence of skip connections.

6.3 Illustrative Examples

This section provides an overview of the usage for *Simplify*. We also illustrate the results obtained for the two use cases discussed, namely optimization for model deployment and optimization during training. Please note that our experiments were performed on different image classification networks, made available by the Torchvision library[3]. Such models are defined for the ImageNet [65] dataset and therefore expect an input of size $B \times 3 \times 224 \times 224$, where B represents the batch size.

6.3.1 Optimization for deployment

This is the most common use case. Here, the simplification procedure is applied to an already trained model on which a pruning criterion has been previously applied. In most cases, a one-line call to the `simplify` method is sufficient: the library performs

[3]https://pytorch.org/vision/stable/models.html#classification

all three steps autonomously and takes care of different architectural patterns, such as residual connections. This is the most common use case. Here, the simplification procedure is applied to an already trained model on which a pruning criterion has been previously applied. Below, we provide a sample code snippet.

```
# Load a pruned model checkpoint
model = torch.load(..)

# Apply simplification.
model.eval()
simplify(model, torch.zeros(1, 3, 224, 224))
```

As shown in the code snippet, the `simplify` function takes as argument the model to be simplified and a tensor representing an input image, filled with zeros, with a batch size of 1 (in this case, we're working with models build for ImageNet, therefore, is of size $1 \times 3 \times 224 \times 224$). It is important to note that the model has to be set to evaluation mode before the simplification procedure so that the automatic update of parameters (such as for Batch Normalization) is disabled. The simplification of the model happens in place.

Table 6.1 shows the inference times (in milliseconds) of different Torchvision models. In this table, we compare the inference speed of the dense models (i.e., models that have not been pruned), the resulting pruned architectures (random, structured pruning with 50% probability), and the simplified model obtained with our proposed library. The benchmarks are run on an Intel(R) Core(TM) i9-9900K CPU, with a batch size of 1, to simulate a one-shot inference of a deployed model. The results are averaged across 1000 different runs for each architecture. It's easy to see that, thanks to *Simplify*, the resulting model is faster and can leverage the applied pruning while remaining a fully-fledged PyTorch network. We can see that, for most network families, *Simplify* allows for a decrease in the actual inference time. It is essential to point out that for some architectures, like MobileNet or SqueezeNet, the library may not lead to significant speed-up as they are already very optimized.

6.3.2 Optimization for training

Most modern network architectures employ Batch Normalization as a way to improve generalization. To avoid losing the Batch Normalization contribution, we provide the ability to prevent the fusion step so that these layers are retained. To further enhance training time, enabling a *training* mode for `simplify` is possible, which helps decrease inference time. Below, we provide a sample code snippet.

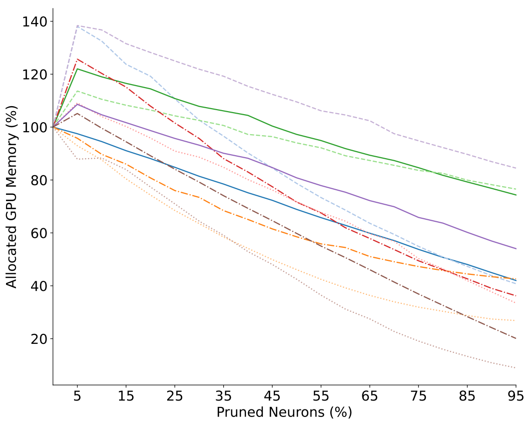
```
for step in range(epochs):
    model.train()
    model = ... # train model and apply pruning

    # Apply simplification
    model.eval()
    simplify(model,
             torch.zeros(1, 3, 224, 224),
             fuse_bn=False,
             training=True)
```

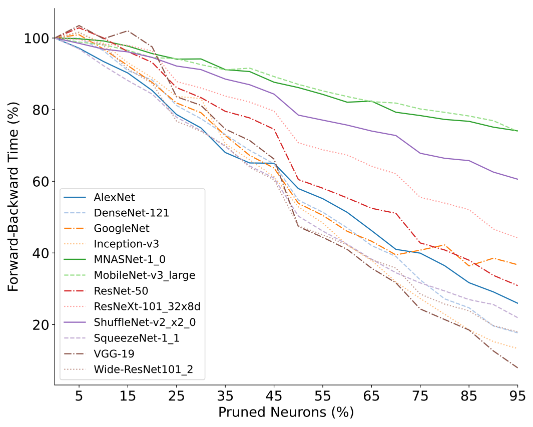
Figure 6.3a shows the reduction in allocated GPU memory for different pruning ratios. While overhead is introduced at low pruning percentages, a reduction in the required memory for performing an optimization step is achieved when pruning a sufficient amount of neurons. In Figure 6.3b, we show the decrease in time required by a network training loop's forward and backward pass. The benchmarks are run on an NVIDIA RTX 2080Ti GPU with an Intel(R) Core(TM) i9-9960X CPU, using a batch size of 64.

6.4 Summary

We propose the PyTorch-compatible library *Simplify*, with the aim of providing a simple-to-use set of procedures to remove zeroed neurons from a neural network architecture. The proposed library solves different issues in the creation of simplified models, such as the propagation of the bias of pruned neurons and the shape constraint of skip connections. The library is composed of three modules that, while designed



(a)



(b)

Fig. 6.3 Simplification during training: (a) Allocated GPU memory for a forward/backward pass of different pruned models (b) Total time for a forward and a backward pass of different pruned models.

to work together, can be used independently from one another according to the required functionality for a specific setting.

In the next chapter, we will use Simplify to provide empirical evidence on the advantages of structurally-pruned models when deployed on resource-constrained devices, without the need for ad hoc hardware or software support.

Chapter 7

On the Role of Structured Pruning for Neural Network Compression

In Chapters 3, 5, and 6, we mentioned that structured pruning procedures could provide better practical benefits when deploying a pruned model on low-power devices without the need for specific software or hardware; this, however, is true if the model is correctly processed (i.e., the zeroed neurons are not involved in the model inference). In Chapter 6, we proposed a library that enables us to remove the pruned neurons from the architecture, reducing the resources required to perform inferences. In this chapter, we will provide tangible results to these speculations. We will compare the performance of models pruned with LOBSTER (Chapter 4) and SeReNe (Chapter 5), deployed on a variety of devices. To this end, we will consider two use cases:

– First, we will investigate the benefits of structured pruning approaches within the MPEG-7 Part 17 neural network compression pipeline. Evaluating how structured pruning can benefit quantization, entropy coding, and inference time.

– Second, we will present HLSinf, an open-source framework for developing custom neural network accelerators for FPGAs that provides efficient support to quantized and pruned neural network models.

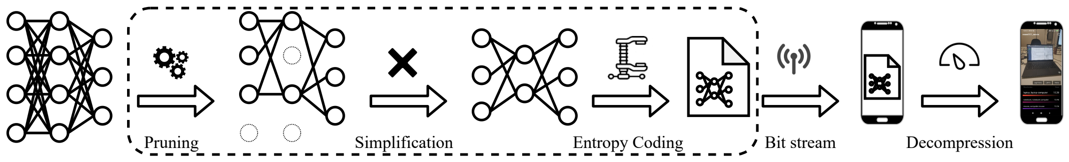


Fig. 7.1 Our experimental setup; the parts within the dashed box are the object of the MPEG-7 part 17 standard.

7.1 Purning in the MPEG-7 Part 17 pipeline

The need for compressing deep neural networks prompted the Moving Pictures Experts Group (MPEG) of the International Organization for Standardization (ISO) to define the upcoming MPEG-7 Part 17 standard *Compression of neural networks for multimedia content description and analysis* [66]. The MPEG neural network compression pipeline includes three stages. First, the number of parameters in the neural network is preliminarily reduced, e.g., pruning away disposable parameters and yielding a sparse network topology. Second, the parameters that survived pruning are quantized over a finite set of values. Third, the quantized parameters are entropy coded with context adaptive arithmetic coding [67], producing a compressed bitstream. Experiments show that a favorable trade-off between compression efficiency and performance could be stricken [68]. While quantization and entropy coding has received more attention, the role of parameter pruning has not been fully explored and the effect of the pruning scheme and the resulting tensor structure on the efficiency of a complete neural network compression pipeline is not understood.

This section will investigate the benefits of structured pruning approaches within the MPEG neural network compression pipeline. We experimentally show that, while the structured approach achieves a lower pruning ratio, it yields better end-to-end compression efficiency. As a bonus, the network topology is also easier to represent in memory once the network is decompressed. Also, inference time is lower as entire operations among tensors are avoided, as we show experimenting on Android devices. We hypothesize that the structured topology of the pruned neural network is the key to the higher efficiency of the entropy coder.

Table 7.1 Experimental results for different network architectures and pruning strategies. Left: percentage of pruned parameters, size of the simplified network topology, and size of the compressed bitstream. Right: inference time on different embedded devices: Raspberry Pi 3B (RPi 3B), Huawei P20 (P20), Xiaomi MI 9 (MI9), and Samsung Galaxy S6 lite (S6L).

Dataset	Architecture	Pruning	Pruning ratio [%]	Simplified topology [MB]	Compressed bitstream [MB]	Inference time [ms] RPi 3B	P20	MI9	S6L
CIFAR-10	VGG-16	No pruning	-	60.0	3.6	647	204	153	251
		LOBSTER	**92.44**	58.61	1.61	610	191	146	242
		SeReNe	47.16	**31.02**	**0.34**	**594**	**99**	**85**	**106**
	ResNet-32	No pruning	-	2.0	0.30	580	32	30	31
		LOBSTER	**81.19**	1.96	0.12	545	32	26	30
		SeReNe	52.80	**1.0**	**0.09**	**536**	**25**	**17**	**25**
CIFAR-100	AlexNet	No pruning	-	94.6	10.1	246	131	84	168
		LOBSTER	**98.90**	48.84	0.40	224	95	67	120
		SeReNe	59.87	**37.07**	**0.20**	**186**	**75**	**53**	**96**
ImageNet	ResNet-101	No pruning	-	178.4	26.24	11919	958	416	1008
		LOBSTER	**87.39**	173.87	9.24	11879	956	403	985
		SeReNe	1.09	**172.53**	**7.51**	**11699**	**929**	**371**	**974**

7.1.1 Proposed compression pipeline

In this section, we describe a neural network compression pipeline as specified in [66] and as illustrated in Figure 7.1 (also including a non-normative evaluation part).

Parameter Pruning

During pruning, some network parameters are removed from the connections graph. We will consider two different pruning strategies: as an unstructured pruning strategy, we use LOBSTER as presented in Chapter 4; as a structured approach, we will use SeReNe, presented in Chapter 5.

Topology Simplification

After pruning, the network undergoes simplification: in this stage, arcs corresponding to neurons without incoming and/or outgoing connections are removed from the topology. Simplification is primarily effective when the network has been pruned with SeReNe. We perform the simplification step using Simplify (Chapter 6).

Quantization and Coding

Finally, the remaining parameters undergo quantization and entropy coding, yielding a compressed bitstream. We rely on scalar quantization of parameters and DeepCABAC [67, 68] for entropy coding, in accordance with the MPEG-7 part 17 standard. DeepCABAC represents an evolution of the context-adaptive binary arithmetic coding used in videos, including model quantization, binarization, and arithmetic coding.[1]

7.1.2 Experimental results

Table 7.1 shows the results of our experiments with the popular VGG-16, ResNet, and AlexNet architectures for image classification over the CIFAR-10, CIFAR-100, and ImageNet datasets. All the architectures are compressed according to the scheme described in the previous section, i.e., we alternatively prune the networks with LOBSTER and SeReNe. The size of the simplified networks refers to the case where the parameters are represented over 32-bit floating-point values, compliant with the IEEE 754 standard. All results are obtained with a fixed quantization step equal to 2^{-15}. As expected, LOBSTER yields the highest compression ratio, i.e., removes more parameters from the network, whereas SeReNe results in more compact simplified topologies. Of course, the more general the task, the least the parameters/neurons to be removed without performance loss (for example, on ImageNet, the parameters removed in proportion are less than for other architectures). Figure 7.2 shows the parameter pruning map for the first convolutional layer of VGG-16 trained over CIFAR-10 (64 convolutional neurons, three filters sized 3×3 per neuron) for both LOBSTER (Figure 7.2a) and SeReNe (Figure 7.2b). Here, black lines represent neurons for which SeReNe pruned all parameters and thus are not represented altogether in the simplified topology, yielding reduced-size simplified networks. As a result, structured pruning always produces the best end-to-end compression efficiency in terms of compressed bitstream size. In particular, for VGG-16 trained on CIFAR-10, the SeReNE-pruned network bitstream is about five times smaller than the LOBSTER reference.

To guarantee optimal encoding into the final bitstream, one needs not only to reduce the number of parameters in the network but also that their entropy is low.

[1]https://github.com/fraunhoferhhi/DeepCABAC

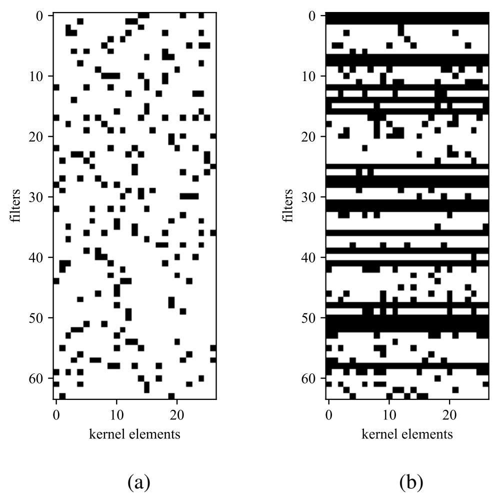


Fig. 7.2 Representation of the first convolutional layer for VGG-16 trained on CIFAR-10: unstructured pruning LOBSTER (a) and structured pruning SeReNe (b). Black pixels are pruned parameters.

Our experiments highlight that SeReNe is very effective also in terms of parameters entropy or compressibility. To spot this effect, let us observe the compression ratio C_p achieved by DeepCABAC after simplification, measured as the ratio between the final compressed bitstream size and the simplified floating-point network. In the VGG-16 case, we get $C_p = 0.0275$ for unstructured and $C_p = 0.011$ for structured pruning: the parameters retained with SeReNe can be compressed about three times more. To summarize, unstructured pruning effectively sets the largest possible amount of parameters to zeros, but the corresponding network tensors are sparse, and the remaining parameters are costly to represent. On the other hand, structured pruning provides a counter-intuitive behavior since one gets more non-zeros parameters, but their structure can be exploited for better compression. Not only can the tensor be trivially simplified (e.g., by altogether dropping some dimensions), but the entropy of the symbols to be encoded with DeepCABAC turns out to be lower. This result is likely due to the excellent interplay between the neural sensitivity regularizer and the context modeling in DeepCABAC: indeed, SeRene amounts to imposing a constraint onto a set of weights in the context of a given neuron; DeepCABAC can exploit the same regularity to represent the same context jointly. On the contrary, unstructured regularization behaves independently on every weight, with lower chances of creating homogeneous contexts for the following entropy coder. Finally, Figure 7.3 shows the compression-accuracy tradeoff for VGG-16 over CIFAR-10 dataset for different DeepCABAC quantization steps $q \in \left[2^{-15}; 1\right]$ range. SeReNe's structured pruning

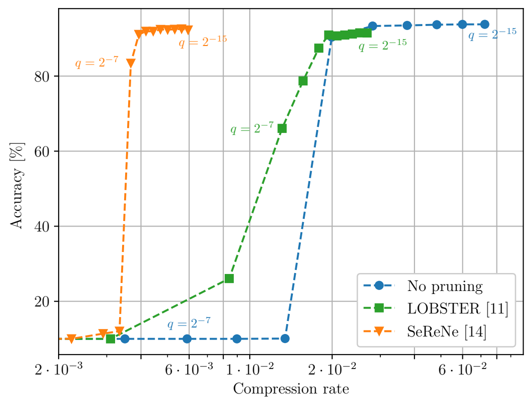


Fig. 7.3 Classification accuracy vs. compression rate for VGG-16 on CIFAR-10 for different pruning strategies. The compression rate is the ratio between the model's size after DeepCABAC and the original size. A lower compression rate means better compression efficiency.

yields comparable accuracy at a far lower encoded bitstream rate, outperforming by a large margin LOBSTER.

Inference Time

Table 7.1 also shows the inference time when the decompressed simplified architecture is deployed on embedded devices. The experiments have been worked on the following devices:

- Raspberry Pi 3B (RPi 3B): Quad Core 1.2GHz Broadcom BCM2837 64bit CPU, 1GB RAM;
- Huawei P20 phone (P20): 4x2.36 GHz Cortex-A73 + 4x1.84 GHz Cortex-A53 processors, 4GB RAM;
- Xiaomi MI 9 phone (MI9): 1x2.84 GHz Kyro 485 + 3x2.42 GHz Kyro 485 + 4x1.80 GHz Kyro 485, 6GB RAM;
- Samsung Galaxy S6 lite tablet (S6L): 4x2.3 GHz Cortex-A73 + 4x1.7 GHz Cortex-A53, 4GB RAM.

Structured sparsity always yields lower inference time as a side benefit of the more compact representation of the network into the device memory and the fewer matrix-vector multiplication required at inference time.

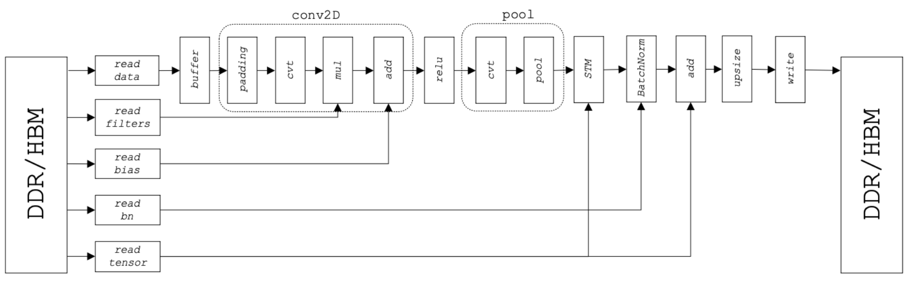


Fig. 7.4 HLSinf architecture. Modules are represented as boxes, and streams as arrows. Modules can be placed in any order.

7.2 Structured pruning for FPGAs

In the context of the "Deep-Learning and HPC to Boost Biomedical Applications for Health" (DeepHealth) project funded by European Union's Horizon 2020 program (grant agreement No 825111), we collaborated with the Universitat Politècnica de València to evaluate the effect of structured pruning approaches on FPGA devices. To perform this evaluation, we used HLSinf: an open-source platform to develop customized FPGA accelerators specifically tailored for pruned and quantized models. This platform, developed by the team of the Universitat Politècnica de València, enables optimal adaptation of the data precision format used in the quantization process. It takes into account the pruning process to flexibly adapt the needed resources on the FPGA to the final accelerator implementation. HLSinf can deploy final implementations with different performances, and various neural network layers support, thus, suiting additional neural network model requirements. Furthermore, we adapted the accelerators to the EDDL (European Distributed Deep Learning) library [69]. EDDL is an open-source platform developed during the DeepHealth project that enables the definition, training, and inference of neural networks on CPUs and GPUs. Thanks to our adaptation, EDDL can now natively support FPGAs with the HLSinf accelerators. Our evaluation demonstrates that quantized and pruned models can primarily benefit performance when combined with HLSinf. Specifically, results show that up to 90x speed up can be achieved on typical medical image-based applications using neural network models on FPGAs.

HLSinf [2] is developed in High-Level Synthesis (HLS). HLS reduces the programming hurdle of FPGAs and enhances productivity. It enables easier adoption and reduces the need for excellent technical skills to develop accelerator instances.

[2]Source code at https://github.com/PEAK-UPV/HLSinf

The main goal of HLSinf is to enable customized accelerators for varying types of operations and data formats, mainly focused on quantized models. Figure 7.4 shows the baseline architecture design, which follows the dataflow model where data flows through the different modules in a streamlined fashion. Arrows represent streams, and boxes represent modules. The modules can be added or removed at design time, thus adapting the accelerator to the chosen neural network model. Read modules take data from the external FPGA memory and feed data into the accelerator through data streams. Data is then processed, and output is produced and written back into memory. Internal buffer modules are used to reduce memory access. The module-based design allows a pipeline of additional operations performed in NNs: activation functions such as ReLU, pooling operations, or other tasks that may be needed. Each module has a stream-based interface.

The platform is designed around the channel-slicing concept. Indeed, the main operation performed by the accelerator is the 2D convolution operation. This operation takes as an input a set of input channels (feature maps from previous layers) and produces a set of output channels (output feature maps). The accelerator handles, in parallel, a group of input channels (CPI, channels per input) and produces a set of output channels (CPO, channels per output). Both CPI and CPO parameters can be instantiated at design time, enabling the implementation of accelerators of different sizes and performances. The accelerator is designed to process CPI input pixels and produce CPO output pixels per clock cycle. All the datapaths within the accelerator can be customized and adapted to the target model. When working with quantized models (which employ different data precision formats for activations, weights, and bias), the HLSinf platform can be parametrized to use either 32-bit floating-point or both fixed-point and integer data types with a specific number of bits.

The most compute-demanding operation in image-based neural networks is the 2D convolution. HLSinf convolution module can be customized in the type of convolution to perform. Currently, direct convolution, Winograd's algorithm [70], and DepthWise Separable convolutions [71] are supported. Figure 7.4 shows the case for the direct 2D convolution. The module is composed of different sub-modules connected through streams. At its input, the module receives bias, filters, and data. Bias values are grouped in consecutive blocks of CPO bias values. For the filters, the convolution operation receives filters of size $CPO \times CPI \times KH \times KW$, where KH and KW represent the filter height and width, respectively. Thus, CPI filters are provided for each output feature map. For the input data, the module receives input pixels from CPI channels (a group of channels) in an interleaved manner (one pixel from each channel). The input data is pre-processed with the *padding* and *cvt* modules. The first one pads the input pixels with horizontal and/or vertical padding (zero-value pixels) and forwards the padded image in cut-through mode; the *cvt* module is in charge of generating frames. A frame is defined as a piece of the input data of size $KH \times KW$ where the convolution has to be performed. This module forwards CPI frames of $KH \times KW$ size on every clock cycle.

Pre-processed data and filters arrive at the *mul* module to perform the convolution operation with CPO blocks of multipliers. Each block multiplies CPI input frames (each one of size $KH \times KW$) by a set of CPI filters (each of size $KH \times KW$). Each block has $KH \times KW$ multipliers and adders. Each block then reduces the output data obtaining one output pixel. On every clock cycle, the module produces and forwards CPO pixels. Figure 7.5 shows the convolution operation. The number of MAC (multiply and accumulate) units used is $CPI \times CPO \times KH \times KW$. Therefore, its computing capabilities depend mainly on CPI and CPO parameters.

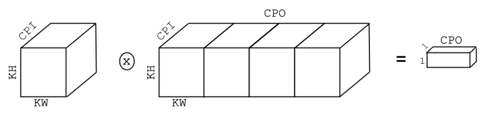


Fig. 7.5 2D convolution operation performed in the *mul* module on every clock cycle.

Finally, the *add* module is responsible for accumulating the output feature maps and adding the bias to each output channel. To do this, the module provides an output buffer of size $H \times W \times CPO$, where H and W are the height and width of the feature map, respectively. Once all input iterations have been performed, the module forwards the output buffer to the next layer and ultimately for storage in memory.

With this design and enough memory bandwidth available at the accelerator's input and output, we can determine the execution time of the accelerator when performing a 2D convolution. Assuming an input of $I \times H \times W$ and O feature maps, its execution time will be $\frac{I}{CPI} \times H \times W \times \frac{O}{CPO}$ clock cycles. The additional layers will add a constant delay in the form of a few cycles (the design is pipelined).

7.2.1 EDDL-HLSinf Support

HLSinf can be designed to run specific compute-intensive tasks on the FPGA. However, not all the layers of a neural network model are suitable or worth being run on such a system. This is the case for lightweight layers (e.g., softmax). Moreover, a fine-grained HW/SW co-design is needed in specific embedded systems. For these reasons, we need a method to perform the inference process in a combined way between the CPU and the FPGA and, simultaneously, adapt a Deep Learning library to use the HLSinf accelerator effectively. To this end, we have selected the EDDL (European Distributed Deep Learning) library [3]. EDDL defines a set of layers as C++ classes and allows the end-user to build a model by connecting layers with a net class. Each layer class provides methods for each possible tensor operation. To allow for an effective HW/SW co-design, we have implemented a new layer in EDDL called HLSinf, which encapsulates all the functionalities of the accelerator. The EDDL engine will run each model layer, and each one will use the target device, either CPU/GPU for regular layers or FPGA for the HLSinf layer. Data transfers between the FPGA and the CPU/GPU memories are performed transparently when needed.

Model Adaptation

EDDL allows training neural network models or loading them using ONNX [72]. To use our HLSinf accelerator, we have designed a new functionality in EDDL, which transforms an input model into a new one with added HLSinf layers where needed. Indeed, the new *toFPGA()* EDDL method enables three transformations of input models. First, layers in the original model are merged into a single one if the HLSinf accelerator can perform all those layers simultaneously. Second, data transformation is implemented when a layer that runs on the CPU/GPU feeds a layer running on the FPGA and vice-versa. For such purposes, a Transform layer is implemented. Figure 7.6 shows part of the VGG16 network adapted with the *toFPGA()* method. Finally, weights are adjusted, and tensors are reorganized as needed by the HLSinf accelerator. All tensors required by the FPGA device are stored in its memory.

[3]Source code at https://github.com/deephealthproject/eddl

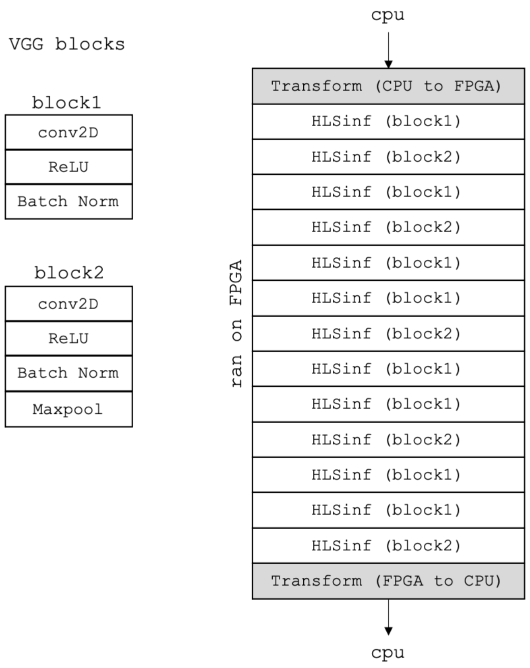


Fig. 7.6 VGG16 model mapped on FPGA. Blocks perfectly match the HLSinf accelerator. A transform layer is added for data movement between the CPU and FPGA.

7.2.2 Experiments

Hardware support

We run our experiments on an Intel i7-7800-X CPU at 3.45GHz and a Xilinx ALVEO U200 FPGA board. FPGA results use a single core to offload computations to the ALVEO board attached to this CPU. The CPU-only results use all the 12 threads available in the machine.

Target models

We considered two use cases to evaluate HLSinf, combining pruning and quantization. The ISIC dataset for skin lesion [73] is used for classification and segmentation for melanoma diagnosis. VGG16 [74] and SegNet [75] were used for classification and segmentation, respectively.

Pruning

We use SeReNe as a pruning procedure, and the resulting models are then processed with Simplify. During the pruning phase, we used a validation set to monitor the performance of the pruned network. For our test, we produced two models for the classification task and one for the segmentation problem. The two classification models are characterized by different accuracy and compression: for the first, called HA (high-accuracy), we target the preservation of the performance of the dense model at the cost of pruning ratio; the second, HCR (high compression), aims at maximizing the removed neurons, allowing for a lower, albeit still acceptable, performance.

Quantization

Alongside pruning, quantization is one of the most widely adopted compression methods. It reduces the necessary amount of memory to store the network's parameters and enables the deployment of a larger model in the same initial memory for better accuracy. Using 8-bit integer (INT8) instead of 32-bit floating-point (FP32) parameters enables performance gains in many fields: 4x reduction in model size, 2-4x reduction in memory bandwidth, and 2-4x faster inference due to faster computing with integer arithmetic. This is usually achieved at the cost of a slight decrease in accuracy. Quantization methods can be roughly divided into two categories [76]: Quantization Aware Training (QAT) and Post-Training Quantization (PTQ). PTQ quantizes both weights and activations for faster inference without re-training the model. QAT models quantization during training, which provides higher accuracy than PTQ schemes. The open-source N2D2 framework [77] was used as a quantization tool in this study. PTQ was chosen [78]: it takes a model trained using FP32 and directly quantizes it to INT8 without re-training or fine-tuning. As such, the overhead of PTQ is negligible. In N2D2, the post-training quantization algorithm is done in 3 steps:

- Weights normalization: All weights are rescaled in the range $[-1.0, 1.0]$.
- Activations normalization: Activations at each layer are rescaled in the range $[-1.0, 1.0]$ for signed outputs, and $[0.0, 1.0]$ for unsigned outputs.

– Quantization: Inputs, weights, biases, and activations are quantized to the desired precision N_{bits}. Conversion ranges from $[-1.0, 1.0]$ and $[0.0, 1.0]$ to $\left[-2^{N_{bits}-1} - 1, 2^{N_{bits}-1} - 1\right]$ and $\left[0, 2^{N_{bits}} - 1\right]$ taking into account all dependencies.

Results

Table 7.2 shows some details of the models used in our experiments. Table 7.2a summarises the pruned models in terms of performance (classification error for VGG and Dice score for SegNet) and percentage of remaining neurons. Table 7.2b shows the results of PTQ applied to the original VGG model for $N_{bits} = 8$. Using both pruning and quantization brings a 100x reduction in memory requirement with less than 2% performance reduction (HA network) and close to 1000x with a 15% performance drop (HCR).

Pruning		
Model	Performance	Remaining neurons
VGG16-HA	22.84 %	36.19 %
VGG16-HCR	35.66 %	11.64 %
SegNet-Pruned	0.83	59.76 %

(a) Pruned models

	FP32		PTQ	
Network	Memory	Error (%)	Memory	Error (%)
VGG16	524 583	21.06	131 146	21.06
VGG16-HA	18 692	22.84	4 673	23.76
VGG16-HCR	2 268	35.66	567	35.86

(b) Quantized models (memory in kB)

Table 7.2 Employed models. a) Pruned models, performance is expressed as classification error for VGG16 and as Dice score for SegNet. b) Quantized models, classification error, and memory footprint.

Table 7.3 shows the inference time of a single 224×224 input image on different models and devices for the skin lesion segmentation and classification problems. Focused on classification, the time needed when running the model on CPU and using the full model (VGG16 network) with 32-bit floating-point arithmetic is 1430.29 ms. Notice the performance improvement when using the FPGA instead. Execution time is reduced by a factor of 2.59 (inference in 552.28 ms). The accelerator is implemented with CPI and CPO factors of 4 and uses 32-bit floating-point precision arithmetic. If we either quantize the model or prune the model, we achieve further

Classification				
Model	Device	Inf. time (ms)	FPS	Speedup
VGG16	CPU	1430.29	0.70	-
VGG16	FPGA	552.28	1.81	2.59×
VGG16-Quantized	FPGA	229.93	4.35	6.22×
VGG16-HA	FPGA	99.33	10.07	14.40×
VGG16-HCR	FPGA	15.09	66.25	94.76×
Segmentation				
Model	Device	Inf. time (ms)	FPS	Speedup
SegNet	CPU	3165.97	0.32	-
SegNet	FPGA	757.22	1.32	4.18×
SegNet-Pruned	FPGA	282.23	3.54	11.22×

Table 7.3 Inference time, FPS, and speedup for ISIC classification and segmentation models for CPU and FPGA.

enhancements when targeting the FPGA device. In particular, the quantized model requires 8-bit integer weights and 32-bit integer activations and bias. The accelerator has been implemented with the specific integer data types, and the result can be seen in the table. The execution time is reduced by a factor of 6.22 (229.93 ms for inference). The improvement comes from a lower data precision format increasing the accelerator parallelism (CPI and CPO) to a factor of 8. The table also shows the benefit of running pruned models on the FPGA. Inference time is, in this case, significantly reduced. 32-bit floating-point precision and CPI and CPO parameters to 4 have been used. The pruned models fit perfectly on the target accelerator, and inference time is reduced by a total factor of 14.4 and 94.8 for both models, respectively (inference time lower than 100 ms in both cases). We can see the same performance trend for the skin lesion segmentation problem. As we move to FPGA, the inference time is reduced by a factor of 4.2. Also, when using the pruned model, the inference time is further decreased by a factor of 11.2.

7.3 Summary

In this chapter, we presented practical applications of pruned network models, which show empirically that pruned and simplified models can achieve significant gain concerning their resource requirements (e.g., lower inference time). We evaluated the role of unstructured and structured parameter pruning approaches in a standardized neural network compression pipeline. The surprising result of our experiments is that structured pruning enables better end-to-end compression despite lower pruning ratios. We explain this finding in part with the structure of the pruned network that is easier to simplify dropping entire elements of the tensors and in part with the lower entropy of the residual symbols processed by the DeepCABAC encoder. To ease the deployment of pruned and quantized models on embedded systems, we introduced HLSinf, an open-source platform for neural network accelerators in FPGAs. The platform has been integrated into the EDDL library, an open-source library for deploying neural network models on CPUs and GPUs. Quantization and pruning techniques have been incorporated in the design, thus exploiting the efficiency of the FPGA resources and optimizing performance compared to CPUs. Results demonstrate the approach's effectiveness and suggest new deployments and support in the platform for combined pruning and quantization strategies.

Chapter 8

Pruning artificial neural networks: a way to find well-generalizing, high-entropy sharp minima

In the previous chapters, we primarily focused on the structure of sparsification procedures. However, another interesting way to classify pruning procedures is *gradual* or *one-shot*: gradual methods slowly remove parameters over multiple iterations and can achieve higher sparsity, while one-shot techniques greedily remove the targeted amount of parameters in a single step. We feel that an exploration of the potential properties of such procedures has been overlooked, especially: are one-shot strategies enough to match gradual pruning approaches? Is there a specific reason we can prune many parameters with minimal or no generalization loss? Is there any hidden property behind these sparse architectures?

In this chapter, we will shed some light on these topics, comparing one-shot pruning strategies to their gradual counterparts, and investigating the benefits of having a much more computationally-intensive sparsifying strategy. Furthermore, we will highlight some local properties of minima achieved using the two pruning strategies. To this end, we propose *PSP-entropy*, a measure of the state of ReLU-activated neurons, to be used as an analysis tool to better understand the obtained sparse network models.

8.1 Local properties of minima

Throughout this thesis, we have encountered different pruning strategies that rely on state-of-the-art optimization strategies: applying very simple optimizing heuristics to minimize the loss function (e.g., SGD [79, 80]), it is nowadays possible to succeed in training neural networks on massive datasets. These problems are typically over-parametrized, and the dimensionality of the deep model trained can be efficiently reduced with almost no performance loss. Furthermore, minimizing non-convex objective functions is generally supposed to make the trained architecture stuck into local minima. However, the empirical evidence shows that something else is happening under the hood: understanding it is generally of interest to improve learning strategies.

Goodfellow *et al.* observed no loss barrier between a generic random initialization for the neural network model and the final configuration [81]. Such a phenomenon has also been observed on larger architectures by Draxler *et al.* [82]. These works are the basis for the Lottery Ticket Hypothesis. However, a secondary yet relevant observation in [81] stated that there is a loss barrier between different neural network configurations showing similar generalization capabilities. Later, it was revealed that typically a low loss path between well-generalizing solutions to the same learning problem could be found [83]. This brief discussion shows that a general approach to better characterizes such minima has yet to be found.

Keskar *et al.* showed why we should prefer small batch methods to large batch ones: they correlate the stochasticity introduced by small-batch methods to the sharpness of the reached minimum [84]. They generally observe that the larger the batch, the sharper the minimum. Also, they observe that the sharper the minimum, the worse the generalization of the neural network model. In general, many works support the hypothesis that flat minima generalize well, and this has also been the strength for a significant part of the current research [85, 84]. However, this does not necessarily mean that no sharp minimum generalizes well, as we will see in Section 8.3.2.

8.2 Towards a deeper understanding: an entropy-based approach

In this section, we propose PSP-entropy, a metric to evaluate the dependence of the output for a given neuron on the target classification task. The proposed measure will allow us to understand the effect of pruning better.

8.2.1 Post-synaptic potential

As introduced in Chapter 3, we can define the post-synaptic potential of the *i*-th neuron of the *n*-th layer as

$$p_{n,i} = f_{n,i}(\theta_{n,i}, \boldsymbol{y}_{n-1}) \tag{8.1}$$

and its output as

$$y_{n,i} = g_{n,i}(p_{n,i}). \tag{8.2}$$

Typically, deep models are ReLU-activated: here on, let us consider the activation function for all the neurons in hidden layers as $g_{n,i}(\cdot) = \text{ReLU}(\cdot)$. Under such an assumption, it is straightforward to identify two distinct regions for neuron activation:

- $p_{n,i} \leq 0$: the output of the neuron will be exactly zero;
- $p_{n,i} > 0$: there is a linear dependence of the output to $p_{n,i}$.

Hence, let us define

$$g'_{n,i}(p_{n,i}) = \begin{cases} 0 & p_{n,i} \leq 0 \\ 1 & p_{n,i} > 0 \end{cases} \tag{8.3}$$

Intuitively, we understand that if two neurons belonging to the same layer, for the same input, share the same $g'(p)$, then they are linearly mappable to one equivalent neuron:

– $p_{n,i} \leq 0$, $p_{n,j} \leq 0$: one of them can be simply removed;

– $p_{n,i} > 0$, $p_{n,j} > 0$: they are equivalent to a linear combination of them.

In the next section, we will formulate a metric to evaluate the degree of disorder in the post-synaptic potentials. Such a measure will aim to have an analytical tool to give us a broader understanding of the behavior of the neurons in sparse architectures. We are not interested in using this approach toward structured pruning in this work.

8.2.2 PSP-entropy for ReLU-activated neurons

In the previous section, we recalled the concept of post-synaptic potential. Some interesting concepts have also been introduced for ReLU-activated networks: we can use its value to approach the problem of *binning* the state of a neuron, according to $g'(p_{n,i})$. Hence, we can construct a binary random process that we can rank according to its entropy. To this end, let us assume we set as input of our neural network model two different patterns, $\mu_{c,1}$ and $\mu_{c,2}$, belonging to the same class c (for those inputs, we aim at having the same target at the output of the neural network model). Let us consider the PSP $p_{n,i}$:

– if $g'(p_{n,i}|\mu_{c,1}) = g'(p_{n,i}|\mu_{c,2})$ we can say there is *low PSP-entropy*;

– if $g'(p_{n,i}|\mu_{c,1}) \neq g'(p_{n,i}|\mu_{c,2})$ we can say there is *high PSP-entropy*.

We can model an entropy measure for PSP:

$$H(p_{n,i}|c) = -\sum_{t=\{0,1\}} \surd\left[g'(p_{n,i}) = t|c\right] \cdot \log_2 \left\{ \surd\left[g'(p_{n,i}) = t|c\right] \right\} \tag{8.4}$$

where $\surd[g'(p_{n,i}) = t|c]$ is the probability $g'(p_{n,i}) = t$ when presented an input belonging to the c-th class. Since we typically aim at solving a multi-class problem, we can model an overall entropy for the neuron as

$$H(p_{n,i}) = \sum_c H(p_{n,i}|c) \tag{8.5}$$

It is essential to separate the contributions of the entropy according to the c-th target class since we expect the neurons to catch relevant features highly correlated to the target classes. Equation equation 8.5 provides us with critical information: the lower its value, the more it specializes for some specific classes. The formulation in equation 8.5 is very general, and it can be easily extended to higher-order entropy, i.e., the entropy of sets of neurons whose state correlates for the same classes. We are ready to use these metrics to shed further light on the findings in Section 8.3.

8.3 Experiments

For our test, we have decided to compare the state-of-the-art one-shot pruning proposed by Frankle and Carbin [13] to our technique from Chapter 4, LOBSTER. Towards this end, we first obtain a sparse network model using LOBSTER; the non-pruned parameters are then re-initialized to their original values, according to the lottery ticket hypothesis [13]. We aim to determine whether the lottery ticket hypothesis applies to the sparse models obtained using high-performing gradual pruning strategies.

As a second experiment, we want to test the effects of different random initialization while keeping the achieved sparse architecture. According to Liu *et al.*, this should lead to similar results to those obtained with the original initialization [86]. Towards this end, we tried 10 different new starting configurations. As a last experiment, we want to assess how important is the structure originating from the pruning algorithm in reaching competitive performances after re-initialization. For this purpose, we randomly define a new pruned architecture with the same number of pruned parameters as those found via LOBSTER. Again, 10 different structures have been tested.

We decided to experiment with different architectures and datasets commonly employed in the relevant literature: LeNet-300 and LeNet-5-Caffe trained on MNIST, LeNet-5-Caffe trained on Fashion-MNIST [31] and ResNet-32 trained on CIFAR-10.[1] For all our training, we used the SGD optimization method with standard hyper-parameters and data augmentation, as defined in the papers of the different compared techniques [13, 86].

[1]https://github.com/akamaster/pytorch_resnet_cifar10

8.3.1 One-shot vs. gradual pruning

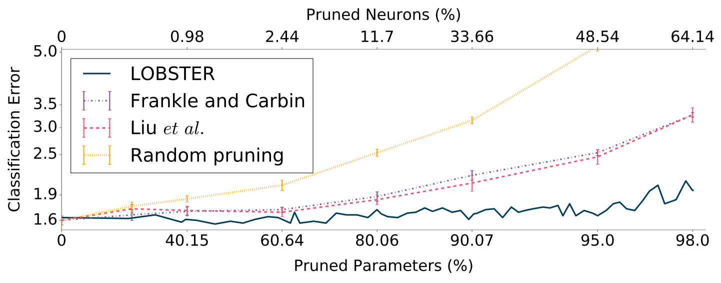
In Figure 8.1, we show, for different percentages of pruned parameters, a comparison between the test accuracy of models pruned using the LOBSTER technique and the models retrained following the approaches we previously defined. We can identify a low compression rate regime in which the re-initialized model can recover the original accuracy, validating the lottery ticket hypothesis. On the other hand, when the compression rate rises (for example, when we remove more than 95% of the LeNet-300 model's parameters, as observed in Figure 8.1a), the retraining approach strives to achieve low classification errors.

As one might expect, other combinations of datasets and models might react differently. For example, LeNet-300 can no longer reproduce the original performance when composed of less than 5% of the initial parameters. On the other hand, LeNet-5, when applied on MNIST, can achieve an accuracy of around 99.20% even when 98% of its parameters are pruned away (Figure 8.1b). This does not happen when applied on a more complex dataset like Fashion-MNIST, where removing 80% of the parameters already leads to performance degradation (Figure 8.1c). Such a gap becomes exceptionally evident when we re-initialize an even more complex architecture like ResNet-32 trained on CIFAR-10 (Figure 8.1d).

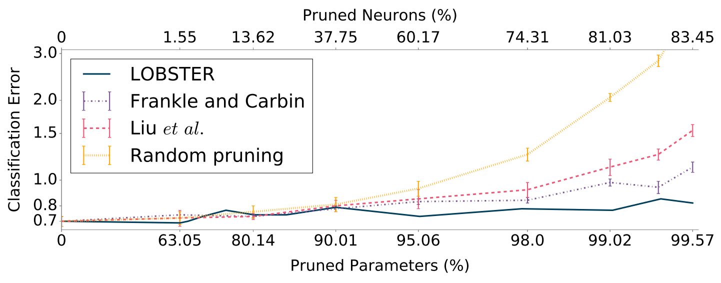
From the reported results, we observe that the original initialization is not always necessary: the error gap between a randomly initialized model and a model using the initial weights' values is minor, with the latter being slightly better. Furthermore, they both fail to recover the performance for high compression rates.

8.3.2 Sharp minima can also generalize well

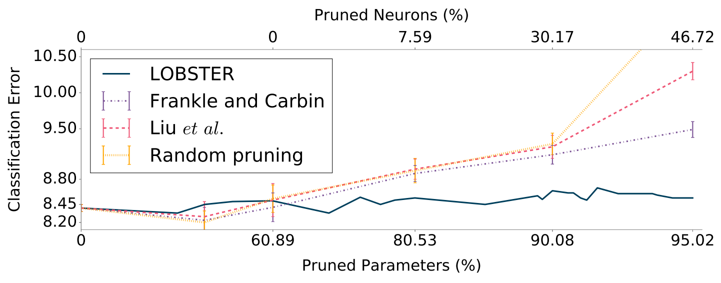
To study the sharpness of local minima, let us focus, for example, on the results obtained on LeNet-5 trained on MNIST. We choose to focus our attention on this particular neural network model since, according to the state-of-the-art and coherently to our findings, we observe the lowest performance gap between gradual and one-shot pruning (as depicted in Figure 8.1b); hence, it is a more challenging scenario to observe qualitative differences between the two approaches. However, all the observations for such a case also apply to the other architectures/datasets explored in Section 8.3.1.



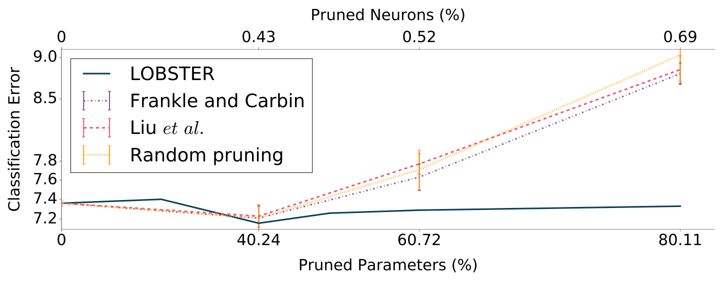
(a)



(b)



(c)



(d)

Fig. 8.1 Test set error for different compression rates: LeNet-300 trained on MNIST (a), LeNet-5 trained on MNIST (b), LeNet-5 trained on Fashion-MNIST (c) and ResNet-32 trained on CIFAR-10 (d).

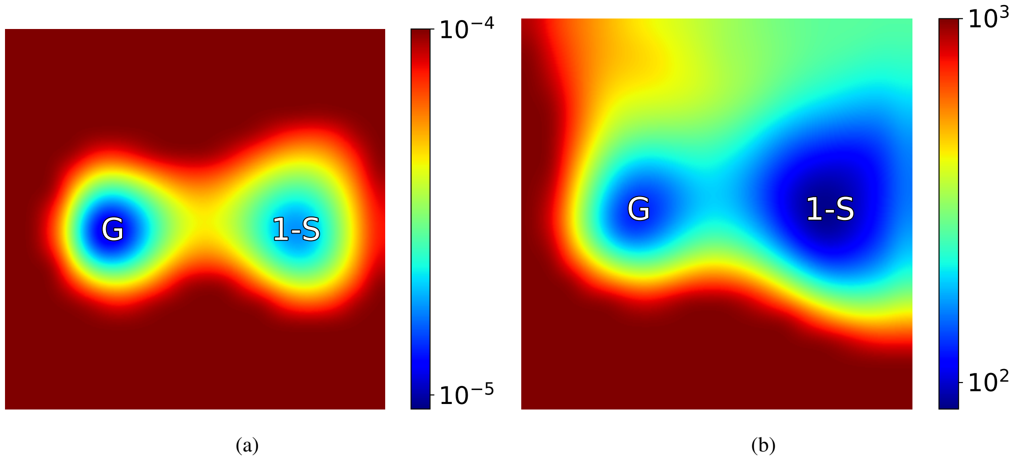


Fig. 8.2 Results of LeNet-5 trained on MNIST with the highest compression (99.57%): (a) plots the loss in the training set and (b) plots the magnitude of the top-5 largest hessian eigenvalues. G is the solution with gradual learning, while 1-S is the best one-shot solution (Frankle and Carbin).

To obtain the maps in Figure 8.2, we follow the approach proposed by [81], and we plot the loss for the neural network configurations between two reference ones: in our case, we compare a solution found with gradual pruning (G) and one-shot (1-S). Then, we take a random orthogonal direction to generate a 2D map. Figure 8.2a shows the loss on the training set between iterative and one-shot pruning for the highest compression rate (99.57% of pruned parameters as shown in Figure 8.1b). According to our previous findings, we see that iterative pruning lies in a lower loss region. Here, we also show the plot of the top-5 Hessian eigenvalues (all positive) in Figure 8.2b, using the efficient approach proposed in [87]. Interestingly, we observe that the solution proposed by iterative pruning lies in a narrower minimum than the one found using the one-shot strategy, despite generalizing slightly better. With this, we do not claim that narrower minima generalize well: gradual pruning strategies enable access to a *subset of well-generalizing narrow minima*, showing that not all the narrow minima generalize worse than the wide ones. This finding raises warnings against second order optimization, which might favor the research of flatter, wider minima, ignoring well-generalizing narrow minima. These non-trivial solutions are naturally found using gradual pruning and cannot be found using one-shot approaches, which focus their effort on larger minima. In general, the sharpness of these minima explains why, for high compression rates, retraining strategies fail in recovering the performance, considering that it is in general harder to access this class of minima.

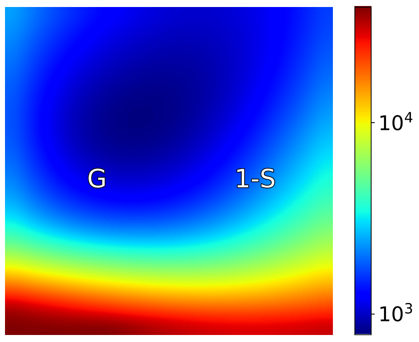


Fig. 8.3 L2 norm of PSP values for LeNet-5 trained on MNIST with 99.57% of pruned parameters.

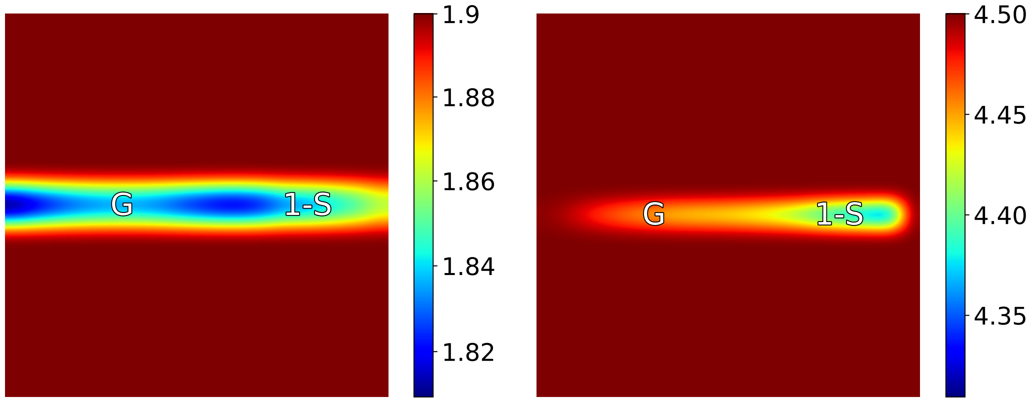
8.3.3 Study on the post-synaptic potential

In Section 8.3.2, we have observed that, as a result, iterative strategies focus on well-generalizing sharp minima. Is there something else yet to say about those?

Let us inspect the average magnitude values of the PSPs for the different solutions: towards this end, we plot the average of their L2 norm values (g^2). As a first finding, gradually-pruned architectures naturally have lower PSP L2-norm values, as shown in Figure 8.3. None of the pruning strategies explicitly minimize the term in g^2: they naturally drive the learning toward such regions. However, the solution showing better generalization capabilities shows lower g^2 values. Of course, there are regions with even lower g^2 values; however, according to Figure 8.2a, they should be excluded since they correspond to high-loss values (not all the low g^2 regions are low-loss).

If we look at the PSP-entropy formulated in equation 8.5, we observe something interesting: gradual and one-shot pruning show comparable first-order entropies, as shown in Figure 8.4a.[2] It is interesting to see that there are also lower entropy regions which however correspond to higher loss values, according to Figure 8.2a. When we move to higher-order entropies, something even more interesting happens: gradual pruning shows higher entropy than one-shot, as depicted in Figure 8.4b (displaying the second-order entropy). In such a case, having a lower entropy means having more groups of neurons specializing in specific patterns which correlate to the target class; on the contrary, having higher entropy yet showing better generalization performance results in having more general features, more agnostic towards a specific class, which still allow a correct classification performed by the output layer. This

[2]the source code is available at https://github.com/EIDOSlab/PSP-entropy.git



(a) (b)

Fig. 8.4 Results on LeNet-5 trained on MNIST with 99.57% of pruned parameters. (a) plots the first order PSP-entropy, while (b) shows the second-order PSP entropy.

counter-intuitive finding has potentially-huge applications in transfer learning and domain adaptation, where it is critical to extract more general features not very specific to the originally-trained problem.

8.4 Summary

In this chapter, we have compared one-shot and gradual pruning on different state-of-the-art architectures and datasets. In particular, we have focused our attention on understanding the potential differences and limits of both approaches toward achieving sparsity in neural network models.

We have observed that one-shot strategies are very efficient to achieve moderate sparsity at a lower computational cost. However, there is a limit to the maximum achievable sparsity, which can be overcome using gradual pruning. The highly-sparse architectures, interestingly, focus on a subset of sharp minima which are able to generalize well, which poses some questions to the potential sub-optimality of second-order optimization in such scenarios. This explains why we observe that one-shot strategies fail in recovering the performance for high compression rates. More importantly, we have observed, contrary to what could be expected, that highly-sparse gradually-pruned architectures are able to extract general features non-strictly correlated to the trained classes, making them unexpectedly, potentially, a good match for transfer-learning scenarios.

Future works include a quantitative study on transfer learning for sparse architectures and PSP-entropy maximization-based learning.

Chapter 9

To update or not to update? Neurons at equilibrium in deep models

Many works of the last two years have received inspiration from the Lottery Ticket Hypothesis in the quest to determine the early lottery tickets. This is, however, a challenging task to solve: Frankle *et al.* [23] shows that there is a region, at the very early stages of learning, where the lottery tickets identified with iterative pruning are "not stable" (meaning that tickets extracted at different moments of this early stages are essentially different). This suggests that, in the very first epochs, the neural network evolves in very different states, making the problem of a priori identifying winning tickets hard [88]. Other works also endorse this, including [89, 90]. On the other hand, different approaches reduce the overall complexity of the iterative training by drawing early-bird tickets [91] (meaning that they learn the lottery tickets when the model has not yet reached full convergence), even reducing the training data [92] or moving the first steps towards structurally-sparse winning tickets, yet still at iterative fashion, applying similar concept as Frankle and Carbin to entire neurons and channels [93]. Unfortunately, training a sparse network with standard optimizers leads to subpar results [94] or the final result does not differ much from magnitude pruning at the end of the training [95]. The causes for such behaviors are still a matter of debate among the community [94, 95].

Taking inspiration from this line of research, in this chapter, we shift the focus from the single parameter to the entire neuron. We propose NEq, an approach to evaluate whether a given neuron is at *equilibrium* for the learning dynamics. If the neuron is in such a state, its parameters have already reached a target configuration and do not require a further update. Unlike many other recent approaches, NEq disables *entire neurons* (hence, in a structured way), does not require prior knowledge of the specific task (e.g., by first training a model to convergence), and automatically self-adapts to the particular learning policy deployed. Unlike pruning techniques, NEq does not remove the neurons' contribution to the output; instead, it only prevents unnecessary updates to their weights: as a result, we reduce the number of operations performed by the back-propagation algorithm and the optimizer.

With NEq, we learn the essential lessons related to the lottery ticket hypothesis, targeting the reduction of computational complexity at training time without exploiting knowledge of pre-training models or rewinding. Hence, we do not target the achievement of sparse architectures. Still, we aim to determine when a whole neuron must be updated or when the computation of the gradient for its parameters is unnecessary. As such, the comparison with the other presented approaches will be unfair, as they require a much greater computational complexity for training because of unstructured sparsity (which introduces an overhead in the representation of the tensors) and the iterative strategies. Furthermore, along the training process, we will observe the possibility that some neurons, already kept in a "frozen" state, might unfreeze, requiring additional update steps. Although resource reallocation has been exploited before [96, 97], our unfreezing is different as it involves learning of a specific target function by the neuron and not learning new ones through their reallocation.

9.1 Neurons at equilibrium

This section will treat the problem of determining when a given neuron and the learning dynamics are at equilibrium. Towards this end, we define $y^t_{n,i,\xi}$ as the output of the i-th neuron when the input ξ is fed to the whole model trained after t epochs. Given a set of inputs $\xi \in \Xi_{val}$ (where Ξ_{val} is the validation set), it is possible to compare each m-th element $y^t_{n,i,m,\xi}$ with $y^{t-1}_{n,i,m,\xi}$, for the same model's input: what changes are the parameters of the model. Figure 9.1 provides an overview of the nomenclature used: in the rest of the section, we will see how to determine when a neuron is at equilibrium.

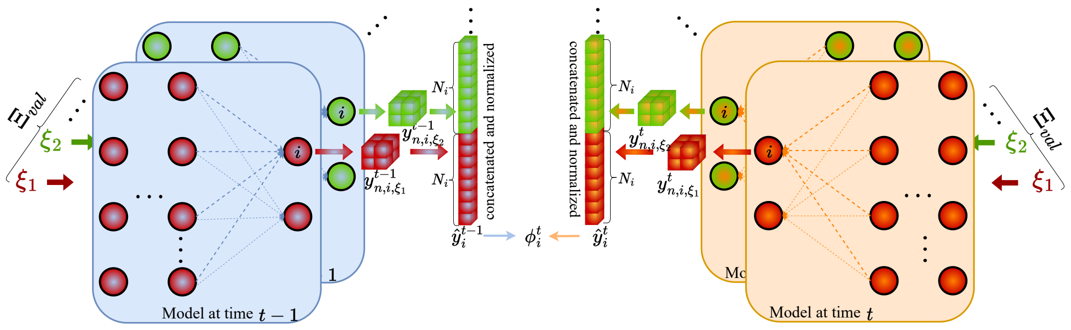


Fig. 9.1 For a given time t, the model (either in blue or orange) receives samples from the validation set (in red or green). The output of the i-th neuron (whose cardinality is N_i) depends on the model's parameters and the specific sample on the validation set. These outputs are squeezed, concatenated and the obtained vector (of size $N_i \cdot \|\Xi_{val}\|_0$, being $\|\Xi_{val}\|_0$ the cardinality of the validation set) is then normalized, obtaining $\hat{y}^t_i$.

9.1.1 Neuronal equilibrium

In this section, we are interested in evaluating when the relationship between the input of the model and the output of the i-th neuron is modified. When this happens, the neuron is at *non-equilibrium*, meaning that its learned function, in the whole picture (or in other words, taking into account the evolution of the neurons in the previous layers as well), is still "evolving". We are interested in identifying the scenarios where the neuron is at *equilibrium* at the net of the interactions with the other neurons. To assess it, let us define the cosine similarity between all the outputs of the i-th neuron at time t and at time $t-1$ for the whole validation set Ξ_{val} as

$$\phi^t_i = \sum_{\xi \in \Xi_{val}} \sum_{m=1}^{M_i} \hat{y}^t_{n,i,m,\xi} \cdot \hat{y}^{t-1}_{n,i,m,\xi}. \tag{9.1}$$

Here, we can determine that, when $\phi_i = 1$, the i-th neuron produces the same (eventually scaled) output between the evaluation at time t and at time $t-1$ for the same input ξ of the model. We say the i-th neuron reaches equilibrium when we have

$$\lim_{t\to\infty} \phi^t_i = k, \tag{9.2}$$

where $k \in [-1;+1]$ is some constant value. We can have the following scenarios:

- $k = 1$. In this case, the two outputs are perfectly correlated, meaning that the relationship bounding the input of the whole model ξ and the output of the specific i-th neuron is maintained.
- $k \in (0;1)$. The outputs correlate, but we are in an oscillatory behavior (in the sense that the cosine similarity varies by a constant value between consecutive evaluations). This effect can be caused by stochastic effects like high learning rate/regularization, small batch size, or a combination of them.
- $k \in [-1;0]$. Also, in this case, we are in the presence of oscillatory behavior, but the outputs are anti-correlated or de-correlated.

9.1.2 Neuron dynamics evaluation

To assess the convergence to equilibrium for equation 9.2, it is essential to evaluate the variation of the similarities ϕ_i^t over time. Towards this end, let us introduce the *variation of similarities*

$$\Delta\phi_i^t = \phi_i^t - \phi_i^{t-1}. \tag{9.3}$$

According to the analysis in Sec. 9.1.1, in this case, we say we reach equilibrium when $\Delta\phi_i^t \to 0$. Hence, it is useful to keep track of the recent evolution over the similarity scores in the model: towards this end, we can introduce the velocity of the similarity variations:

$$v_{\Delta\phi_i}^t = \Delta\phi_i^t - \mu_{eq} v_{\Delta\phi_i}^{t-1}, \tag{9.4}$$

where μ_{eq} is the momentum coefficient. We can rewrite equation 9.4 making the similarity scores explicit, obtaining

$$v_{\Delta\phi_i}^t = \begin{cases} \phi_i^t + \sum\limits_{\alpha=1}^{t} (-1)^\alpha \left[(\mu_{eq})^{\alpha-1} + (\mu_{eq})^\alpha \right] \phi_i^{t-\alpha} & \mu_{eq} \neq 0 \\ \phi_i^t - \phi_i^{t-1} & \mu_{eq} = 0, \end{cases} \tag{9.5}$$

where $(\cdot)^\alpha$ indicates power of α. If we assume $\phi^t \in [0;1] \forall t$ (which is the case of ReLU-activated neurons), to prevent equation 9.5 from exploding, we need to set

$\mu_{eq} \in [0; 0.5]$. We can extend this setup to any layer if we assume that the neurons in the trained model will not reach equilibrium with anti-correlated outputs.

9.1.3 Selection of trainable neurons at non-equilibrium

To evaluate when a given neuron has reached equilibrium, exploiting equation 9.2, we can say that the i-th neuron is at equilibrium when it can satisfy

$$\left| v^t_{\Delta\phi} \right| < \varepsilon, \qquad \varepsilon \geq 0. \tag{9.6}$$

It is essential to notice that once equation 9.6 is satisfied for a specific t, in case something changes in the learning dynamics (for example, the learning rate is re-scaled), there may exist some $t' > t$ such that the constraint is not satisfied anymore. When this happens, it means that the neuron is driven towards *new states* and is no anymore at equilibrium. Hence, it requires to be updated again.

9.1.4 Overall training scheme

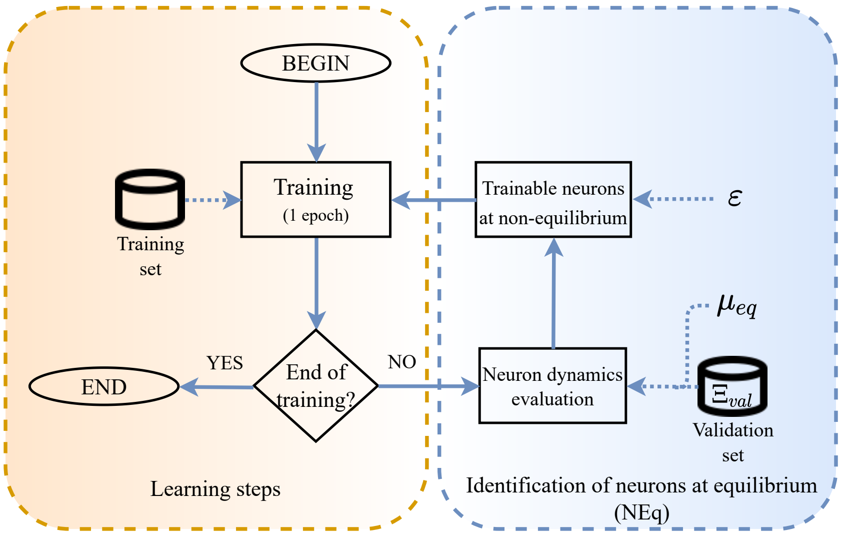


Fig. 9.2 Overall training scheme. In orange is the standard training part, and in blue is the neuron equilibrium evaluation and selection stages (we name this part NEq).

The overall training scheme is summarized in Figure 9.2. The model is trained for one epoch, after which neurons at equilibrium are identified. We split this into two phases: in the first (neuron dynamics evaluation), the velocity of the similarities is evaluated according to equation 9.4, while in the second (trainable neurons at non-equilibrium), the hidden neurons at non-equilibrium, which will be trained for the next epoch, are identified according to equation 9.6. All the neurons are considered at non-equilibrium by default for the first epoch. The evaluation of neurons at equilibrium is agnostic to the general training strategy, which can include arbitrary re-scaling for the learning rate/hyper-parameters or the most common optimizers. The following section will test this procedure on different architectures, tasks, and learning strategies.

9.2 Experiments

This Section reports the experiments supporting the approach as presented in Section 9.1.4. First, we will perform an ablation study, analyzing single contributions for the introduced hyper-parameters and providing an overview of neuronal equilibrium along the training process; then, we will test the proposed technique on state-of-the-art network architectures, datasets, and learning policies. All experiments were performed using 8 NVIDIA A40 GPUs, and the source code uses PyTorch 1.10.[1]

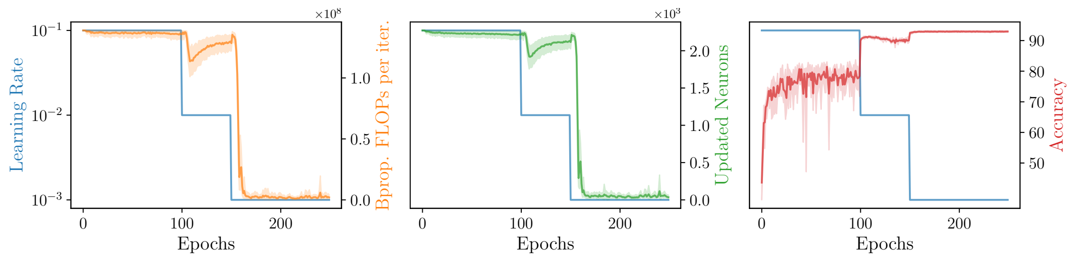
9.2.1 Ablation study

We performed our ablation study training a ResNet-32 [32] model on CIFAR-10 [98]. Unless differently specified, following the hyper-parameters setup of [64], the model is trained using SGD as an optimizer, with a starting learning rate $\eta = 0.1$ and momentum $\mu_{opt} = 0.9$ and a weight decay of 5×10^{-4} for 250 epochs. The learning rate is decayed by a factor of 10 after 100 and 150 epochs, using the formulation as in equation 9.4, with $\|\Xi_{val}\|_0 = 50$, $\mu_{eq} = 0.5$, and $\varepsilon = 0.001$.

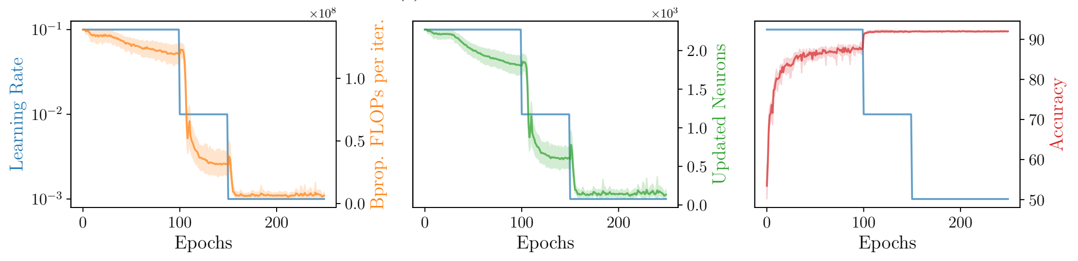
SGD vs Adam

To show that our technique automatically self-adapts to the training policy, we compare the evolution of the FLOPs required for a back-propagation step and the number of updated neurons of two different training of the ResNet-32: one using

[1]the source code is available at https://github.com/EIDOSLAB/NEq.



(a) ResNet-32 trained with SGD.



(b) ResNet-32 trained with Adam.

Fig. 9.3 Back-propagation FLOPs (left, orange), updated neurons (center, green), and classification accuracy (right, red) for ResNet-32 trained on CIFAR-10.

the SGD optimizer with $\mu_{opt} = 0.9$, and the other using the Adam optimizer. For Adam, we leave the hyper-parameters to their default values ($\eta = 0.001$, $\beta_1 = 0.9$, and $\beta_2 = 0.999$) and use the same weight decay as for SGD (5×10^{-4}). Figure 9.3 shows the trends for the two training procedures. In the first phase of the train, where η is high, the amount of the trained neurons (and the FLOPs required for the backward pass) is higher. This is related to the general lack of equilibrium in the network's neurons: at high learning rates, the configuration of the neurons' parameters is subjected to high stochastic noise. As the train progresses and the network moves toward its final configuration, fewer and fewer neurons need to be updated. Noticeably, Adam drives the neurons towards equilibrium faster, as expected; however, in simple tasks like the considered one, it converges to lower accuracy scores (92.96% for SGD and 92.01% for Adam). Furthermore, at the first learning rate decay (epoch 100), for SGD, the number of updated neurons decreases and then increases; such a phenomenon is not present in the Adam case. This is explained by the different working principles of the two optimizers: SGD explores the solution space looking for large minima, searching for configurations that prevent equilibrium in high learning rate regimes.

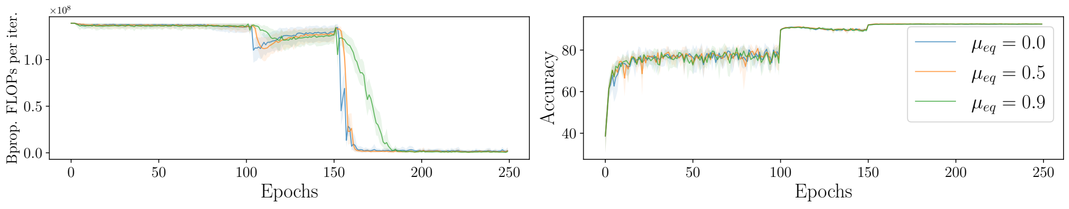
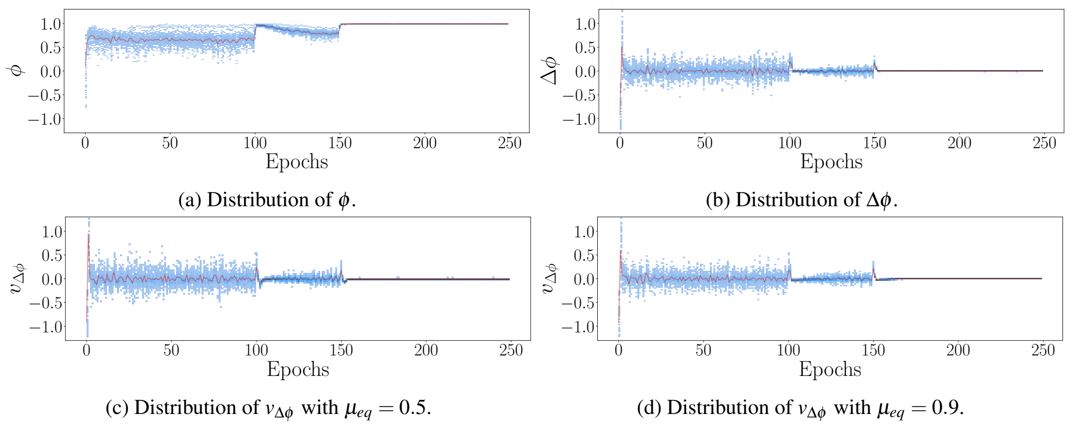


Fig. 9.4 Back-propagation FLOPs (left) and accuracy (right) for different values of μ_{eq} for ResNet-32 trained on CIFAR-10.



(a) Distribution of ϕ.

(b) Distribution of $\Delta\phi$.

(c) Distribution of $v_{\Delta\phi}$ with $\mu_{eq} = 0.5$.

(d) Distribution of $v_{\Delta\phi}$ with $\mu_{eq} = 0.9$.

Fig. 9.5 Neuronal equilibrium-related quantities for ResNet-32 trained on CIFAR-10. The red line indicates the average.

Distribution of ϕ & choice of μ_{eq}

Looking at different values for μ_{eq} in Figure 9.4, we observe for all the values a convergence to similar accuracy. Despite without warranty from the theory, we tested a very large value for the momentum coefficient (0.9): the convergence of $v^t_{\Delta\phi_i}$ shows that the neurons are in general in a significantly correlated case of equilibrium, with very high values for k in equation 9.2, which is also empirically observed in Figure 9.5a. However, including a very large value for μ_{eq} maintains the memory of very old variations, producing a sub-optimal reduction in terms of FLOPs reduction. We find that a good compromise, supported by the findings as in Section 9.1.2, is to set μ_{eq} to 0.5. Figure 9.5 reports the distribution for the velocities for ϕ, $\Delta\phi$, and $v_{\Delta\phi}$, observing that the average converges to a specific k for each of the three learning rates used.

Table 9.1 Ablation for ResNet-32 trained on CIFAR-10.

(a) Ablation on $\|\Xi_{val}\|_0$.

$\|\Xi_{val}\|_0$	**Bprop. FLOPs per iteration**	**Accuracy**
500	84.73M ± 629.15K	92.70 ± 0.12
250	82.62M ± 613.14K	92.80 ± 0.43
100	84.82M ± 628.05K	92.81 ± 0.15
50	84.81M ± 629.12K	92.96 ± 0.21
25	84.49M ± 629.37K	92.62 ± 0.28
10	84.91M ± 627.11K	92.70 ± 0.27
5	84.34M ± 619.37K	92.57 ± 0.48
2	85.76M ± 617.11K	92.80 ± 0.24
1	85.56M ± 626.09K	92.77 ± 0.23

(b) Ablation on ε.

ε	**Bprop. FLOPs per iteration**	**Accuracy**
0.0	136.71M ± 15.34K	92.62 ± 0.23
0.0001	124.45M ± 161.76K	92.65 ± 0.40
0.0005	89.29M ± 589.71K	92.69 ± 0.19
0.001	83.62M ± 629.22K	92.96 ± 0.21
0.005	65.65M ± 591.07K	91.72 ± 0.37
0.01	52.53M ± 590.30K	91.23 ± 0.32
0.05	16.10M ± 254.94K	86.80 ± 0.29
0.1	4.97M ± 186.54K	83.90 ± 0.66
0.5	1.78M ± 137.92K	76.78 ± 2.57

Impact of the validation set size and ε

Tab. 9.1 provides an empirical evaluation of the impact on the performance and on the FLOPs varying the validation set size. We indeed do not observe a significant effect on the performance of the model changing it. Interestingly, the approach produces excellent results even for extremely low cardinality for the validation set (down to even a single image): this can be explained by the presence of convolutional layers (the only fully-connected layer is the output layer, excluded by default) which even with small images produce high-dimensionality output in every neuron (Figure 9.1) and by the homogeneity of the considered dataset. Investigating the impact of ε, we find a drop in performance for very high values of ε, identifying a good compromise for classification tasks to 0.001.

9.2.2 Main experiments

In this section, we show the results of the proposed method. For our experiments, we focused on different state-of-the-art architectures trained on standard classification and semantic segmentation datasets. All the learning policies used are borrowed from other works and are un-optimized to test the adaptability of NEq.

ResNet-32 trained on CIFAR-10. The training spans 250 epochs, using SGD as optimizer with momentum $\mu_{opt} = 0.9$, weight decay 5×10^{-4} and initial learning rate $\eta = 0.1$, reduced by a factor of 10 after 100 and 150 epochs. To evaluate the

Table 9.2 Results of the application of NEq to each experimental setup, compared to the stochastic approach. We report the average FLOPs per iteration at backpropagation, and the final performance of the model evaluated on the test set (values annotated with † report the classification accuracy, values annotated with ‡ report the mean IoU).

Dataset	**Model**	**Approach**	**Bprop. FLOPs per iteration**	**Performance**
CIFAR-10	ResNet-32	Baseline	138.94M $\pm$ 0.0M	92.85% $\pm$ 0.23%†
		Stochastic ($p = 0.2$)	112.99M $\pm$ 0.00M (-18.68%)	92.78% $\pm$ 0.19% (-0.07%)†
		Stochastic ($p = 0.5$)	69.75M $\pm$ 0.00M (-49.8%)	91.88% $\pm$ 0.27% (-0.97%)†
		Stochastic*	86.34M $\pm$ 0.00M (-37.85%)	92.23% $\pm$ 0.25% (-0.62%)†
		Neq	84.81M $\pm$ 0.63M (-38.96%)	92.96% $\pm$ 0.21% (+0.11%)†
ImageNet-1K	ResNet-18	Baseline	3.64G $\pm$ 0.0G	69.90% $\pm$ 0.04%†
		Stochastic ($p = 0.2$)	2.94G $\pm$ 0.00G (-19.26%)	69.42% $\pm$ 0.16% (-0.48%)†
		Stochastic ($p = 0.5$)	1.85G $\pm$ 0.00G (-49.11%)	69.18% $\pm$ 0.03% (-0.72%)†
		Stochastic*	2.82G $\pm$ 0.00G (-22.58%)	69.45% $\pm$ 0.06% (-0.45%)†
		Neq	2.80G $\pm$ 0.03G (-23.08%)	69.62% $\pm$ 0.06% (-0.28%)†
	Swin-B	Baseline	30.28G $\pm$ 0.00G	84.71% $\pm$ 0.04% †
		Stochastic ($p = 0.2$)	24.65G $\pm$ 0.00G (-18.6%)	84.54% $\pm$ 0.04% (-0.83%)†
		Stochastic ($p = 0.5$)	16.15G $\pm$ 0.00G (-46.67%)	84.40% $\pm$ 0.02% (-0.31%)†
		Stochastic*	11.02G $\pm$ 0.00G (-63.67%)	84.27% $\pm$ 0.04% (-0.44%)†
		Neq	10.78G $\pm$ 0.02G (-64.39%)	84.35%$\pm$0.02% (-0.36%)†
COCO	DeepLabv3	Baseline	305.06G $\pm$ 0.0G	67.71% $\pm$ 0.02%‡
		Stochastic ($p = 0.2$)	248.69G $\pm$ 0.00G (-18.48%)	67.11% $\pm$ 0.02% (-0.60%)‡
		Stochastic ($p = 0.5$)	163.42G $\pm$ 0.00G (-46.43%)	66.91% $\pm$ 0.04% (-0.80%)‡
		Stochastic*	229.00G $\pm$ 0.00G (-24.93%)	67.02% $\pm$ 0.03% (-0.69%)‡
		Neq	217.29G $\pm$ 0.04G (-28.77%)	67.22% $\pm$ 0.04% (-0.49%)‡

neuronal equilibrium we used a $\|\Xi_{val}\|_0$ of 50 images, $\mu_{eq} = 0.5$, and $\varepsilon = 0.001$. We used a batch size of 100 images during training.

ResNet-18 trained on ImageNet-1K [99]. This model was trained with SGD as an optimizer for 90 epochs, with $\eta = 0.1$, reduced by a factor of 10 every 30 epochs, $\mu_{opt} = 0.9$ and weight decay 10^{-4} using a batch size of 128. We used a $\|\Xi_{val}\|_0$ of 1.2k images, $\mu_{eq} = 0.5$, and $\varepsilon = 0.001$.

Swin Transformer [100] (Swin-B) trained on ImageNet-1K. We used the Swin-B architecture to test our technique on more modern models and training policies. Here we trained the model from a pre-trained checkpoint trained on ImageNet-21K, following the official GitHub repository[2] released under the MIT License. We used a $\|\Xi_{val}\|_0$ of 1.2k images, $\mu_{eq} = 0.5$, and $\varepsilon = 0.001$.

DeepLabv3 [101] trained on COCO [102]. Besides classification tasks of varying complexity, we tested our procedure on a semantic segmentation problem. We used DeepLabv3 with a ResNet-50 backbone and the COCO dataset for this experiment. To train the network, we followed the state-of-the-art procedure defined in PyTorch[3]. We evaluated the neuronal equilibrium using a $\|\Xi_{val}\|_0$ of 320 images, $\mu_{eq} = 0.5$, and $\varepsilon = 0.02$.

9.2.3 Discussion

The results (average over five different runs) are reported in Table 9.2. For each experiment, we compare our technique with a "stochastic" approach. Namely, we randomly halt the update of a given neuron at every epoch and with probability p. We test on three different probabilities: 0.2, 0.5, and a probability as close as possible to the average over the one achieved by NEq - indicated with "*". To evaluate the effectiveness of the proposed procedure, we focus on the average computational complexity of the back-propagation for a single update iteration (expressed in FLOPs) and the network generalization capabilities at the end of the training. In all the considered scenarios, it is possible to observe a reduction of FLOPs with very marginal or no performance drop for NEq. Compared to the stochastic approach, with fixed probabilities, the amount of saved computation is similar in all the scenarios considered. Still, the loss in performance varies, depending on the specific architecture/dataset. On the contrary, NEq remains consistent in performance, self-adapting to the particular setup and saving the largest FLOPs for the given performance. Furthermore, the performance loss is lower when testing the stochastic approach with the same FLOPs saving (hence, even letting that information leak in favor of the stochastic approach).

[2]https://github.com/microsoft/Swin-Transformer
[3]https://github.com/pytorch/vision/tree/main/references/segmentation

9.3 Summary

The Lottery Ticker Hypothesis [13] showed the existence of sub-graphs in deep models which, when trained in isolation, can match the original performance of the whole model. Finding these sub-graphs is, however, a complex task, as in the first stages of the learning, the model itself is at non-equilibrium. Identifying these with a dynamic strategy, without requiring a posterior over the whole training process, is a crucial task to be solved towards computational resource saving. Differently from the vast majority of the literature, which focuses on the identification of sub-graphs without any substantial computational saving (as they rely on iterative or roll-back algorithms), we have introduced the knowledge of neuronal equilibrium, looking for entire structures of the deep model at equilibrium, not requiring further optimization and gradient computation, which self-adapts to very specific experimental setups on very different learning scenarios. This work opens the doors to a deeper understanding of the deep neural network's learning dynamics and to developing new training strategies exploiting this knowledge.

The proposed approach analyzes the behavior of an entire neuron. However, empirical experiments show that there could be further improvements considering ensembles of neurons. For example, Figure 9.5 shows the average value for the similarities close to a constant, but many neurons are still away from the convergence value, meaning that these neurons, at isolation, are still not at equilibrium: is the scenario changing when considering the dynamics of groups of neurons? Furthermore, to validate the adaptability of NEq to the most popular training schemes, no optimization of the hyper-parameters for the training procedure has been performed (as it is out of scope for our evaluation). However, higher savings in computational complexity are possible by tuning the training strategy as well. In such a direction, prospectively, it will be of interest to design more efficient learning strategies that consider the concept of neuronal equilibrium.

Chapter 10

Conclusions

In this thesis, we explored the problem of decreasing the resource requirements of a deep network model during inference via neural network pruning. Pruning allows us to remove many elements (parameters or neurons) from an architecture, reducing the model's memory footprint and inference time. In Chapter 3, we went over the main concepts of neural network pruning while also providing an overview of the state-of-the-art. In particular, we presented the four main categories of pruning procedures: unstructured, in which parameters are removed independently; structured, in which entire neurons are removed from the architecture; one-shot, usually faster albeit fewer parameters are removed; gradual, generally slower, but allow for a more significant reduction in the number of parameters. Exploring these differences allowed us to propose multiple pruning-related procedures and expand our understanding of the effect of pruning on neural network models.

In Chapter 4, we proposed LOBSTER: an unstructured, gradual pruning technique. LOBSTER uses a sensitivity-based regularization to push toward zero the least influential parameters and promote sparsity in the architecture. In this context, we define the sensitivity of a parameter as the derivative of the loss function with respect to the target parameter. Unlike standard L_2 regularization, the sensitivity is aware of the global contribution of the parameters to the loss and can self-tune its effect. Moreover, LOBSTER computes the sensitivity by exploiting the already available gradient of the loss function, avoiding the additional computations of other sensitivity-based approaches. With extensive testing, we showed that LOBSTER achieves competitive results for different combinations of datasets and architectures.

LOBSTER, while able to remove many parameters, result in unstructured sparsification, which is harder to exploit without particular sparse-oriented software or hardware. For this reason, in Chapter 5, we presented SeReNe. SeReNe is a structured, gradual pruning procedure that applies a neuron-oriented regularization according to the contribution of the neuron's post-synaptic potential to the network's output. Our experiments show that the technique can prune a satisfactory amount of neurons while maintaining a good generalization performance. Since SeReNe could be computationally intensive, we proposed some faster variations, albeit less performing.

To better exploit the structure in the models pruned using SeReNe, we presented Simplify in Chapter 6. Simplify is a PyTorch-compatible library that removes the zeroed neurons from a pruned neural network. Simplify allows fully exploiting the induced structured sparsity in a network model without needing ad hoc sparse-oriented software or hardware and automatically handles bias propagation and residual layers. Evaluation with different state-of-the-art architectures shows that simplified models achieve faster inference time when compared to both their dense and pruned-only counterparts. By using Simplify, in Chapter 7, we provided some empirical evidence of the benefits of structured pruning. To this end, we investigated the contribution of pruning in the MPEG-7 Part 17 neural network compression pipeline, verifying that structured pruning presents better synergy with quantization and entropy coding, enabling better end-to-end compression and inference speed of the deployed model. Also, in the same chapter, we presented HLSinf, an open-source platform for neural network accelerators in FPGAs. Integrated into the EDDL library, this platform eases the deployment of pruned and quantized models and exploits the efficiency of FPGAs.

While much importance was given to the structure of the pruning procedure, in Chapter 8, we explored and compared the properties of gradual and one-shot pruning and proposed PSP-entropy to measure the state of ReLU-activated neurons. Comparing one-shot and gradual pruning, we observed that the latter focus on a subset of sharp, well-generalizing minima, which can explain why one-shot approaches fail to recover the performance for high sparsity ratios. Also, we observed that gradual pruning generates architectures able to extract more general features making them a good candidate for transfer-learning scenarios.

Finally, in Chapter 9, we shifted the focus of pruning from identifying the less useful parameters during the forward pass to evaluating which set of neurons has to be updated for each training epoch. The proposed technique, NEq, allows us to find such neurons: we leverage the concept of neuronal equilibrium to freeze the gradient computation for neurons that no longer need updates. This reduces the time required for computing the gradients of the neural network during the backpropagation step. Various experiments on different tasks showed that NEq allows us to reduce the number of operations needed to perform backpropagation substantially.

Chapter 11

Publications

11.1 Conference papers

- **Andrea Bragagnolo**; Enzo Tartaglione; Marco Grangetto "To update or not to update? Neurons at equilibrium in deep models." *Advances in Neural Information Processing Systems 36* (2022).
- José Flich; Laura Medina; Izan Catalán; Carles Hernández; **Andrea Bragagnolo**; Fabrice Auzanneau; David Briand "Efficient Inference Of Image-Based Neural Network Models In Reconfigurable Systems With Pruning And Quantization." *2022 IEEE International Conference on Image Processing* (2022). DOI: 10.1109/ICIP46576.2022.9897752.
- **Andrea Bragagnolo**; Enzo Tartaglione; Attilio Fiandrotti; Marco Grangetto "On the role of structured pruning for neural network compression." *2021 IEEE International Conference on Image Processing* (2021). DOI: 10.1109/ICIP42928.2021.9506708.
- Enzo Tartaglione; **Andrea Bragagnolo**; Marco Grangetto et al. "Pruning artificial neural networks: A way to find well-generalizing, high-entropy sharp minima." *International Conference on Artificial Neural Networks* (2020). DOI: 10.1007/978-3-030-61616-8_6.

11.2 Journal papers

- **Andrea Bragagnolo**; Carlo Alberto Barbano "Simplify: A Python library for optimizing pruned neural networks." *SoftwareX 17* (2022).
  DOI: 10.1016/j.softx.2021.100907. Online ISSN: 2352-7110.

- Enzo Tartaglione; **Andrea Bragagnolo**; Attilio Fiandrotti; Marco Grangetto "Loss-based sensitivity regularization: towards deep sparse neural networks." *Neural Networks 146* (2022).
  DOI:10.1016/j.neunet.2021.11.029. Online ISSN: 1879-2782. Print ISSN: 0893-6080.

- Enzo Tartaglione; **Andrea Bragagnolo**; Francesco Odierna; Attilio Fiandrotti; Marco Grangetto "Serene: Sensitivity-based regularization of neurons for structured sparsity in neural networks." *IEEE Transactions on Neural Networks and Learning Systems* (2021).
  DOI: 10.1109/TNNLS.2021.3084527. Online ISSN: 2162-2388. Print ISSN: 2162-237X.

11.3 Patents

- Enzo Tartaglione, Marco Grangetto, Francesco Odierna, **Andrea Bragagnolo**, and Attilio Fiandrotti. "Method and apparatus for pruning neural networks." *IT201900018821A1*, issued April 15, 2021. *US20220284298A1*, filed September 8, 2022. *EP4035087A1*, filed August 3, 2022.

References

[1] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. In *ICLR 2015 : International Conference on Learning Representations 2015*, 2015.

[2] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel Ziegler, Jeffrey Wu, Clemens Winter, Chris Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. In H. Larochelle, M. Ranzato, R. Hadsell, M.F. Balcan, and H. Lin, editors, *Advances in Neural Information Processing Systems*, volume 33, pages 1877–1901. Curran Associates, Inc., 2020.

[3] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 770–778, 2016.

[4] Andrew Howard, Mark Sandler, Bo Chen, Weijun Wang, Liang-Chieh Chen, Mingxing Tan, Grace Chu, Vijay Vasudevan, Yukun Zhu, Ruoming Pang, Hartwig Adam, and Quoc Le. Searching for mobilenetv3. In *2019 IEEE/CVF International Conference on Computer Vision (ICCV)*, pages 1314–1324, 2019.

[5] Yao Lu, G. Lu, R. Lin, Jinxing Li, and D. Zhang. Srgc-nets: Sparse repeated group convolutional neural networks. *IEEE Transactions on Neural Networks and Learning Systems*, 31:2889–2902, 2020.

[6] Jiaxiang Wu, Cong Leng, Yuhang Wang, Qinghao Hu, and Jian Cheng. Quantized convolutional neural networks for mobile devices. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 4820–4828, 2016.

[7] Hrushikesh N Mhaskar and Tomaso Poggio. Deep vs. shallow networks: An approximation theory perspective. *Analysis and Applications*, 14(06):829–848, 2016.

[8] A. Brutzkus, A. Globerson, E. Malach, and S. Shalev-Shwartz. Sgd learns over-parameterized networks that provably generalize on linearly separable data. 2018.

[9] Yann LeCun, John S Denker, and Sara A Solla. Optimal brain damage. In *Advances in neural information processing systems*, pages 598–605, 1990.

[10] Durk P Kingma, Tim Salimans, and Max Welling. Variational dropout and the local reparameterization trick. In *Advances in Neural Information Processing Systems*, pages 2575–2583, 2015.

[11] Enzo Tartaglione, Skjalg Lepsøy, Attilio Fiandrotti, and Gianluca Francini. Learning sparse neural networks via sensitivity-driven regularization. In *Advances in Neural Information Processing Systems*, pages 3878–3888, 2018.

[12] Song Han, Jeff Pool, John Tran, and William Dally. Learning both weights and connections for efficient neural network. In *Advances in neural information processing systems*, pages 1135–1143, 2015.

[13] J. Frankle and M. Carbin. The lottery ticket hypothesis: Finding sparse, trainable neural networks. 2019.

[14] Enzo Tartaglione, Daniele Perlo, and Marco Grangetto. Post-synaptic potential regularization has potential. In *International Conference on Artificial Neural Networks*, pages 187–200. Springer, 2019.

[15] Song Han, Jeff Pool, John Tran, and William Dally. Learning both weights and connections for efficient neural network. In *Advances in Neural Information Processing Systems*, pages 1135–1143, 2015.

[16] Michael C Mozer and Paul Smolensky. Skeletonization: A technique for trimming the fat from a network via relevance assessment. In *Advances in neural information processing systems*, pages 107–115, 1989.

[17] Wei Wen, Chunpeng Wu, Yandan Wang, Yiran Chen, and Hai Li. Learning structured sparsity in deep neural networks. In *Advances in Neural Information Processing Systems*, pages 2074–2082, 2016.

[18] Christos Louizos, Max Welling, and Diederik P Kingma. Learning sparse neural networks through *l*_0 regularization. *arXiv preprint arXiv:1712.01312*, 2017.

[19] Enzo Tartaglione, Skjalg Lepsøy, Attilio Fiandrotti, and Gianluca Francini. Learning sparse neural networks via sensitivity-driven regularization. In *Advances in Neural Information Processing Systems*, pages 3878–3888, 2018.

[20] Dmitry Molchanov, Arsenii Ashukha, and Dmitry 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.

[21] Aidan N. Gomez, Ivan Zhang, Kevin Swersky, Yarin Gal, and Geoffrey E. Hinton. Learning sparse networks using targeted dropout. *CoRR*, abs/1905.13678, 2019.

[22] Xia Xiao, Zigeng Wang, and Sanguthevar Rajasekaran. Autoprune: Automatic network pruning by regularizing auxiliary parameters. *Advances in neural information processing systems*, 32, 2019.

[23] Jonathan Frankle, Gintare Karolina Dziugaite, Daniel Roy, and Michael Carbin. Linear mode connectivity and the lottery ticket hypothesis. In *International Conference on Machine Learning*, pages 3259–3269. PMLR, 2020.

[24] Emma Strubell, Ananya Ganesh, and Andrew McCallum. Energy and policy considerations for deep learning in NLP. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pages 3645–3650, Florence, Italy, July 2019. Association for Computational Linguistics.

[25] N. Lee, Thalaiyasingam Ajanthan, and P. Torr. Snip: Single-shot network pruning based on connection sensitivity. *ArXiv*, abs/1810.02340, 2019.

[26] Chaoqi Wang, Guodong Zhang, and Roger Grosse. Picking winning tickets before training by preserving gradient flow. In *International Conference on Learning Representations*, 2020.

[27] Hidenori Tanaka, Daniel Kunin, Daniel L Yamins, and Surya Ganguli. Pruning neural networks without any data by iteratively conserving synaptic flow. *Advances in Neural Information Processing Systems*, 33:6377–6389, 2020.

[28] Steven Dalton, Nathan Bell, Luke Olson, and Michael Garland. Cusp: Generic parallel algorithms for sparse matrix and graph computations.

[29] Steven Dalton, Nathan Bell, Luke Olson, and Michael Garland. Basic linear algebra for sparse matrices on nvidia gpus.

[30] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278 – 2324, November 1998.

[31] Han Xiao, Kashif Rasul, and Roland Vollgraf. Fashion-mnist: a novel image dataset for benchmarking machine learning algorithms. *CoRR*, abs/1708.07747, 2017.

[32] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. *CoRR*, abs/1512.03385, 2015.

[33] D. Molchanov, A. Ashukha, and D. Vetrov. Variational dropout sparsifies deep neural networks. volume 5, pages 3854–3863, 2017.

[34] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. U-net: Convolutional networks for biomedical image segmentation. In *International Conference on Medical image computing and computer-assisted intervention*, pages 234–241. Springer, 2015.

[35] Noel Codella, Veronica Rotemberg, Philipp Tschandl, M Emre Celebi, Stephen Dusza, David Gutman, Brian Helba, Aadi Kalloo, Konstantinos Liopyris, Michael Marchetti, et al. Skin lesion analysis toward melanoma detection 2018: A challenge hosted by the international skin imaging collaboration (isic). *arXiv preprint arXiv:1902.03368*, 2019.

[36] Philipp Tschandl, Cliff Rosendahl, and Harald Kittler. The ham10000 dataset, a large collection of multi-source dermatoscopic images of common pigmented skin lesions. *Scientific data*, 5(1):1–9, 2018.

[37] Junjie Bai, Fang Lu, Ke Zhang, et al. Onnx: Open neural network exchange. https://github.com/onnx/onnx, 2019.

[38] Igor Pavlov. Lzma sdk (software development kit), 2007.

[39] K. Ullrich, M. Welling, and E. Meeds. Soft weight-sharing for neural network compression. *5th International Conference on Learning Representations, ICLR 2017 - Conference Track Proceedings*, 2019.

[40] Yiwen Guo, Anbang Yao, and Yurong Chen. Dynamic network surgery for efficient dnns. *Advances in Neural Information Processing Systems*, pages 1387–1395, 2016.

[41] Yann LeCun, Léon Bottou, Yoshua Bengio, Patrick Haffner, et al. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[42] Han Xiao, Kashif Rasul, and Roland Vollgraf. Fashion-mnist: a novel image dataset for benchmarking machine learning algorithms. *CoRR*, abs/1708.07747, 2017.

[43] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *3rd International Conference on Learning Representations, ICLR 2015 - Conference Track Proceedings*, 2015.

[44] Davis Blalock, Jose Javier Gonzalez Ortiz, Jonathan Frankle, and John Guttag. What is the state of neural network pruning? In I. Dhillon, D. Papailiopoulos, and V. Sze, editors, *Proceedings of Machine Learning and Systems*, volume 2, pages 129–146, 2020.

[45] Liqiang Lu, Jiaming Xie, Ruirui Huang, Jiansong Zhang, Wei Lin, and Yun Liang. An efficient hardware accelerator for sparse convolutional neural networks on fpgas. In *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 17–25. IEEE, 2019.

[46] PyTorch. PyTorch sparse tensor, 2021.

[47] Xuda Zhou, Zidong Du, Shijin Zhang, Lei Zhang, Huiying Lan, Shaoli Liu, Ling Li, Qi Guo, Tianshi Chen, and Yunji Chen. Addressing sparsity in deep neural networks. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 38(10):1858–1871, 2019.

[48] Chaoyang Zhu, Kejie Huang, Shuyuan Yang, Ziqi Zhu, Hejia Zhang, and Haibin Shen. An efficient hardware accelerator for structured sparse convolutional neural networks on fpgas. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 28(9):1953–1965, 2020.

[49] PyTorch issue. Sparse filter layers (more specifically convolutions), 2020.

[50] PyTorch issue. Module prune has no effect on speed and memory consumption, 2020.

[51] PyTorch issue. Pruning doesn't affect speed nor memory usage, 2020.

[52] PyTorch issue. Effectiveness of pruning, 2020.

[53] Gao Huang, Zhuang Liu, Laurens Van Der Maaten, and Kilian Q. Weinberger. Densely connected convolutional networks. In *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 2261–2269, 2017.

[54] Yihui He, Xiangyu Zhang, and Jian Sun. Channel pruning for accelerating very deep neural networks. In *2017 IEEE International Conference on Computer Vision (ICCV)*, pages 1398–1406, 2017.

[55] Alexey Kruglov. Channel-wise pruning of neural networks with tapering resource constraint. *arXiv preprint arXiv:1812.07060*, 2018.

[56] Hengyuan Hu, Rui Peng, Yu-Wing Tai, and Chi-Keung Tang. Network trimming: A data-driven neuron pruning approach towards efficient deep architectures. *arXiv preprint arXiv:1607.03250*, 2016.

[57] Alex Krizhevsky. One weird trick for parallelizing convolutional neural networks. *arXiv preprint arXiv:1404.5997*, 2014.

[58] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 1–9, 2015.

[59] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jon Shlens, and Zbigniew Wojna. Rethinking the inception architecture for computer vision. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 2818–2826, 2016.

[60] Mingxing Tan, Bo Chen, Ruoming Pang, Vijay Vasudevan, Mark Sandler, Andrew Howard, and Quoc V Le. Mnasnet: Platform-aware neural architecture search for mobile. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 2820–2828, 2019.

[61] Saining Xie, Ross Girshick, Piotr Dollár, Zhuowen Tu, and Kaiming He. Aggregated residual transformations for deep neural networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1492–1500, 2017.

[62] Ningning Ma, Xiangyu Zhang, Hai-Tao Zheng, and Jian Sun. Shufflenet v2: Practical guidelines for efficient cnn architecture design. In Vittorio Ferrari, Martial Hebert, Cristian Sminchisescu, and Yair Weiss, editors, *Computer Vision – ECCV 2018*, pages 122–138, Cham, 2018. Springer International Publishing.

[63] Forrest N Iandola, Song Han, Matthew W Moskewicz, Khalid Ashraf, William J Dally, and Kurt Keutzer. Squeezenet: Alexnet-level accuracy with 50x fewer parameters and < 0.5 mb model size. *arXiv preprint arXiv:1602.07360*, 2016.

[64] Sergey Zagoruyko and Nikos Komodakis. Wide residual networks. pages 87.1–87.12, September 2016.

[65] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, et al. Imagenet large scale visual recognition challenge. *International journal of computer vision*, 115(3):211–252, 2015.

[66] The Motion Picture Expert Group. ISO/IEC DIS 15938-17 Compression of neural networks for multimedia content description and analysis. 2021.

[67] David Neumann, Felix Sattler, Heiner Kirchhoffer, Simon Wiedemann, Karsten Müller, Heiko Schwarz, Thomas Wiegand, Detlev Marpe, and Wojciech Samek. Deepcabac: Plug & play compression of neural network weights and weight updates. In *2020 IEEE International Conference on Image Processing (ICIP)*, pages 21–25. IEEE, 2020.

[68] Simon Wiedemann, Heiner Kirchhoffer, Stefan Matlage, Paul Haase, Arturo Marban, Talmaj Marinč, David Neumann, Tung Nguyen, Heiko Schwarz, Thomas Wiegand, et al. Deepcabac: A universal compression algorithm for deep neural networks. *IEEE Journal of Selected Topics in Signal Processing*, 14(4):700–714, 2020.

[69] Michele Cancilla, Laura Canalini, Federico Bolelli, Stefano Allegretti, Salvador Carrión, Roberto Paredes, Jon A. Gómez, Simone Leo, Marco Enrico Piras, Luca Pireddu, Asaf Badouh, Santiago Marco-Sola, Lluc Alvarez, Miquel Moreto, and Costantino Grana. The deephealth toolkit: A unified framework to boost biomedical applications. In *2020 25th International Conference on Pattern Recognition (ICPR)*, pages 9881–9888, 2021.

[70] Andrew Lavin and Scott Gray. Fast algorithms for convolutional neural networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 4013–4021, 2016.

[71] Francois Chollet. Xception: Deep learning with depthwise separable convolutions. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, July 2017.

[72] Junjie Bai, Fang Lu, Ke Zhang, et al. Onnx: Open neural network exchange. https://github.com/onnx/onnx, 2019.

[73] Noel Codella, Veronica Rotemberg, Philipp Tschandl, M Emre Celebi, Stephen Dusza, David Gutman, Brian Helba, Aadi Kalloo, Konstantinos Liopyris, Michael Marchetti, et al. Skin lesion analysis toward melanoma detection 2018: A challenge hosted by the international skin imaging collaboration (isic). *arXiv preprint arXiv:1902.03368*, 2019.

[74] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[75] Vijay Badrinarayanan, Alex Kendall, and Roberto Cipolla. Segnet: A deep convolutional encoder-decoder architecture for image segmentation. *IEEE Trans. Pattern Anal. Mach. Intell.*, 39(12):2481–2495, 2017.

[76] Amir Gholami, Sehoon Kim, Zhen Dong, Zhewei Yao, Michael W. Mahoney, and Kurt Keutzer. A survey of quantization methods for efficient neural network inference, 2021.

[77] https://github.com/CEA-LIST/N2D2, 2019.

[78] Markus Nagel, Mart van Baalen, Tijmen Blankevoort, and Max Welling. Data-free quantization through weight equalization and bias correction, 2019.

[79] Léon Bottou. Large-scale machine learning with stochastic gradient descent. In *Proceedings of COMPSTAT'2010*, pages 177–186. Springer, 2010.

[80] Martin Zinkevich, Markus Weimer, Lihong Li, and Alex J Smola. Parallelized stochastic gradient descent. In *Advances in neural information processing systems*, pages 2595–2603, 2010.

[81] Ian J Goodfellow, Oriol Vinyals, and Andrew M Saxe. Qualitatively characterizing neural network optimization problems. *International Conference on Learning Representations, ICLR 2015*, 2015.

[82] Felix Draxler, Kambis Veschgini, Manfred Salmhofer, and Fred Hamprecht. Essentially no barriers in neural network energy landscape. In Jennifer Dy and Andreas Krause, editors, *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*, pages 1309–1318, Stockholmsmässan, Stockholm Sweden, 10–15 Jul 2018. PMLR.

[83] Enzo Tartaglione and Marco Grangetto. Take a ramble into solution spaces for classification problems in neural networks. In *International Conference on Image Analysis and Processing*, pages 345–355. Springer, 2019.

[84] Nitish Shirish Keskar, Dheevatsa Mudigere, Jorge Nocedal, Mikhail Smelyanskiy, and Ping Tak Peter Tang. On large-batch training for deep learning: Generalization gap and sharp minima. *arXiv preprint arXiv:1609.04836*, 2016.

[85] Pratik Chaudhari, Anna Choromanska, Stefano Soatto, Yann LeCun, Carlo Baldassi, Christian Borgs, Jennifer Chayes, Levent Sagun, and Riccardo Zecchina. Entropy-sgd: Biasing gradient descent into wide valleys. *International Conference on Learning Representations, ICLR 2017*, 2017.

[86] Zhuang Liu, Mingjie Sun, Tinghui Zhou, Gao Huang, and Trevor Darrell. Rethinking the value of network pruning. *International Conference on Learning Representations, ICLR 2019*, 2019.

[87] Noah Golmant, Zhewei Yao, Amir Gholami, Michael Mahoney, and Joseph Gonzalez. pytorch-hessian-eigentings: efficient pytorch hessian eigendecomposition, October 2018.

[88] Enzo Tartaglione. The rise of the lottery heroes: why zero-shot pruning is hard. *arXiv preprint arXiv:2202.12400*, 2022.

[89] Ari Morcos, Haonan Yu, Michela Paganini, and Yuandong Tian. One ticket to win them all: generalizing lottery ticket initializations across datasets and optimizers. *Advances in neural information processing systems*, 32, 2019.

[90] Eran Malach, Gilad Yehudai, Shai Shalev-Schwartz, and Ohad Shamir. Proving the lottery ticket hypothesis: Pruning is all you need. In Hal Daumé III and Aarti Singh, editors, *Proceedings of the 37th International Conference on Machine Learning*, volume 119 of *Proceedings of Machine Learning Research*, pages 6682–6691. PMLR, 13–18 Jul 2020.

[91] Haoran You, Chaojian Li, Pengfei Xu, Yonggan Fu, Yue Wang, Xiaohan Chen, Richard G Baraniuk, Zhangyang Wang, and Yingyan Lin. Drawing early-bird tickets: Towards more efficient training of deep networks. *arXiv preprint arXiv:1909.11957*, 2019.

[92] Zhenyu Zhang, Xuxi Chen, Tianlong Chen, and Zhangyang Wang. Efficient lottery ticket finding: Less data is more. In *International Conference on Machine Learning*, pages 12380–12390. PMLR, 2021.

[93] Tianlong Chen, Xuxi Chen, Xiaolong Ma, Yanzhi Wang, and Zhangyang Wang. Coarsening the granularity: Towards structurally sparse lottery tickets. *arXiv preprint arXiv:2202.04736*, 2022.

[94] Utku Evci, Fabian Pedregosa, Aidan Gomez, and Erich Elsen. The difficulty of training sparse neural networks. *arXiv preprint arXiv:1906.10732*, 2019.

[95] Jonathan Frankle, Gintare Karolina Dziugaite, Daniel Roy, and Michael Carbin. Pruning neural networks at initialization: Why are we missing the mark? In *International Conference on Learning Representations*, 2021.

[96] Volker Tresp, Ralph Neuneier, and Hans-Georg Zimmermann. Early brain damage. *Advances in neural information processing systems*, 9, 1996.

[97] Yang He, Guoliang Kang, Xuanyi Dong, Yanwei Fu, and Yi Yang. Soft filter pruning for accelerating deep convolutional neural networks. *arXiv preprint arXiv:1808.06866*, 2018.

[98] Alex Krizhevsky, Vinod Nair, and Geoffrey Hinton. Cifar-10 (canadian institute for advanced research).

[99] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[100] Ze Liu, Yutong Lin, Yue Cao, Han Hu, Yixuan Wei, Zheng Zhang, Stephen Lin, and Baining Guo. Swin transformer: Hierarchical vision transformer using shifted windows. In *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*, 2021.

[101] Liang-Chieh Chen, George Papandreou, Florian Schroff, and Hartwig Adam. Rethinking atrous convolution for semantic image segmentation. *arXiv preprint arXiv:1706.05587*, 2017.

[102] Tsung-Yi Lin, Michael Maire, Serge Belongie, James Hays, Pietro Perona, Deva Ramanan, Piotr Dollár, and C Lawrence Zitnick. Microsoft coco: Common objects in context. In *European conference on computer vision*, pages 740–755. Springer, 2014.