arXiv:2102.00554v1 [cs.LG] 31 Jan 2021

Sparsity in Deep Learning: Pruning and growth for efficient inference and training in neural networks

TORSTEN HOEFLER, ETH Zürich, Switzerland
DAN ALISTARH, IST Austria, Austria
TAL BEN-NUN, ETH Zürich, Switzerland
NIKOLI DRYDEN, ETH Zürich, Switzerland
ALEXANDRA PESTE, IST Austria, Austria

The growing energy and performance costs of deep learning have driven the community to reduce the size of neural networks by selectively pruning components. Similarly to their biological counterparts, sparse networks generalize just as well, if not better than, the original dense networks. Sparsity can reduce the memory footprint of regular networks to fit mobile devices, as well as shorten training time for ever growing networks. In this paper, we survey prior work on sparsity in deep learning and provide an extensive tutorial of sparsification for both inference and training. We describe approaches to remove and add elements of neural networks, different training strategies to achieve model sparsity, and mechanisms to exploit sparsity in practice. Our work distills ideas from more than 300 research papers and provides guidance to practitioners who wish to utilize sparsity today, as well as to researchers whose goal is to push the frontier forward. We include the necessary background on mathematical methods in sparsification, describe phenomena such as early structure adaptation, the intricate relations between sparsity and the training process, and show techniques for achieving acceleration on real hardware. We also define a metric of pruned parameter efficiency that could serve as a baseline for comparison of different sparse networks. We close by speculating on how sparsity can improve future workloads and outline major open problems in the field.

The supreme goal of all theory is to make the irreducible basic elements as simple and as few as possible without having to surrender the adequate representation of a single datum of experience -

Albert Einstein, 1933

1 INTRODUCTION

Deep learning shows unparalleled promise for solving very complex real-world problems in areas such as computer vision, natural language processing, knowledge representation, recommendation systems, drug discovery, and many more. With this development, the field of machine learning is moving from traditional feature engineering to neural architecture engineering. However, still little is known about how to pick the right architecture to solve a specific task. Several methods such as translational equivariance in convolutional layers, recurrence, structured weight sharing, pooling, or locality are used to introduce strong inductive biases in the model design. Yet, the exact model size and capacity required for a task remain unknown and a common strategy is to train overparameterized models and compress them into smaller representations.

Biological brains, especially the human brain, are hierarchical, sparse, and recurrent structures [Friston 2008] and one can draw some similarities with the inductive biases in today's artificial neural networks. Sparsity plays an important role in scaling biological brains—the more

Authors' addresses: Torsten Hoefler, htor@inf.ethz.ch, ETH Zürich, Zürich, Switzerland, 8092; Dan Alistarh, dan.alistarh@ist.ac.at, IST Austria, Klosterneuburg, Austria, 3400; Tal Ben-Nun, talbn@inf.ethz.ch, ETH Zürich, Zürich, Switzerland, 8092; Nikoli Dryden, ndryden@ethz.ch, ETH Zürich, Zürich, Switzerland, 8092; Alexandra Peste, alexandra.peste@ist.ac.at, IST Austria, Klosterneuburg, Austria, 3400.

neurons a brain has, the sparser it gets [Herculano-Houzel et al. 2010]. Furthermore, research has shown that a human brain starts sparse, has an early phase of densification followed by massive pruning, and then remains at a relatively stable sparsity level. Yet, even fully-grown brains change up to 40% of their synapses each day [Hawkins 2017]. Many of today's engineered pruning techniques have intuitive biological analogies, which we will mention throughout the text and discuss in Section 8. Yet, the computational substrates (biological tissue vs. CMOS) result in very different constraints.

Artificial deep learning models are traditionally dense and over-parameterized, sometimes to the extent that they can memorize random patterns in data [Zhang et al. 2017] or that 95% of the parameters can be predicted from the remaining 5% [Denil et al. 2014]. This may be linked to empirical evidence suggesting that over-parameterized models are easier to train with stochastic gradient descent (SGD) than more compact representations [Glorot et al. 2011a; Kaplan et al. 2020; Li et al. 2020a; Mhaskar and Poggio 2016]. Brutzkus et al. [2017] and Du et al. [2019] show that such gradient descent techniques provably train (shallow) over-parameterized networks optimally with good generalization. Specifically, they show that over-parameterization leads to a strong "convexity-like property" that benefits the convergence of gradient descent. Recent theoretical results [Allen-Zhu et al. 2019; Neyshabur et al. 2018] seem to support these findings and indicate that training dynamics and generalization rely on overparameterization.

This over-parameterization comes at the cost of additional memory and computation effort during model training and inference. In particular, for inference on mobile and battery-driven devices and in cost-conscious settings, sparse model representations promise huge savings. Concretely, sparse models are easier to store, and often lead to computational savings. Furthermore, overparameterized models tend to overfit to the data and degrade generalization to unseen examples. Following Occam's razor, sparsification can also be seen as some form of regularization, and may improve model quality by effectively reducing noise in the model. Specifically, the framework of Minimum Description Length provides an attractive formulation with a Bayesian interpretation and a clear interpretation as data compression [Grünwald 2007], as we discuss later.

Many, especially older, works centered on improved generalization through sparsification. Early research [Mozer and Smolensky 1988] focused on models with tens to hundreds of parameters also describe better interpretability of their sparsified versions. However, with today's models using millions or billions of parameters, it is to be seen if sparsity improves explainability and interpretability significantly. The recent work of Bartoldson et al. [2020] models pruning as "noise" similar to dropout or data augmentation to explain generalization. Other recent works found that sparsity can improve robustness against adversarial attacks [Cosentino et al. 2019; Gopalakrishnan et al. 2018; Guo et al. 2018; Madaan et al. 2020; Rakin et al. 2020; Sehwag et al. 2020; Verdenius et al. 2020].

A larger group of works recently focused on improving the computational efficiency while maintaining the model accuracy. Modern networks are computationally expensive to use — for example, Inception-V3 [Szegedy et al. 2016], a state of the art object recognition network, requires 5.7 billion arithmetic operations and 27 million parameters to be evaluated; and GPT-3 [Brown et al. 2020], an experimental state of the art natural language processing network requires 175 billion parameters (350 GiB assuming 16 bits per parameter) to be evaluated. Furthermore, training such deep neural models becomes increasingly expensive and the largest language models already require supercomputers for training, potentially costing millions of dollars per training run [Brown et al. 2020]. Thus, it is important to investigate sparsity during the training process to manage the costs of training.

The results we survey show that *today's sparsification methods can lead to a 10-100x reduction in model size, and to corresponding theoretical gains in computational, storage, and energy efficiency,*

all without significant loss of accuracy. If those speedups are realized in efficient hardware implementations, then the gained performance may lead to a phase change in enabling more complex and possibly revolutionary tasks to be solved practically. Furthermore, we observe that the pace of progress in sparsification methods is accelerating, such that even during the last months while we worked on this report, several new methods that improve upon the state of the art have been published.

We aim to provide an overview of the key techniques and ideas, while covering some of the necessary mathematical background. Due to space constraints, we keep our descriptions brief—we always refer the interested reader to the original papers which describe the ideas in full detail. We structure the discussion along various axes: which elements of a neural network are sparsified, when are they sparsified, and how can they be sparsified. Furthermore, we consider sparse training and the need to re-add connections during training to maintain a constant model complexity after sparsification. We also outline the development of results in various areas of sparsification.

In general, the flurry of different techniques, tasks, models, and evaluation settings causes a wide spread in the community. This leads to many incomparable results and makes it hard to determine the state of the art and whether method A is better than method B. Furthermore, we found that nearly every basic approach has been invented at least twice. Blalock et al. [2020] also point at these problems and they propose a common benchmark and methodology to go forward. We aim at summarizing the existing techniques, and first focus on purely qualitative aspects of designing models in Sections 2-5. Then, in Sections 6 and 7, we explain a selection of architectures implementing combinations of those designs including performance results. Sections 8-10 provide a general discussion, list open problems, and conclude the overview.

1.1 Overview of Model Compression Techniques

We first present the landscape of approaches to compress models in order to improve computational and memory efficiency. We differentiate between six main techniques:

- **Down-sizing models** creates smaller dense networks to solve the same task. Model distillation [Hinton et al. 2015] or Neural Architecture Search [Elsken et al. 2019] are typical examples of techniques to find small dense models.
- **Operator factorization** decomposes operators, for example the matrix multiplication of dense layers, into smaller operators. For matrices, operators can be decomposed via singular value decomposition [Sainath et al. 2013], while more general tensors can be decomposed via tensor train decomposition [Kanjilal et al. 1993; Zhao et al. 2017].
- **Value quantization** seeks to find a good low-precision encoding for values in the networks, such as weights, activations, or gradients. Various floating point and integer formats can be used to encode data efficiently leading to a smaller number of bits than standard 32 or 64 bit datatypes.
- **Value Compression** can be used to compress model structures and values (e.g., weights) either with generic entropy-based methods [Han et al. 2016b] or loss-bounded type-specific methods using correlation across values [Jin et al. 2019].
- **Parameter sharing** can lead to model compression by exploiting redundancy in the parameter space. Such redundancy can also be fostered during the training process [Plummer et al. 2020].
- **Sparsification** can lead to more efficient models that continue to operate in high-dimensional feature spaces but reduce the representational complexity using only a subset of the dimensions at a time. Practically, such methods can reduce complexity by zeroing out subsets of the model parameters.

All of these methods lead to reduced memory requirements and all schemes, except for parameter sharing, can also reduce the computational complexity. These schemes can be combined into an efficient inference and training approach and various surveys cover subsets of this space in detail [Cheng et al. 2020; Choudhary et al. 2020; Deng et al. 2020]. In this paper, we focus on the most complex and, in our view, most powerful of those techniques: sparsification, also known as "pruning" in some contexts. Reed [1993] provides an overview of early sparsification techniques until 1993—since then, the literature has evolved significantly. A second "AI winter" in the late 1980s and early 1990s appears to have significantly reduced interest in and funding for artificial intelligence research and development [Russell and Norvig 2020, Sec. 1.3], [Nilsson 2009, Sec. 24.4], and activity in neural networks subsequently waned for nearly two decades. Deep learning (re-)started its success story around 2012 with convolutional neural networks for image recognition. Since then, more than 266 papers, comprising 4,089 pages focusing on ideas and techniques for sparsity in deep networks appeared, which we categorize and summarize below. We aim to provide an intuitive and comprehensive overview of the most important ideas. Yet, at a compression rate of 97.9% and more than 420 citations, we almost surely miss specific ideas or works.

Fig. 1 shows the volume of scientific publications on various aspects of sparsity over the last three decades. The first papers in the late 80's and 90's focus on very small models and their generalization and interpretability properties. The whole field of neural networks was rather inactive during the early 2000's until the breakthroughs in image recognition circa 2012, followed by a resurgence of interest in optimization of sparse networks. During the late 2010's, numerous accelerators and optimization techniques were designed to specifically aim at optimizing sparse deep neural networks. The meaning of the labels will be clarified later in this paper.

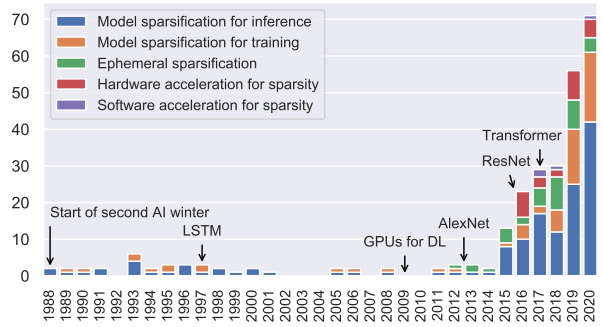


Fig. 1. Literature development over the years.

One of the main drivers behind the massive progress in deep learning between the 90's and today was the nearly 1 million times increase in computational capability delivered by Moore's law, Dennard scaling, and architectural specializations with GPUs and specialized machine learning accelerators. With the ending of those scaling laws and specialization opportunities, these developments will hit their natural limits and progress may stall. *We see sparsity as potentially achieving a second significant "jump" in computational capability as, even with current methods, it has the promise to increase computational and storage efficiency by up to two orders of magnitude.*

1.1.1 Document structure. We aim to provide a comprehensive overview to a diverse set of readers. Section 1.2 introduces all mathematical background for the different sparsification approaches and can be skipped by experienced readers as well as readers that are mostly looking

for intuition. Section 2 provides an executive summary of how pruning schemes work. Sections 3 and 4 dive deeply into different schemes for removal and growth (weight addition) during training and pruning while Section 5 describes details of various ephemeral (per example) sparsification schemes. We consider examples of pruning for full convolutional and transformer architectures in Section 6. Section 7 overviews various approaches for improving the performance of sparse models, ranging from software to specialized hardware implementations. In Section 8, we summarize and extrapolate the most significant observations in the field and we provide ten research challenges in Section 9.

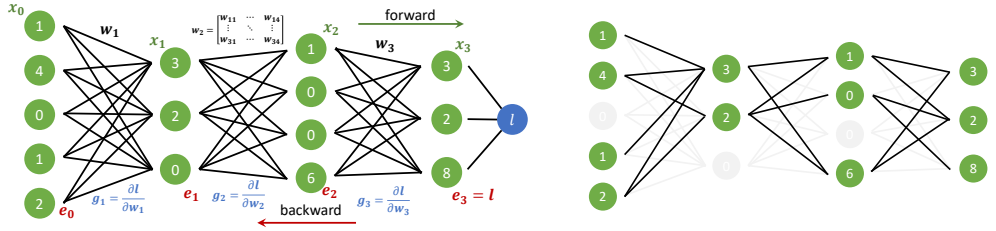
If your goal is to get a quick executive overview of the field, then we recommend studying Sections 2 and 8 while skimming Sections 3, 4, 5, and 7, especially the overview figures and tables therein. If your main interest lies in the hardware engineering aspects, then we recommend to at least get the executive overview mentioned before and study Section 7 in detail. Similarly, if you are a neural network architect looking for sparsification best practices, we recommend the executive overview in combination with details in Section 6 and the references therein. Researchers in the field may want to examine the whole document carefully to get a deep overview of all aspects and focus efforts especially on the challenging problems in Section 9. Finally, readers can get a view of each section from the first 1–2 paragraphs to decide whether to dive deeper into each subject.

1.2 Background and Notation

We start by providing some background on deep learning inference and training to introduce our notation. Experienced readers may wish to skip to the next section. Deep learning models (or "networks") consist of a graph of parameterizable layers (or "operators") that together implement a complex nonlinear function f. We consider a general supervised learning setting, where we are given a training set, comprised of pairs of input examples $\mathbf{x} \in \mathcal{X}$ and outputs $\mathbf{y} \in \mathcal{Y}$. The goal is to learn the function $f : \mathcal{X} \mapsto \mathcal{Y}$, parameterized by weights $\mathbf{w} \in \mathbb{R}^d$, such that given input $\mathbf{x}$, the prediction $f(\mathbf{x}; \mathbf{w})$ is close to $\mathbf{y}$. We usually assume that $\mathcal{X}$ represents a vector of features describing an element drawn from a true input distribution $\mathcal{D}$ that captures the characteristics of typical inputs but cannot be measured or described concisely (e.g., cat pictures). Applying the function $f(\mathbf{x}; \mathbf{w})$ is performed by transforming the input $\mathbf{x}$ layer by layer to generate the output - this process is called *inference*, or in a training setting the *forward pass*.

The process of finding a network to solve a specific task can be decomposed into two phases: (1) *design* or *engineer* the network structure, and (2) *train* the network's weights. The network structure is traditionally designed manually and not changed during the training process. Training iterations start with a forward pass, which is similar to inference but stores the inputs of each layer. The quality of the result $f(\mathbf{x}; \mathbf{w})$ of the forward pass is evaluated using a loss function $\ell : \mathcal{Y} \times \mathcal{Y} \mapsto \mathbb{R}$ to estimate the accuracy of the prediction, $\ell\big(\mathbf{y}, f\,(\mathbf{x}; \mathbf{w})\,\big)$, where $(\mathbf{x}, \mathbf{y})$ is the sample pair. Many loss functions are known, such as the L_2 distance or the cross-entropy between the predicted output $f\,(\mathbf{x}; \mathbf{w})$ and the expected one $\mathbf{y}$. The following backward pass propagates the loss ("*error*") from the last layer in the reverse direction. At each learnable (parametric) layer, the backward pass uses the adjoint of the forward operation to compute a gradient g and update the parameters ("*weights*") using a *learning rule* to decrease ℓ (for the current example pair). This method is repeated iteratively for many different examples drawn from $\mathcal{D}$ until the function $f(\mathbf{x}; \mathbf{w})$ provides the desired accuracy. This accuracy is typically evaluated on a separate set of examples that were not used to train the model in order to measure the *generalization* capabilities of the model to unseen examples drawn from $\mathcal{D}$.

We now introduce some further notation and mathematical background, which will be useful to understand some of the following pruning schemes. Parts of this section follow the notation and general approach of [Molchanov et al. 2017; Singh and Alistarh 2020].



(a) Dense MLP training example

(b) Sparsified MLP

Fig. 2. Training, Inference, and Sparsification examples.

Let us consider the simple case of a multilayer perceptron shown in Fig. 2a with the typical input layer $\mathbf{x_0}$, two hidden layers $\mathbf{x_1}, \mathbf{x_2}$, and output layer $\mathbf{x_3}$, with rectified linear units (ReLU) $\sigma_R(x) := \max(0, x)$ as *activation* functions. We denote the number of neurons in layer i as $|\mathbf{x_i}|$. The forward pass can be written as a series of matrix-vector products $f(\mathbf{x_0}; \mathbf{w}) = \sigma_R(\mathbf{w_3} \cdot \sigma_R(\mathbf{w_2} \cdot \sigma_R(\mathbf{w_1 x_0} + \mathbf{b_1}) + \mathbf{b_2}) + \mathbf{b_3})$, where $\mathbf{x_0}$ is the input ("feature") vector. Here, the network function $f(\mathbf{x_0}; \mathbf{w})$ is parameterized by weight matrices $\mathbf{w_1}$ with dimensions $|\mathbf{x_0}| \times |\mathbf{x_1}|$, $\mathbf{w_2}$ with dimensions $|\mathbf{x_1}| \times |\mathbf{x_2}|$, and $\mathbf{w_3}$ with dimensions $|\mathbf{x_2}| \times |\mathbf{x_3}|$; and bias vectors $\mathbf{b_i}$ with dimensions $|\mathbf{x_i}|$ for layer i (we usually omit biases for brevity). Subscripts identify the layer—we omit them for equations that apply to all layers. Sometimes, we treat the concatenation of all weight matrices as a vector—this will be obvious from the context. It is already apparent that the $O(|\mathbf{x_i}| \cdot |\mathbf{x_{i+1}}|)$ storage and compute may overparameterize the model for a large number of neurons.

Fig. 2b shows a sparsified version of Fig. 2a. It shows that the third input feature and all its adjacent weights are removed (grayed out). Furthermore, two hidden neurons and their weights as well as various other weights have been removed. Removing neurons or input features corresponds to removing rows or columns in the layer weight matrices while single weights remove elements of the matrices.

1.2.1 The Deterministic Formulation for Training. Training a deep neural network minimizes a loss. In the deterministic case, the (empirical) training loss L is defined as the average loss over training examples, i.e., $L(\mathbf{w}) = \frac{1}{N} \sum_{n=1}^{N} \ell(\mathbf{y}[n], f(\mathbf{x}[n]; \mathbf{w}))$. In the following, we fix $d \geq 1$ to be the total number of parameters in the model, and we will omit the indexing by sample when clear from context. The loss function L will be a d-dimensional loss function over the model parameters $L : \mathbb{R}^d \to \mathbb{R}$.

The most common training scheme is stochastic gradient descent (SGD), which is based on a first-order approximation to the loss function L. This method utilizes automatic differentiation (AD) to compute the derivative ("gradient") of the loss with respect to the weights in a layer $\mathbf{g_1} = \frac{\partial L}{\partial \mathbf{w_1}}$ and $\mathbf{g_2} = \frac{\partial L}{\partial \mathbf{w_2}}$ at the specific example $\mathbf{x}$. Reverse mode (aka. adjoint) AD stores the intermediate results of the forward pass and applies the loss function L that returns an error ("distance") with respect to the desired model output. This can be done by applying the chain rule to the compound function $f(\mathbf{x}; \mathbf{w})$ and propagating the error backward through all operators. For example, the gradient of the second layer is $\mathbf{g_2} = \frac{\partial L}{\partial \mathbf{w_2}} = \frac{\partial L}{\partial \mathbf{e_2}} \frac{\partial \mathbf{e_2}}{\partial \mathbf{w_2}}$. The gradients are then used with a learning rule function R to update the weights for the next iteration: $\mathbf{w}^{(i+1)} = \mathbf{w}^{(i)} + R(\mathbf{g}, \mathbf{w}^{(i)})$.

The Jacobian matrix. The Jacobian of an arbitrary function $F : \mathbb{R}^d \to \mathbb{R}^m$ is the matrix of first-order partial derivatives of a vector-valued function with respect to its inputs. For example, the Jacobian matrix for the loss function $L : \mathbb{R}^d \to \mathbb{R}$ with respect to the weights is a $1 \times d$ matrix of

partial derivatives with respect to each individual weight. If we write w_1 for the first individual weight and $\mathbf{w_1}$ for the set of weights in the first layer (similarly for gradients), then the Jacobian matrix is defined as $\mathbf{J} = \nabla_{\mathbf{w}} L = \left[\frac{\partial L}{\partial w_1} \frac{\partial L}{\partial w_2} \cdots \frac{\partial L}{\partial w_d} \right] = [g_1 g_2 \dots g_d]$. More generally, the Jacobian also arises when we consider the matrix of partial derivatives for a specific layer's outputs with respect to its inputs. Intuitively, the Jacobian matrix encodes the rate of change of a given vector-valued function's outputs with respect to its inputs.

The Hessian matrix. For a twice-differentiable loss L, the Hessian matrix is the matrix of second-order derivatives of the loss function with respect to the weights, mathematically expressed as $\mathbf{H} = \nabla^2_{\mathbf{w}} L$. Intuitively, its role is to express the local geometry ("curvature") of the loss around a given point $\mathbf{w}$, leading to a faithful approximation of the function in a small neighborhood $\delta\mathbf{w}$ around the point $\mathbf{w}$. The second-order approximation of the function, which includes the first-order (gradient) term and the second-order (Hessian) term, is also referred to as the *local quadratic model* for the loss. Following the Taylor expansion, where we assume that the higher-order terms are negligible, this leads to the approximation

$$L(\mathbf{w} + \delta\mathbf{w}) \approx L(\mathbf{w}) + \nabla_{\mathbf{w}} L \, \delta\mathbf{w} + \frac{1}{2} \delta\mathbf{w}^\top \mathbf{H} \, \delta\mathbf{w}.$$

For clarity, note that here we take $\mathbf{w}$ to be a column vector, which implies that each term in the above expression is a scalar.

1.2.2 The Probabilistic Formulation. The above deterministic formulation inherently assumed a deterministic "correct" output label corresponding to each input example. However, it is just as reasonable to consider that each input example $\mathbf{x}$ has some probability of being assigned a given label $\mathbf{y}$, rather than the output being fixed.

We can formalize this intuition following Martens and Grosse [2015]. Given input examples $\mathbf{x} \in \mathcal{X}$ and outputs $\mathbf{y} \in \mathcal{Y}$, we assume that input vectors $\mathbf{x}$ are drawn from a distribution Q_x, and that the corresponding outputs y are drawn from a conditional distribution $Q_{y|x}$, leading to an underlying joint probability distribution defined as $Q_{\mathbf{x},\mathbf{y}} = Q_{\mathbf{x}} Q_{\mathbf{y}|\mathbf{x}}$. We will assume that the marginal probability distribution over input samples $Q_{\mathbf{x}}$ is well-approximated by the empirical distribution $\widehat{Q}_{\mathbf{x}}$ over the inputs in our training set. Intuitively, this means that we trust the sampling distribution used to generate the input dataset to be representative of the true distribution.

In this context, the goal of learning is to minimize the distance between the target joint distribution $Q_{\mathbf{x},\mathbf{y}}$, and a learned joint distribution $P_{\mathbf{x},\mathbf{y}}(\mathbf{w})$, where $\mathbf{w}$ is the model. It is standard for this distance to be measured in terms of the Kullback-Leibler (KL) divergence between distributions. Alternatively, we can cast this as the task of predicting the output $\mathbf{y}$ given an input $\mathbf{x}$, i.e., training a model $\mathbf{w}$ to learn the conditional distribution $P_{\mathbf{y}|\mathbf{x}}(\mathbf{w})$, where $P_{\mathbf{y}|\mathbf{x}}(\mathbf{w})$ is the probability of a given output given a certain input, which should be close to the true distribution $Q_{\mathbf{y}|\mathbf{x}}$. In the following, we omit the explicit dependency of $P_{\mathbf{y}|\mathbf{x}}$ on $\mathbf{w}$ when clear from context. In this formulation, we can obtain an equivalence between the standard loss we considered above and the negative log-likelihood of the probability density function corresponding to the output distribution of the model with parameters $\mathbf{w}$, which we denote by $p_{\mathbf{w}}$. Formally, for a sample $(\mathbf{x}_n \mathbf{y}_n)$ in the probabilistic formulation, we have:

$$\ell\big(\mathbf{y}, f(\mathbf{x}; \mathbf{w})\big) = -\log\big(p_{\mathbf{w}}(\mathbf{y}|\mathbf{x})\big).$$

The Fisher Matrix. Intuitively, the role of the Fisher matrix is very similar to that of the Hessian matrix, but in the *probabilistic* setting, where our notion of distance is the KL divergence between the model's output distribution and the true output distribution. More precisely, assuming the probabilistic view, the Fisher information matrix F [Ly et al. 2017] of the model's conditional

distribution $P_{y|x}$ is defined as

$$F = \mathrm{E}_{P_{\mathbf{x},\mathbf{y}}} \left[\nabla_{\mathbf{w}} \log p_{\mathbf{w}}(\mathbf{x},\mathbf{y}) \, \nabla_{\mathbf{w}} \log p_{\mathbf{w}}(\mathbf{x},\mathbf{y})^{\top} \right] . \tag{1}$$

It can be proved that the Fisher matrix in fact satisfies $F = \mathrm{E}_{P_{\mathbf{x},\mathbf{y}}} \left[-\nabla^2_{\mathbf{w}} \log p_{\mathbf{w}}(\mathbf{x},\mathbf{y}) \right]$. Matching the original intuition, we can express $P_{\mathbf{y},\mathbf{x}} = Q_{\mathbf{x}} P_{\mathbf{y}|\mathbf{x}} \approx \widehat{Q}_{\mathbf{x}} P_{\mathbf{y}|\mathbf{x}}$.

Further, it is known [Ly et al. 2017] that, if the model's output conditional distribution $P_{\mathbf{y}|\mathbf{x}}$ matches the conditional distribution of the data $\widehat{Q}_{\mathbf{y}|\mathbf{x}}$, then the Fisher and Hessian matrices are in fact equivalent. In practical terms, this means that, if $\mathbf{w}$ is an accurate set of parameters for the model, we can approximate the Hessian matrix at $\mathbf{w}$ with the Fisher matrix. In turn, this is useful since the Fisher matrix can be more efficiently approximated, as we will see below.

The Empirical Fisher. In practical settings, it is common to consider an approximation to the Fisher matrix introduced in Eq. (1), where we replace the model distribution $P_{\mathbf{x},\mathbf{y}}$ with the empirical training distribution $\widehat{Q}_{\mathbf{x},\mathbf{y}}$. Then we can simplify the expression of empirical Fisher $\hat{F}$ as follows,

$$\hat{F} = \mathrm{E}_{\widehat{Q}_{\mathbf{x}}} \left[\mathrm{E}_{\widehat{Q}_{\mathbf{y}|\mathbf{x}}} \left[\nabla \log p_{\mathbf{w}}(\mathbf{y}|\mathbf{x}) \nabla \log p_{\mathbf{w}}(\mathbf{y}|\mathbf{x})^{\top} \right] \right] \overset{(a)}{=} \frac{1}{N} \sum_{n=1}^{N} \underbrace{\nabla \ell \left(\mathbf{y}_n, f\left(\mathbf{x}_n; \mathbf{w}\right) \right)}_{\nabla \ell_n} \nabla \ell \left(\mathbf{y}_n, f\left(\mathbf{x}_n; w\right) \right)^{\top},$$

where (a) uses the equivalence of the loss between the probabilistic and deterministic settings. In the following discussion, we will use a shorthand ℓ_i to denote the loss for a particular training example $(\mathbf{x}[i], \mathbf{y}[i])$, and refer to the *true Fisher* when describing the matrix defined in Eq. (1). Thus, the above formula describes a fairly popular approximation, which equates the Fisher matrix with the *empirical* Fisher. For a more detailed exposition on various aspects of this topic, we refer the reader to Kunstner et al. [2019]; Ly et al. [2017]; Martens and Grosse [2015]; Singh and Alistarh [2020].

1.2.3 The Bayesian Formulation. We now provide a brief primer on Bayesian inference, which will be useful to understand the variational pruning approaches presented in the later sections. Our presentation follows [Molchanov et al. 2017].

We start from the probabilistic formulation above, in which, given a dataset $S = \{(\mathbf{x}[i], y[i])\}_{i=1}^{N}$ our goal is to identify a set of parameters $\mathbf{w}$ which approximates the "correct" distribution of outputs $p(y[i]|\mathbf{w}, \mathbf{x}[i])$ for any given input $\mathbf{x}[i]$. In Bayesian learning, it is assumed that we have some prior knowledge on $\mathbf{w}$, in the form of a *prior* distribution over models, $p(\mathbf{w})$. After observing some of the data, we can form the *posterior* distribution by following Bayes' rule

$$p(\mathbf{w}|S) = p(S\,|\mathbf{w}) p(\mathbf{w}) / p(S).$$

This process is called Bayesian Inference. However, computing the posterior distribution is often not possible in practice, as it requires computing the marginal likelihood $p(S) = \int p(S|\mathbf{w}) p(\mathbf{w}) d\mathbf{w}$, which is an intractable integral for most complex models. Therefore, certain simplifying assumptions are usually made, to enable an efficient approximation of the posterior distribution.

One specific technique for Bayesian Inference that relies on such simplifying assumptions is Variational Inference. Here, the posterior distribution $p(\mathbf{w}|S)$ is approximated by a parametric distribution $q_{\phi}(\mathbf{w})$. The quality of this distributional approximation is measured in terms of the KL divergence $D_{KL}(q_{\phi}(\mathbf{w}) \| p(\mathbf{w}|S))$, and the task of finding $p(\mathbf{w}|S)$ is translated into an optimization problem in the space of variational parameters ϕ. In this context, the optimal value of ϕ can be found by maximizing the following *variational lower bound* of the marginal log-likelihood of the data:

$$\mathcal{L}(\phi) = \sum_{i=1}^{N} \mathbb{E}_{q_\phi}[\log p(y[i]|\mathbf{x}[i], \mathbf{w})] - D_{KL}(q_\phi(\mathbf{w})\|p(\mathbf{w})). \quad (2)$$

The first term is called the expected log-likelihood, which is often denoted by $L_S(\phi)$, representing the model's loss, whereas the second term acts as a regularizer, enforcing that the parametric distribution $q_\phi(\mathbf{w})$ should stay close to the prior $p(\mathbf{w})$.

One important issue with the above framework is that, for complex models, optimizing the above variational lower bound is intractable, due to the integration required for computing $L_S(\phi)$. Instead, it is common to estimate $L_S(\phi)$ by sampling, and optimize the lower bound stochastically. A series of technical advances generally known as "reparametrization tricks" allow to obtain unbiased, differentiable, minibatch-based Monte-Carlo estimators of the expected log-likelihood term above for large-scale models.[1] We refer the interested reader to [Kingma et al. 2015; Kingma and Welling 2013; Molchanov et al. 2017; Rezende et al. 2014] for details.

Variational Dropout. To illustrate this technique, we will use the same notations as [Molchanov et al. 2017] and consider a single fully-connected layer with I input neurons and O output neurons before the non-linear activation function. Taking M to be the minibatch size, we denote the $M \times O$ output matrix by B, the $M \times I$ input matrix as A, and the $I \times O$ layer weight matrix as W. Notice that $B = AW$.

Dropout [Hinton et al. 2012] is a popular regularization method for neural networks, which injects multiplicative random noise Ξ to the layer input A, at each iteration of the training procedure. Mathematically,

$$B = (A \odot \Xi)W,$$

where the entries of Ξ denoted by ξ_{mi} follow a given distribution $p(\xi)$. The original variant of dropout used a constant parameter $p \in (0, 1)$ called *dropout rate*, and drew the random variables as $\xi_{mi} \sim \text{Bernoulli}(1 - p)$.

Srivastava et al. [2014a] reported that Gaussian dropout, where the noise is drawn from a continuous distribution $\xi_{mi} \sim \mathcal{N}(1, \alpha = \frac{p}{1-p})$, works as well as the discrete counterpart. Interestingly, this procedure has a non-trivial Bayesian interpretation, as was shown in [Kingma et al. 2015].

Specifically, applying Gaussian noise $\xi_{mi} \sim \mathcal{N}(1, \alpha)$ to a weight w_{ij} is equivalent to sampling the weight's value from a parameterized normal distribution centered at w_{ij}, denoted as $q(w_{ij} \mid \theta_{ij}, \alpha) \sim \mathcal{N}(w_{ij}|\theta_{ij}, \alpha\theta_{ij}^2)$. Thus, instead of viewing each w_{ij} as a parameter, each weight can be seen as a random variable parameterized by θ_{ij}, which controls the weight's variance.

Following this interpretation, Gaussian Dropout training can be seen as equivalent to standard stochastic optimization of the expected log-likelihood over the parameters θ_{ij}, in the special case where we draw a single sample of the weights $W \sim q(W|\theta, \alpha)$ per minibatch to estimate the expectation, and where we use a log-uniform prior distribution over the weights. Sparse Variational Dropout [Molchanov et al. 2017] extends this idea, and explicitly uses $q(W|\theta, \alpha)$ as an approximation for the posterior distribution. Thus, the parameters θ and α of the distribution $q(W|\theta, \alpha)$ can be optimized via stochastic variational inference. This means that $\phi = (\theta, \alpha)$ are the so-called *variational parameters*, introduced above. To avoid the problem of high variance of stochastic gradients for large values of α_{ij} as reported in [Kingma et al. 2015], Molchanov et al. [2017] introduce an *additive noise reparameterization*, in which the optimization is done directly over (θ, σ^2), with $\sigma^2 = \alpha\theta^2$, instead of (θ, α).

[1] The main idea is to represent the parametric noise $q_\phi(\mathbf{w})$ as a deterministic differentiable function $\mathbf{w} = g(\phi, \varepsilon)$ of some non-parametric noise $\varepsilon \sim p(\varepsilon)$. This trick allows one to obtain an unbiased estimator of the gradient of the log-likelihood term, $\nabla L_S(\phi)$.

Practically, Variational Dropout provides a way to train the dropout rate α by optimizing the variational lower bound we introduced above. Interestingly, however, the dropout rate becomes a variational parameter to be optimized, and not a simple hyper-parameter. This allows one to train individual dropout rates for each layer, neuron, or even weight. While the basic technique was introduced by Kingma et al. [2015], it was Molchanov et al. [2017] who first investigated the effects of training individual dropout rates, and showed that Variational Dropout can effectively sparsify DNNs. We discuss this latter paper and its follow-ups in Section 3.7.

1.2.4 Convolutional Layers as Designed Sparsity. Particularly common in deep learning are convolutional operators. Convolutions perform a weighted average over local regions of neurons, incorporating local information and reducing the number of weights at the same time. Convolutional Neural Networks (CNNs) have been proven to be highly successful for image classification [He et al. 2016], segmentation [He et al. 2017], and many other tasks. The convolution operator itself and its variants can be seen as a sparse version of fully connected layers (Fig. 3). Instead of connecting

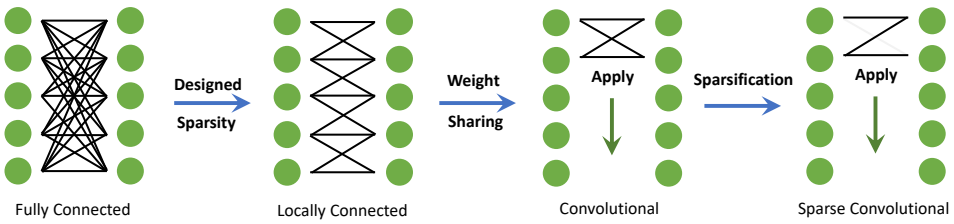


Fig. 3. Convolutional operators as sparse fully-connected operators for a single input and output channel.

every pair of neurons in the input and output layers, we prune the connections to contain only local surroundings based on the operator's convolution kernel size, strides, padding, and other factors such as dilation. The new operator contains a unique *filter* for each output neuron, also known as a Locally Connected Network (LCN) [Ngiam et al. 2010], which are used for specializing filters for different spatial regions [Grönquist et al. 2020] as shown in the 2nd part of Fig. 3. Olshausen and Field [1996] even argue that sparsity is essential property to encode vision operations.

In order to provide translational equivariance, these operators are sparsified yet again by way of *weight sharing*, reusing the local filters in each output neuron as shown in the 3rd part if Fig. 3. In a typical convolutional layer, the input is divided into C_{in} "channels" and the output into C_{out} channels or "features", multiplying and summing each input channel with a unique set of C_{out} filters. This yields the formula for the convolutional operator: $o_{j,k,l} = \sum_{m=0}^{C_{in}-1} \sum_{k_y=0}^{K_y-1} \sum_{k_x=0}^{K_x-1} x_{m,k+k_y,l+k_x} \cdot W_{j,m,k_y,k_x}$ for a filter size $K_x \times K_y$. Fig. 3 shows only one input channel and one output channel for simplicity. As we will discuss in the following, further sparsity can be introduced in CNNs, as well as other DNN classes as shown in the last part of Fig. 3.

2 OVERVIEW OF SPARSITY IN DEEP LEARNING

The utility of sparsification lies in two very different areas: (1) improved generalization and robustness and (2) improved performance for inference and/or training. We now provide a general overview of sparsification in deep learning, starting with an observation of typical sparsity-accuracy tradeoffs. We then discuss sparse storage formats, a taxonomy of element removal, and sparsification schedules. All discussions apply to both inference and training.

2.1 Generalization

Generalization performance is one of the most important aspects of a deep learning model. It measures how well the model performs for unseen data drawn from the same distribution as the training data but was not used for training. Most, if not all, sparsification follows Occam's hill [Rasmussen and Ghahramani 2001] shown as a sketch (green line) in Fig. 4: As we start to sparsify, initially the accuracy increases due to the reduction of learned noise. Intuitively, the smaller model forms a stronger regularizer forcing the learning algorithm to "focus" on more important and general aspects of the model (Part A in the figure). Then, the model reaches an often extended range of sparsities where the performance remains stable and maybe slightly decreases (Part B). Eventually, with high sparsity, the quality quickly degrades (Part C).

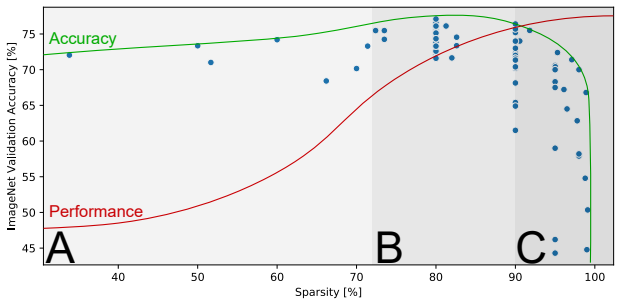


Fig. 4. Typical test error vs. sparsity showing Occam's hill (network: ResNet-50 on Top-1 ImageNet).

If we observe the computational performance of the model, we often see a curve similar to the red line in Fig. 4: initially, for low sparsity, performance grows slowly due to overheads in storing sparse structures and controlling sparse computations. Then, for moderate and high sparsity, we see a sustained growth of performance before it usually levels off at extremely high sparsities where storage and control overheads dominate. For most practical purposes and sparsities, the performance increases with growing sparsity, the area of diminishing returns only applies to extreme sparsities which deep learning models have yet to reach. In general, achieving highest performance at a specific sparsity level is complex—most techniques to store and exploit sparsity are only efficient within a limited sparsity interval and/or distribution of non-zero elements.

2.2 Performance and model storage

Sparsification reduces the necessary operations to evaluate a model as well as the memory necessary to store the model by removing nonessential elements. In some cases, for example, when whole neurons or filters are removed, we can use associativity and distributivity of linear algebra to transform a sparsified structure into a smaller dense structure. However, if we remove random elements of a weight matrix, we need to store the indices of the remaining non-zero elements.

The storage overheads for indexing m non-zero elements in a space of size n vary from bitmaps with n bits to absolute coordinate schemes using $m \log(n)$ bits. Many different formats cover the whole space and the optimal scheme depends on the sparsity, the structure, and the required access patterns (e.g., streaming, transposed, or random access). More generally, finding space-optimal indexing schemes falls into the class of integer compression problems and hundreds of sparse matrix indexing techniques exist [Pooch and Nieder 1973]. Here, we focus on a small illustrative subset.

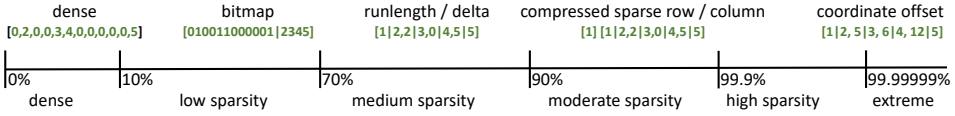


Fig. 5. Simple sparse storage formats.

Let us assume we have to store the positions of m elements, each of size k bits in a space of n elements, i.e., $m \leq n$. Fig. 5 overviews a sketch of the schemes described below and shows a range of sparsity where they are most beneficial. The exact scheme depends on many architectural factors and also the exact size of each weight. The simplest scheme stores one bit per element in a **bitmap** (BM) that stores a map with n bits, each bit indicating whether an element is present. It is efficient for relatively dense structures and requires $o = n$ additional bits. The next simpler scheme, **coordinate offset** (COO), stores each non-zero element together with its absolute offset. This scheme lives at the other end of the sparsity spectrum and is most efficient for hyper-sparse structures because it requires $o = m\lceil \log_2 n \rceil$ additional bits. This offset scheme can be extended with **runlength encoding** (sometimes also known as **delta coding**) where only the difference between two elements is stored. If the maximum difference between the indices of two neighboring elements after sorting by index is $\hat{d}$, then those can be encoded with $o = m\lceil \log_2 \hat{d} \rceil$ bits. If the offsets vary highly, then we could use a **zero-padded delta offset** scheme where we reduce the bit-width to $\lceil \log_2 \bar{d} \rceil$. Here, $\bar{d} < \hat{d}$ represents the expected difference—for all elements that are larger than $\bar{d}$ apart, we add zero values in $\bar{d}$ intervals. The overhead now depends on the distribution of distances and this scheme works best when little padding is necessary.

In the high-sparsity regime, schemes known from scientific and high-performance computing such as **compressed sparse row** (CSR), **compressed sparse column** (CSC), and more general fiber-based schemes can store indices of matrices and tensors, respectively. We exemplify these **dimension-aware schemes** using CSR: CSR represents the indices in an $n = n_c \times n_r$ matrix using column and row index arrays. The column array is of length m and stores the column indices of each value in $\lceil \log_2 n_c \rceil$ bits. The row array is of length n_r and stores the offsets of each row in the value array in $\lceil \log_2 m \rceil$ bits. The overhead is $o = m\lceil \log_2 n_c \rceil + n_r \lceil \log_2 m \rceil$ and other dimension-aware schemes are similar.

Let us consider an example with $n_c = n_r = 10^4 \rightarrow n = 10^8$, $k = 8$ and m ranging from 100-0%. The storage overhead for bitmaps is lowest for rather dense representations. No sparse storage scheme offers benefits for less than 10% sparsity. The bitmap index fares best between 10-70% sparsity and the delta encoded scheme (assuming $\hat{d} < 1000$) is best for sparsity higher than 80%. The offset index and dimension-aware schemes could work best in very high sparsity and hyper-sparse environments with very high $\bar{d}$ but it is unclear if such high sparsity is to be expected for deep models. The highest sparsity reported in the literature to date is up to 99.9% [Lin et al. 2020].

2.3 What can be sparsified?

We now provide a summary of which elements of a deep learning model can be sparsified. Fig. 6 shows an overview. First, we differentiate between *model* (also *structural*) and *ephemeral* sparsification. Model sparsification changes the model and can be considered as a generalization of neural architecture search (NAS). NAS summarizes a class of methods to automatically find better deep learning models and Elsken et al. [2019] provide an overview.

Model sparsification changes the model but does not change the sparsity pattern across multiple inference or forward passes. The two main elements, weights and neurons can be sparsified. Elements in specialized layers, such as filters in convolutional layers or heads in attention layers

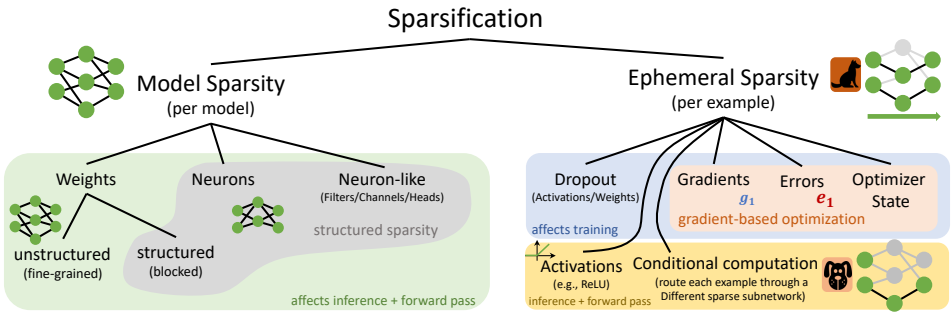


Fig. 6. Overview of DNN elements to sparsify.

are similar to neurons in the context of pruning and can be removed as well. Neuron, filter, and head sparsification reduces simple parameters of the model, can shrink it substantially, and results in a new model that is essentially dense (i.e., can efficiently be executed on the same hardware as the original model) [Sharma et al. 2017]. If we sparsify arbitrary weights, the resulting model may be unstructured and we may need to remember indices as described before. This adds overheads for index structures and leads to less efficient execution on hardware that is optimized for dense computations. However, weight sparsification "is very fine-grained and makes pruning particularly powerful." [Prechelt 1997]. Thus, approaches for structured weight sparsification have been developed to reduce indexing overheads and improve efficiency of execution. These approaches typically store contiguous blocks of the weights instead of single elements. We overview model sparsification techniques in Sections 3 and 4.

Ephemeral sparsification is a second class of sparsification approaches—it is applied during the calculation of each example individually and only relevant for this example. The most obvious structural sparsification applies to activations—in fact, the well-known ReLU and SoftMax operators lead to a natural sparsification. Both set values to zero by a fixed threshold (rounding in case of SoftMax). One can also consider random activation sparsity as in dropout [Srivastava et al. 2014b] (see Section 5.2) or top-k sparsification as used in [Ahmad and Scheinkman 2019; Makhzani and Frey 2015]. A second set of ephemeral sparsity elements are related to the gradient-based training values. The back-propagation phase of SGD uses errors and gradients to update the weights. Both can be sparsified to only update weights partially (see Section 5.3). This can have a similar effect to ephemeral sparsification in the forward pass and lead to significant performance improvements, especially in distributed settings. An option here is to delay the communication/update of small local gradient contributions until they are significant [Renggli et al. 2019]. Another important class of ephemeral techniques is conditional computation, where the model dynamically decides a sparse computation path for each example. We overview ephemeral sparsification techniques in Section 5.

2.4 When to sparsify?

While ephemeral sparsity is dynamically updated for each example and configured with a small number of parameters during inference and training, model sparsity follows a more complex NAS-like procedure. Model sparsity is thus often trained with a *schedule*. We differentiate three different classes of training schedules illustrated in Fig. 7. Each of those schedules could be used iteratively in an outer train-sparsify loop [Sun et al. 2015].

2.4.1 Sparsify after training. The **train-then-sparsify** is the most common schedule type and uses a standard dense training procedure that is run to convergence in T iterations (green area

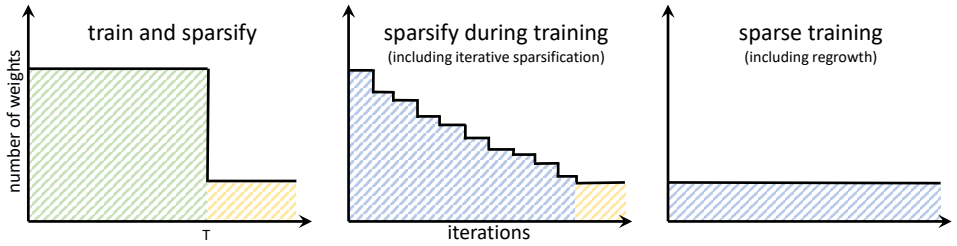


Fig. 7. Overview of structural sparsification schedules.

in Fig. 7) followed by a sparsification of the fully trained model. Beginning from the earliest works [Janowsky 1989], the model is typically re-trained ("fine tuned") after the sparsification to reach significantly higher accuracy (yellow area in Fig. 7). This schedule type aims at improving performance and/or generalization during inference. It provides the best baseline for model quality because we can always compare the sparsified model quality with the original dense model. Furthermore, since we are starting from a dense model, training does not change such that existing hyperparameter settings and learning schedules can be re-used. Some early works even show that pruning before the model has converged can reduce the final accuracy [Engelbrecht and Cloete 1996].

2.4.2 Sparsify during training. The **sparsify-during-training** schedule starts sparsification of the model before it has been trained to convergence and is usually cheaper than the train-then-sparsify schedule. Furthermore, training a dense model to convergence may allow for overfitting that is hard to correct with pruning alone. Schedules that gradually sparsify during training may follow a pruning schedule that also corrects for approximation errors due to premature pruning in early iterations. Such schemes often train the dense model for some iterations before sparsification starts and end with a sparse trained model. Early work [Finnoff et al. 1993] advocates a fixed schedule to sparsify during the training before the model converges to improve the quality of solutions using early stopping. In general, sparsifying during training already reaps potential performance benefit of sparsity early on but could lead to less efficient convergence and is often more brittle to configure via hyperparameters [Ghosh and Tumer 1994]. Furthermore, this approach needs to hold the dense model in memory at the beginning of the operation and thus does not enable the use of smaller-capacity devices.

Some methods take advantage of this limitation and do not reduce the memory consumption during the training process. Instead of deleting pruned weights and gradients, they use binary masks to determine the presence or absence of weights and update even masked weights during backpropagation to enable better weight regrowth/selection (see Section 5). For example, Wortsman et al. [2019] and Lin et al. [2020] keep the full weights around to implement an efficient search through different sparse architectures by turning weights on and off during training.

The sparsification schedule, i.e., how fast to prune how many elements, is of central importance to this method. Prechelt [1997] observes that a fixed pruning schedule can reduce the generalization ability of the network substantially. He also observes that the distribution of weight values during training is roughly normal with the mean and variance increasing during the process. Pruning reduces the variance and raises the mean, then during early training the variance increases and the mean decreases before training proceeds as before with increasing mean and variance. Prechelt uses the generalization loss to characterize the amount of overfitting and adjust the pruning rate dynamically during training. The pruning rate increases with growing generalization loss and

saturates at a maximum value. This method demonstrates a significant gain in generalization ability for well-tuned static-dynamic schedules.

Another approach, Iterative hard thresholding (IHT), is a technique where training schedules of dense and sparse iterations are combined [Jin et al. 2016]. IHT iterates the following two steps: (1) prune all but the top-k weights by magnitude (implements an L_0 constraint, see Section 3.6) and fine-tune the sparsified network to the task for s iterations, and (2) re-enable the pruned weights and train the dense network for d iterations. The outer loop is running for i iterations with a total of s_i sparsified and d_i dense training steps. The first step regularizes the network while the second step relaxes the optimization to "learn better representations" [Jin et al. 2016]. Han et al. [2017] use a similar scheme where they run three steps during training: (1) (traditional) dense training to convergence, (2) magnitude-pruning followed by retraining, and (3) dense training. All steps are performed for multiple iterations but the overall scheme is not repeated. They show that this dense-sparse-dense scheme leads to significantly higher generalization performance. Carreira-Perpinan and Idelbayev [2018] use a similar scheme of sparsification followed by training. They only re-enable a subset of the weights while others are masked out by a learned mask using a penalty term. They argue that magnitude-based pruning (see Section 3.2) arises naturally in their scheme but the "soft pruning" approach selects better weights allowing for higher sparsity. All those schemes aim to improve the "learnability" of the model by supporting the standard stochastic gradient descent (SGD) algorithm.

SGD training dynamics and sparsity. Similarly to reduced neuroplasticity as biological brains age [Jones et al. 2006], studies of deep neural networks show that the importance of elements is determined relatively early on in training. Specifically, Shwartz-Ziv and Tishby [2017] argue that SGD-based training of deep neural networks happens in two phases: (1) a drift phase that quickly minimizes the empirical risk (training error), and (2) a diffusion phase that compresses the internal representation. Similarly, Achille et al. [2019] describe two phases of training where the first phase discovers the important connections and their topology between layers and the second phase fine-tunes this relatively fixed pattern. Michel et al. [2019] show that the most important heads in transformers (see Section 6.2) are identified in the first 10 epochs. Ding et al. [2019b] observe that identifying weights for later elimination happens early in the training process and weights are rarely re-added late in the process. We call this phenomenon *early structure adaptation* in the following.

You et al. [2020] and Golub et al. [2019] directly utilize early structure adaptation during the training process where they freeze the sparsity pattern after some iterations. You et al. [2020] propose to use low-cost approximate training to identify the best sparse structure before starting the actual training of the network. Their work is inspired by Li et al. [2020b], who show that a large learning rate in earlier iterations helps the model to memorize easy to fit patterns that are later refined. Specifically, they show that for structured pruning of feature maps in convolutional networks, quick training at low precision and large learning rates leads to a good approximation of the sparse network structure. In general, early structure adaptation is reflected in learning rate schedules and most sparsification schemes use large learning rates for denser models and drastically reduce the rate with growing sparsity.

2.4.3 Sparse training. The **fully-sparse training** schedule starts with a sparse model and trains in the sparse regime by removing and adding elements during the training process. Narasimha et al. [2008] showed early that this scheme can even outperform separate growing or pruning approaches for neuron-sparse training of simple MLPs. Weight-sparse training often uses complex hyperparameter settings and schedules. However, it enables to train very high-dimensional models whose dense representations would simply not fit into the training devices.

We differentiate between **static** and **dynamic sparsity** during sparse training. Dynamic sparsity combines pruning and regrowth of elements during the training process, while static sparsity prunes once before the training starts and does not update the model structure during training.

Dynamic sparsity during training. We start with schemes that iteratively prune and add (regrow) elements during the training phase. A general overview of pruning techniques is provided in Section 3 while growth techniques are described in Section 4. Dynamic sparse training can use any combination of those schemes—we highlight some successful approaches below.

The number of elements and the sparsity does not necessarily have to remain constant throughout training. NeST [Dai et al. 2018a], for example, uses a training schedule that is inspired by the development of the human brain [Hawkins 2017]. It uses three stages to arrive at the final network architecture: (1) a random seed architecture ("birth brain"), (2) a growth phase ("baby brain") where neurons and connections are added, and (3) a pruning phase ("adult brain") where weights and neurons are removed. SET [Mostafa and Wang 2019] combine magnitude pruning and random regrowth to maintain a balanced parameter budget throughout training. This and many other schemes focus on different ways to regrow connections, those are outlined in Section 4.

Fixed sparsity during training. Networks can also be trained with a fixed sparsity structure determined before training starts. This structure can either be hand-tuned such as "structured sparsity" for transformers [Child et al. 2019], sparsity determined in a pre-training phase [You et al. 2020], or data-independent (randomly initialized) sparsity [Bourely et al. 2017; Changpinyo et al. 2017; Prabhu et al. 2018; Su et al. 2020].

Liu et al. [2019b] question the hypothesis that one must train an overparameterized model and then prune it in order to achieve acceptable accuracy. They show that for neuron and filter removal (structured sparsity), training a smaller model with standard random weights suffices. They show examples for CNNs on CIFAR-10 and ImageNet. They achieve state-of-the-art for neuron pruning but fail for weight pruning (unstructured sparsity) on the large ImageNet dataset where fine-tuning still improves performance. They conclude that one can train sparse models from scratch without pruning if the architecture and hyperparameters are chosen well and that such sparse training may improve performance.

SNIP's [Lee et al. 2019] single shot network pruning approach identifies unstructured sparsity in the network in a data-driven way *before* training. Specifically, the scheme aims to classify (the initial random) weights as important based on an influence to the loss metric proposed nearly 30 years earlier by Mozer and Smolensky [1988]: $I_w^{(1)} = \left|\frac{\partial L}{\partial w} w\right|$, where I_w represents the importance of weight w evaluated for a single batch. They suggest to choose the batch size equal to the number of result classes. Then, the least important weights are removed and the network is trained in a standard way. ESPN [Cho et al. 2020] uses a similar technique but trains the network for a small number of iterations before sparsification in order to quickly establish more structure using early structure adaption in DNN training.

Wang et al. [2020b] observed that for sparsities above 99%, SNIP eliminates nearly all weights in some layers, effectively creating a bottleneck. Following this observation, they note the importance of "gradient flow", the ability to propagate gradients through the network. They observe that SNIP can hinder gradient flow and performs worse than random pruning at high sparsity [de Jorge et al. 2020], because it considers the gradient for each weight in isolation. Tanaka et al. [2020] even show cases where SNIP disconnected networks, rendering them untrainable, by removing all weights of a layer, a phenomenon they name "layer collapse". Wang et al. [2020b] detect bottlenecks through a reduction in the norm of the gradient. They propose Gradient Signal Preservation (GraSP), a scheme that considers gradient flows and only prunes weights that decrease the gradient norm (i.e., slow the

training of the whole network) least *after* being pruned. GraSP redefines SNIP's gradient-magnitude product of importance to the Hessian-gradient-magnitude product: $I_w^{(2)} = \delta\mathbf{w}\mathbf{H}g_\mathbf{w}$, with $\delta\mathbf{w}$ being a selection vector for w: $\delta\mathbf{w} = (0, \ldots, -w, \ldots, 0)$. They also show that GraSP improves upon SNIP in very sparse regimes. A similar observation of a "minimal layer (junction) density" to maintain a given accuracy was made earlier by Dey et al. [2019].

Verdenius et al. [2020] criticize the complexity of GraSP and introduce the small-step iterative SNIP-it for unstructured and SNAP-it for structured pruning, all before training. They follow the intuition that some elements that may be of medium importance initially, gain importance with increased pruning, roughly following the gradient flow argument. By iteratively removing elements according to $I_w^{(1)}$ followed by a re-assessment of the importance scores similar to SNIP, information bottlenecks are prevented at a much lower complexity than GraSP. de Jorge et al. [2020] derive a similar iterative algorithm as well as a variant that slightly improves performance by reanimating weights excluded in earlier iterations. They suggest to use more data during the structure finding phase and show a 5x improvement of performance over GraSP while achieving similar quality. This scheme achieves state-of-the art results today but leads to a lower accuracy than pruning of fully-trained ResNets. Verdenius et al. [2020] also found that random initialization is a very strong baseline, hinting at the idea of data-free initialization methods, which we discuss next.

The authors of SNIP complemented their initial pruning scheme with a *data-free* pruning that only considers the structure of the network [Lee et al. 2020a]. They consider the "signal propagation" across layers: better signal propagation leads to better properties during training, which leads to better networks (loss minima). Starting from a random pruning, they propose to increase the signal propagation through each layer by adjusting the initial weights using a gradient descent method. This method initializes weight matrices $\mathbf{w}$ to full rank such that the combination of sparse topology and the weight is layer-wise orthogonal. The authors argue and show empirically that such randomly structured but orthogonally initialized networks can be trained to achieve the same or higher accuracy than dense networks with the same number of parameters. Hayou et al. [2020] provide additional theoretical evidence for the efficacy of this initialization scheme and show how ResNets can be effectively initialized. Verdenius et al. [2020] and de Jorge et al. [2020] also use this scheme for initializing networks pruned in a data-dependent way. With such data-free schemes, the pruning ratio still needs to be fine-tuned per layer. Su et al. [2020] propose a fixed sparsity schedule ("smart-ratio") for ResNet and VGG that decreases for larger layers. Other networks would need to be tuned accordingly.

Tanaka et al. [2020] propose to overcome layer collapse by ensuring a minimal flow through the sparse network. They also show that iterative magnitude pruning avoids layer collapse, providing additional support for Verdenius' and Hayou's iterative schemes. They use the L_1 path norm in addition to SNIP's gradient-magnitude product to avoid layer collapse and reach extreme sparsity. It remains unclear whether the performance-accuracy tradeoff at those sparsity levels (for which layer collapse would happen) justifies the cost of avoiding it.

Another fixed sparsity training method, Neural Tangent Transfer [Liu and Zenke 2020], uses a dense teacher to derive a sparse model without requiring labels that follows a similar training trajectory as the dense one.

2.4.4 Ephemeral sparsity during training. Most efficient training methods would take advantage of both ephemeral and model sparsity during training (see Section 5 for an overview). In an empirical study, Raihan and Aamodt [2020] observe that training is less robust with respect to sparsifying activations in the forward pass and gradients in the backward pass. Based on those findings, they design the SWAT method that eliminates small weights during the forward pass and both small weights and activations during the backward pass using a simple top-k method.

2.4.5 Sparsify for transfer learning and fine tuning. In transfer learning, large pre-trained and somewhat generic networks are specialized to an often narrower task than the original broad training goal. This specialization is another opportunity for pruning and potentially parameters can be pruned during the process [Mehta 2019; Molchanov et al. 2017]. The schedule for such pruning during fine-tuning is similar to the train and prune schedule: a model is trained to convergence and then pruned. However, the difference is in the training dataset and corresponding distribution. The dataset used for fine-tuning is different from the original dataset—often it corresponds to a specific subset, but sometimes it could represent a distributional shift. So in some sense, the pre-trained network can be seen as a more intelligent (non-random) weight initialization as basis for a shorter learning process. Also, data sets for fine-tuning are often much smaller.

Given these characteristics, different pruning mechanisms are used in practice. Specifically, Molchanov et al. [2017] and Sanh et al. [2020] use first order (gradient-based, see Section 3.4) pruning for transfer learning to capture the change from the pre-trained weights to the new weights. Mehta [2019] use magnitude-based pruning to transfer sparse networks during fine-tuning. Those and related methods are summarized in Section 3.4. Chen et al. [2020] showed that task-specific fine-tuning of the BERT transformer network can result in 40-90% sparsity in final weights using iterative magnitude pruning. They found that most fine-tuned networks have a task-specific structure while the masked language modeling task that was used for pre-training generates universal sparse networks that even transfer well to other tasks.

Manessi et al. [2018] investigate how well sparse models can be used to transfer their knowledge to other tasks. They show that for various image recognition tasks, moderately sparse models transfer well with either negligible accuracy loss or even a small gain in one example.

2.4.6 General Sparse Deep Learning Schedules. Fig. 8 shows a prototypical training algorithm for a pruned network. The sparse training process can be described as a series of steps, each can be

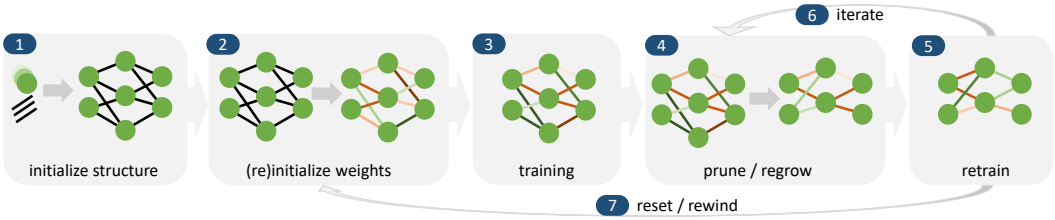


Fig. 8. Overview of sparsification schedules. Different weight values are indicated by different colors, the darker the lower the magnitude (black=zero), red indicates positive weights, green indicates negative weights.

skipped and some steps can be iterated multiple times. Step (1) initializes the network structure, this can either load a description of the network structure from disk or be built using a framework as is usually done for dense networks. However, it could also generate a random network structure or use a sparse network construction strategy such as SNIP (see Section 2.4.3).

Step (2) initializes the weights of the network, typically randomly or in transfer learning settings with pre-trained weights. For sparse networks, one could use specialized initialization strategies such as synaptic flow (see Section 2.4.3). Different weight values are indicated by different colors in Fig. 8, the darker the lower the magnitude (black=zero), red indicates positive weights, green indicates negative weights.

Step (3) trains the network for a defined number of iterations or until convergence. This training can be done with an unmodified dense training schedule or with a sparsity-inducing schedule (e.g.,

regularization, see Section 3.6). This initial training may be run until convergence or stop early for iterative methods.

Step (4) prunes and regrows various elements (see Section 2.3) using the different techniques explained in Sections 3 and 4, respectively.

Step (5) may retrain the network either for a fixed number of iterations or to convergence (this step is relatively often skipped but generally improves model accuracy).

Steps (6) and (7) indicate possible loops in the training process. Step (6) is often used in iterative training/sparsification schedules to achieve highest quality. Step (7) could be used to reset weight values, which is sometimes done (see Section 8.3).

Why retraining? Even though many pruning schemes pick the least important elements, the degradation of model quality greatly varies (see Section 6). Janowsky [1989] point out that "There is no a priori reason why their initial values should remain optimal after the pruning process". In fact, many works have shown that retraining immediately following each pruning step and fine-tuning after the last pruning step are both crucial for well-performing sparsification schedules.

In particular, we observe that many methods follow the pruning (or weight masking) step with re-training the resulting sparse network, a process also known as "fine-tuning." When the sparsification is performed in multiple steps (usually called gradual or iterative pruning), then several fine-tuning periods may be applied.

The approach of choosing which elements to remove based on the difference in loss immediately observed after removal inherently assumes that the accuracy after fine-tuning correlates perfectly with the accuracy *before* fine-tuning, i.e., immediately after pruning was applied. This assumption was validated to some extent in the analysis of [He et al. 2019a], which exhibited a correlation between the two accuracies. However, other references, notably [Singh and Alistarh 2020], observe that the SGD fine-tuning process can serve to "level" the performance of various schemes, to the extent that large gains in terms of quality immediately following the pruning step for a specific method can be erased to a large extent after fine-tuning. Fig. 9 provides an illustration of this phenomenon, as well as of the structure of a gradual pruning schedule.

Specifically, the sparsity targets shown on the graph are increased progressively, starting at 5%, until they reach the final 95% target. Fine-tuning periods of fixed length are applied between pruning steps, and a longer fine-tuning period follows the last pruning step. Observe the loss of accuracy immediately following the pruning steps, for both methods. Further, notice the significantly better performance of the second-order WoodFisher method immediately following a pruning step, but also the fact that the difference between the methods largely levels off before the next pruning step, due to SGD fine-tuning. Ultimately, the second-order method does achieve higher accuracy than the magnitude-based one (by 0.4% Top-1 validation accuracy), but this difference is lower than what one may expect based on the difference immediately following the pruning step.

Update frequency of sparse model structures. All methods described above allow to choose a sparsification frequency through the number of iterations in the (re)training steps. While ephemeral sparsification schemes are applied to each example in each minibatch, structural changes to the model often benefit from delays to reduce noise (cf. momentum) and amortize the often expensive rearrangement of data structures over multiple examples. This is consistent with biological brains where neurotransmitters are activated at high frequency while plastic structural changes happen relatively infrequently (e.g., during sleep [De Vivo et al. 2017; Diering et al. 2017]).

Tuning the right update frequency for structural changes is crucial to the performance of the final model [Jin et al. 2016]. There have not been many structured studies on how to tune this new hyperparameter but it seems related to the choice of minibatch size and ideas such as gradient noise [McCandlish et al. 2018] may be a good starting point. Raihan and Aamodt [2020] show that

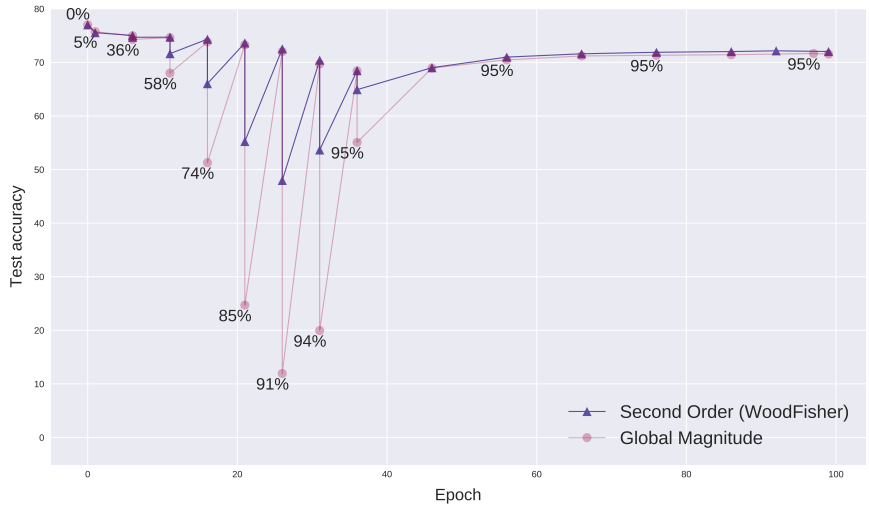


Fig. 9. An illustration of a standard gradual pruning schedule including fine-tuning periods, applied to ResNet-50 on the ImageNet dataset. The graph depicts the evolution of the validation accuracy for two different methods (global magnitude pruning and WoodFisher [Singh and Alistarh 2020]) across time.

a higher update frequency is better for training based on ephemeral weight and activation sparsity. Many works also consider tempering hyperparameters on a specific schedule during training (e.g., sparsification probability [Guo et al. 2016]), similarly to other hyperparameters (e.g., learning rate schedules).

2.5 Ensembles

One interesting use-case for sparsification is to enable ensemble models with a limited parameter and compute budget. Instead of having a single model within the budget, one could train an ensemble of multiple smaller models and average or otherwise combine their outputs to make a final selection. Collins and Kohli [2014] show that 2–3 ensemble models can improve the performance of image recognition tasks over a single model with the same parameter budget.

3 SELECTING CANDIDATES FOR REMOVAL

The core operation in any sparsification scheme is to select candidate elements to be removed. The most intuitive and most precise data-driven way to select elements for removal is to evaluate the network with and without the elements in question [Suzuki et al. 2001]. However, this simple leave-some-out approach to just train the network with and without the neurons or weights removed poses obvious scalability challenges as it needs to train $\binom{n}{k}$ networks with n elements total and k removal candidates. Another simple method is to select elements to be removed at random, which is related to the theory of compressive sensing and can be quite effective in some settings [Changpinyo et al. 2017; Mittal et al. 2018]. However, guiding the removal by some metric of importance has been shown to perform best to achieve compressed models with high sparsity in practice. In the following, we provide an overview of such selection methods.

The various schemes for element removal form the basis of different sparsification methods. Unfortunately, comparative studies such as Gale et al. [2019] have not identified a clear winner, thus, we aim to provide a comprehensive overview of the known methods. We will not quantify the efficacy of each scheme here, because this depends on the exact setting of network architecture, hyperparameters, learning rate schedule, learning task etc., and different works can hardly be compared. Instead, we will focus on the intuition behind each scheme, and describe specific results in their experimental context for some network architectures in Section 6. We provide a set of references for each method for more details. Fig. 10 provides a coarse classification of existing methods to select candidates for removal and a roadmap for this section.

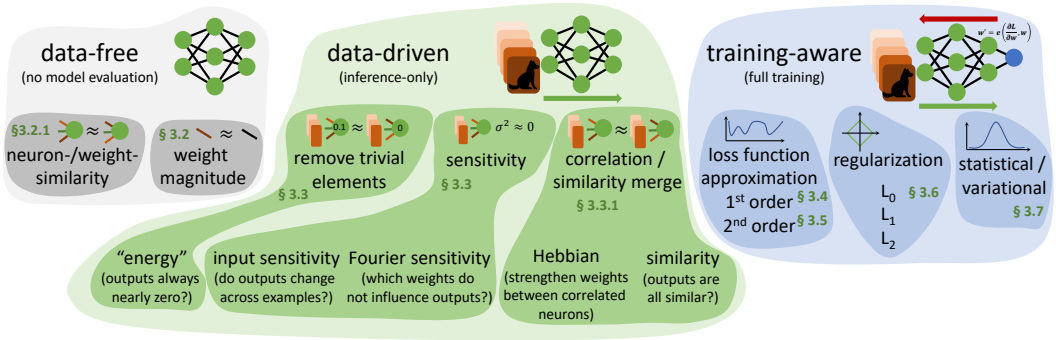


Fig. 10. Overview of schemes to select candidate elements for removal during sparsification

3.1 Structured vs. unstructured element removal

As discussed in Section 2.2, fine-grained unstructured weight sparsity requires storing the offsets of non-zero elements and handling the structure explicitly during processing. Both add significant cost to processing and storing sparse deep neural networks. Structured sparsity constrains sparsity patterns in the weights such that they can be described with low-overhead representations such as strides or blocks. This reduces the index storage overhead and simplifies processing (see Section 7). Structured sparsity promises highest performance and lowest storage overheads but it may lead to worse models because it may limit the degrees of freedom in the sparsification process.

One simple example of structured sparsity is the removal of whole neurons in a fully-connected layer: the resulting computations for the forward or backward pass after removing a neuron are simple dense matrix multiplications from which a whole row/column was removed (weights of all incoming and outgoing connections). A similar argument applies to the removal of convolutional filters [Polyak and Wolf 2015] and transformer heads [Michel et al. 2019].

Strided sparsity [Anwar et al. 2017] considers structured weight sparsification at the granularity of channels (removing whole feature maps in a layer), kernels (removing all connection between two features in consecutive layers), or a strided kernel structure (remove all connections between features with a particular stride). For example a stride-2 weight vector could be $w = [0.2, 1.9, 0, 1.3, 0, 0.3, 0, 1.2, 0, 0.4]$ where after an initial offset of one, every other element is zero. The storage of this vector would simply require to memoize the offset, stride, and non-zero elements, e.g., $\hat{w} = [1, 2, 0.2, 1.9, 1.3, 0.3, 1.2, 0.4]$.

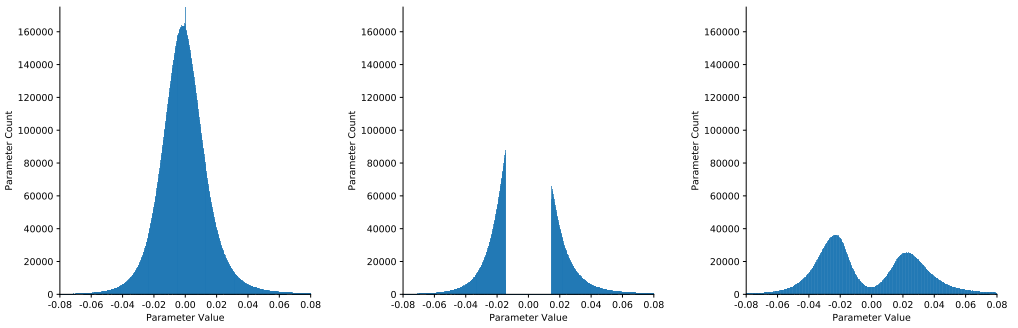
Convolutional layers can not only benefit from structured sparsity by dropping whole filters or kernels. If we write the convolution operator in matrix form (sometimes called im2col [Chellapilla et al. 2006]), we can sparsify groups in those matrices. Here, each input map may have a different

non-zero structure which is shared across all output maps. Lebedev and Lempitsky [2015] showed that this scheme, together with a regularizing training procedure and magnitude-based pruning, can sparsify filters effectively. They also find that the resulting filters are shrunk towards the center and remain largely circular. Meng et al. [2020] learn filter shapes using L_1 regularization. A similar scheme sparsifies the connections between filters—not all output filters in layer i are connected to all input filters in layer $i+1$. Specifically, Changpinyo et al. [2017] choose fixed random connectivity between the filters at each layer.

Structured pruning often uses similar schemes to unstructured pruning, sometimes with minor modifications to prune whole sets of weights. For each of the following pruning methods, we will outline its extension to structured sparsity if it is not obvious.

3.2 Data-free selection based on magnitude

One of the simplest, but also most effective, selection schemes is removing weights with the smallest absolute magnitude. This intuitive approach of removing small weights has been discussed ever since in the early 90's as a simple and effective technique [Hagiwara 1993] and always fares surprisingly well [Gale et al. 2019; Thimm and Fiesler 1995]. It is often used together with re-training the sparsified network [Han et al. 2016b] and training schedules where the sparsity is gradually increased over time [Zhu and Gupta 2017]. It can be applied to either individual weights or arbitrary groups of weights using $\sum |W_i|$ for structured pruning (e.g., blocks or rows/columns for whole neuron pruning). As we will see in the next section, this scheme even has a strong theoretical justification, under some assumptions.



(a) Dense Network (76.0%) (b) 70% Pruned (36.1%) (c) After 3-epoch Retraining (71.4%)

Fig. 11. Magnitude pruning of weights for ResNet-50 and Top-1 ImageNet validation accuracy.

As the weight values usually follow a normal distribution with a zero mean, pruning by magnitude can remove the bulk of the weights around zero as shown in Fig. 11. Part (a) shows the weight distribution before pruning, part (b) right after pruning with the condition $|w| \leq x$ for $x = 0.17$, and part (c) after retraining.

An obvious question is how to choose the magnitude x below which to prune. Besides fixing a weight budget and keeping the top-k weights globally or per layer, one could learn sparsification thresholds per layer. Kusupati et al. [2020] propose a method to learn those thresholds during the normal SGD step. They replace the original weights w with thresholded weights $w' = sgn(w) \cdot ReLU(|w| - \alpha_l)$, where α_l is a learnable pruning threshold per layer. The loss is computed with respect to w' and layer-wise α_l are learned via SGD. This scheme can easily be extended to structured sparsity as noted above. Another approach uses a reinforcement learner to derive the best values for

each layer. He et al. [2019a] proposed a DDPG agent [Lillicrap et al. 2019] to optimize for different scenarios such as a resource constraint or a target accuracy.

Magnitude pruning is often used during sparse training schedules to maintain an approximately constant connection density during training [Dettmers and Zettlemoyer 2019; Guo et al. 2016; Mocanu et al. 2018]. Bellec et al. [2018] slightly modify the scheme to fix a weight to zero if the SGD optimizer would flip its sign during training.

Han et al. [2016b] popularized magnitude pruning for modern deep neural networks as part of neural network compression for inference. Li et al. [2017] prune whole filters with the smallest sum of absolute weights in convolutional layers. Several works [Narang et al. 2017; See et al. 2016; Ström 1997] use magnitude pruning to prune recurrent neural networks as well as sparse training. Works related to the lottery ticket hypothesis also use magnitude pruning (see Section 8.3).

3.2.1 Other Data-free methods. Magnitude pruning is not the only scheme that does not consider training examples. Various other schemes solely base pruning decisions on the structure of the network. Since these methods do not depend on examples, they can be used as a pre- or post-processing step for data-driven methods.

A simple scheme compares sets of weights between different neurons. Specifically, if a fully connected layer has N output neurons, we create an $N \times N$ matrix and compare the input weights between all neurons. Now we can simply merge k similar neurons into a single neuron, multiply all weights by k, and add all biases. Srinivas and Babu [2015] showed that this method works well for small networks but prunes less for large networks. Coreset pruning [Mussay et al. 2020] enables a precise tradeoff between sparsity and approximation error. The authors show improved accuracy at 90% sparsity for very small example networks.

While data-free methods, especially magnitude pruning, are often very effective and can provide state-of-the-art results, several works have shown that more precise methods can achieve significantly better results, especially at high sparsity [Sanh et al. 2020]. Furthermore, data-free schemes often require expensive retraining to recover an accuracy as close to the original performance as possible. An obvious way to improve precision is to consider the influence of the training data (and implicitly its distribution) in the pruning selection. This leads us to the class of data-driven pruning schemes.

3.3 Data-driven selection based on input or output sensitivity

This class of selection methods considers the statistical sensitivity of the output of neurons or the whole network with respect to the training data. In those methods, a set of examples (potentially all of the training data) is used to determine directly which elements should be removed to maintain or improve prediction accuracy while sparsifying the network. Elements with very small or zero change with respect to deviation of the input examples contribute less in the entire network since their outputs are approximately constant to the variation in their inputs. Thus, such a sensitivity measure can be employed to define the relevance of an element for the function of a network and low-relevance elements can be removed.

The first scheme follows this intuition and removes neurons that show very little variation in their output across various input examples [Sietsma and Dow 1988]. After removing, we add their output to the next neurons' biases. Similarly, if two neurons in a layer always produce the same (or opposite) output for all inputs, we can remove one of those and adjust the other one's outgoing weights without changing the overall function. Castellano et al. [1997] generalize this scheme and formulated it in terms of solving a linear system of equations to change the weights after removing a neuron in order to minimize the change of output values across the dataset. They compute new weights for all units that consumed the output of the removed unit to minimize the

change in their inputs. They pose the problem using a least-squares metric and optimize it with a conjugate gradient method. Their scheme considers networks where layers can be skipped and it does not require hyperparameter tuning. They also mention the possibility to remove individual weights and later develop a similar scheme to prune input nodes ("features") [Castellano and Fanelli 2000]. In a similar scheme, Chandrasekaran et al. [2000] model the outputs of hidden units as linear combinations of outputs of other neurons. A neuron can be pruned if its output is well approximated by a linear combination of other units' outputs.

Such schemes can also be applied to filters in convolutional networks. Luo et al. [2017] phrase the filter pruning problem in terms of its output sensitivity to the following layer. They prune filters that, across the whole minibatch, change the output of a layer least. They define an optimization problem and solve it using a simple greedy strategy. Yu et al. [2018] define an importance metric that aims to minimize the error in the input to the fully connected classification layers (the "final response layer") in CNNs. This captures information flows that span multiple layers. Ding et al. [2019a] use "centripetal SGD" to train the network towards similar filter weights that can later be pruned. One could also use a geometric interpretation and find filters that are close to the geometric median of all filters [He et al. 2019b].

A simple generalization is to consider the sensitivity of neuron outputs (either model or layer) *with respect to elements in earlier layers* (including inputs). Zeng and Yeung [2006] define a direct measure of the output sensitivity of a neuron with respect to deviations in its inputs. They multiply this sensitivity by the sum of the absolute outgoing weights of the neuron to compute the relevance for pruning. The weights are included because they amplify the sensitivity as input to the next layer. Engelbrecht and Cloete [1996] define different measure using a *sensitivity matrix S* that captures the change of a neuron i in a layer k with respect to small perturbations of a neuron j in an earlier layer l. They first define the sensitivity with respect to a single example as $S_{ij,lk} = \frac{\partial f_{k,i}}{\partial f_{l,j}}$, where $f_{k,i}$ is the output of neuron i in layer k. To consider all training examples, they summarize the matrix using a mean square method into an average sensitivity matrix, which is then used to prune neurons that have low significance with respect to all output neurons. Tartaglione et al. [2018] later apply a similar scheme to weights but instead of determining the output sensitivity at the end of training, they use a weight update rule that penalizes a weight's absolute magnitude by output sensitivity during training. These simple sensitivity metrics can be seen as early predecessors of the methods basing on a first order Taylor expansion of the loss function (see Section 3.4).

A related scheme is *contribution variance* which is based on the observation that some *connections* have very similar outputs across examples in the whole training set [Thimm and Fiesler 1995]. Thus, if a connection (a source neuron multiplied by the weight) has little variance across all training examples, then it can be removed and added to the bias of the target neuron. Hagiwara [1993, 1994] proposes an even simpler scheme to prune neurons based on their "energy consumption", basically the value of activations throughout training. They prune "low-energy" neurons during training and refine the network with a simple magnitude-based weight pruning. A similar scheme prunes neurons whose activations are mostly zero for many examples—Hu et al. [2016] define the intuitive "Average Percentage of Zeros" (APoZ) pruning criterion. This scheme works well for ReLU activation functions that set negative values to zero. This scheme only distinguishes zero and non-zero values. DropNet [Tan and Motani 2020] uses the average magnitude of activations for pruning to achieve higher fidelity.

One could also consider the *variation of model output depending on variation of each weight* in a spectral sense. Here, the relevance of a weight can be assessed by its contribution to the variance of the model output. Lauret et al. [2006] propose to use the "Fourier Amplitude Sensitivity Test" (FAST) for determining the relevance of weights. The main idea is to simulate periodic oscillation

with frequency ω_i of each weight i in a fixed interval $[l, u]$. Large Fourier amplitudes at the weight's frequency ω_i and its harmonics indicate that the output is sensitive to the weight. Then, perform simulation runs to compute the contribution of each weight variation to the total output through this analysis and remove neurons whose weights are contributing less than 5% of the total output variance. The number of necessary simulation runs to disentangle the weights grows linearly with the number of weights. Han and Qiao [2013] combine a similar scheme to prune neurons in a single hidden layer. In order to find the best number of neurons for the model, they prune neurons based on their output variance across samples from the input distribution. They use FFTs to determine the change in output for inputs that vary within the input distribution. They add neurons and improve model capacity if the mean-square training error exceeds a bound.

One benefit of FAST is that it suffices to have upper and lower bounds on the features to roughly approximate the input distribution—detangling the selection process from the data. Afghan and Naumann [2020] also only rely on the the size of the interval that the input values live in. Together with the maximum partial derivative of that input with respect to a specific output, they define a measure of significance for neurons to make pruning decisions.

Many of the schemes above can be applied to any neuron in any layer. However, some study the "feature selection problem" to prune input neurons ("features"). Many datasets have inputs with very little information, for example, the four corner pixels in the digit-recognition task for MNIST play a very small role in the actual task output. Engelbrecht et al. [1995] propose a sensitivity analysis to identify input neurons that are of little relevance and can be pruned. For this, they start from a fully-trained network and compute each output's sensitivity with respect to each input $s_{ij}^{(e)} = \frac{\partial o_i}{\partial x_j}$ for each example e. They then use either a mean square, sum of absolute values, or maximum to summarize the sensitivity of an input value for the whole dataset. They then prune based on the resulting metric, re-train, and optionally repeat the procedure.

3.3.1 Selection based on activity and correlation. One simple observation is that, in many networks, some neurons are often activated together, relating to the Hebbian observation "neurons that fire together wire together" [Hebb 1949]. Several sparsification schemes are based on this observation. A simple sparsification scheme could merge neurons that have very similar output activations and simply adapt their biases and rewire the network accordingly. A similar idea has been used in "data free" schemes described in Section 3.2.1.

In a method that could be seen as a generalization of APoZ (yet, it was developed earlier), Sietsma and Dow [1988]; Sietsma and Dow [1991] observe that some neurons are producing very similar outputs for all examples during inference. They identify such pairs of similar-output neurons across the training examples and remove redundant ones. Kameyama and Kosugi [1991] extend the idea by fusing those neurons and accumulate their weights and biases to minimally affect the sparsified networks to reduce the re-training time. Suau et al. [2019] perform principal component analysis of max-pooled filter and neuron outputs to select the number of filters for a layer. They use either Principal Component Analysis or KL divergence to compute the number for each layer and then remove the most correlate neurons or filters.

A different method would strengthen connections between correlated neurons: we could preferentially drop weights between weakly correlated neurons and maintain connections between strongly correlated neurons. Sun et al. [2015] found that this method works particularly well to refine fully trained networks and leads to better generalization and good sparsification for pruning a convolutional network for face recognition.

While data-driven sensitivity-based schemes consider the outputs across the examples drawn from the input distribution, they purely aim at minimizing the impact on the input-output behavior of the network. Thus, if the network has a low accuracy, it will not gain from such pruning methods.

We could now consider the training loss function itself in the pruning process and use it to improve the model accuracy of the pruned network as much as possible.

3.4 Selection based on 1st order Taylor expansion of the training loss function

Gradient-based first order methods are most successful for learning weights in deep neural networks. It is thus not far-fetched to also apply similar methods to the selection of weights. Since gradients of the weights are computed during the normal optimization process, one can easily re-use those for determining weight importance. Furthermore, gradient computations are generally cheap, so one could employ them together with additional, so called *gating* elements to select arbitrary elements (weights, neurons, filters, etc.) for removal.

If we consider the loss function $L(\mathbf{w})$ at any time during the training process, we can write a small perturbation at $\mathbf{w}$ as

$$\delta L = L(\mathbf{w} + \delta\mathbf{w}) - L(\mathbf{w}) \approx \nabla_{\mathbf{w}} L\delta\mathbf{w} + \frac{1}{2}\delta\mathbf{w}^\top \mathbf{H}\ \delta\mathbf{w},$$

where $\nabla_{\mathbf{w}} L\delta\mathbf{w}$ and $\frac{1}{2}\delta\mathbf{w}^\top \mathbf{H}\ \delta\mathbf{w}$ are the first and second order Taylor expansion of L, respectively. (It is usual to assume that the influence of higher order terms is negligible and thus they are ignored.) In this and the next section, we describe how to use those terms to view pruning as part of the model optimization process.

A first and probably simplest approach to prune weights is to consider the *total weight change* during training. Here, we store the sum of all updates during the training and prune the weights that have changed least [Golub et al. 2019; Karnin 1990]. Molchanov et al. [2019] use a squared gradient-weight product as first-order approximation to a neuron's or filter's importance. The intuition is that if weights are changed little from their initial random values during the network's learning process, then they may not be too important. This method would be identical to sparsification techniques based on absolute magnitude (see Section 3.2) if we consider the change with respect to a (contrived) starting state of all-zero weights.

One generic way to decide whether elements can be removed is to use a gradient based scheme with respect to a binary *gating* function that regulates whether to include that element or not. Then, during training, differentiate that function at the positions $1 \longrightarrow 1 - \delta$ to determine its importance. Mozer and Smolensky [1988] uses this technique to "trim fat" neurons from networks in order to improve generalization. They define the gradient of a function α_i that disables ("gates") a neuron i in a fully-trained network as measure of its relevance. The transfer function of a fully-connected layer l changes to $f_l = \sigma_R(W_l \cdot \alpha \odot f_{l-1})$ where α is a vector with the same size as f_{l-1} and $\odot$ stands for the element-wise Hadamard product. This method requires two backprop stages—one for the weights and another one for the gate perturbation $\frac{\partial L}{\partial \alpha_i}$. The method can now prune the least important neurons iteratively and stops when it observes a large jump in $\frac{\partial L}{\partial \alpha_i}$. Lee et al. [2019] and Xiao et al. [2019] apply a very similar method based on the absolute value of the gradients to gate weights in the model.

The Tri-state ReLU [Srinivas and Babu 2016] unit is a generalization of element gating and can be used to learn neuron pruning. It is defined as:

$$\text{tsReLU}(x) = \begin{cases} wx, & x \geq 0 \\ wdx, & \text{otherwise.} \end{cases}$$

Both w and d are learnable binary parameters; w is similar to the gating function above and $d = 1$ turns the nonlinearity into the identity function. If we use a single d for each layer, then we can remove the whole layer for $d = 1$. We note that for $d = 0$ and $w = 1$ the Tri-state ReLU is identical to the traditional ReLU. Learning binary parameters is as tricky as described above and Srinivas

and Babu choose the simple function $w(1-w)$ as regularizer with final rounding and constrain the values of d and w to the interval $[0, 1]$. This can be interpreted as learning the parameters of a binomial distribution, where each Bernoulli trial indicates whether the weight is chosen or not. More general schemes for learning discrete parameters are described in Section 3.6.1. Srinivas et al. [2016] use the maximum likelihood (simple rounding as before, see Section 3.3) of this formulation to gate weights during training. You et al. [2019] use a 1st order approximation of the loss function (the gradient-weight product, see [Molchanov et al. 2017]) to select filters to prune structurally.

One could also investigate the *Jacobian matrix* after training has progressed for some iterations. Zhou and Si [1999] and Xu and Ho [2006] found that the Jacobian is usually not full rank, which means that the gradients for some weights are correlated. Zhou and Si use QR factorization of the Jacobian matrix to determine which weights are redundant while Xu and Ho use QR factorization on the output of hidden nodes to determine redundant neurons. Both approaches benefit from the nonlinearity (e.g., sigmoid or ReLU) creating the rank deficiency due to saturation or cut-off.

Specifically pruning during *transfer learning* can benefit from first order gradient information. Molchanov et al. [2017] use the magnitude of the gradients to prune full feature maps to improve the inference efficiency of fine-tuned CNNs. They use the absolute value of the gradient to determine whether a parameter should be removed or not. It seems intuitive to consider the change of parameters during fine-tuning. Movement pruning [Sanh et al. 2020] recognizes that the direction of the gradient plays a crucial role: if the pre-trained weights move towards zero for fine-tuning examples, then they are more likely to be less important (prunable) than if they move away from zero. Their technique accumulates the parameter movement and uses this as task-specific information for pruning.

Ding et al. [2019b] propose global sparse momentum to change the gradient flow during back-propagation. They classify the weights into two sets based on their importance during training. The important set is updated with the gradients during backprop while the other set does not receive gradient updates but follows weight decay to be gradually zeroed out. The importance of parameters is determined by the magnitude of the gradients and the weights as $S_w = \left|\frac{\partial L}{\partial w} w\right| = |g_w w|$ (similar to sensitivity-based approaches). The selection of the two sets is performed at each iteration such that weights may move from the unimportant into the important set during training. While the authors point out that this "re-selection" is important for the overall accuracy of the model, they also observe that it happens rarely and decreases during the training process following the early structure adaptation observation (see Section 2.4.2).

3.5 Selection based on 2nd order Taylor expansion of the training loss function

The question of selecting the "least significant" set of weights to remove from a fully-trained model relative to the difference in loss with respect to the current model was considered in the work of Le Cun et al. [1990], followed by Hassibi and Stork [1992]. These references consider an "optimization" approach to pruning, trying to answer the question of which parameter to remove in order to minimize the corresponding loss increase, under the assumption that the second-order Taylor approximation of the loss around the dense model is exact. Their frameworks differ in terms of assumptions, with the latter work being more general. We will present them jointly, outlining the differences at the end.

3.5.1 Pruning as an optimization task. Let us again consider the Taylor expansion of the loss function at $\mathbf{w}$

$$\delta L = L(\mathbf{w} + \delta\mathbf{w}) - L(\mathbf{w}) \approx \nabla_{\mathbf{w}} L \delta\mathbf{w} + \frac{1}{2}\delta\mathbf{w}^\top \mathbf{H}\, \delta\mathbf{w},$$

where the model perturbation $\delta\mathbf{w}$ is chosen so that it zeroes out a single weight $\mathbf{w}_i$ in position i and leaves the other ones unchanged, i.e., $\delta\mathbf{w} = (0, \ldots, -\mathbf{w}_i, \ldots, 0)$. Since we are assuming that the model $\mathbf{w}$ is trained to a local minimum, the (zero) gradient term can be ignored, and the problem reduces to finding the weight $\mathbf{w}_i$ whose pruning perturbation $\delta\mathbf{w}_i$ minimizes the expression

$$\frac{1}{2}\delta\mathbf{w}_i^\top \mathbf{H}\, \delta\mathbf{w}_i.$$

This minimization problem can be solved exactly via the method of Lagrange multipliers, to yield the following "saliency measure", which is associated to each weight $\mathbf{w}_i$

$$\rho_i = \frac{\mathbf{w}_i^2}{2\,[\mathbf{H}^{-1}]_{ii}}, \tag{3}$$

where $[\mathbf{H}^{-1}]_{ii}$ denotes the ith diagonal element of the *inverse Hessian matrix* of the loss L of the given model $\mathbf{w}$. To choose which weights to prune, one can sort the weights in decreasing order of this pruning statistic, the lowest-value weight being the best candidate for removal.

Interestingly, this procedure suggests that the value of the remaining weights should also change, and provides the corresponding optimal perturbation $\delta\mathbf{w}^*$. This is as follows:

$$\delta\mathbf{w}^* = -\frac{w_i \mathbf{H}^{-1}\mathbf{e}_i}{[\mathbf{H}^{-1}]_{ii}}. \tag{4}$$

The work of Hassibi and Stork [1992]; Le Cun et al. [1990] provided the first derivations for this metric, and numerical methods for computing this metric on tiny networks, with tens or hundreds of parameters.

Optimal Cell Damage (OCD) [Cibas et al. 1996] applies a very similar technique to prune the *input features* to the network. The scheme uses the sum of the saliencies ρ_i of all outgoing weights of an input value to compute the saliency of that input. The authors find it to perform worse than approaches based on regularization (see Section 3.6).

3.5.2 Magnitude pruning as a special case. To gain some intuition, let us consider the above pruning statistic when the Hessian is the identity, possibly rescaled by a constant. Intuitively, this would mean that the Hessian matrix is diagonally-dominant, and that its diagonal entries are roughly uniformly distributed. In this case, a quick examination of the above equations will yield that following the statistic is equivalent to pruning the weight of *lowest magnitude*, as the saliency measure becomes proportional to the square of each weight. As noted, the weight magnitude is a popular pruning criterion in practice, e.g., [Blalock et al. 2020; Gale et al. 2019; Singh and Alistarh 2020]. We do note that this structural assumption on the Hessian is somewhat strong, and may not hold in practice.

3.5.3 Discussion of assumptions and guarantees. The OBD/OBS method offers a powerful mathematical framework for pruning. However, the framework comes with a few important assumptions and limitations that are worth noting:

(1) The original framework assumes that pruning is performed upon a well-trained model, whose loss gradient $\nabla_{\mathbf{w}} L$ is negligible. Follow-up work [Singh and Alistarh 2020] has shown that the formulation can be extended to the case where the gradient is non-zero.
(2) The framework inherently assumes that (a) the Hessian matrix is *invertible* at the point where pruning is performed, and that (b) the pruning perturbation is *small*, and in particular that the Hessian matrix is *constant* along the direction of the pruning perturbation (this is necessary in order to ignore the higher-order terms). This constraint is addressed by practical schemes

by either performing *gradual pruning* of the weights, or by re-computing the Hessian along the pruning direction, as we will detail in the next section.

(3) Importantly, the above derivation holds if we are willing to remove a single weight at a time, and to re-compute the Hessian upon each new removal. Clearly, this would be infeasible for modern networks, so, to apply this method at scale and remove several weights in a step, one assumes that the correlations between removed weights are negligible.[2]

(4) Finally, we note that the early work of Le Cun et al. [1990] introduced the above formulation under the assumption that the Hessian matrix is *diagonal*, and applied this method on small-scale networks. Hassibi and Stork [1992] generalized this diagonal approximation, and presented efficient numerical methods for estimating the inverse Hessian under additional assumptions, which we will detail in the next section.

3.5.4 A Simple Illustration. We now provide an intuitive example for the workings of the different methods based on the Taylor expansion of the loss function. Fig. 12 shows the function $L(x_1, x_2) = 2x_1^2 + 0.5x_2^2$. Let us assume that SGD found an approximation of the minimum at the point $(x_1^*, x_2^*) = (0.1, -0.3)$. (Clearly, in this example, the optimum is $(0, 0)$ but we use $(0.1, -0.3)$ for illustration.) The gradient of L at this point is 0.1 but it is common for second-order methods to assume that it is negligible since the model is well-optimized.

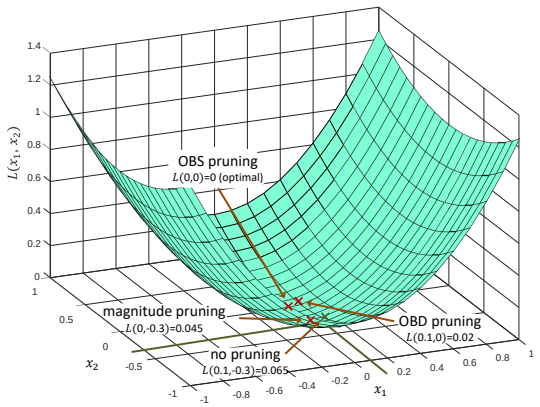


Fig. 12. Example function $L(x_1, x_2) = 2x_1^2 + 0.5x_2^2$ with estimated minimum at point $(0.1, -0.3)$.

The function value is $L(0.1, -0.3) = 0.065$. Magnitude pruning would evaluate the absolute values for x_1^* and x_2^* and decide to prune x_1, getting us to a pruned function value $L^{MAG}(0, -0.3) = 0.045$. OBD assumes that the Hessian is diagonal (which holds here but may not in general) and would dampen the absolute values of the weights by their inverse Hessian diagonals (i.e., x_1 is doubled and x_2 is halved), and would decide to remove x_2, achieving a better function value $L^{OBD}(0.1, 0) = 0.02$. Relative to OBD, OBS has two main differences. First, OBS does not assume that the Hessian is diagonal, which is more general. In our case, this would lead to the same saliency values, so x_2 would be removed. Second, OBS would also update x_1's value to adjust for the fact that x_2 is now set to zero. Concretely, we can follow Equation (4) to obtain that x_1 should be updated by $\delta x_1 = -0.1 \cdot \frac{0.5}{0.5} = -0.1$. Thus, the updated sparse point given by OBS is $(0, 0)$, leading to $L^{OBS}(0, 0) = 0$, which in this simple case is optimal.

[2]Technically, it could be that the removal of the lowest weight in the order of the pruning statistic would cause the second-lowest weight to become significantly more important towards the loss.

3.5.5 Large-scale pruning based on second-order information. The key question addressed by subsequent work on applying second-order methods to pruning has been how to apply such methods at the scale of deep neural networks, where the dimension parameter is in the millions or even in the billions. Calculating the pruning metric above requires estimating the diagonal of the *Hessian inverse*, which faces several hurdles, as the Hessian is hard to store, let alone invert, and may technically not even be invertible.

Layerwise Optimal Brain Surgeon (L-OBS). One extension of this classical approach to deeper networks, called L-OBS, was proposed by Dong et al. [2017], by defining separate layer-wise objectives, and by approximating the Hessian matrix at the level of carefully-crafted blocks, which follow the neuron structure of the network. The paper showed superior results relative to the layer-wise magnitude pruning baseline.

The Empirical Fisher Approximation to the Hessian. A common approach, first proposed by Hassibi and Stork [1992] has been to leverage the *empirical Fisher* approximation to the Hessian matrix. This approximation should hold under the following assumptions: (1) the task being considered is a classification task, e.g., whose output is given via a SoftMax function; (2) the model whose Hessian we wish to estimate is already well-optimized, and in particular its output distribution approximates well the true output distribution. Then, following our discussion of the empirical Fisher, one can approximate the Hessian matrix via

$$H \simeq \frac{1}{N} \sum_{j=1}^{N} \nabla \ell_j \cdot \nabla \ell_j^\top,$$

where N is the number of samples used for the approximation, $\nabla \ell_j$ is the gradient of the loss at sample j, and $\cdot$ denotes the outer product. (Recall that, for this approximation to hold, the model's output distribution should match well with the true output distribution.)

Fisher pruning. Theis et al. [2018] provide an example application of this approximation. Specifically, they assume a diagonal approximation of the empirical Fisher matrix, i.e., only compute the diagonal elements, and invert the resulting diagonal matrix. They apply this technique to perform structured pruning of gaze prediction models.

Approaches based on low-rank inversion. One can then leverage the observation that this approximation is effectively a sum of rank one matrices to estimate its inverse, via the classic Sherman-Morrison formula. We obtain the following recurrence, which integrates the series of gradients $(\nabla \ell_j)_{j=1}^N$ taken over individual samples into an approximation to the Fisher matrix:

$$\widehat{H}_{j+1}^{-1} = \widehat{H}_j^{-1} - \frac{\widehat{H}_j^{-1} \nabla \ell_{j+1} \nabla \ell_{j+1}^\top \widehat{H}_j^{-1}}{N + \nabla \ell_{j+1}^\top \widehat{H}_j^{-1} \nabla \ell_{j+1}}, \tag{5}$$

where initially $\widehat{H}_0^{-1} = \lambda I_d$, and λ is a small dampening parameter, usually assumed to be small. This approach was initially proposed by Hassibi and Stork [1992], and then re-discovered by Amari [1998] in the different context of optimization via natural gradient. Both these references apply the method at small-scale, specifically on single-layer neural networks.

Recently, Singh and Alistarh [2020] revisited this method at the large scale of modern deep neural networks. Specifically, they proposed a block-diagonal approximation of the above approach, and showed that it leads to an accurate local prediction of the loss along the direction of pruning, relative to the magnitude, diagonal Fisher, and to the K-FAC approximations. They then apply this method to both one-shot and gradual pruning, leading to state-of-the-art accuracy for unstructured

pruned models in both cases. Specifically, they show that the accuracy drop at a single pruning step, when computed using their method, can be significantly lower than using other methods, which leads to higher accuracy following fine-tuning steps. They also show that results can be further improved by taking the first-order (gradient) term into account, and by re-estimating the Hessian along the direction of pruning.

Extensions of OBD/OBS. Several non-trivial extensions of the OBD/OBS framework were presented in the early 90s. Pedersen et al. [1996], for example, propose the following host of improvements. First, they extend the method so that pruning is performed with respect to an estimate of the generalization error, rather than the loss. For this, they use a framework for the estimation of the generalization error given by Moody [1991][3]. Second, they incorporate the weight decay term into the OBS metric, following earlier work by Hansen et al. [1994]. Third, they recognize and address the problem of "nuisance parameters," described in brief as the issue that, if eliminating an output weight w_o, all the weights in the corresponding hidden unit are practically pruned as well. Thus, their method eliminates these parameters from the model as well, to avoid spurious contributions from them.

Other uses of the Fisher matrix. The relatively simple structure of the empirical Fisher matrix inspired additional approaches. For example, Tamura et al. [1993] and Fletcher et al. [1998] use singular value decomposition of the Fisher matrix to determine the ideal number of neurons in each hidden layer. Assuming that outputs are linearly activated, they use the rank of the resulting covariance matrix of maximum likelihood to compute the number of neurons in the compressed network.

Kronecker-Factored Approximate Curvature (K-FAC). An alternative approximation for the Fisher matrix (and thus, for the Hessian) is a family of methods based on the *Kronecker-Factored Approximate Curvature (K-FAC)* [Martens and Grosse 2015]. The method has been originally developed for the purposes of optimization, i.e., to determine an efficient pre-conditioner for the gradient update.

Following Singh and Alistarh [2020], we illustrate the method through a simple example. Consider a fully-connected network with ℓ layers. Let us denote the pre-activations of layer i by $\mathbf{s}_i$. Then, they can be written as $\mathbf{s}_i = W_i \mathbf{a}_{i-1}$, where W_i is the weight matrix at the i^{th} layer and a_{i-1} denotes the activations from the previous layer, which represent the input of the i^{th} layer.

Following the chain rule, the gradient of the objective function L with respect to the weights in layer i is

$$\nabla_{W_i} L = \text{vec}(\, \mathbf{g}_i \mathbf{a}_{i-1}^\top \,).$$

Above, we denote by $\mathbf{g}_i$ the gradient of the objective with respect to the pre-activations s_i of this layer, which implies that $\mathbf{g}_i = \nabla_{s_i} L$. Using the fact that $\text{vec}(\mathbf{u}\mathbf{v}^\top) = \mathbf{v} \otimes \mathbf{u}$, where $\otimes$ denotes the Kronecker product, we can simplify our expression of the gradient with respect to W_i as

$$\nabla_{W_i} L = \mathbf{a}_{i-1}^\top \otimes \mathbf{g}_i.$$

Given the above, observe that we can now write the block of the Fisher matrix which corresponds to layers i and j as follows:

$$\begin{aligned} F_{i,j} = \mathrm{E}\left[\nabla_{W_i} L \, \nabla_{W_j} L^\top\right] = \mathrm{E}\left[(\mathbf{a}_{i-1} \otimes \mathbf{g}_i)\,(\mathbf{a}_{j-1} \otimes \mathbf{g}_j)^\top\right] &\overset{(a)}{=} \mathrm{E}\left[(\mathbf{a}_{i-1} \otimes \mathbf{g}_i)\left(\mathbf{a}_{j-1}^\top \otimes \mathbf{g}_j^\top\right)\right] \\ &\overset{(b)}{=} \mathrm{E}\left[\mathbf{a}_{i-1}\mathbf{a}_{j-1}^\top \otimes \mathbf{g}_i \mathbf{g}_j^\top\right], \end{aligned} \tag{6}$$

[3] A similar approach, but using a different estimator, is given by Burrascano [1993].

where, in steps (a) and (b) we have used the transpose and mixed-product properties of Kronecker product. The expectation is taken over the model's distribution, as in the formulation of Fisher.

Finally, the Kronecker-Factored Approximate Curvature (K-FAC) approximation for $\widetilde{F}$ can be written as

$$\widetilde{F}_{i,j} = \mathrm{E}\left[\mathbf{a}_{i-1}\mathbf{a}_{j-1}^{\top}\right] \otimes \mathrm{E}\left[\mathbf{g}_i\mathbf{g}_j^{\top}\right] = \widetilde{A}_{i-1,j-1} \otimes \widetilde{G}_{i,j}. \tag{7}$$

Essentially, we have moved the expectation inside the expression, and applied it prior to performing the Kronecker product. This is a significant analytical assumption, since in general the expectation of the Kronecker product would not be equal to the Kronecker product of the expectations of its terms.

The advantage of this approximation is that it allows one to compute the inverse of K-FAC approximated Fisher efficiently. This is because the inverse of a Kronecker product is equal to the Kronecker product of the inverses. This implies that instead of inverting one matrix of size $n_{i-1}n_i \times n_{j-1}n_j$, one only needs to invert two smaller matrices $\widetilde{A}_{i,j}$ and $\widetilde{G}_{i,j}$, of sizes $n_{i-1} \times n_{j-1}$ and $n_i \times n_j$, respectively, where we denote the number of neurons in layer ℓ by n_ℓ.

One potential issue with this approach is that it is especially-crafted for fully-connected layers. If we wish to apply it to the case of convolutional or recurrent neural networks, the Kronecker structure needs to be further manipulated to yield an efficient approximation, as shown in [Ba et al. 2016a; Martens and Grosse 2015].

The K-FAC approximation has found several applications in optimization [Ba et al. 2016a; Osawa et al. 2019] and reinforcement learning [Wu et al. 2017]. Specifically in the case of pruning, Wang et al. [2019]; Zeng and Urtasun [2019] present applications to unstructured and structured pruning, respectively.

More precisely, Wang et al. [2019] introduces a technique called EigenDamage, which consists of (1) a novel reparameterization of the neural network in the Kronecker-factored eigenbasis (KFE), and then (2) the application of the Hessian-based structured pruning framework described above, in this basis. As an intermediate technical step, the paper provides an extension of the OBD/OBS framework to the case of *structured* pruning, with the key difference that the correlations between weights inside the same structure must be taken into account. The method is validated experimentally on the CIFAR-10 and Tiny-ImageNet datasets, for pruning residual networks.

Concurrent work by Zeng and Urtasun [2019] used a similar K-FAC-based approximation of the Hessian, but applied it to *unstructured* pruning. Relative to layer-wise pruning schemes, their approach, called MLPrune, has the advantage that it provides an approximate *global* saliency metric. Specifically, this allows the user to set a global average sparsity percentage, and the technique will automatically distribute sparsity among layers, proportionally to their sensitivity to pruning.

3.6 Selection based on regularization of the loss during training

A large class of sparsification approaches uses the well-known technique of regularization, in which we add penalty terms to the cost function, for example, $L'(\mathbf{x}, \mathbf{w}) = L(\mathbf{x}) + P(\mathbf{w})$. Here, $L(\mathbf{x})$ is the original loss function and $P(\mathbf{w})$ is a penalty term defined on the weights. Penalty functions can be defined with respect to arbitrary elements in the network (e.g., gating terms for neurons [Zhuang et al. 2020]) or metrics (e.g., required floating point operations [Molchanov et al. 2017]) and are generally easy to implement. The penalty will guide the search function to the desired output (e.g., sparse weights) and reduce the complexity of the model. The former leads to a sparse, smaller, and potentially faster model and the latter may lead to improved generalization. Mukherjee et al. [2006] show a strong link between stability and generalization. The choice of penalty term is most crucial for the success of the method. The resulting problem is often non-convex and can hardly be characterized theoretically. In fact, penalty terms can introduce additional local minima [Hanson

and Pratt 1989], which makes the optimization landscape harder to navigate. Furthermore, tuning the regularization parameters often requires a delicate balancing between the normal error term and the regularization term to guide the optimization process. Even more, regularization may require fine-tuning per layer [Lauret et al. 2006]. Yet, well-tuned regularization terms are essential to deep learning training and sparsification.

One of the first penalty terms that was shown to significantly improve generalization was weight decay [Krogh and Hertz 1991], where the weight update rule adds a reduction in absolute magnitude: $w' = (1 - \lambda)w - \alpha g$, with the decay factor λ and the learning rate α. Weight decay is similar to an L_2 normalization for an α-specific parameterization of the decay factor. Weight decay is a standard techniques for improving generalization today and it can be combined with magnitude pruning for sparsification.

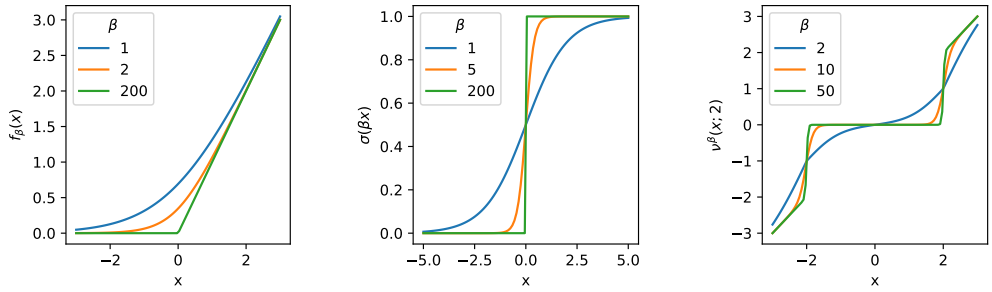
3.6.1 L_0 norm. The most obvious penalty term to generate sparse weights is the L_0 norm of the weights:

$$P(\mathbf{w}) = \alpha \|\mathbf{w}\|_0 = \alpha \sum_i \begin{cases} 0 & w_i = 0 \\ 1 & w_i \neq 0 \end{cases},$$

which simply counts the number of non-zero elements, weighted by a penalty term α. Unfortunately, optimizing this metric directly is hard due to the discrete nature (binary, either zero or non-zero) of the problem, which cannot be differentiated. In fact, the problem is NP-complete [Ge et al. 2011]. Louizos et al. [2018] approximate the L_0 norm using differentiable non-negative stochastic gating variables to determine which weights to set to zero. Their method can be used with gradient-based optimization maintaining the original learning schedules. However, as with a similar method by Srinivas et al. [2016], it may suffer from the stochastic nature of parameter selection (see [Savarese et al. 2020]): during training, new masks (weight structures) are sampled at each iteration for the forward pass. This may introduce noise into the training process if the sampling has a high variance. Furthermore, it leads to a discrepancy in the training and inference performance if a fixed deterministic sample is used at inference time. Verdenius et al. [2020] even find that tuning hyperparameters for L_0-based schemes is particularly hard to an extent that they could not apply the method to a different network.

Estimating discrete functions. The main complexity lies in selecting the non-differentiable binary gating variables whose gradient is zero almost everywhere. The possibly simplest approach is *Straight-through Estimators* [Bengio et al. 2013a] that simply ignore the derivative of the non-contiguous binary function during backpropagation (treat it as if it was an identity function). Several works use this simple trick to optimize arbitrary element gating functions ([Li et al. 2021; Sanh et al. 2020; Srinivas et al. 2016; Wortsman et al. 2019]). Others find it to be unstable at minima and suggest variants of ReLU [Yin et al. 2019]. Xiao et al. [2019] point out that hard thresholding does not support weight reanimation and they suggest "softer" selection functions such as the Leaky ReLU or Softplus shown in Fig. 13a.

A second direction to estimate discrete functions is to design parameterizable continuous approximations. Luo and Wu [2019] and Savarese et al. [2020] choose the sigmoid function as such a continuous approximation to the Heaviside step function ($H(x) = 1$ if $x > 0$, and 0 otherwise). They introduce a varying "temperature term" β to control the smoothness: $\sigma(\beta x) = \frac{1}{1+e^{-\beta x}}$. For high β, $\sigma(\beta x)$ approximates the Heaviside step function better but is "harder" to train. Fig. 13b shows the function for various values of β. Furthermore, they continuously sparsify during deterministic training by rounding the mask to $H(x)$ in the forward pass. A key aspect of this method is the adoption of an exponential schedule for the development of β from 1 to an upper bound selected as a hyperparameter. For regularization during training, they use the differentiable L_1 norm $||\sigma(\beta x)||_1$.



(a) Softplus approximation of the ReLU function.

(b) Sigmoid approximation of the Heaviside step function.

(c) Approximation of magnitude pruning.

Fig. 13. Various approximations for non-differentiable step functions. The β parameter regulates the temperature choosing between approximation quality and smoothness.

Azarian et al. [2020] use another sigmoid-based "soft pruning" estimator and combine it with layer-wise threshold learning. They also observe that pruning needs to performed slowly but they use an iterative scheme with a fixed temperature but increasingly aggressive penalty parameter.

One could also directly learn the threshold for magnitude pruning during training. Manessi et al. [2018] propose to use a soft version of the threshold linear function: $v^\beta(x,t) = ReLU(x - t) + t\sigma(\beta(x-t)) - ReLU(-x-t) - t\sigma(\beta(-x-t))$. Here, t is the threshold parameter and β is an approximation factor, as before. We show the varying "sharpness" of the curve in Fig. 13c. This function reduces x to near-zero in the range $[-t:t]$ while t can be learned through SGD. Manessi et al. [2018] then tune β as hyperparameter and apply another fixed parameter to round the values in the learned pruning interval to zero.

Top-k. Yu et al. [2012] and Collins and Kohli [2014] specify a hard limit to the number of parameters k and simply prune all but the top-k weights by magnitude. Both report that this scheme outperforms other "soft" regularization schemes. Collins and Kohli [2014] define a simple greedy scheme to select layers to sparsify and thus distribute the weights. Xiao et al. [2019] regularize gating variables, which is essentially an L_0 regularizer and train it via a hard sigmoid straight-through estimator [Hubara et al. 2016].

Polarization. A related approach for pruning is polarization [Zhuang et al. 2020] where the regularizer is defined to pull some gating elements to zero and others away from zero:

$$R(\alpha) = t\|\alpha\|_1 - \|\alpha - \bar{\alpha}\mathbf{1}_n\|_1 = \sum_{i=1}^{n} t|\alpha_i| - |\alpha_i - \hat{\alpha}|,$$

where $\bar{\alpha} = \frac{1}{n}\sum_{i=1}^{n}\alpha_i$. The effect of the term $-\|\alpha - \bar{\alpha}\mathbf{1}_n\|_1$ added to the L_1 norm is to separate small and large weights—it reaches its maximum when all α_i are equal and its minimum when half are equal to zero and the other half is equal [Zhuang et al. 2020].

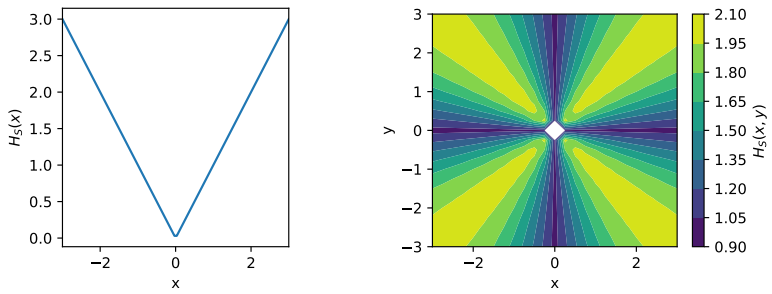
3.6.2 L_1 norm. The L_1 norm is the tightest convex relaxation of the L_0 norm that is almost everywhere differentiable. It has been popularized through the well-known lasso technique [Tibshirani 1996]. The left side of Fig. 15 visualizes Lasso in three dimensions. As opposed to L_1, the penalty is

not discrete but linear, i.e., the sum of absolute magnitude of the weights:

$$P(\mathbf{w}) = \alpha\|\mathbf{w}\|_1 = \alpha \sum_i |w_i|.$$

While L_1 norms lead to very small weight values, they usually do not reduce weights to exactly zero and magnitude-based thresholding is often used to sparsify models [Collins and Kohli 2014]. Williams [1995] uses a penalty term proportional to the logarithm in the L_1 norm to achieve better generalization through sparsification. Liu et al. [2015b] use L_1 sparsification for convolutional networks. Chao et al. [2020] use a carefully tuned L_1 proximal gradient algorithm which can provably achieve directional pruning with a small learning rate after sufficient training, and show that their solution reaches similar minima "valleys" as SGD.

Related regularization approaches. L_1 norm regularization has multiple shortcomings: First, it shrinks all parameters in the weight matrices with the same speed and second, it is also invariant to a scaling of the parameters, i.e., $||x\mathbf{w}||_1 = |x| \cdot ||\mathbf{w}||_1$. Yang et al. [2020b] address both shortcomings by use the square of the Hoyer regularizer (Fig. 14), which represents the almost anywhere differentiable scale-invariant ratio between L_1 and L_2 norms: $H_S(\mathbf{w}) = \frac{(\sum_i |w_i|)^2}{\sum_i w_i^2}$. This operator can also be applied in a group setting for structured pruning operations (see below).



(a) One-Dimensional (b) Two-Dimensional

Fig. 14. Squared Hoyer regularizer for inputs with varying dimensions.

Another related method, the shrinkage operator [Tibshirani 1996] has significantly better empirical and theoretical properties than simple thresholding after L_1 regularization: $w' = (|w|-\delta)_+ sgn(w)$ with $(x)_+$ representing the positive component of x and δ is acts as a weight threshold. This operator will zero out weights that would change sign and δ implements thresholding.

Layer-wise regularization. While regularization as part of the overall loss function is most common, one could also imagine a layer-wise regularization to restrict the focus of the optimization problem to a smaller scope. Aghasi et al. [2017] use an L_1 norm regularizer for the weights at each layer while keeping the layer output ϵ-close to the original output: $\mathbf{w}' = \arg\min\|\mathbf{w}\|_1$ s.t., $\|\sigma_R(\mathbf{w}'x_{l-1}) - \sigma_R(\mathbf{w}x_{l-1})\| \leq \epsilon$, where $\mathbf{w}'$ are the sparsified weights. For the special but very common case of ReLU ($\sigma_R(\cdot)$), they use the "cut off" to provide a convex relaxation to this optimization problem.

3.6.3 Grouped regularization. The group lasso generalizes the lasso operator to a setting where variables are segmented into predefined groups, for which either all group members should be non-zero or zero together [Yuan and Lin 2006]. We define a vector $\mathbf{y}$ of E examples and a feature

matrix $\mathbf{X}$ of size $E \times N$, all with mean zero. Suppose that the N elements are divided into G groups, and the matrix $\mathbf{X_g}$ contains only examples of group g with the corresponding coefficient vector β_g and n_g is the size of group g. The group lasso is defined as solving the convex optimization problem:

$$\min_{\beta \in \mathbb{R}^p} \left(\left\| \mathbf{y} - \sum_{g=1}^{G} \mathbf{X_g} \beta_g \right\|_2^2 + \lambda \sum_{g=1}^{G} \sqrt{n_g} ||\beta_g||_2 \right).$$

It is easy to see that, if all groups are of size one, the original lasso is (up to factors) recovered:

$$\min_{\beta \in \mathbb{R}^p} \left(\frac{1}{E} ||\mathbf{y} - \mathbf{X}\beta||_2^2 + \lambda ||\beta||_1 \right).$$

Friedman et al. [2010] point out that the group lasso does not promote sparsity within groups, which can be achieved with a small tweak to the regularization term, arriving at the sparse group lasso:

$$\min_{\beta \in \mathbb{R}^p} \left(\left\| \mathbf{y} - \sum_{g=1}^{G} \mathbf{X_g} \beta_g \right\|_2^2 + \lambda_1 \sum_{g=1}^{G} ||\beta_g||_2 + \lambda_2 ||\beta||_1 \right).$$

The middle two parts of Fig. 15 visualize group lasso and sparse group lasso with three dimensions and two groups. Group lasso uses a simple L_2 norm within each group while its sparse variant even attempts to sparsify within groups, adjustable by parameters.

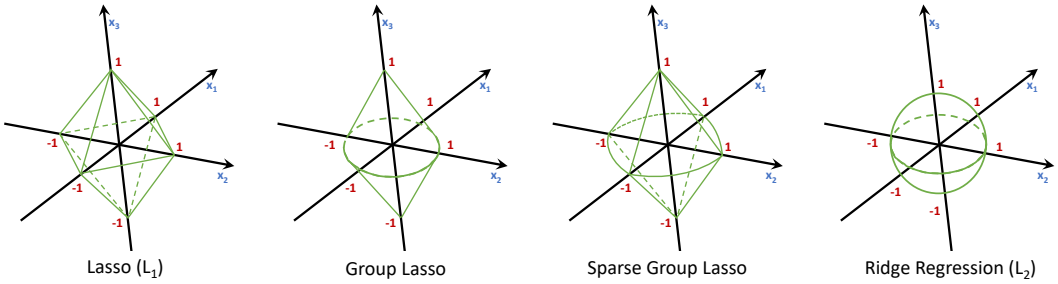


Fig. 15. Lasso vs. ((Sparse) Group) Lasso with $G_1 = \{x_1, x_2\}$ and $G_2 = \{x_3\}$ vs. Ridge Regression.

A simple definition of such a group is to assign all outgoing weights of either input or hidden neurons to a group [Scardapane et al. 2017]. Thus, if a group is zeroed during optimization, then the corresponding neuron/input can be removed from the network. For convolutional layers, groups could be used to sparsify filters or channels [Wen et al. 2016]. At a much coarser granularity, groups could also tie whole layers together and optimize the overall model structure [Wen et al. 2016]. Group lasso can also be used to keep important structures of the network, such as residual connections intact [Gordon et al. 2018].

Pan et al. [2016] use regularization on both the input and output of each neuron to facilitate neuron pruning. Their method, DropNeuron, is similar to group Lasso and they define the regularizer as the sum over all L_2 norms of all neuron's inputs or outputs: $L_i = \lambda_{l_i} \sum_{l=1}^{L} \sum_{n=1}^{n_{l-1}} ||W_{:,n}^{l}||_2$ and $L_o = \lambda_{l_o} \sum_{l=1}^{L} \sum_{n=1}^{n_{l-1}} ||W_{n,:}^{l}||_2$, where $W_{n,:}^{l}$ and $W_{:,n}^{l}$ are the input and output weights of neuron n in layer l, respectively. The authors propose to use the sum of both regularization terms with carefully tuned parameters λ_{l_i} and λ_{l_o} as penalties to further sparsify after a magnitude pruning step on the weights.

A somewhat similar scheme adds scaling factors, which are related to gating variables, to each filter [Liu et al. 2017]. Those scaling factors can be merged into a batch normalization layer and thus

do not lead to additional values. Liu et al. then penalize the factors with an L1 norm before pruning them by magnitude globally. Gordon et al. [2018] use this scheme in a grow/prune algorithm for neurons and Kang and Han [2020] extend the scheme to consider the effects of ReLU operations and the bias of batch normalization to also prune neurons that are mostly zero. Huang and Wang [2018] generalize this scheme and add scaling factors to neurons, groups, or whole layer blocks in various convolutional networks. They train the factors with an Accelerated Proximal Gradient method. Ye et al. [2018] use a similar scheme by adding factors to the batch normalization of filter outputs. They use ISTA [Beck and Teboulle 2009] as a sparsifier for those factors, eventually pulling the output of each filter to a constant. They then remove the corresponding filter and merge the removed constant into the biases of the next-layer elements.

3.6.4 Other regularization techniques. Similar regularization approaches can also be used to promote low-rank matrices for the weights such that a later compression by factorization is more effective [Alvarez and Salzmann 2017] or promote similarity of weights and filters [Ding et al. 2019a]. Yet, such schemes are outside the scope of our work.

Chauvin [1989] adds an *neuron energy penalty* term $P(o) = \mu_{en} \sum_{i=1..|o|} e(o_i^2)$ over the output neurons. The positive monotonic energy function $e(\cdot)$ and the scaler μ_{en} are parameters to the method. This penalty will decrease the magnitude of the neurons and implicitly the weights, which can then be used to sparsify the network.

Tartaglione et al. [2018] use a penalty term that is based on the *output sensitivity* of each neuron to the parameters. This sensitivity measures the relevance of the parameters to a specific output. If the sensitivity of an output neuron with respect to a specific parameter is small, then setting it to zero will change the output little. They use a regularization term to gradually decrease the absolute value of low sensitivity parameters and eventually set them to zero once they pass a certain threshold. This method can be applied during training but the authors suggest to start from a pretrained network.

Pruning can be modeled as a special case of weight quantization by breaking the error down to the contributions of the quantization error at each bit width [van Baalen et al. 2020]. They use powers-of-two bit-widths with a gating term α_i for each width i, including a general gating term for zero bits (pruned weights): $w = \alpha_2(w_2 + \alpha_4(\epsilon_4 + \alpha_8\epsilon_8)))$, where w_i and ϵ_i are the weights quantized to i bits and the quantization error with respect to the ith bit-width, respectively; α_2 prunes the whole weight.

3.6.5 Potential issues. Azarian et al. [2020] observe that L_1 (and L_2) regularization fails to prune networks with batch normalization layers. This is because batch normalization layers can rescale the output of previous layers arbitrarily and thus eliminate any regularization penalty. In practice, such weights would become simply very small while *keeping their original relative values* (i.e., not benefiting pruning decisions), followed by an upscaling in the batch normalization layer such that model performance is not influenced.

3.7 Variational selection schemes

Other methods for selecting candidate elements to be removed from the network rely on a Bayesian approach or draw inspiration from the minimum description length (MDL) principle [Grünwald 2007]. Namely, one can assume a distribution across the elements of a neural network (e.g., over individual weights or neurons), and prune elements based on their variance. The intuition behind this approach is that elements with high variance would have little contribution to the final network performance, and therefore it might be beneficial to remove them. We now discuss methods based on variants of this approach, and refer the reader to Section 1.2.3 for the relevant mathematical background.

Variational dropout. Sparse Bayesian learning [Tipping 2001] is a framework originally used for obtaining sparse models, such as the "relevance vector machine" (RVM), through carefully designed prior distributions, without additional manual tuning of hyperparameters. More recent advances in variational inference [Kingma et al. 2015; Kingma and Welling 2013; Rezende et al. 2014] have enabled the use of Bayesian learning techniques for large-scale models, such as neural networks. The connection between variational inference and dropout [Kingma et al. 2015], together with the idea of defining the relevance of a weight in terms of its variance during training have motivated Sparse Variational Dropout (Sparse VD) [Molchanov et al. 2017] as a method for pruning neural networks.

As described in Section 1.2.3, Sparse VD approximates a posterior probability distribution for each weight $\mathbf{w} \sim \mathcal{N}(\mathbf{w}|\theta, \alpha\theta^2)$, where the pair (θ, α) corresponding to individual $\mathbf{w}$ consists of the variational parameters learned by optimizing the variational lower bound. Srivastava et al. [2014a] observed empirically that Gaussian dropout has a similar performance to regular binary dropout for $\alpha = \frac{p}{1-p}$; following this observation, weights $\mathbf{w}$ with large values of α, for example $\log \alpha \geq 3$, have corresponding Binary Dropout rates $p > 0.95$, which suggests that these weights $\mathbf{w}$ can be set to zero during testing. This approach is also intuitive: large values of α correspond to high amounts of multiplicative noise in $\mathbf{w}$, which would hurt the performance of the network, unless these weights are set to 0. The benefits of this approach are that no additional hyperparameters need to be tuned, and at the end of training the weights corresponding to large values of α can be dropped in one-shot, without additional fine-tuning of the sparse network. However, one disadvantage is that this new model has twice as many parameters as the original network; additionally, the authors reported difficulties in training the model from scratch and have proposed either starting from a pretrained model, or having a "warm-up" period in which the KL-regularizing term of the bound is gradually introduced. Although the original paper reports results only on smaller datasets such as MNIST and CIFAR-10, Gale et al. [2019] has shown that Sparse VD can also sparsify large models at ImageNet scale. We do note that in this case Sparse VD achieves high sparsity, but has high variance in the results with respect to final accuracy and average model sparsity.

One intriguing question that is not entirely resolved in the literature is whether methods such as Sparse VD applied at scale are truly "variational". Namely, how different are variances of the weights considered redundant, from those of the un-pruned parameters. Following the intuition presented in [Molchanov et al. 2017], for the weights $\mathbf{w} \sim \mathcal{N}(\theta, \sigma^2)$ corresponding to large α it is desirable to have $\theta = 0$, which in turn favors values close to zero for $\sigma^2 = \alpha\theta^2$; this would prevent large amounts of multiplicative noise that would corrupt the model quality.

To examine this question, we reproduced the results for CIFAR-10 presented in [Molchanov et al. 2017], focusing on the converged values of the variational parameters. Specifically, we separated the weights corresponding to large values of α, which are eventually pruned, from the remaining weights, and studied the differences for log-variances $\log \sigma^2$. Surprisingly, all values of $\log \sigma^2$ were very close to -15, which was also the value used at initialization. Such a small initial value of all $\log \sigma^2$ was chosen by the authors to prevent the training process from diverging. Reproducing the same experiment at a larger scale for ResNet-50 trained on ImageNet using the implementation from Gale et al. [2019] revealed the same behavior: variances of the model's weights are all very small (close to $\mathbf{e}^{-15}$) and do not move during training. In this case, the threshold $\log \alpha = \log \frac{\sigma^2}{\theta^2}$ will make decisions very similar to global magnitude pruning. A distinctive behavior could be observed on Transformer networks, as implemented in [Gale et al. 2019], where the weights corresponding to large α generally had smaller $\log \sigma^2$ than the pruned weights, while the values of $\log \sigma^2$ moved significantly from their initial value. In spite of the intriguing observation that for CNNs, Sparse VD has a very similar behavior to global magnitude pruning, it is worth noting that for models

trained using variational dropout, a large proportion of the weights can be pruned immediately after training, with a small drop in test accuracy. This is in contrast with magnitude pruning methods, which require fine-tuning to recover from the drop in performance, and suggests a powerful regularization effect in Sparse VD, which is not always reflected in the final variances of the weights.

Structured Bayesian pruning. Although Sparse VD can lead to sparse neural networks, the unstructured sparsity achieved can rarely accelerate inference today. If the goal is acceleration, then structured sparsity is a more desirable outcome, and Neklyudov et al. [2017] showed how this can be achieved using the Bayesian dropout framework. The authors propose using a truncated log-normal distribution as the approximate posterior, where $\theta \sim \text{LogN}(\mu, \sigma^2) \iff \log(\theta) \sim \mathcal{N}(\mu, \sigma^2)$; here the variational parameters (μ, σ^2) are shared across different groups, such as neurons or convolutional filters. This has the advantage that log-normal noise does not change the sign of its input, as the noise is non-negative both during train and test. Furthermore, using truncated versions of both the log-uniform prior and log-normal posterior gives a closed form solution of the KL divergence term used in the variational lower-bound. To obtain a sparse solution, the authors propose thresholding neurons by their corresponding signal-to-noise ratio (SNR); intuitively, neurons with low SNR are mostly propagating noise and therefore should be set to zero. The authors show acceleration for their method on smaller datasets, such as MNIST and CIFAR-10.

Soft weight sharing. Ullrich et al. [2017] propose combining soft weight sharing with pruning to compress neural networks. The idea of soft weight sharing [Nowlan and Hinton 1992] is to compress a neural network by assigning its weights to different clusters. This is done using empirical Bayes methods, in which the prior over the parameters is learned during the training process. Following Nowlan and Hinton [1992], Ullrich et al. [2017] define the prior over the weights of a neural network as a mixture of Gaussians. One of the mixture components has a zero mean and a chosen mixture probability close to one, which will enforce a certain sparsity level for the resulting neural network. Thus, the proposed soft weight-sharing algorithm for compression starts from a pre-trained network and after optimizing the corresponding variational lower-bound, the resulting weights are assigned to the most probable cluster from the Gaussian mixture prior.

Bayesian pruning with hierarchical priors. Louizos et al. [2017] use the variational inference framework and the minimum description length (MDL) principle to compress neural networks, by defining hierarchical sparsity inducing priors to prune neurons. The MDL principle [Grünwald 2007] states that the best hypothesis is the one that uses the smallest number of bits to communicate the sum between the model's complexity cost and the data misfit error; thus, MDL is directly related to compression. Additionally, it has been well understood that variational inference can be reinterpreted through MDL [Hinton and Van Camp 1993]. With this theoretical support, Louizos et al. [2017] define a zero-mean Gaussian prior over the weights of a neural network, where the variance is sampled from a separate distribution, for example a log-uniform or half-Cauchy. This formulation enables weights within the same neuron or feature map to share the corresponding scale variable in the joint prior, which encourages structured sparsity. Furthermore, the optimal fixed point precision for encoding the weights can be determined from the posterior uncertainties, which in turn leads to quantized networks.

Bayesian pruning for recurrent neural networks. Earlier works have focused on inducing sparsity in standard feed-forward neural networks. Yet, Bayesian pruning methods have also been successfully applied to recurrent neural networks (RNNs) [Kodryan et al. 2019; Lobacheva et al. 2018]. Lobacheva et al. [2018] use Sparse VD [Molchanov et al. 2017] to prune individual weights of an LSTM or follow the approach from Louizos et al. [2017] to sparsify neurons or gates and show results on

text classification or language modeling problems. Kodryan et al. [2019] use instead the Automatic Relevance Determination (ARD) framework, in which a zero-mean element-wise factorized Gaussian prior distribution over the parameters is used, together with a corresponding Gaussian factorized posterior, such that a closed-form expression of the KL divergence term of the variational lower bound is obtained. Subsequently, the Doubly Stochastic Variational Inference (DSVI) method is used to maximize the variational lower bound and the weights for which the prior variances are lower than a certain threshold are set to zero.

Related methods. Dai et al. [2018b] prune neurons based on a simple layer-wise information bottleneck, an information-theoretic measure of redundancy. For this, they penalize the "inter-layer mutual information using a variational approximation" to sparsify. Their Variational Information Bottleneck Networks modify the loss function to contain a term that compares the mutual information from layer i to layer $i+1$ with the mutual information between layer i and the final result. With the optimization goal to minimize the former and maximize the latter, they prune based on their KL-divergence. Engelbrecht [2001] prunes based on the variance of sensitivity of inputs and neurons, and could therefore be seen as variational. Specifically, their method dictates that if the sensitivity of a parameter varies very little across the training set, then it can be pruned.

3.8 Other selection schemes

3.8.1 Genetic algorithms. Like any optimization problem, pruning can also be modeled using genetic algorithms [White and Ligomenides 1993; Whitley and Bogart 1990]. The population is created from multiple pruned versions of the neural network and each is trained separately. New networks are created using mutation, reproduction, and cross-over parameter selection. These populations are then rewarded for smaller numbers of parameters and for improved generalization. However, this approach is not practical for modern large compute-intensive training due to the high complexity of training ensembles of models.

3.8.2 Sampling-based pruning with guarantees. Another method for selecting candidate elements for pruning relies on an approach different from the Bayesian framework. Namely, Baykal et al. [2018], propose using a subset of the data to estimate the relative importance, or "empirical sensitivity" of incoming edges to a neuron; this allows the definition of an importance sampling distribution over the incoming edges, which in turn leads to sparse weight matrices. The proposed algorithm has theoretical guarantees in terms of the sparsity level obtained, as well as generalization guarantees for the sparse network. Furthermore, the framework can be improved to allow for structured pruning of neurons. Following work [Liebenwein et al. 2020] has extended the idea of sampling-based pruning to removing filters from CNNs, while also providing guarantees on the size and final output of the pruned network.

3.8.3 Diversity and quantized networks. Diversity networks [Mariet and Sra 2017] employ Determinantal Point Processes to select a subset of "diverse neurons" in each layer while fusing other similar neurons. It starts from fully-trained networks and does not require fine-tuning.

Quantized neural networks already employ an approximation function that could also be used to guide pruning decisions. Guerra et al. [2020] use a metric related to the distance between quantized and full-precision weights (i.e., the rounding error) in binary or quantized networks for selecting filters to prune.

Some neurons or filters may learn properties of the training set distribution that are not relevant to distinguish between classes within that distribution. Tang et al. [2021] propose to generate "knockoff" features that draw from the same distribution but are independent of the example's label. They feed the example and the knock-off into the same network and compare scaling factors for

filters (cf. filter sensitivity). Then they prune the features that have a large sensitivity for knockoff inputs and a relatively small sensitivity for real inputs.

3.9 Parameter budgets between different layers

All of these schemes define several hyperparameters to adjust sparsity — be it based on the value of the elements themselves, or be it based on a target sparsity level (*top-k*). One remaining question is about whether or not these parameters should be chosen per layer/operator or globally for the whole model.

Earlier works implicitly choose the sparsity level globally, such as "drop the bottom 90% of all parameters $\mathbf{w} = \mathbf{w_1} \cup \mathbf{w_2} \cup \cdots \cup \mathbf{w}_\ell$". See et al. [2016] found that global selection without differentiating layers performs best for pruning of RNNs. It was recognized soon that, especially for networks with very different layer types, e.g., convolutional and densely connected, different layers should be treated differently. Furthermore, empirical evidence suggests that even the same layer types should be sparsified differently depending on their position in the network (earlier vs. later layers). One can now consider introducing different sparsities for each layer separately [Mocanu et al. 2018], requiring to tune potentially complex hyperparameters.

Later schemes automatically determine a good parameter budget per layer to reduce the hyperparameter complexity. A simple option would be to link the sparsity to properties of the layer, such as the ratio of weights to neurons or kernel dimensionality [Evci et al. 2020]. Parameter budgets can also be redistributed during training depending on various saliency metrics. For example, Mostafa and Wang [2019] drop small magnitude parameters during training and preferentially re-add parameters in layers with larger loss gradients (i.e., layers that have been pruned less).

Figure 16 shows the distribution of sparsity across the various layers of a ResNet-50 network for different methods (see Sections 3 and 6.1 for details). An interesting and seemingly general

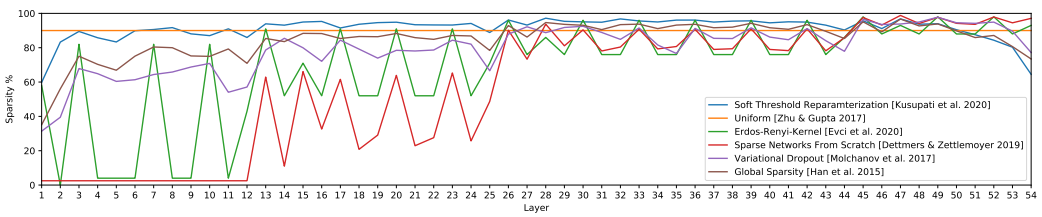


Fig. 16. Distribution of sparsity across layers for ResNet-50 and various sparsification methods.

observation is that many global schemes that can balance the parameters across layers automatically tend to assign more parameters to earlier layers than later ones [Sanh et al. 2020]. Many practitioners even disable pruning of the first layers because they empirically found this to yield higher accuracy. Tuning sparsity across layers is an important consideration for practical sparsification.

3.10 Literature overview

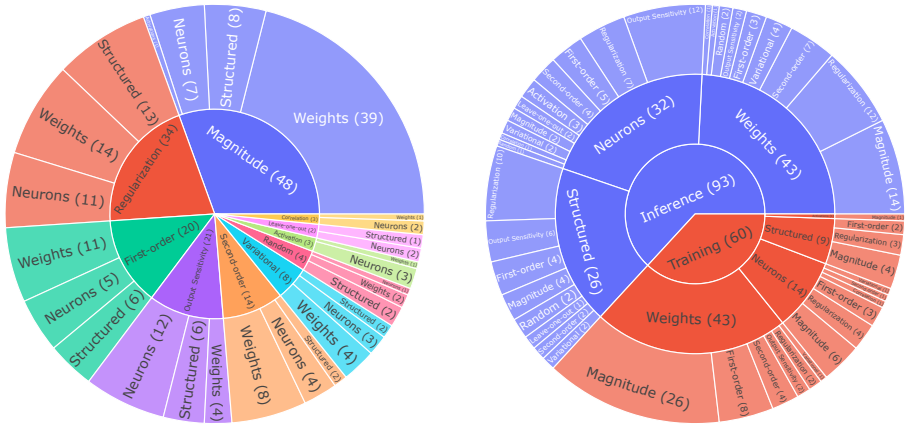
After describing the flurry of different approaches, we attempt to overview the landscape of the literature to provide some information about the popularity of the various techniques. Figures 17 and 18 show various different views of the same data summarizing all surveyed papers from 1988 to 2020. We classified each paper in three different categories: (1) the candidate element to be removed, (2) the method to choose elements for removal, and (3) whether the authors discuss optimizing inference or improving training (type). The different candidate elements are, as

described in Section 2.3, neurons, weights, convolutional filters, transformer heads, transformer hidden dimensions, and inputs. The different methods follow the structure of this section.

(a) Selection method by candidate. (b) Candidates by type. (c) Methods by type.

Fig. 17. Statistics of how many papers combine a specific selection method, to prune a specific candidate element for training or inference (type).

Fig. 17a shows that nearly 50% of all papers focus on weight sparsification, closely followed by neuron sparsification. Other structured schemes and inputs form a minority. Of the weight sparsification schemes, the vast majority uses simple magnitude pruning followed by first and second order schemes. Fig. 17b shows that more than 60% of the papers focus on inference while training is recently gaining popularity. Most inference works focus on pruning either neurons, weights, or filters while pruning to improve training largely focuses on weights. Fig. 17c allows us to compare popular pruning methods for inference and training. Inference is interestingly dominated by regularization approaches, closely followed by magnitude pruning while training focuses on magnitude.



(a) Candidates by selection method. (b) Method by candidates by type.

Fig. 18. Statistics of how many papers combine a specific selection method, to prune a specific candidate element for training or inference (type).

Fig. 18a shows that 50% of the works focus on either magnitude pruning or regularization. Magnitude pruning is most often used for weights while regularization is equally applied to all

element types. Here, we summarize filters, blocks, and heads into a single "structured" category. Fig. 18b shows an overview including all three classification dimensions. It illustrates once more the dominance of pruning weights by magnitude, followed by sensitivity-based neuron pruning.

4 DYNAMIC PRUNING: NETWORK REGROWTH DURING TRAINING

Fully-sparse training schedules remove elements during training but also need to re-add other elements in order to ensure that the model remains of approximately the same size. The process is very similar to architecture search in that it traverses the space of possible model architectures. If we prune and re-add neurons, then relatively simple schemes to add new neurons perform well [Han and Qiao 2013; Narasimha et al. 2008] because the order of neurons in a layer is insignificant. However, weights are more complex because re-adding the best weights is as crucial as removing the right weights. Yet, it is often much harder because the information for all non-existent weights is the same: they are zero. Additional hints, such as the gradient or Hessian magnitude could be used but cause additional overheads in terms of memory and compute and invalidate some of the benefits of sparsity. We will now describe the various schemes put forward to select weights to add to a sparse model during training.

4.1 Random or uniform regrowth

The simplest weight addition scheme is to activate a random new weight during training, which essentially leads to a random walk in the model space [Bellec et al. 2018]. Mocanu et al. [2018] show that this scheme leads eventually to power-law graphs following a preferential attachment rule. They also draw parallels to biological brains and argue that weight removal can be seen as natural selection, similar to synaptic shrinking during sleep [De Vivo et al. 2017; Diering et al. 2017], and weight addition can be seen as natural mutation.

Similar to layer-wise pruning, layer-wise addition can also lead to improved accuracy. The main idea is to add parameters preferentially in layers that would be sparsified less. Mostafa and Wang [2019] initially distribute all parameters according to a fixed fraction to all layers. After magnitude pruning, they add new parameters proportionally to the number of parameters retained in each layer to strengthen the significant layers.

Uniformly adding filters or neurons via a "width multiplier" to layers as part of an iterative grow/prune methodology has also been shown to be effective [Gordon et al. 2018].

Based on the observation that the optimization process benefits from large dense models (see Section 8.5), one could argue that learning in a dense space should be beneficial. Golub et al. [2019] realize that the initial (random) weight values that were pruned influence the non-pruned weights during the optimization process (not at inference, where they are removed). Since those weights have been generated with a pseudo-random number generator, the authors propose to simply recompute them on demand for training.

4.2 Based on gradient information

One simple way to determine which weights should be added is to observe the gradients during the backwards pass, including those gradients for zero weights. While this immensely increases memory and computation overheads and removes some of the benefits of sparse computations, it provides good information about the importance of specific weights: if the gradients are large, then those weights would be important. Dai et al. [2018a] show that adding weights by largest gradient is biologically plausible and related to Hebbian learning. They show that this scheme is mathematically identical to adding a new synapse between two highly stimulated neurons in adjacent layers.

The simplest version of this scheme to to compute gradients for all parameters. Lin et al. [2020] keep a full copy of all weights up to date during training and only deactivate them with a mask in the forward pass. This enables an efficient search through different architectures with various pruning methods for the dense model. They show good results using simple magnitude pruning every k iterations. Wortsman et al. [2019] uses a similar scheme and but restricts the flow of gradient information through pruned weights. In this scheme, gradients flow *to* pruned weights but their contribution is not forwarded to other weights. Dettmers and Zettlemoyer [2019] compute a momentum term that captures gradients over multiple iterations as a criterion for growing weights. While the memory and compute overheads are significant, these methods still reduce the number of arithmetic operations substantially compared to dense training. They can be combined with layer-wise redistribution strategies to focus the addition of new neurons to more efficient layers. Dettmers and Zettlemoyer [2019] find in an ablation study that updating pruned weights during training is critical for final model accuracy.

One way to reduce gradient storage is to compute it only layer-by-layer and discard it after layer-wise regrowth decisions [Evci et al. 2020]. This reduces the memory overheads but potentially decreases the accuracy due to noise in the instantaneous gradients. They use three different schemes to determine the number of parameters per layer: (1) the same uniform fraction for each layer, (2) scaling the number of weights with the number of neuron's ("Erdős Rènyi"), and (3) incorporating the kernel dimension into the scaling factor. Another way to reduce gradient storage is to only compute the top-$(k + d)$ gradients [Jayakumar et al. 2020] for k non-zero weights. In this way, the additional d gradients can be seen as a "halo zone" of the most relevant gradients to be added.

4.3 Locality-based and greedy regrowth

Biological brains are sparse structures with hierarchical sparsity distributions that are locally dense and globally sparse [Betzel et al. 2017]. It is now perceivable that local connectivity could also benefit deep neural networks. Ström [1997] decays the probability for adding a new weight exponentially with the distance between neurons, leading to a hierarchically sparse structure.

Simple greedy schemes that start from a trained network, remove all neurons and add the most beneficial neurons provide theoretical guarantees albeit with limited sparsification. Ye et al. [2020] show a scheme that adds neurons based on maximum loss reduction and Zhuang et al. [2019] add filters based on minimizing a gradient-based sensitivity.

5 EPHEMERAL SPARSIFICATION APPROACHES

In biological brains, model sparsity is one important component. However, activity sparsity is at least as important: the connections among neurons are fixed on a longer time-scale ranging from hours to days while the electrical signals appear and disappear on a millisecond time-scale. Not all 86 billion neurons of the human brain are active at any moment and are controlled by complex activation and inhibition signals. While it is hard to estimate the exact activity factor of this asynchronous system, several works suggest that only around 10% of the neurons are active at any moment [Kerr et al. 2005]. This is necessary to keep the human brain's energy budget around 20W (≈20% of a typical human's operating budget, as the most expensive organ).

Deep neural networks use ephemeral sparsification to mimic that behavior: activation functions such as ReLU inhibit certain signals by shutting down whole paths through the network, implicitly selecting the information-rich paths specific to each input problem. We can also extend ephemeral sparsity to the backpropagation learning process where we can sparsify gradients and errors during training. Ephemeral sparsity has initially been used as a regularizer but it is increasingly seen as another opportunity to save memory and energy during processing of neural networks.

We start by describing inference sparsification where neural activations are set to zero during inference and the forward pass of training. Then we consider sparsification during training. We start with the various forms of dropout, a set of techniques to sparsify networks during the forward pass of training to improve generalization. Gradient sparsification has received special attention to reduce the communication overheads in distributed data parallelism [Ben-Nun and Hoefler 2018]. We then discuss less common options to sparsify back-propagated errors between layers and the optimizer state.

5.1 Sparsifying neuron activations

The output activations of any ReLU-based neural network layer are naturally sparse [Glorot et al. 2011b] since, intuitively, on random inputs, half of the output values of such a layer would be zero. In practice, it appears that the fraction of sparse activations is significantly higher than 50%. This phenomenon does not currently have an analytical explanation, but it has been leveraged by several hardware architecture proposals (see Section 7.2). Specifically, Rhu et al. [2018] were among the first to perform an in-depth analysis of activation sparsity on a range of large-scale convolutional models with ReLU activations, showing high sparsities of up to 90% in some layers, well in excess of the 50% predicted by the structure of the ReLU activation.

This phenomenon has inspired a line of work on compressing the activation maps in a neural network for memory and computational gains, and potentially augmenting this natural sparsity. The standard technique for reducing the memory footprint of activation maps is *quantization*, see e.g., Mishra et al. [2017]. Since quantization is not the main focus of this work, we do not detail this approach here. For sparsifying activations, Alwani et al. [2016] suggested to *stochastically* prune activations, although the objective is not to gain performance, but to design a defense to adversarial attacks. To further reduce the size of activations, Gudovskiy et al. [2018] suggested converting fixed-point activations into vectors over the smallest finite field $GF(2)$ followed by nonlinear dimensionality reduction (NDR) layers embedded into the structure of the neural network. The technique results in a factor of two decrease in memory requirements with only minor degradation in accuracy, while adding only bitwise computations. At the same time, we note that the technique requires modifying the network structure, and additional retraining. Both these techniques incur low, but persistent, accuracy loss. Activation sparsity can also be used to significantly reduce memory consumption during the training process [Liu et al. 2019].

More recently, Georgiadis [2019] proposed and investigated the use of L_1-regularization applied to the activation maps, and showed that it can result in a significant increase in sparsity—up to 60% relative to naturally-occurring activation sparsity on a range of CNNs for image classification on ImageNet. Further, he investigated a range of encoding techniques for the activations, and evaluated them in terms of their resulting compression factors. Kurtz et al. [2020] followed up on this idea, and showed that Hoyer regularization [Hoyer 2004], a popular regularizer in the context of sparse recovery, is superior to L_1 regularization, in the sense that it provides higher activation sparsity with lower accuracy loss. The paper goes on to introduce a series of thresholding methods that are complementary to regularization, in the sense that they zero out activation values that are close to, but not exactly, zero. In addition, this paper describes a complete set of algorithms for leveraging activation sparsity for fast inference on CPUs, showing end-to-end inference speedup for activation-sparsified models. Concurrent work by Dong et al. [2019] also introduced an algorithmic framework for obtaining computational speedups on models where layers have extremely high input sparsity. Their method is different from Kurtz et al. [2020], but appears to require higher input sparsity to ensure speedup. In particular, it is applied to tasks such as LiDAR-based detection, or character recognition, in which inputs (and therefore further activations) are naturally extremely sparse.

Other operators such as GELU or SoftMax may also sparsify to some degree, be it through rounding towards zero with limited precision. Since those two operators are often used in transformers, see Section 6.2.

5.2 Dropout techniques for training

Dropout [Hinton et al. 2012; Srivastava et al. 2014a] is a regularizing operator in DNNs that forces the network to "prevent co-adaptation" of neurons during training. Specifically, dropout is a data-free sparsifier that uses Bernoulli random sampling (with p typically ranging from 0.01 to 0.5) to zero out neurons and nullify their contributions. During training, the neuron-masking vector, which is randomly sampled at every step, is kept stored in memory in order to mark the neurons to be ignored during backpropagation. At inference-time, no dropout masks are applied, i.e., the entire set of neurons is considered. The operator is applied mostly on the activations of fully connected layers, and is widely used to increase generalization in MLPs, CNNs, and Transformers. An interesting property of dropout is that it induces sparsity in activations [Srivastava et al. 2014a], likely due to the repeated ephemeral sparsification. The sparsity factor was observed to increase with the dropout probability p.

There are several interpretations to dropout's generalization effect. The initial line of research claims that neuron "co-adaptation" (a concept borrowed from genetics) harms generalization, and dropout prevents it by "making the presence of other hidden units unreliable" [Srivastava et al. 2014a]. Baldi and Sadowski [2013] characterize dropout in neural networks as simultaneously training an ensemble of an exponentially large set of networks, each one generated by the different masked versions, and that at inference-time their sum is taken (similarly to ensembles). Another interpretation originates from Bayesian statistics [Gal and Ghahramani 2016; Molchanov et al. 2017]. The claim is that dropout is an approximating distribution to the posterior in a Bayesian neural network with a set of random weights. It is shown that dropout's minimization objective reduces the epistemic uncertainty of a DNN, or more specifically the KL-divergence with a Gaussian process [Gal and Ghahramani 2016].

Over the years, several successful extensions and generalizations of dropout were proposed. DropConnect [Wan et al. 2013] drops out weights instead of activations. Srivastava et al. [2014a] proposed to replace the Bernoulli distribution with a normal $\mathcal{N}(1, 1)$ distribution in order to add multiplicative noise. Other variants of dropout specialize to certain operators: For convolutions, instead of random activation subsets, SpatialDropout [Tompson et al. 2015] drops entire feature maps, and DropBlock [Ghiasi et al. 2018] drops contiguous spatial regions. For recurrent neural network units, ZoneOut [Krueger et al. 2017] modifies information propagation through sequences by randomly selecting between the old hidden state and the new hidden state of the RNN unit, dropping the hidden state update. Stochastic Depth [Huang et al. 2016], Drop-Path [Larsson et al. 2017], and LayerDrop [Fan et al. 2020] are more coarse-grained versions of dropout, dropping layer weights and outputs of entire subgraphs of DNNs to prevent co-adaptation of paths and increase regularization.

The variational interpretation has been used to generalize the dropout operator in various ways. Concrete Dropout [Gal et al. 2017] uses the Concrete distribution [Maddison et al. 2017] instead of Bernoulli sampling, which results in increased generalization as well as the ability to evaluate epistemic uncertainty of the results. Variational dropout [Kingma et al. 2015] uses Bayesian principles to define a variational dropout probability specific to each neuron based on measured noise during training, foregoing the data-free property of dropout to reduce the gradient variance. Molchanov et al. [2017] makes use of variational dropout to select weights to prune (see Section 3.7).

Gomez et al. [2019] also propose a modification to the original dropout procedure to "prepare" the learned network structure for pruning. Their targeted dropout stochastically selects a set of

weights or neurons to drop that may be pruned later. Specifically, they rank weights and neurons (activation outputs) by their magnitude and apply dropout only to a fraction of those deemed less important. For this, they select the $\gamma|W|$ elements with lowest magnitude and drop each of those with probability α. This scheme allows lower-valued elements to emerge from the set of unimportant values during training.

5.3 Gradients

Gradient sparsification aims to introduce sparsity in the gradients of parameters during training. While there are exceptions, this is primarily done in order to compress the gradients communicated as part of distributed data-parallel training (see Ben-Nun and Hoefler [2018] for an overview). In this context, gradient sparsification is a subset of the more general area of communication compression, which also includes quantization and low-rank approximations (see Tang et al. [2020] for a broad overview of this area). The key intuition is that the gradients produced by SGD are noisy, and therefore identifying and removing unimportant sub-components should not have a significant impact on convergence or may even have a regularizing effect, while enabling compression.

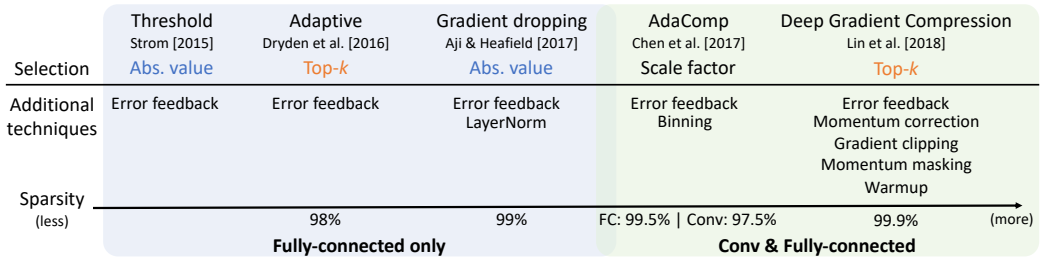


Fig. 19. Overview methods for magnitude-based gradient sparsification.

5.3.1 Magnitude-based gradient sparsification. Most methods for gradient sparsification select gradients to remove based on magnitude, on the assumption that smaller gradients are relatively less important. The first work on gradient sparsification, Strom [2015], is prototypical. A fixed threshold τ is introduced as a hyperparameter and only gradient components of absolute magnitude larger τ are applied directly to the model. The remaining values are quantized to a single bit per component based on their sign, and each is packed into a single 32-bit integer representing the index and quantized value. The other key feature is *error feedback* [Seide et al. 2014], where each worker locally accumulates the error introduced by its compression and incorporates the residual into the next iteration, by simply adding it to the gradient. Using this method, Strom [2015] showed that communication bandwidth was reduced by three orders of magnitude for training a DNN for acoustic modeling, with no reduction in accuracy.

Absolute cut-off magnitudes are hard to pick because different networks or layers within a network may have gradients of different magnitudes, and the magnitude may change during training. Dryden et al. [2016] use a form of top-k selection, whereby a fixed proportion of the positive and negative gradients are retained. They use sampling to find an absolute threshold for top-k selection in linear time. They also quantize those top-k gradients to a single bit [Seide et al. 2014], compress them based on entropy, and utilize all rounding errors through error feedback.

Subsequent works improved upon these by refining the methods for selecting gradients or incorporating other tricks. Aji and Heafield [2017] use a single proportion for all gradients, and select it globally for all layers, finding that layer normalization [Ba et al. 2016b] is sufficient to

keep gradients on a similar scale. Sun et al. [2017] performs top-k sparsification of gradients as part of sparsifying all computation in backpropagation (see Section 5.4). Chen et al. [2017] study sparsification for CNNs in addition to fully-connected networks. They formalize *binning*, where compression is applied separately to subsets of a layer's gradients. This ensures that sampling windows are small enough to effectively capture different gradient dynamics within a single layer. They also use a self-adjusting threshold based on a scale factor, rather than a fixed top-k threshold. Lin et al. [2018] introduces a number of tricks to improve the convergence of top-k sparsification, including incorporating momentum into error feedback, gradient clipping, stopping momentum on excluded gradients, and a warmup phase with less sparsity. This can result in orders-of-magnitude communication-compression; however, their results appear to be quite sensitive to hyper-parameterization. Sun et al. [2019] further approximate the gradient momentum and incorporate local update steps.

Fig. 19 provides an overview of these methods, their key components, and the sparsity they are able to achieve (we omit the sparsity for Strom [2015], as they focus on very different applications and the sparsity results are not comparable). Gradient sparsification has steadily improved in the amount of sparsity it can introduce, with Lin et al. [2018] achieving up to 99.9% sparsity. Compared to pruning for weights or activations, gradients seem to be significantly more amenable to sparsity.

5.3.2 Variance-based gradient sparsification. The convergence of SGD is significantly impacted by the variance of the stochastic gradients used. However, sparsification can increase the variance in the resulting sparse gradients, and hence slow convergence. Alistarh et al. [2017] noticed that, when stochastically quantizing gradient vectors normalized by their L_2-norm to only three quantization levels: 0, 1, and -1, in expectation all but a $\Theta(\sqrt{n})$ fraction of the n gradient values will be set to zero. This results in non-trivial compression, but also induces high additional variance, which hurts convergence. To alleviate this issue, Wangni et al. [2018] first propose rand-k sparsification, where k gradients are retained at random, biased by their absolute value, and the rest zeroed; the remaining gradients are then rescaled to ensure the gradient is unbiased. They then develop algorithms to select the optimal sparsification strategy given a variance budget. In practice this turns out to be similar to choosing an appropriate k for top-k sparsification. Similarly, Wang et al. [2018] considers the problem of minimizing variance subject to a sparsity budget. They also consider the more general problem of sparsifying arbitrary atomic decompositions, rather than just element-wise sparsification. Concurrently, Tsuzuku et al. [2018] also identify variance as a key metric, and use the variance of gradients within a mini-batch, rather than their magnitude, as a criterion for sparsification. The variance can be computed for relatively little extra cost during standard backpropagation. Using variance as a sparsification metric thus has attractive theoretical properties, and Tsuzuku et al. [2018] show that it matches or outperforms Strom [2015]'s threshold sparsification on CIFAR-10 and ImageNet.

5.3.3 Other methods for gradient sparsification. A variety of other approaches to sparsification have also been studied. Ivkin et al. [2019] use count sketches on each worker to approximate large gradients, and the sketches are communicated. Lim et al. [2019] combine sparsification with ternary quantization, and use a tunable sparsity factor to control how many values are rounded to zero. Basu et al. [2020] studies the convergence of the combination of sparsity, quantization, and local updates, showing this converges at the same rate as SGD in certain settings. Wang et al. [2020b] apply top-k sparsification in the frequency domain after applying an FFT to gradients.

5.3.4 Convergence of sparsified gradient methods. There have been several theoretical analyses of the convergence of sparsified gradient methods. Concurrently, Alistarh et al. [2018]; Jiang and Agrawal [2018]; Stich et al. [2018] show that sparsified gradient methods converge at roughly the

same rate as standard SGD, provided error feedback is used. These works differ in terms of the assumptions made and guarantees provided: for instance, Stich et al. [2018] consider the case where a single node compresses its gradient via sparsification with error correction ("memory"), assuming a convex objective function, and provides very strong convergence guarantees, similar to those of regular SGD. By way of comparison, Alistarh et al. [2018] consider non-convex objectives, and the multi-node case, but require an additional analytic assumption for their convergence bounds. Overall, these works provide a strong theoretical justification to the previous empirical results, in particular highlighting the importance of error feedback for convergence. Karimireddy et al. [2019] extend these results to more general settings. However, these results are for sparsifying an entire model's gradients, as opposed to layer-wise operation. Dutta et al. [2020] further extend these convergence results and show that layer-wise compression is theoretically better. They also experiments and show that, while this usually holds in practice, there do exist cases in practice where sparsifying an entire model out-performs layer-wise sparsification. Tang et al. [2019] provide a convergence analysis for the case where, in addition to workers sparsifiying their individual gradients before communication, the aggregated gradient is also sparsified before being communicated back to the workers. This situation is common in practice, but was neglected in previous analyses.

5.3.5 Runtime support for sparse gradient summation. Sparse communication was first implemented in the parameter server setting, where all workers communicate with a single central parameter server. However, many scalable high-performance distributed training systems perform communication without a central store using allreduces. Extending sparse communication to this case is challenging. Dryden et al. [2016] implements a ring-based allreduce that includes custom reduction operators that uncompress vectors, sum them, and recompress them with the same hyperparameters. Shi et al. [2019a] proposes a similar mechanism, global top-k, where instead of using the top k gradients from each worker, only the top k gradients among all workers are used. Shi et al. [2019b] provides convergence results for this approach. Renggli et al. [2019] propose SparCML, a framework for efficiently performing distributed aggregations of sparse vectors. They combine sparse vectors and retain all non-sparse coordinates; as this may eventually result in dense vectors, SparCML includes a mechanism to switch from sparse to dense or even dense-quantized representations.

5.3.6 Gradient sparsification for better accuracy. The prior approaches have primarily focused on sparsification in order to reduce communication bandwidth. Shokri and Shmatikov [2015] investigate such methods in the context of privacy, while Sinha et al. [2020] study top-k sparsification to improve the training quality for GANs [Goodfellow et al. 2014a]. When training a GAN, a critic network is used to identify whether samples produced by the generator are "bad". This work uses the critic to select the best k samples in each mini-batch to perform updates with.

Note that the core idea behind parallelizing mini-batch SGD consists essentially of computing an average of the gradients of the samples within a mini-batch, which functions as a lower-variance estimate of the full gradient. Computing this average is a special case of the more general distributed mean estimation problem. Several works have tried to achieve optimal communication bounds for this and related problems [Davies et al. 2020; Huang et al. 2019; Konečnỳ and Richtárik 2018; Suresh et al. 2017]. We note however that the above gradient sparsification approaches do not solve exact distributed mean estimation, since the approximation to the true mean is inherently lower-dimensional; instead, they use error feedback to correct for the inherent error.

5.4 Errors and optimizer state

In addition to the gradients of parameters, the gradients of a layer's input, or the "errors", can also be sparsified. Sun et al. [2017] introduces meProp ("minimal effort backpropagation") which applies top-k sparsification to the errors to reduce flops. This also necessarily leads to sparse gradient updates, as only k rows (or columns) of the resulting gradient matrix are non-zero. The top-k sparsification is first applied to the gradient of the loss initially computed in backpropagation, and then reapplied after every fully-connected layer to keep the errors sparse. Wei et al. [2017] demonstrate that this scheme can lead to 95% gradient sparsity.

Whether the optimizer state can be sparsified and the benefits of sparse optimizer states have yet to be explored. We expect it to lead to more memory efficient training algorithms.

5.5 Dynamic networks with conditional computation

Dynamic networks where outputs of previous layers determine a path through the network increase model capacity without increasing the computational cost. Conditional computation achieves this by *routing* the computation through the network without touching all weights. Many practical approaches use various trained gating techniques (binary or continuous, deterministic or stochastic) [Almahairi et al. 2016; Bengio et al. 2016, 2013b] or use switching methods that explicitly select the next "expert" [Jacobs et al. 1991; Jordan and Jacobs 1994; Shazeer et al. 2017]. Both approaches lead to ephemeral sparsity during the execution.

Recently, mixture of experts models have achieved impressive success in natural language processing. Shazeer et al. [2017] define a Mixture of Experts (MoE) layer to contain n expert subnetworks $E_1, \ldots, E_n$ and a gating network G that outputs a sparse n dimensional vector. The function of this layer can be written as $y = \sum_{i=1}^{n} G(x)_i E_i(x)$, where $G(x)$ selects (gates) the relevant experts. One way to implement a k-sparse gating function is to use a top-k method. Shazeer et al. [2017] use a noisy top-k gating where they add tunable Gaussian noise to the selection function to improve load balancing the experts. A typical basic gating function is $G(x) = \mathrm{softmax}(W_g x)$ with learned weights W_g. Lepikhin et al. [2020] apply this idea to transformer networks to train a model with 600 billion parameters by using a similar gating function for $k = 2$ and stochastic load balancing across the experts to enable large-scale parallel training. Switch transformers [Fedus et al. 2021] evolve the model further and show that MoE sparsity can improve pretraining speed by up to 7x compared to a dense model and supports models with extreme capacity of up to a trillion parameters. They show that $k = 1$ (a single expert) performs best and they design a load balancing loss term for the gating function.

Conditional computation during inference requires quick decision making at low overhead. Runtime Neural Pruning [Lin et al. 2017], uses a Markov decision process to determine the path through the network. Its parameters learned by a reinforcement learner, the path through the network is determined at inference time. Here specifically, the agent determines which channels are important to be considered for a specific input. During training, two networks are trained in tandem: the original ("backbone") convolutional network and the decision network that guides filter selection at runtime. Chen et al. [2020] show a reinforcement learner used at runtime to select convolutional channels during runtime with low storage. Several similar approaches use gating modules [Liu and Deng 2018] or routing [Rosenbaum et al. 2017].

Other similar approaches, such as product key networks [Lample et al. 2019] increase model capacity without increasing the computation using key-value store-like memory layers. This vast topic of dynamic memory networks is outside the scope of this overview.

6 SPARSE DEEP LEARNING ARCHITECTURES

After describing the building blocks of sparsification methods, we continue to highlight specific applications and results that were achieved applying these methods. Many works prune for a specific goal such as performance/inference latency [Gordon et al. 2018], memory consumption [Li et al. 2020a], or energy efficiency [Yang et al. 2017]. There, several sparsifying techniques are often combined, for example, regularization and magnitude pruning [He et al. 2017; Yang et al. 2017]. Layer-wise sensitivity schemes or data-free methods can then be used to improve the performance further. Many schemes iterate over a mix of such techniques and their carefully engineered combinations with pruning schedules can result in impressive gains for specific purposes [Han et al. 2016b; Yang et al. 2017].

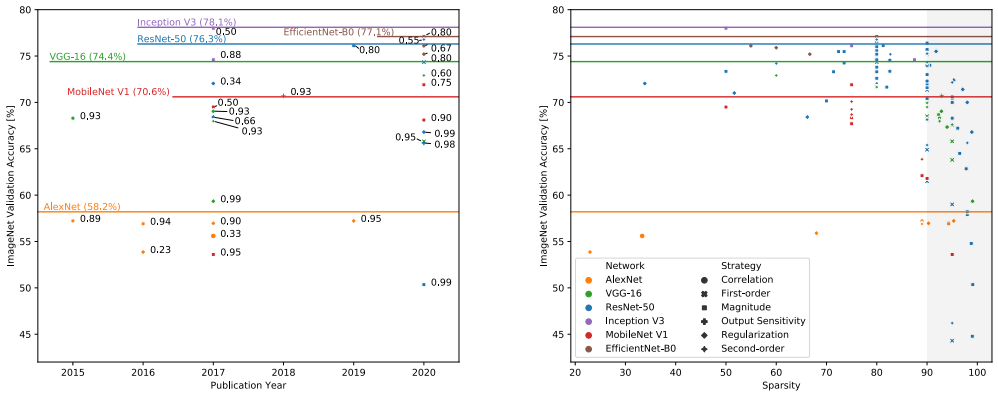
Each methodology represents a combination of specific elements to sparsify, a sparsification schedule, a removal method, and (optionally), a re-addition method. Each result is measured by the authors of the original work and can be reproduced through the description in the original paper. As pointed out by Blalock et al. [2020], these works do not always follow a consistent experimental discipline and thus many results are unclear, and may not be fully interpretable from both an accuracy and performance perspective [Hoefler and Belli 2015]. Thus, when comparing works, we rely on the author's results and only do so to provide a rough overview of the relative performance of these methods. Different setups and problem statements are likely to shift this balance—however, we believe that we can derive several important observation from the quantitative results.

Here, we focus on more recent results after 2015 when broader interest in deep neural networks emerged and works solved large-scale data-driven problems from computer vision, natural language processing, and related domains, that are still relevant today.

6.1 Sparsifying convolutional neural networks

CNNs with diverse structures have recently become the primary target for sparsification, and diverse architectures were successfully pruned. As opposed to MLPs, CNNs contain combinations of convolutional operators (Section 1.2.4), fully connected layers, skip connections, and other statistical normalization operators such as batch normalization. The composition of these operators determine which sparsification strategies would be effective. Convolutions are used from the inputs onwards to compute feature maps, whereas the fully connected layers are used as classifiers. The convolutional operators can be pruned in a structured or unstructured manner, but typically less than fully connected layers, due to the fewer and structured connections between the inputs and the output.

In Fig. 20 we see the development of accuracy in CNNs over time (Fig. 20a) and sparsity (Fig. 20b), where in the former we highlight the two extremes on the Pareto front of sparsity — best validation accuracy and highest compression ratio — for every year and every strategy. We see that over the years research was able to increase compression and accuracy at the same time, and the composition of pruning strategies (see Section 3 for details) changed, but magnitude constitutes the majority of the reviewed works. From Fig. 20b, we see that two regions emerge: dense to moderate sparsity (0–90%), and moderate to high sparsity (marked in darker background). In the lower compression ratios, magnitude-based pruning works relatively well (especially when iterative pruning is applied), achieving state of the art accuracy for all studied networks. However, when >90% sparsity is desired, regularization, first-order, and second-order sparsification yield the best networks in the sparsity-accuracy tradeoff. Below we review the history and methodology behind the papers shown in the figures. First, we focus on the convolutional operator and modifications to the CNN architecture. Then we discuss approaches for pruning CNNs and the derived training schemes.



(a) Best accuracy and sparsity over time

(b) Sparsity vs. accuracy

Fig. 20. Accuracy of pruned CNNs. Marker shape indicates pruning strategy and labels indicate sparsity.

6.1.1 CNN architecture evolution. Over-parameterization in convolutional operators was already noted by Szegedy et al. [2015]. In order to reduce the computational requirements and memory footprint of CNNs, the authors proposed the GoogLeNet architecture, using "Inception" modules that trade large convolution kernels with 1×1 convolutions and smaller convolutions following dimensionality reduction. This was later improved to chaining separable 1D convolutions instead of 2D in the "Inception V3" CNN [Szegedy et al. 2016], and with depth-wise separable convolution [Sifre and Mallat 2014] in the parameter-efficient MobileNets [Howard et al. 2017], both of which can be seen as handmade sparse formulations of convolutions. Kuzmin et al. [2019] provides a survey about structured compression of convolutions, including tensor decompositions, channel pruning, and probabilistic compression. A recent popular technique to reduce the size of CNNs and increase their parameter efficiency is Neural Architecture Search (NAS) [Tan et al. 2019], formulating the process as a meta-optimization problem. EfficientNet [Tan and Le 2020], the current state-of-the-art CNN for image classification, uses NAS to construct their base EfficientNet-B0 network, and defines a compound method to scale it up while retaining parameter efficiency.

6.1.2 CNN sparsification. In unstructured pruning, the popular paper on model compression by Han et al. [2016b] combines magnitude-based sparsification, quantization, weight sharing, and Huffman coding into a compression scheme able to reduce AlexNet and VGG-16 on the ImageNet dataset by 35× and 49×, respectively, without loss of accuracy. They were able to sparsify those models by more than 90% when manually tuning the sparsification level per layer. The authors show that convolutional layers should be sparsified less (15–65%) than fully connected layers (91–96%). Their compression scheme starts from a fully-trained baseline model, performing re-training to reach the original accuracy.

Using a scheme based on neuron output correlation, Sun et al. [2015] demonstrate 33% improved accuracy for the DeepID2+ face recognition CNN when sparsified by 74%, and retaining the same accuracy with sparsification of up to 88%. The authors also discuss the sparsification re-training scheme, showing that a fully-sparse training approach could not match the performance of the dense-trained, then sparsified network. They conjecture that, with a sparser model, the randomized initial values of the weights play a more significant role than in a dense model. Thus, training a sparse model is more prone to converge to suboptimal local minima than a dense network, agreeing with later proposed theory [Frankle and Carbin 2019].

Some works advocate for training the sparse network in tandem with the dense network, or by modifying the training process to promote sparsity. One example is the effect of dropout on sparse networks (see Section 5.2). Zhou et al. [2016] propose a forward-backward splitting method to enforce sparsity as regularization, pruning 61.3% of VGG-13 and 65.4% of AlexNet parameters with 1.7% and 0.53% accuracy degradation respectively. Tartaglione et al. [2018] use sensitivity-driven pruning until the network drops below the required accuracy. They achieve higher sparsity than earlier magnitude-based mechanisms. Molchanov et al. [2017] use variational dropout (see Section 3.7) to prune weights starting from relatively small pre-trained networks. For those networks, they show record sparsity levels for small networks: 98.5% for LeNet-300-100 and 99.996% for LeNet-5 with 98.1% and 99.3% accuracy on MNIST, respectively. Training takes twice the number of operations for forward and backward but converges equally fast on LeNet and MNIST. VGG-style networks on CIFAR-10 and CIFAR-100 could be sparsified by more than 97% at similar accuracy.

Dong et al. [2017] use 2nd order OBS pruning per layer to limit its computational complexity. They approximate the inverse of the Hessian matrix with the Woodbury matrix identity. The achieved compression ratios are similar or slightly better than magnitude-based pruning. However, they show that after applying 2nd order pruning, the resulting network (before retraining) has a much higher quality than the ones obtained with magnitude pruning (<5% vs. >80% error for LeNet and <50% vs. >73% error for VGG and AlexNet). This reduces the number of iterations needed to re-train the model to near-original accuracy (>200× less for LeNet, >40× less for AlexNet, and >12× less for VGG-16).

Guo et al. [2016] observe that the process of sparsification can benefit from re-adding weights during training that were erroneously pruned before. For this, they maintain the set of all weights during the whole training process, including the pruned ones, and mask them during the forward pass. This allows them to later re-add pruned weights if they reach a certain magnitude. Furthermore, they specify a pruning schedule to decrease the sparsification probability over time. They demonstrate that this method significantly improves upon earlier methods [Han et al. 2016b] that use iterative retraining — specifically, they show that the number of iterations to prune AlexNet can be reduced from 4.8M to 0.7M (6.9×) while improving the sparsity from 89% to 94% (2×). Using the same method, they compress LeNet-5 and LeNet-300-100 by 99% and 98%, respectively. They again show that convolutional layers that already share weights compress less (46-97%) than fully-connected layers (93-99%).

Despite prior claims against fully-sparse training schemes, and due to growing CNN memory footprints, several recent works attempt to improve such schemes to produce usable networks. Bellec et al. [2018] use a fully-sparse training schedule to enable training higher-dimensional models that would not fit in a dense configuration. They use a variant of magnitude based pruning and random weight addition and show that this method outperforms densely-trained methods if the target sparsity is very high (>95%). They showed that longer training leads to improving generalization. The paper also studies aspects of transfer learning and pre-training, in that the sparsified architecture quickly adapts to similar learning tasks. Mocanu et al. [2018] use a similar training schedule and show that it can improve accuracy while pruning by more than 96%. They also show that the degree distribution of sparsely learned connections follows a power law. Mostafa and Wang [2019] refines fully-sparse training for CNNs by automatically adjusting the parameter budget across layers. Their method may require more operations to converge than hand-tuned schedules, as sparsity may only slowly be redistributed to the later fully-connected layers. Their sparsely-trained models achieved significantly better performance than dense models of the same size. Dettmers and Zettlemoyer [2019] perform fully-sparse training and point out that parameter redistribution is especially important for larger layers. They use a cosine decay schedule for the

pruning rate across iterations and achieve similar performance with a 95% sparse VGG on CIFAR-10, and slightly outperform prior approaches with a 90% sparse ResNet-50 on ImageNet, achieving 72.3% accuracy, while reducing the required computations between 2.7× and 5.6×.

In more recent, parameter-efficient networks, state-of-the-art pruning techniques become more adaptive to the training process. Azarian et al. [2020] propose soft pruning, where sparsifying a weight is a continuously differentiable function, and L_0 regularization. The authors prune ResNet and EfficientNet-B0, where the latter attains 76.1% accuracy, compared with a 77.1% accuracy for the dense counterpart. He et al. [2019a] use a reinforcement learning approach combined with a CNN embedding scheme to prune the network. Their first-order approach to sparsification is able to sparsify ResNet-50 to 80%, keeping the same baseline top-1 validation accuracy of 76.1%. Evci et al. [2020] train networks fully-sparse with pruning based on magnitude and re-addition based on instantaneous gradients. They decay the sparsification probability using a cosine schedule and stop sparsification before the end of training. The method attains good generalization for ResNet-50, with 76.6% at 80% sparsity and 75.7% accuracy at 90% sparsity, while reducing the computations compared with dense training. Singh and Alistarh [2020] use second-order information (specifically, inverse Hessian-vector products using an approximation based on the empirical Fisher Information Matrix) to estimate which weights to prune. The authors report that with gradual sparsification, ResNet-50 can be pruned with no extra epochs to higher accuracies than existing approaches that do the same (76.8% at 80% sparsity). Gale et al. [2019] provide a systematic study of various pruning strategies: random (baseline), magnitude pruning, L_0 [Louizos et al. 2018], and variational Bayes [Molchanov et al. 2017] applied to ResNet-50 and transformer networks, ranging from 50–98% sparsity. Their main result is that simple magnitude pruning is competitive, if the pruning schedule and per-layer distribution is tuned (76.52% accuracy for 80% sparse ResNet-50 and 75.2% for 90% sparsity after 100 epochs). They report that variational dropout performs best only for very high sparsity (>95%), but tuned magnitude pruning remains close. They also show that both variational dropout and L_0-based pruning can be up to 3× slower and uses 2× more memory than magnitude pruning. Their general conclusion is that well-tuned magnitude pruning is probably the most practical pruning method.

Fig. 21 shows an overview of the computational intensity (flop count) for inference of sparsified ReNet-50 models. It shows that one can save between 50-80% of the operations without significant loss in accuracy, leading to a potential speedup of up to 5x.

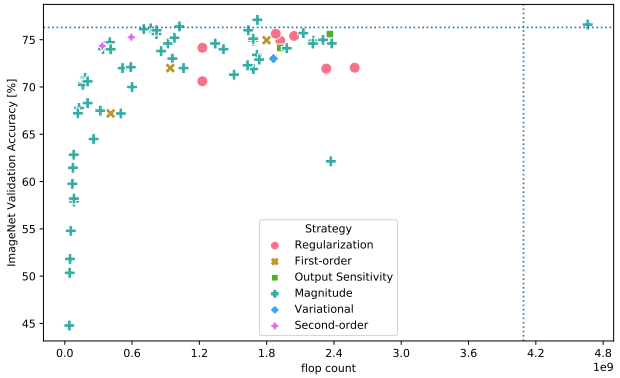
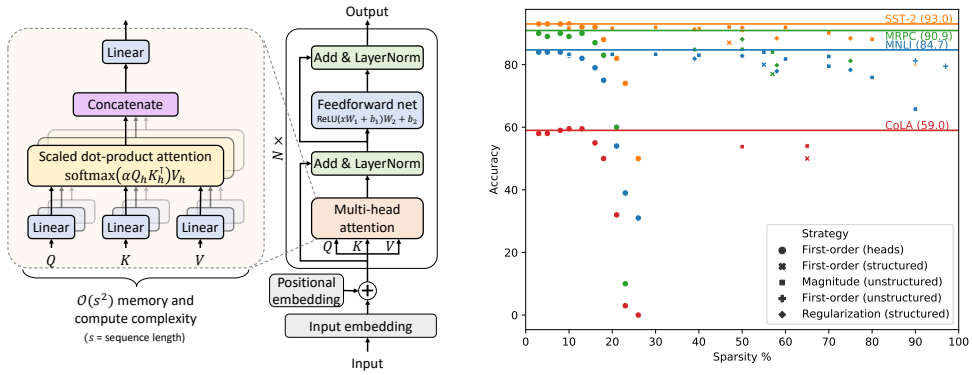


Fig. 21. Flop count and resulting accuracy of state-of-the-art pruning methods for ResNet-50 over ImageNet. Dotted lines represent dense baselines.

6.2 Sparsifying transformer networks

Transformers [Vaswani et al. 2017] are a class of sequence transduction models that have led to breakthroughs in natural language processing and are recently expanding into other fields such as computer vision [Dosovitskiy et al. 2021]. Widely used transformer models include the original Transformer [Vaswani et al. 2017] for language translation as well as language models such as BERT [Devlin et al. 2019] and GPT-3 [Brown et al. 2020]. The key idea behind transformers is to generalize prior work on shallow language embeddings to deep, multi-layer embeddings, while being more parallelizable in training than RNNs. Like CNNs, transformer architectures are a combination of a variety of operators, which we illustrate in Fig. 22a. The key primitive is multi-head



(a) Transformer architecture [Vaswani et al. 2017]. (b) Accuracy of pruned BERT-base on selected tasks.

Fig. 22. Overview of transformers.

attention, introduced by Vaswani et al. [2017]. Each attention "head" performs scaled dot-product attention to identify how elements of one sequence should relate to elements of another sequence. A transformer layer is then composed of a multi-head attention layer followed by a feedforward network (sometimes called "expert layers"), with layer normalization [Ba et al. 2016b] and residual connections. The full transformer network consists of one or more stacks of transformer layers, with embedding layers at the beginning.

Transformers are often very large, ranging from about 110 M parameters in BERT-base to hundreds of billions [Brown et al. 2020] or trillions [Fedus et al. 2021; Lepikhin et al. 2020] in the largest, best-performing models. As it is infeasible to deploy such large models in production situations, compression is critical. Indeed, Li et al. [2020a] show that training a large, over-parameterized transformer and then compressing it results in better accuracy than training a smaller model (e.g., a 75% sparse 24-layer RoBERTa model [Liu et al. 2019a] outperforms a 3-layer model on MNLI, while being the same size). Complementary to pruning, many other approaches to compressing transformers have been developed; Ganesh et al. [2020] provides an overview for BERT specifically, and Gupta and Agrawal [2020] for deep learning models for text in general.

Fig. 22b presents an overview of sparsity results for pruning BERT-base [Devlin et al. 2019] for four downstream natural language understanding tasks from the General Language Understanding Evaluation (GLUE) [Wang et al. 2019] benchmark: the Stanford Sentiment Treebank (SST-2) [Socher et al. 2013], the Microsoft Research Paraphrase Corpus (MRPC) [Dolan and Brockett 2005], the Multi-Genre Natural Language Inference corpus (MNLI) [Williams et al. 2018], and the Corpus of Linguistic Acceptability (CoLA) [Warstadt et al. 2019]. BERT-base consists of 110M parameters (including embedding layers), with twelve transformer layers, each with twelve attention heads.

Compared to results on pruning CNNs (Fig. 20), there are two qualitative differences: There are relatively few results with large accuracy degradation, and there are relatively few results with very high sparsity levels. This stems from much of the work on pruning BERT being focused either on understanding what the model has learned or on the Lottery Ticket Hypothesis (see Section 8.3). In many of these works, iterative pruning only continues while the pruned model remains close to the original accuracy.

We can observe several qualitative trends among methods, which generally agree with the results on CNNs. For very low sparsity levels, structured head pruning performs very well, but it rapidly degrades as important heads are pruned. At moderate sparsity levels (40–80%), unstructured magnitude pruning performs very well, and outperforms structured pruning. When >90% sparsity is desired, however, only the first-order movement pruning method [Sanh et al. 2020] reports results, and achieves high accuracy on MNLI.

6.2.1 Structured sparsification. There has been much study of the importance of different components of transformers; for BERT, this is referred to as "BERTology" [Rogers et al. 2021]. For example, while attention heads are important for training, several works showed that most of the heads can be pruned after training with only minor accuracy loss. Michel et al. [2019] and Voita et al. [2019] study the importance of heads in two concurrent and complementary works.

Voita et al. [2019] analyze the linguistic properties and importance of each head and conclude that specific heads take on specific roles, such as representing "positional", "syntactic", and "rare words" functions. Using a simple stochastic gating scheme [Louizos et al. 2018] to prune heads, they can remove 80% of heads and lose only 0.15 BLEU on a English-Russian translation task [Jan et al. 2019] and 92% of heads at a loss of 0.25 BLEU on OpenSubtitles [Lison et al. 2019].

Michel et al. [2019] show similar results with a first-order head importance score for pruning. Using an iterative greedy process to test model quality with each head removed, they are able to prune 20–40% of attention heads with an insignificant decrease in quality. They also find that the importance of heads is transferable across tasks and that the importance of heads is determined early in the training process, hinting that early structure adaptation may also apply to heads.

However, *multi-head attention layers account for only about a third of the parameters in BERT*, which limits the overall compression level, and for some tasks, Michel et al. [2019] show that pruning too many heads is detrimental to accuracy. Prasanna et al. [2020] extended the importance metric of Michel et al. [2019] to also prune entire feedforward networks in a layer, using a similar iterative pruning process that continues for as long as the model retains over 90% of the original's accuracy. With this, they show that BERT can be pruned to 40–65% sparsity on a variety of GLUE benchmark tasks. They also show that, for low sparsity, even random structured removal achieves good performance.

McCarley et al. [2020] evaluate a larger set of pruning approaches that can remove attention heads and slices of feedforward and embedding layers and use a gating mechanism $\alpha_j \in \{0, 1\}$ to select components for removal. They compare four techniques for pruning: (1) random pruning as a baseline; (2) a first-order "gain" metric that computes $g_i = |\partial L / \partial \alpha_i|_{\alpha_i=0}$ for each example; (3) a leave-one-out score, where the loss for each element removed is computed separately, and elements that cause a small loss on removal are retained; and (4) a sampled L_0 regularization. Finally, they apply distillation using the unpruned model as a teacher for the pruned model. The main finding is that L_0 regularization performs best and can prune 40–75% of the elements in BERT and RoBERTa models while loosing about 5 points F1 score on the Natural Question benchmark task [Kwiatkowski et al. 2019].

Wang et al. [2020a] use a modified L_0 regularization to prune all weight matrices in a transformer. For each weight matrix W, they first reparameterize it as a low-rank factorization $W = PQ$, and

then introduce a diagonal pruning matrix G, so that $W = PGQ$. The pruning matrix allows the model to learn to keep the best rank-1 components of the weight matrix. They use L_0 regularization to promote sparsity, with an additional term added to allow for control of the desired sparsity level.

Building on the idea that layers in transformers learn disparate tasks [Rogers et al. 2021] and that some layers may be less important than others [Tenney et al. 2019], two works have pruned larger-scale structures. Lin et al. [2020] prune entire residual blocks (i.e., either the entire multi-head attention layer or feedforward network) by identifying blocks whose nonlinear part has small activations. These blocks are then pruned and replaced by a simple identity map. To do this, they adapt ϵ-ResNets [Yu et al. 2018] and augment each residual block with a gating function if the non-linear component is less than ϵ. Once a layer's activations fall below ϵ, it will cease to contribute to the output, and its gradients will no longer be updated, leading to weight collapse. In a similar vein, Fan et al. [2020] introduce LayerDrop, a form of structured dropout that stochastically drops entire transformer layers during training. To reduce model size for inference, they also explore different ways to completely remove layers, and find that the simple approach of dropping the layers at depth d such that $d \equiv 0 \left(\mathrm{mod} \left\lfloor \frac{1}{p} \right\rfloor\right)$, where p is the dropout probability, performs best.

6.2.2 Unstructured sparsification. Simple iterative magnitude pruning has also been applied for unstructured sparsification, with several conclusions. Prasanna et al. [2020] compared it with structured pruning using Michel et al. [2019]'s first-order importance metric and found that unstructured magnitude pruning typically results in networks that are both smaller and retain better accuracy. Gordon et al. [2020] showed that a pretrained BERT model can be pruned to up to 40% sparsity without affecting the performance of downstream tasks, but beyond that performance begins to degrade. Surprisingly, they also show that fine-tuning a pretrained model and then pruning it does not result in better performance, and conclude that one need only prune BERT once after pretraining instead of for each downstream task. Chen et al. [2020] show a similar result in the context of the Lottery Ticket Hypothesis (see Section 8.3). They find that magnitude pruning can prune a pretrained BERT model to up to 70% sparsity without compromising performance on the pretraining objective, and that such networks transfer universally to downstream tasks. In contrast, they find that while pruning for a particular downstream task may result in higher sparsity levels without compromising performance on that task, such networks do not transfer as well.

Guo et al. [2019a] conduct experiments showing that using L_1 or L_2 regularization can cause divergence during training, and that the regularization should be decoupled from the gradient update, in line with prior work on optimization [Loshchilov and Hutter 2019]. To prune, they instead develop Reweighted Proximal Pruning, which uses reweighted L_1 minimization instead of regularization, and use a proximal algorithm to find the sparsity pattern, rather than backpropagation.

Sanh et al. [2020] argue that for transfer learning, what matters is not the magnitude of a parameter, but whether it is important for the downstream task. They introduce movement pruning (see Section 3.4), a first-order method which prunes parameters that shrink during fine-tuning, regardless of their magnitude. Movement pruning is able to achieve significantly higher performance than magnitude- or L_0-based pruning for very high levels of sparsity (e.g., 97% sparse), and can be combined with distillation to further improve performance.

6.2.3 Sparse attention. Scaled dot-product attention requires a dot-product between two sequences of length N ($QK^\top$), which produces an alignment matrix for the two sequences. Producing this matrix requires both $O(N^2)$ time and memory; as sequence lengths in transformers range from 128 to 2,048, this can be a large bottleneck. This, combined with the intuition that one does not need to compute full attention to get good model performance, has resulted in a large body of work on so-called efficient transformers. Tay et al. [2020] provide a survey of this field; we focus here

on sparsity. Recent work has also started to develop benchmarks focused specifically on efficient transformers [Tay et al. 2021].

Yun et al. [2020] provide broad theoretical results showing that $O(n)$ connections in an attention layer is sufficient to universally approximate any sequence-to-sequence function if the following properties are met: (1) every token attends to itself; (2) a chain of connections covers all tokens; and (3) each token connects to all other tokens after a fixed number of transformer layers. This provides a rigorous basis for the intuition that each input token need only be able to route to each other token through successive layers.

Many approaches to sparse attention satisfy these requirements by sparsifying the $QK^\top$ computation, including restricting attention to local neighborhoods [Parmar et al. 2018], star topologies [Guo et al. 2019b], combinations of strided and fixed sparsity patterns [Child et al. 2019], sliding windows [Beltagy et al. 2020], and local attention plus a fixed number of tokens that attend globally [Zaheer et al. 2020]. SAC [Li et al. 2020] learns a *task-specific* sparsity structure using an LSTM edge predictor.

The SoftMax computation in each attention head can also be modified to maintain its ranking while inducing sparsity and satisfying the above requirements. Zhao et al. [2019] take a direct route and apply top-k sparsification to the attention weights. The sparsity patterns can also be learned directly using generalizations of SoftMax, such as α-entmax [Correia et al. 2019] or sparsegen-lin [Cui et al. 2019]. These build on earlier work, predating transformers, that aimed to induce sparsity in attention mechanisms to either improve performance or interpretability, including sparsemax [Martins and Astudillo 2016], constrained sparsemax [Malaviya et al. 2018], and fusedmax [Niculae and Blondel 2017].

7 SPEEDING UP SPARSE MODELS

Sparse networks do not always execute faster than dense networks using current machine learning frameworks on today's hardware. Sanh et al. [2020] demonstrate that small dense models often perform faster on current hardware than sparse models of the same and even smaller size despite the generally higher accuracy and parameter efficiency of sparse models. Han et al. [2017] show that even 90% sparse workloads execute slower on a GPU than computing 90% zeros densely and Yu et al. [2017] show that an 89% sparse AlexNet executes 25% slower on CPU than its dense version. In general, unstructured sparsity is not well supported on today's architectures. Some cases of structured sparsity can be mapped to dense matrix operations (e.g., neuron, filter, or head sparsity) and can thus trivially use existing optimized frameworks or libraries such as cuDNN [Chetlur et al. 2014]. Other structured sparsity approaches such as blocks of weights would require support from the frameworks to be executed efficiently. We will discuss algorithmic and hardware solutions to support sparsity on practical systems in this section.

Training for sparsity can be especially expensive on some architectures. For example, regularization methods (e.g., Louizos et al. [2018]), schemes using gating variables (e.g., Mozer and Smolensky [1988]), and various other techniques [Molchanov et al. 2017; Sanh et al. 2020] double the number of trainable parameters. Furthermore, variational methods are often expensive in both memory and compute during training [Gale et al. 2019]. Those techniques may be even slower than fully dense training in a sparsified training schedule. Thus, we recommend a careful analysis of memory, data movement, and computational overheads when considering the performance of a method (see Ivanov et al. [2020]).

7.1 Algorithmic and software support for sparse models

Sparse computations have a long history in the context of linear algebra and scientific computing. However, sparsities in those fields are often two orders of magnitude higher than in today's deep

learning (> 99.9% vs. 50 – 99% [Gale et al. 2020]) and it was long considered not beneficial to attempt sparse computations on less than 99% sparse matrices. Furthermore, many scientific computing workloads have close-to banded non-zero patterns that can often be compressed as hierarchical matrices. Those structures lead to high temporal locality and many libraries such as Intel's MKL are tuned for those patterns [Park et al. 2017]. As we will see below, sparsified neural networks have different characteristics in their non-zero distributions. Thus, scientific computing kernels such as the sparse BLAS or cuSPARSE are only optimized for scientific computing workloads and supported formats aimed at high sparsities such as compressed sparse row. We do not cover the many elegant approaches developed for very high sparsity here—albeit they may become very relevant to sparse deep learning if the trend to higher sparsity continues. Instead, we focus on approaches developed for sparsity levels observed in today's deep learning workloads.

We describe basics of storing unstructured sparse matrices in Section 2.2. Many practical schemes use run-length or delta-encoding with padding for offsets [Han et al. 2016a]. Furthermore, it is common to combine quantization with index storage to achieve aligned number formats. For example, Han et al. [2017] pack a 12-bit integer value with a 4-bit index value into a 16-bit element that is naturally aligned to DRAM page and PCIe transaction boundaries. This format would store the sparse vector $v = [0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 3, 2, 0, 0, 0]$ as $v' = [2|1, 15|0, 0|3, 0|2]$, where, for example, the padding entry "15|0" decodes to 15 (first 4 bits) zeros followed by the value 0 (last 12 bits). Sparse weights are often stored column-wise for inference using a compressed sparse column (CSC) format to facilitate the sparse matrix-vector multiplication. Gale et al. [2020] show various techniques to tune such sparse computations to GPU architectures.

Park et al. [2017] tune unstructured sparsity for convolutional layers by implementing an optimized sparse-with-dense matrix multiplication. They only consider sparsity in the convolutional kernels and not in the activations, which, despite of up to 85% sparsity was slower than sparse-dense in their experiments. Using a simple but effective performance model, they guide the sparsification such that the resulting model achieves highest performance. While they demonstrate their approach in conjunction with dynamic network surgery [Guo et al. 2016], it is applicable as a regularization or direct performance metric to many, if not most of the sparsification approaches discussed in Section 3. A general observation is that there is a range of sparsity where workloads can efficiently utilize CPUs: too low sparsity leads to high overheads managing it but also too high sparsity leads to a performance reduction on CPUs. This is due to the fact that higher sparsity increases the relative storage overhead of the index structure and decreases the relative compute load. Since CPUs have a fixed ratio of memory bandwidth per compute operation, too high sparsity will underutilize the compute units and be bound by the well-known data locality and movement bottlenecks [Ivanov et al. 2020; Unat et al. 2017]. This implies that an accelerator needs to be carefully tuned to the expected workload, making a detailed co-design between the data science aspects of sparsification and the engineering aspects of representation and dataflow mandatory.

7.1.1 Structured sparsity. Various sparsity structures have been used in the deep learning context to manage the storage overhead. They vary from completely unstructured storage where the offset for each single element needs to be encoded to structured storage formats that only store offsets of blocks or other elements arranged with a fixed structure. In Section 2.2, we analyze the storage complexity in terms of the number of parameters needed to describe the structure of an irregular matrix. A blocked format with block size B would reduce the storage overhead by a factor of B. Fig. 23 shows an overview. Blocked formats can be defined for any set of dimensions, the figure shows a one-dimensional format with blocks of size three and a two dimensional format with blocks of size 4 (2×2) as used in Cambricon-S [Zhou et al. 2018]. Here, the offsets are only stored once for each block of non-zeros. Another promising format is block-balanced. This format specifies a

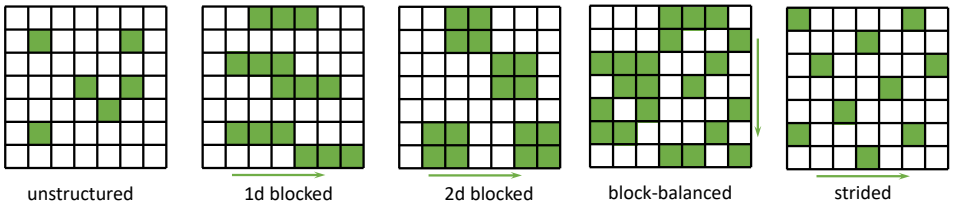


Fig. 23. Overview of sparse structures in deep learning—non-zero elements are colored.

fixed block size and a fixed number of non-zeros per block. Here, one would only need to encode the offsets of the non-zeros for each fixed-size block. Nvidia's Ampere microarchitecture [Nvidia 2020] uses a bitmap to store the non-zeros in blocks of size four with 50% sparsity. The figure above shows blocks of size seven with exactly three non-zero elements each. The strided format [Anwar et al. 2017] in Fig. 23 shows the most compact but also most constrained format. It fixes each 5th element of the matrix to have a non-zero value and all others zero, leading to a constant-size representation. In general, sparse matrix storage formats can use arbitrary encodings to minimize the representational overhead. MPI datatypes [Gropp et al. 2014] form an elegant hierarchy with clear performance properties [Gropp et al. 2011] and can provide further inspiration for specific designs.

7.1.2 Tuned block sparsity in practice. Elsen et al. [2019] demonstrate speedups with sparse representation for inference on mobile devices. They focus on single-weight and weight-block sparsity and optimized implementations of CNNs on ARM CPUs and WebAssembly and release the XNNPACK library. They primarily focus on a medium sparsity range between 70–95% and they optimize for caches and data movement, which has been shown to be a major bottleneck in deep learning systems [Ivanov et al. 2020]. They investigate the influence of various block-sizes on model accuracy and show that the shape of blocks (e.g., 1×4, 2×2, or 4×1) is irrelevant and just the size matters. They also show that larger models suffer less from large block sizes.

Scalpel [Yu et al. 2017] combines weight-blocking and neuron pruning into a scheme to support SIMD platforms. They sparsify weights in blocks the same size as the width of SIMD units and use a modified CSR format for storing the sparse weights. They prune by root mean square magnitude of weight blocks. For ARM microprocessors, their pruning scheme reduces the necessary sparsity required to achieve a speedup from 70% to 50% and on Intel CPUs, their scheme reduces the necessary sparsity from 50% to less than 10%. DeftNN [Hill et al. 2017] optimizes a whole row or column pruning scheme based on similarity for inference on GPUs achieving a 1.5x speedup.

Han et al. [2017] introduce block-balanced pruning that restricts blocks ("sub-matrices") to the same sparsity ratio. Thus, when loading blocks in parallel, the accelerator can process them in approximately the same time avoiding load imbalance. They find such pruning does not reduce the model quality significantly for large enough blocks. In an even simpler approach, Dey et al. [2019] fix the degree of each neuron in an MLP, leading to balanced row and column sparsity of the weight matrix.

PruneTrain [Lym et al. 2019] focuses on accelerating training using a group-lasso regularization method and pruning during training. The authors mention that the freed memory from pruning can be reinvested during the training to increase the minibatch size. This fits well into existing training schedules [Smith et al. 2018].

Mao et al. [2017] specifically analyze the impact of structured sparsity on the accuracy of CNNs. They consider four levels of increasing structure in convolutional layers: (0) unstructured weight sparsity, (1) dimension-wise (blocked) weight sparsity, (2) kernel-level sparsity, and (3)

filter-level sparsity. When considering storing the weights array as $W[C, K, H, W]$ (Channel, Kernel, Height, Width), then each of the four levels would require the following addressing per element: (0) $W[C, K, H, W]$, (1) $W[C, K, H, :]$ or $W[C, K, :, W]$, (2) $W[C, K, :, :]$, and $W[C, :, :, :]$. They show that, for a simple magnitude-based (sum per block) pruning scheme, the top-5 accuracy degrades with increasing block size at sparsity levels of 60–77%.

7.2 Hardware acceleration for sparse models

Numerous hardware accelerators have been designed to accelerate deep neural networks, see [Reuther et al. 2020; Sze et al. 2017] for an overview. Here, we focus on a summary of important techniques implemented in hardware accelerators that have *explicit support for sparse computations* in deep learning. Dave et al. [2020] provide a comprehensive and generic survey including more architectures, techniques, and technical details on this topic. Accelerator designs are based on the observation that typical workloads have 50–90% ephemeral activation sparsity and up to 99% weight sparsity. Activation sparsity is either induced by ReLU operations or autoencoders [Noh et al. 2015] and generative adversarial networks [Goodfellow et al. 2014b] that insert zeros in the upsampling phase of the decoder. Furthermore, as we outline in the previous sections, weights can often be structurally sparsified to 95% (or more) without significant loss in accuracy.

7.2.1 Inference accelerators. We start with an overview of sparse inference accelerators that typically aim at latency-sensitive batch sizes of one where the central operation is sparse matrix-vector (SpMV, or weight-activation) multiplication. Similarly to dense DNN accelerators, sparse accelerators can achieve record performance up to nearly 22 TOp/W [Zhang et al. 2019a]. Different layer types can be expressed in terms of a small number of primitives. For example, fully-connected layers can be expressed as (sparse) matrix-vector multiplication. Convolutional layers can similarly be expressed as sparse matrix-vector or matrix-matrix multiplication [Lavin and Gray 2016]. Conversely, a 1×1 convolution can be expressed as a fully-connected operator. Similarly, recurrent layers can be unrolled into a series of matrix-vector multiplications. So any device that can process a convolution *or* fully-connected layer can process all layers. Yet, accelerators are tuned for the specifics of layer types and network architectures. Thus, we structure our overview by different architectures.

Sparse CNN inference accelerators. We start with an overview of sparse CNN accelerators (including fully-connected layers). Minerva [Reagen et al. 2016] uses a hand-tuned threshold to prune small activation values in MLPs that save weight-fetch bandwidth and arithmetic operations. This saves 50% of the power consumption on top of other optimizations such as quantization. Eyeriss [Chen et al. 2017] clock-gates PEs that would process zero activation values of convolutional layers to save energy.

Han et al. [2016a] show Efficient Inference Engine (EIE), an inference architecture optimized for sparse models with parameter sharing. Their architecture supports both sparse matrices as well as sparse activation vectors and aims at fully-connected layers in CNNs. To enable fine-grained parallelism, they distribute the columns of the weight matrix to the processing elements (PEs). At the input, they scan the activations for non-zero entries and broadcast them to all PEs, where they are accumulated into a local partial sum. They balance the load through queues at the PEs that buffer non-zero activation values to avoid synchronization. The authors showed empirically that a queue depth of four values is sufficient to achieve good load balance. Finally, the output activations are summed and compressed through a hierarchical non-zero detection tree. Their silicon implementation is 13x faster and 3,400x more energy efficient than an Nvidia Pascal Titan X GPU.

Zena [Kim et al. 2018] introduces a scheme that uses both weight and activation sparsity for convolutional layers. Other sparse DNN accelerators such as Cambricon-X [Zhang et al. 2016], SCNN [Parashar et al. 2017], Eyeriss v2 [Chen et al. 2019], and Cnvlutin [Albericio et al. 2016] use a combination of similar ideas to achieve between 2–15x speedups and 2–10x lower energy consumption. Niu et al. [2020] design an FPGA-based accelerator for the spectral processing (based on FFT and Hadamard products in the frequency domain) of sparse convolutional layers. They keep the input activations in SRAM and stream the sparse kernels. A similar design [Niu et al. 2019] streams activations with stationary weights. Both have limited reuse due to the limited BRAM (on-chip SRAM) on FPGAs. Both store weights (kernels) in COO format arranged in device DRAM and Niu et al. [2020] uses a covering algorithm to optimize locality.

Sparse RNN inference accelerators. A second class of accelerators aims at sparse recurrent (RNN, LSTM) inference accelerators. Han et al. [2017] later show Efficient Speech Recognition Engine (ESE), an FPGA accelerator design for LSTM models using block-balanced sparsity for load balancing. ESE stores (sparse) activations in fast memory with the (sparse) weights being streamed, while the (dense) output is accumulated into a fast output memory. Their overall design achieves 3x performance and 11.5x energy efficiency improvements on a Xilinx Ultrascale (XCKU60) FPGA compared to an Nvidia Pascal Titan X GPU. Those systems are designed for the typical cases of 50–70% activation sparsity as well as 90% weight sparsity. MASR [Gupta et al. 2019] proposes an ASIC design for sparse RNNs as used in speech recognition. They exploit sparsity in weights and activations and different from EIE, they use a bitmask scheme to store indices using a relatively moderate sparsity of 66%.

Predicting sparsity in the results. Most accelerators utilize either sparsity in the input activations, in the weights, or both. However, one could also aim to predict sparsity in the output activations (i.e., the result of the computation). SparseNN [Zhu et al. 2017] show that such a prediction scheme can improve performance by up to 70% while halving power consumption. The key technique is a light-weight prediction of the non-zero pattern in the output. LRADNN [Zhu et al. 2016] use Singular Value Decomposition (SVD) of the weight matrix: $W = UV$, where $W \in \mathbb{R}^{m \times n}$, $U \in \mathbb{R}^{m \times r}$, and $V \in \mathbb{R}^{r \times n}$, where U and V are the first left- and right-singular vectors, respectively. The prediction is then simply performed by computing a mask $m = sgn(UVx)$ for the input activations x. For small enough r, the computation can be 20x faster than evaluating the full layer and the generated sparse output mask can be used to avoid computing zero elements. SparseNN [Zhu et al. 2017] improves upon this scheme by learning U and V through back propagation. They estimate the derivation of the sign function with a well-known straight-through estimator (see Section 3.6.1).

Fixed block sparsity and systolic arrays. Block sparsity (either blocks of weights or full neurons) reduces the overhead for indices and control logic and thus can efficiently be used to optimize software for any hardware type (see [Yu et al. 2017]). Cambricon-S [Zhou et al. 2018] adds explicit support for block sparsity to the Cambricon series of accelerators. They skip both zero neurons and blocks of weights for arbitrary layer types. They observe that large weights tend to cluster in 2D and 4D in fully-connected and convolutional layers, respectively. Based on this observation, they define 2D and 4D block-sparse weight formats. The block sizes are tuned as hyperparameters and the authors observed that permissible block sizes for ResNets are particularly small where other ("fatter") networks allow bigger blocks. They show 1.7x better performance and 1.37x better energy efficiency than the fine-grained Cambricon-X accelerator.

Most of the sparse accelerator architectures define logic that feeds each unit separately. However, most dense accelerators use systolic arrays to perform the matrix operations. Yet, sparsity would not use those fixed structures of systolic arrays efficiently. Kung et al. [2018] pack sparse filters

into a dense structure to use efficient systolic arrays for sparse computations. They first select columns of sparse filters/weights that have minimal overlap between the non-zero elements. Then, they combine those into a single column retaining the largest values. For example, consider the following four columns: $c_1 = [0, 2, 0, 3, 0]$, $c_2 = [1, 0, 2, 1, 0]$, $c_3 = [1, 2, 0, 1, 0]$, and $c_4 = [0, 0, 0, 0, 4]$. If we were to pack three columns, we would select c_1, c_2, and c_4 with the minimal overlap and pack them into the single dense column $c_p = [1, 2, 2, 3, 4]$—note that only the 4th index conflicts in c_1 and c_2 and the large value is chosen. The group size and number of allowed conflicts are hyperparameters. The authors show that this scheme, combined with a moderate amount of retraining is efficient for small networks. Squeezeflow [Li et al. 2019] use a conceptually similar scheme to compress sparse filters. Instead of combining different filters, they decompose sparse convolutions into effective and ineffective and map the effective ones to a dense representation for efficient processing. Compact [Zhang et al. 2019b] regularizes sparse runlength-encoded activations to be processed in a systolic array.

7.2.2 Training accelerators. Since the (inference) forward-pass is a part of training, one could assume that accelerators designed for inference can also be used in the forward pass of training. While this is partially true, it comes with additional complications. For example, during training, one needs to store the activations. Furthermore, specialized formats such as EIE's CSC format cannot easily be accessed (in transposition) during the backward pass. Thus, specific accelerators are designed for sparse training. Yang et al. [2020a] show a co-design approach of a sparse training algorithm and hardware to design Procrustes, an accelerator specific to the Dropback pruning algorithm [Golub et al. 2019]. They observe that batch normalization layers "shift" values away from zero and essentially eliminate sparsity in the gradients. Procrustes thus exploits only structural weight sparsity by storing weights in a compressed block format. Their design is up to 4x faster and 3.36x more energy efficient than traditional dense training accelerators. Zhang et al. [2019] use the observation of early structure adaptation together with an iterative pruning schedule with magnitude pruning to accelerate training by up to 40%.

More generic accelerators such as SparTen [Gondimalla et al. 2019] and SIGMA [Qin et al. 2020] are not specialized to particular layer types and focuses on general sparse matrix-vector products. Both architectures can support arbitrary reuse of matrices or their elements and both are using (blocked) bitmap storage to implement sparse vector products. Thus, their architecture is not specific to any layer type. Yet, the sparse matrix storage format determines ranges of sparsity where it performs most efficiently (see Section 2.2). The used bitmap format performs best for relatively dense operations. Similarly, Nvidia's Ampere micro-architecture supports "structured sparsity" to accelerate the processing of blocks of four values with up to 50% sparsity [Nvidia 2020].

All those architectures are designed for the relatively modest sparsity in today's deep neural networks. One could expect new breakthroughs to enable higher sparsity closer to those in scientific computing (>99.9%). Then, another class of accelerators, such as SpArch [Zhang et al. 2020], Indirection Stream Semantic Registers [Scheffler et al. 2020], or Extensor [Hegde et al. 2019] would play a bigger role.

7.2.3 Overview of accelerators for sparse deep learning. Table 1 shows an overview of existing accelerator architectures with sparsity support. Most accelerators are designed for inference and most can also be used for the feed-forward pass during training—albeit not always efficiently. We underline accelerators that are specifically designed for training.

Some accelerators aim at either sparse matrix vector (SpMV) or sparse matrix matrix (SpMM) multiplications that can be used for several layer types (e.g., fully connected, convolutional using the Winograd scheme, or RNNs). Others are optimized specifically for convolutional or recurrent layers. The second column of the table (**Ops**) shows the operation (layer) types that the accelerators were

Accelerator	Ops	w mem	y mem	y comp	Load Balancing	Reuse
Cnvlutin [2016]	conv	-	COO*	x	group neurons	output
EIE [2016a]	FC	CSC	-	x	activation queuing	output
Minerva [2016]	FC	-	-	x	N/A	-
Cambricon-X [2016]	SpMM	CSC	-	x	N/A	output
Eyeriss [2017]	conv	-	-	x	-	row
ESE [2017]	LSTM	CSC	-	x	block-balanced	output
SCNN [2017]	Conv	CSC	CSC	x	N/A	input act.
SparseNN [2017]	FC	-	-	x	N/A	N/A
Cambricon-S [2018]	SpMM	COO	-	x	group output neurons	output
Zena [2018]	Conv	BM	BM	x	dynamic group alloc.	N/A
Eyeriss v2 [2019]	SpMM	CSC	CSC	x	activation queuing	row
SparTen [2019]	SpMM	BM	BM	x	precomputed greedy	output
MASR [2019]	RNN	BM	BM	x	dyn. act. assignment	N/A
SPEC^2 [2019]	Conv	COO	-	-	-	weight/kernel
Eager Pruning [2019]	SpMM	BM	-	x	dynamic output	weight
Spectral CNN [2020]	Conv	COO	-	-	-	input act.
Sigma [2020]	SpMM	BM	BM	x	-	input/weight
Procrustes [2020a]	SpMM	CB	-	-	split minibatch for LB	minibatch

Table 1. Overview of accelerators for sparse deep learning; those with explicit training support are underlined.

optimized for explicitly (FC = fully connected, LSTM = Long Short Term Memory, RNN = Recurrent Layers - all SpMV and Conv = Convolutional Layer - all SpMM via im2col). If an accelerator aims at both FC and Conv, we mark it with SpMM as a superset. As mentioned above, most accelerators can process all layer types at varying efficiency.

The column "**w mem**" lists the storage scheme for weights. A "-" means that weights are stored densely and zeros are computed explicitly. The columns "**y mem**" and "**y comp**" list whether activations are stored compressed and whether they are computed. Some accelerators store zeros but filter them before the computation engine. When we write CSC (Compressed Sparse Column), we include runlength encoding even though, in some special cases, the column offsets are managed outside the format. Cnvlutin uses a blocked COO format and Procrustes uses a compressed block (CB) format. The last two columns show specific techniques for **load balancing** and **reuse**. They are described in the section above and listed as a summary.

8 DISCUSSION

We do not understand all details of the inner workings of deep neural networks and how pruning influences them. Specifically, why can networks be pruned and what is the best pruning methodology remain as open questions. In this section, we provide a set of hypotheses, intuitive explanations, and possible assumptions to foster our understanding of the landscape and the characteristics of this gap in understanding. All those are speculation and intended to help readers to develop a better feeling for the area as well as inspire new research directions that could shed more light onto aspects of sparse neural network science and methodology.

A general observation in most works is that sparse models outperform dense models given the same parameter budget. Some works even show that sparse models outperform dense models with larger number of parameters [Elsen et al. 2019; Lee et al. 2020a]. A similar set of observations seems obvious but is worth stating: pruning is most efficient for architectures that are overparameterized. Some authors state that switching to a better architecture may be more efficient than pruning [Blalock et al. 2020]. This implies that, when showing relative pruning rates (e.g., 99%),

one should always consider the degree of over-parameterization or what we call the "parameter efficiency" (see Section 8.7 and "Rule 1" in [Hoefler and Belli 2015]).

8.1 Relation to Biological Brains

Throughout the document, we have used many metaphors linking approaches to biological brains, whose structure inspired the general idea of all neural networks. While such metaphors can be very useful to build an intuition and provide a possible direction, they have to be considered carefully. Biological brains and computers work with fundamentally different compute substrates. For example, the three-dimensional arrangement of the brain encodes structural information nearly for free and learns through neuroplasticity. Silicon devices cannot adapt their wiring structure easily, and thus the simulation of structural properties leads to overheads in terms of memory (encoding the structure) as well as compute (controlling the execution). It is thus possible to design mechanisms that are not common in biological systems but outperform biologically more plausible mechanisms in silicon-based compute substrates and architectures. After all, not many animals have wheels and airplanes do not flap their wings. Nevertheless, Leonardo da Vinci discovered the principle of dynamic soaring by studying birds.

A successful method to guide innovation is to be inspired by biological phenomena and engineer systems in a refinement and optimization step given our technical understanding of the problem. For example, the visual cortex does not utilize weight sharing like convolutional networks do, however, in silicon, it seems to be the most efficient technique given that weight sharing reduces redundancy during feature detection and enables reuse for performance [Unat et al. 2017]. A second example could be the optimization process. While we currently use SGD to train networks, it remains unclear whether biological brains use similar methods. Recent discoveries have shown a possible relationship to Hebbian learning [Millidge et al. 2020] and argue that SGD may be biologically plausible, albeit some gaps remain [Lillicrap et al. 2020].

Various pruning approaches have been directly inspired by biological brains [Ahmad and Scheinkman 2019] but have not demonstrated state of the art results for large-scale networks and complex tasks. They advocate sparse high-dimensional representation spaces. Biological brains have very large sparse layers in a relatively shallow architecture with less than ten layers. We believe that this is a very interesting direction for further exploration and inspiration if it is augmented with theoretical reasoning and solid engineering.

8.2 Permutation Groups and Information Loss

One interesting observation is that every parameterized dense network is an element in an exponentially large equivalence class, which will generate the same output for each input. Specifically, Changpinyo et al. [2017] prove the following lemma: "any dense convolutional neural network with no cross-channel nonlinearities, distinct weights and biases, and with l hidden layers of sizes n_1, n_2, ..., n_l, has at least $i = \Pi_{i=1}^{l} n_i!$ distinct equivalent networks which produce the same output." This suggests that the information content of sparsified networks may be exponentially larger. Ahmad and Scheinkman [2019] show a similar result with respect to noise robustness in high-dimensional vector spaces.

8.3 Sparse subnetworks for training and lottery tickets

Some works hinted at specific subnetworks that may exist during training which could lead to a good sparse structure [Cohen et al. 2017; Sun et al. 2015]. See et al. [2016] demonstrated that re-training a sparse RNN with the same structure results in networks that perform well but not as well as the pruned-from-dense variants. Frankle and Carbin [2019] analyze the relation between initialization and statically sparse training. They state the "Lottery Ticket Hypothesis": "dense,

randomly-initialized, feed-forward networks contain subnetworks (winning tickets) that—when trained in isolation—reach test accuracy comparable to the original network in a similar number of iterations." For shallow vision networks, they find winning tickets by magnitude pruning and show that re-training them with static sparsity starting from the initial weights, they reach similar or higher accuracy in the same number of iterations. They also demonstrate that random initialization, with the same structure, does not suffice. Zhou et al. [2020] empirically show that one may not need the exact weights at initialization to train lottery tickets but the signs may be sufficient.

8.3.1 Pruning is all you need - networks without weight training. Several researchers argue that initial subnetworks with their random weights can perform well [Ramanujan et al. 2020; Zhou et al. 2020]. Furthermore, winning tickets already identify sub-networks with non-random accuracies even without training. In fact, training to find such a "supermask" can produce a network that achieves 93.5% accuracy in MNIST and 65.4% accuracy on CIFAR-10 at around 50% sparsity without changing the random initial weights. In "pruning is all you need", Malach et al. [2020] prove that, with high probability, any network can be approximated with ϵ accuracy by pruning a polynomially larger network. This means that pruning could be used to train a network without changing the weights at all. Orseau et al. [2020] and Pensia et al. [2020] later prove that a logarithmically larger network (except depth) suffices. Specifically, any ReLU network of width n and depth d can be ϵ-approximated by sparsifying a $O(\log(nd))$ wider and two times deeper random network, with high probability [Pensia et al. 2020].

8.3.2 Lottery tickets in large networks. Analysis of the original lottery ticket hypothesis already indicated problems with larger CNNs, which could be fixed with a decreased learning rate. Liu et al. [2019b] showed that with the best learning rate for larger networks, keeping the original initialization does not improve the final accuracy over random initialization. Gale et al. [2019] also could not reproduce the hypothesis for larger networks. Frankle et al. [2020b] later argue that the hypothesis ("with rewinding") applies also to larger networks if one uses the values after some initial optimization steps at iteration r. They demonstrated that 0.1–7% of the total iterations are sufficient for 50–99% sparse networks. In line with early structure adaptation (see Section 2.4.2), they conclude that early pruning could be a promising approach. However, finding the right r remains tricky and the authors investigate the influence of "noise" through the ordering of batches on the training process and result [Frankle et al. 2020a]. Specifically, they investigate the difference in test accuracy for a model that is a smooth interpolation between two models trained with different orders. They consider networks with small such error and allow the orders to only diverge after iteration r. The empirical results show that r relates to the iteration for which a working lottery ticket can be derived by rewinding. Frankle et al. [2020c] empirically analyze the early iterations of training large networks.

Renda et al. [2020] compare the standard single-shot fine-tuning after pruning to "weight rewinding". Rewinding resets the weights after pruning to the values of a previous SGD iteration i. Then, they retrain (fine-tune) with the same learning rate schedule (from the original iteration i) in a process called "Iterative Magnitude Pruning". A modification to the scheme simply uses the same learning rate schedule but without resetting the weights. However, both Savarese et al. [2020] and Chen et al. [2020] find rewinding to be less efficient than fine-tuning from the most recent weights for image recognition and natural language processing tasks. They show for a variety of medium-sized ResNets and GNMT as well as BERT that weight rewinding outperforms fine-tuning but is itself outperformed by just rewinding the learning rate to the first iteration. Ding et al. [2019b] found that a simple selection based on 1st order information outperforms the simple magnitude-based scheme. Morcos et al. [2019] show that lottery tickets can transfer across different datasets and optimizers. A general conclusion could be that fully sparse training is possible (see

Section 2.4.3), especially if applied iteratively (see Section 2.4.6) but rewinding has not been proven effective.

8.4 Structured vs. unstructured pruning

Several works found that unstructured/fine-grained (e.g., weight) pruning maintains a better accuracy per element than structured/coarse-grained (e.g., filter, neuron) pruning [Gomez et al. 2019; Han et al. 2016b; Lee et al. 2020a; Ullrich et al. 2017]. However, structured pruning approaches achieve much higher computational performance on modern devices [Lym et al. 2019; Wen et al. 2016]. Thus, structured sparse models could afford a higher number of iterations to train and more floating point operations during inference to achieve the same overall efficiency/cost. Furthermore, unstructured sparsity has a higher relative representational overhead of indices for each fine-grained element as discussed in Section 2.2. It remains to be seen what level of granularity will be most efficient for the coming computer architectures.

We also observe that random pruning at network initialization works significantly better for neurons and filters than for weights [Gomez et al. 2019]. For neurons and filters, most works that nearly reproduce the state of the art are achieved with post-training sparsification, indicating that this form of architecture search is efficient. This is also intuitive because the specific location of neurons or filters no standard fully-connected and convolutional layers is irrelevant. For weights, this very structure matters and thus random pruning at initialization performs generally worse. Thus, we recommend different schemes for structured vs. unstructured pruning in order to utilize training resources best.

8.5 Optimization algorithms during model training

Stochastic gradient descent (SGD) is the de-facto standard algorithm in training deep neural networks. Most of the works investigating sparse training suggest that SGD is sensitive to the parameters as well as the network structure. Several show empirically that training larger models is more compute-efficient than training smaller models [Glorot et al. 2011a; Kaplan et al. 2020; Li et al. 2020a; Mhaskar and Poggio 2016]. We conjecture that this may be explained by the iterative optimization process and the ability to use additional dimensions to "route around" hills in the loss landscape. Thus, high-dimensional dense spaces help to elude local minima in the loss landscape as illustrated in Fig. 24: the left side shows a two dimensional function $f(x_1, x_2)$ and the loss function L as contour lines. Yellow areas are valleys and blue areas are hilltops. The red dashed line shows the value $x_2 = 0$, emulating a sparsified model, which is shown in the right plot. Here, we plot the (same) loss function on the x axis. We show two possible starting points s_1 and s_2 and SGD trajectories in green on both sides. We see that the x_2 dimension can be used to circumvent the leftmost hill when starting from s_1 in the two-dimensional model and proceed to the lowest minimum in the middle. However, when we sparsify x_2 in the right model, SGD will work in the projected subspace with less degrees of freedom and converge to the suboptimal minimum.

Furthermore, when sparsified, the Lipschitz constant of the loss function increases [Evci et al. 2020; Lee et al. 2020b] and complicates the optimization further. Modern techniques such as momentum can improve the optimizer but then may require more iterations [Lee et al. 2020b].

We may attribute this "weakness" of SGD to its fundamental property of linear first-order descent. Domingos [2020] further hardens this claim by showing that models trained with SGD are approximately kernel machines.

As we have seen, iteratively applying pruning improves the quality of pruned models significantly. If we now see this overall optimization process as a series of (linear) SGD iterations mixed with (nonlinear) pruning steps, this new optimization process implements a guided nonlinear search. At each pruning step, the function is perturbed in a guided way (depending on the pruning

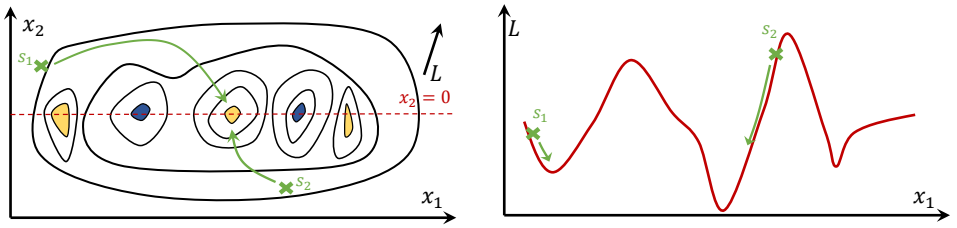


Fig. 24. SGD in a 1D loss landscape.

methodology, see Section 3) and then again minimized with SGD. At each pruning step, the model may evade a local minimum that SGD alone may not be able to overcome. For well tuned schedules, this scheme seems to approximate an efficient learning algorithm.

Bartoldson et al. [2020] model pruning as "noise injection" to explain improved generalization capabilities, which would fit this mental framework. They specifically consider the drop of test accuracy right after pruning in iterative pruning schemes. They show empirically that a higher drop relates to better generalization of the final model. They suggest that smaller models may not be the only reason for improved generalization and carefully tuned magnitude pruning schedules can improve generalization by "flattening" the loss landscape.

8.6 Emerging Benchmarks

Interpreting pruning results and comparing different methods is difficult due to the wide range of experimental setups, tasks, techniques, and hyperparameters used. This issue has already been identified by Blalock et al. [2020] who propose a standard methodology together with a set of benchmarks to solve this issue. One could imagine standard setups such as MLPerf [Mattson et al. 2020] or the Deep500 infrastructure [Ben-Nun et al. 2019] for performance measurements. We note that even before such a benchmark is widely accepted by the community, some datasets, tasks, and network architectures are emerging as *de-facto* benchmarks for pruning. We recommend researchers to use those as comparison points. As we point out above, ResNet-50 on ImageNet and BERT on the GLUE tasks seem excellent candidates for such standard benchmark sets for both model sparsity and performance.

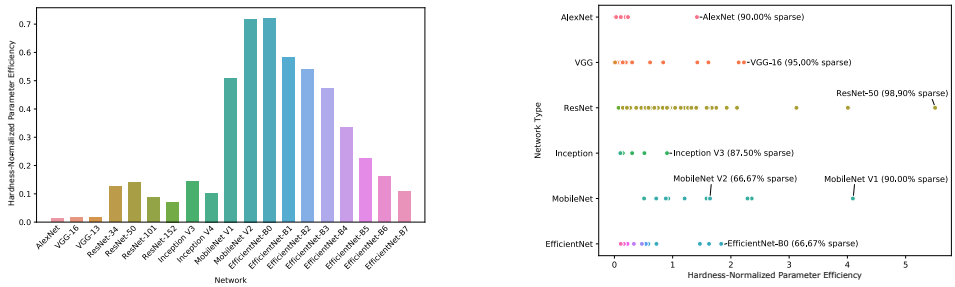
We observe that the achieved sparsity at high accuracies strongly correlates with the attention that certain models received in the literature. For example, ResNet-50 is well tuned and thus shows higher achieved parameter efficiencies relative to other models. Thus, they effectively define the state of the art—however, this observation also means that one cannot easily reason about the "prunability" of a certain architecture without extensive experiments on a level playing field.

For toy examples, the MNIST dataset with the LeNet-300-100 and LeNet-5 networks can act as a good calibration. The state of the art is above 98% accuracy with less than 1% of the original parameters. However, we insist that this task alone is not indicative of good performance of a method. More meaningful tasks are larger convolutional networks on more complex tasks such as CIFAR-100 and ImageNet. In order to track progress, we recommend that those should always be reported when analyzing new pruning methodologies even though better architectures for these tasks (or better tasks) may exist. Additionally, in our experience global magnitude pruning is a good baseline method for a wide range of scenarios, see e.g., Singh and Alistarh [2020] for results.

8.7 Parameter Efficiency

One could define the general concept of parameter efficiency as "How much does the average parameter contribute to the overall quality of the model?". We observe that, when pruned, the parameter efficiency often increases while the overall model quality decreases. Bianco et al. [2018] propose *accuracy density* as a measure of parameter efficiency. It is defined as the validation accuracy (in percentage) divided by the number of parameters (in millions). With the metric, the authors show clear benefits for MobileNet (both versions) over ResNet-50, but also a benefit of AlexNet and SqueezeNet, both under 60% top-1 accuracy, over VGG-16 (with 71.6% accuracy). When extended to pruned DNNs, accuracy density increases disproportionally, with sparse but inaccurate models ranked highest and orders of magnitude of difference. It is thus apparent that not every validation sample is as easy to predict as the others, and the measure should *not* be linear with the count of correct predictions.

To deal with parameter efficiency in the face of varying classification difficulty, we define a slightly modified measure called *hardness-normalized parameter efficiency*. Instead of computing the ratio of accuracy to parameters, we normalize the number of correct predictions by their relative difficulty. To estimate classification difficulty for ImageNet, we fit a function through the state-of-the-art DNN-based image classifiers over the years (Fig. 25), and then evaluate the number of correct classifications by the inverse function to obtain the hardness-normalized correct predictions, and divide by the number of parameters (in millions).



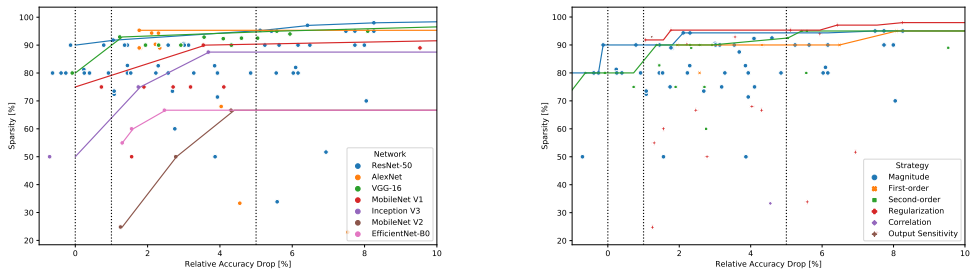
(a) Dense Networks (b) Sparse Networks

Fig. 25. Parameter efficiency of state-of-the-art DNNs on the ImageNet dataset, with color indicating DNN type. Hardness-normalized parameter efficiency is normalized based on a logarithmic fit of the top-1 validation accuracy of DNNs over the years 2012–2020: $f(x) = 5704.7 \cdot \ln x + 30908$ (as correctly predicted images).

The hardness-normalized parameter efficiencies of popular dense and corresponding sparse CNNs are presented in Fig. 25a and 25b, respectively. For dense networks, we can see that parameter efficiency similarly increases for ResNets and MobileNets over AlexNet and VGG, but that VGG variants are actually more parameter efficient than AlexNet, despite being twice larger. EfficientNet-B0 is roughly on the same parameter efficiency as MobileNet (v2), which is reasonable given that the former network is a mobile-sized baseline, albeit produced via Neural Architecture Search. For the sparsified networks, most of the pruned networks are more parameter efficient than the best dense networks. We see that the top ranked CNN is a pruned ResNet-50 [Savarese et al. 2020], which can achieve 66% validation accuracy with only ≈281,000 parameters. The second best network is a pruned MobileNet (v1) with 68% accuracy for ≈423,000 parameters. It may be interesting to investigate this metric in more depth (e.g., with different normalization scales) to understand whether the efficiency per parameter increases monotonically with smaller networks or whether

the decrease in model quality leads to a decrease in parameter efficiency as well. This could provide some insight into optimal sparsity levels.

Parameter Slack. Figure 26 shows a relative view of the same data. It shows what sparsity level is achievable if we allow a fixed decrease in accuracy, relative to the dense baseline. Since the sparsity is relative to the original network (and its parameter efficiency), it is hard to compare different networks in this figure; instead, we recommend to consider the curve of each network in isolation with the vertical dotted lines at markers of 0%, 1%, and 5% accuracy loss budget. (This metric is inspired in part by the MLPerf ImageNet rules [Mattson et al. 2020].) The results suggest that architectures such as AlexNet or VGG-16 have significantly higher parameter slack than, e.g., MobileNet or Inception. Fig. 26a shows the data grouped by network type. It allows to reason about "parameter slack", i.e., the steeper the curve, the higher the percentage of parameters which can be removed while preserving some percentage of the baseline accuracy. Fig. 26b shows the same data but grouped by element removal scheme and thus allows a comparison of different schemes.



(a) Grouped by network

(b) Grouped by strategy

Fig. 26. Relative validation ImageNet accuracy loss for different pruning densities, strategies, and neural networks. Solid lines represent best-performing networks, whereas dotted lines represent accuracy thresholds (e.g., 1% relative accuracy reduction is the maximum allowed by MLPerf ImageNet rules [Mattson et al. 2020]). Negative accuracy drop means improvement in generalization.

Apart from parameters, model information content (and thus parameter efficiency) is also encoded in the data type of the weights themselves. This is explicitly clear in binarized networks that have only values $\in \{-1, 1\}$. In these cases, the additional zero value adds another piece of information, similar to ternary networks [Li et al. 2021]. Networks with larger weight data types also benefit from the sparsity, however, it remains unclear whether the overhead of storing the non-zero structure (see Section 2.2) is worth the gain in parameter efficiency.

Sparsification versus manual design. An interesting observation is that sparsification of older architectures often does not achieve the same gains as architectures developed later. Yet, many breakthrough results in the area of efficient convolutional neural networks can be seen as manually defined sparsifiers, such as bottleneck layers or depthwise separable convolutions [Howard et al. 2017; Iandola et al. 2016]. The resulting optimized networks are often harder to sparsify. In some sense, this manual design of inductive biases into the network is similar to feature engineering that deep neural networks replaced to begin with. Newer works on transformers suggest the more automated way of "train big and then prune" [Li et al. 2020a]. Here, we rely on the learning process to automatically discover good network designs. It is to be shown whether such automated methods can compete with hand-crafted biases for modern networks such as transformers.

We close the discussion on parameter efficiency with an observation: *Interestingly, the fact that most of the (very different) methods presented in the literature reach similar results in terms of accuracy at a given sparsity (within relative 1%) suggests that there are inherent compression thresholds which may be hard to overcome.*

8.8 Generalization and biases

It is a surprising fact that neural networks can be heavily pruned without impacting their overall accuracy. Yet this raises a question: *Is top-level accuracy sufficient to capture the effects of pruning when the neural network representation has changed so dramatically?* In recent work, Hooker et al. [2019] show that using the unstructured iterative magnitude pruning of Zhu and Gupta [2017] on CNNs for image classification results in a large degradation in accuracy for a small number of classes in tasks such as ImageNet, compared to the model's overall decrease. These classes were typically less well represented in the training data. Interestingly, they also find that, compared with pruning, quantization results in a much smaller impact to different classes. Further, they find that pruned models are significantly more brittle under distribution shifts, such as corrupted images in ImageNet-C [Hendrycks and Dietterich 2019] or naturally adversarial images in ImageNet-A [Hendrycks et al. 2019].

Hooker et al. [2020] build on these results and show that the increased errors on certain classes caused by pruning can amplify existing algorithmic biases. On CelebA [Liu et al. 2015a], a dataset of celebrity faces with significant correlations between demographic groups, pruning increases errors on underrepresented subgroups. For example, pruning a model trained to identify people with blond hair to 95% sparsity increased the average false-positive rate for men by 49.54%, but by only 6.32% for others.

The biases and brittleness introduced by pruning may limit the utility of pruned models, especially in situations that often deal with protected attributes and are sensitive to fairness, such as facial recognition or healthcare. This is unfortunate, since these domains typically deploy models in resource-constrained environments where pruning is particularly valuable. Therefore, it is important to study the finer-grained impacts of pruning, rather than just the overall accuracy. Identifying the impact of pruning methods beyond iterative magnitude pruning, and developing more robust pruning methods, are critical open problems.

8.9 Best practices

We now focus more on the practical aspects of pruning and conclude the discussion with a set of recommendations we identified based on the body of literature in the field. We first note that a flurry of simple approaches enables reaching moderate sparsity levels (e.g., 50–90%) at the same or even increased accuracy. It seems that any non-silly scheme achieves some sparsification and that there is an inherent robustness in the networks themselves. However, reaching higher sparsity levels (e.g., >95%) requires more elaborate pruning techniques where we may be reaching the limit of gradient-based optimization techniques for learning. We now provide best practices in five categories that we recommend everyone to follow when performing pruning in practice.

1. Pruning strategy. In general, highest sparsity is achieved using regularization methods in combination with iterative pruning and growth schedules. These methods have high computational costs, sometimes causing a five-fold increase in training overheads, e.g., Savarese et al. [2020]. Regularization methods are relatively hard to control and require numerous hyperparameters. The simplest training method, magnitude pruning, is easiest to control for target sparsity and accuracy in many practical settings. In most training methods, it is important for the structure search to

enable weights to regrow, especially in phase of early structure adaptation at the beginning of training.

2. Retraining/fine-tuning. If the focus of sparsity is to improve inference, then retraining/fine-tuning is an essential part of a sparsification schedule. Gradually pruned sparsification schedules perform best and it is most efficient to start each iteration from the most trained/last set of weights.

3. Structure. Structured pruning seems to provide a great tradeoff between accuracy and performance on today's architectures. This is partly due to the fact that hardware and frameworks are tuned for dense blocked computations. Furthermore, structural pruning can form a strong bias towards powerful mechanisms like locally connected layers that, together with weight sharing, yield convolutional layers.

4. Distribution. The sparsity distribution across layers/operators needs to be considered carefully. For this, one could hand-tune the sparsity levels for each operator type and position in the network. For example, dense layers can often be pruned more than convolutional layers and the first layer in a convolutional network can hardly be pruned. A simpler scheme may use a global sparsity and a learned allocation strategy.

5. Combined ephemeral and model sparsity. Any sparse deep neural network should combine both ephemeral and model sparsity. For example, dropout often functions as a "pre-regularizer" and can benefit generalization greatly if enough data is available. Furthermore, ephemeral and model sparsity lead to a multiplicative benefit in terms of needed arithmetic operations.

9 CHALLENGES AND OPEN QUESTIONS

We now outline ten central challenges and open questions in the field to inspire future research.

(1) **Sparse training.** Can we use sparsity to train gigantic models whose dense version would not fit into the hardware budget? How do we sparsely train models without accuracy loss?
(2) **Structured vs. unstructured.** How does a structural bias influence the accuracy performance and model size tradeoff?
(3) **Hardware co-design.** How do we co-design hardware architectures and pruned models? What is the tradeoff between cost, accuracy, and structured sparsity?
(4) **Multi-objective pruning.** What is the best way to prune for multiple objectives simultaneously, e.g., lowest energy consumption for a certain memory size?
(5) **Architecture design.** Should one use neural architecture search (NAS) for finding efficient networks or can pruning replace NAS?
(6) **Theory of sparse learning.** What is the relationship between sparsity, learning dynamics, and generalization?
(7) **Sparse representations.** What is the representational power of sparse neural networks? Could parameter efficiency be defined rigorously?
(8) **Method generalization.** Which of the pruning methods for MLPs or CNNs generalize to transformers or other neural architectures?
(9) **Data-free sparsity.** Can we design one-shot and data-free methods that rival the accuracy of data-dependent methods?
(10) **Fairness and bias.** How do we design more robust sparse models and sparsification approaches? How do we prevent adversarial attacks on sparsified models?

We do not explicitly list brain-related research challenges because our work focuses primarily on the engineering aspects of sparsity for which biological analogies are certainly a major inspiration but act mainly as a means to an end.

10 CONCLUSIONS AND OUTLOOK

We show that sparsity can already lead to a theoretical 10–100x improvement in efficiency. Furthermore, larger networks appear to provide more opportunity for pruning [Gale et al. 2019; Sanh et al. 2020] so the compression trend is likely to continue as architectures get larger. Specifically, training extremely large models with sparse methods will provide many opportunities. Our detailed analysis of data science and engineering aspects enables a targeted hardware-software co-design for next-generation deep learning architectures that exploit the potentially huge speedups.

We also expect that there remains potential in the data science aspects of sparsity, especially in the areas of very high sparsity (>99%) as well as sparse training of large models in very high-dimensional spaces. Both could lead to significant breakthroughs in future deep learning systems.

Acknowledgments

We thank Doug Burger, Steve Scott, Marco Heddes, and the respective teams at Microsoft for inspiring discussions on the topic. We thank Angelika Steger for uplifting debates about the connections to biological brains and Sidak Pal Singh for his support regarding experimental results.

REFERENCES

Alessandro Achille, Matteo Rovere, and Stefano Soatto. 2019. Critical Learning Periods in Deep Neural Networks. (2019). arXiv:cs.LG/1711.08856

Sher Afghan and Uwe Naumann. 2020. Interval Adjoint Significance Analysis for Neural Networks. In *International Conference on Computational Science*. Springer, 365–378.

Alireza Aghasi, Afshin Abdi, Nam Nguyen, and Justin Romberg. 2017. Net-Trim: Convex Pruning of Deep Neural Networks with Performance Guarantee. (2017). arXiv:cs.LG/1611.05162

Subutai Ahmad and Luiz Scheinkman. 2019. How Can We Be So Dense? The Benefits of Using Highly Sparse Representations. (2019). arXiv:cs.LG/1903.11257

Alham Fikriand Aji and Kenneth Heafield. 2017. Sparse Communication for Distributed Gradient Descent. In *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing*. 440–445. arXiv:cs.CL/1704.05021

J. Albericio, P. Judd, T. Hetherington, T. Aamodt, N. E. Jerger, and A. Moshovos. 2016. Cnvlutin: Ineffectual-Neuron-Free Deep Neural Network Computing. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*. 1–13. https://doi.org/10.1109/ISCA.2016.11

Dan Alistarh, Demjan Grubic, Jerry Li, Ryota Tomioka, and Milan Vojnovic. 2017. QSGD: Communication-Efficient SGD via Gradient Quantization and Encoding. (2017). arXiv:cs.LG/1610.02132

Dan Alistarh, Torsten Hoefler, Mikael Johansson, Nikola Konstantinov, Sarit Khirirat, and Cédric Renggli. 2018. The convergence of sparsified gradient methods. In *Advances in Neural Information Processing Systems*. 5973–5983. arXiv:cs.LG/1809.10505

Zeyuan Allen-Zhu, Yuanzhi Li, and Zhao Song. 2019. A Convergence Theory for Deep Learning via Over-Parameterization. (2019). arXiv:cs.LG/1811.03962

Amjad Almahairi, Nicolas Ballas, Tim Cooijmans, Yin Zheng, Hugo Larochelle, and Aaron Courville. 2016. Dynamic Capacity Networks. (2016). arXiv:cs.LG/1511.07838

Jose M. Alvarez and Mathieu Salzmann. 2017. Compression-aware Training of Deep Networks. (2017). arXiv:cs.CV/1711.02638

Manoj Alwani, Han Chen, Michael Ferdman, and Peter Milder. 2016. Fused-layer CNN accelerators. In *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Press, 22.

Shun-ichi Amari. 1998. Natural Gradient Works Efficiently in Learning. *Neural Computation* 10, 2 (1998), 251–276. https://doi.org/10.1162/089976698300017746 arXiv:https://doi.org/10.1162/089976698300017746

Sajid Anwar, Kyuyeon Hwang, and Wonyong Sung. 2017. Structured pruning of deep convolutional neural networks. *ACM Journal on Emerging Technologies in Computing Systems (JETC)* 13, 3 (2017), 1–18.

Kambiz Azarian, Yash Bhalgat, Jinwon Lee, and Tijmen Blankevoort. 2020. Learned Threshold Pruning. (2020). arXiv:cs.LG/2003.00075

Jimmy Ba, Roger Grosse, and James Martens. 2016a. Distributed second-order optimization using Kronecker-factored approximations. (2016).

Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E Hinton. 2016b. Layer normalization. (2016). arXiv:cs.LG/1607.06450

Pierre Baldi and Peter J Sadowski. 2013. Understanding Dropout. In *Advances in Neural Information Processing Systems*, C. J. C. Burges, L. Bottou, M. Welling, Z. Ghahramani, and K. Q. Weinberger (Eds.), Vol. 26. Curran Associates, Inc.,

2814–2822. https://proceedings.neurips.cc/paper/2013/file/71f6278d140af599e06ad9bf1ba03cb0-Paper.pdf

Brian R. Bartoldson, Ari S. Morcos, Adrian Barbu, and Gordon Erlebacher. 2020. The Generalization-Stability Tradeoff In Neural Network Pruning. (2020). arXiv:cs.LG/1906.03728

Debraj Basu, Deepesh Data, Can Karakus, and Suhas N Diggavi. 2020. Qsparse-local-SGD: Distributed SGD with quantization, sparsification, and local computations. *IEEE Journal on Selected Areas in Information Theory* 1, 1 (2020), 217–226. arXiv:stat.ML/1906.02367

Cenk Baykal, Lucas Liebenwein, Igor Gilitschenski, Dan Feldman, and Daniela Rus. 2018. Data-dependent coresets for compressing neural networks with applications to generalization bounds. *arXiv preprint arXiv:1804.05345* (2018).

Amir Beck and Marc Teboulle. 2009. A Fast Iterative Shrinkage-Thresholding Algorithm for Linear Inverse Problems. *SIAM J. Img. Sci.* 2, 1 (March 2009), 183–202. https://doi.org/10.1137/080716542

Guillaume Bellec, David Kappel, Wolfgang Maass, and Robert Legenstein. 2018. Deep Rewiring: Training very sparse deep networks. (2018). arXiv:cs.NE/1711.05136

Iz Beltagy, Matthew E. Peters, and Arman Cohan. 2020. Longformer: The Long-Document Transformer. (2020). arXiv:cs.CL/2004.05150

Tal Ben-Nun, Maciej Besta, Simon Huber, Alexandros Nikolaos Ziogas, Daniel Peter, and Torsten Hoefler. 2019. A Modular Benchmarking Infrastructure for High-Performance and Reproducible Deep Learning. (2019). arXiv:cs.DC/1901.10183

Tal Ben-Nun and Torsten Hoefler. 2018. Demystifying Parallel and Distributed Deep Learning: An In-Depth Concurrency Analysis. (2018). arXiv:cs.LG/1802.09941

Emmanuel Bengio, Pierre-Luc Bacon, Joelle Pineau, and Doina Precup. 2016. Conditional Computation in Neural Networks for faster models. (2016). arXiv:cs.LG/1511.06297

Yoshua Bengio, Nicholas Léonard, and Aaron Courville. 2013a. Estimating or Propagating Gradients Through Stochastic Neurons for Conditional Computation. (2013). arXiv:cs.LG/1308.3432

Yoshua Bengio, Nicholas Léonard, and Aaron Courville. 2013b. Estimating or Propagating Gradients Through Stochastic Neurons for Conditional Computation. (2013). arXiv:cs.LG/1308.3432

Richard F Betzel, John D Medaglia, Lia Papadopoulos, Graham L Baum, Ruben Gur, Raquel Gur, David Roalf, Theodore D Satterthwaite, and Danielle S Bassett. 2017. The modular organization of human anatomical brain networks: Accounting for the cost of wiring. *Network Neuroscience* 1, 1 (2017), 42–68.

Simone Bianco, Remi Cadene, Luigi Celona, and Paolo Napoletano. 2018. Benchmark Analysis of Representative Deep Neural Network Architectures. *IEEE Access* 6 (2018), 64270–64277. https://doi.org/10.1109/access.2018.2877890

Davis Blalock, Jose Javier Gonzalez Ortiz, Jonathan Frankle, and John Guttag. 2020. What is the state of neural network pruning? (2020). arXiv:cs.LG/2003.03033

Alfred Bourely, John Patrick Boueri, and Krzysztof Choromonski. 2017. Sparse Neural Networks Topologies. (2017). arXiv:cs.LG/1706.05683

Tom B Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. In *Advances in Neural Information Processing Systems*. arXiv:cs.CL/2005.14165

Alon Brutzkus, Amir Globerson, Eran Malach, and Shai Shalev-Shwartz. 2017. SGD Learns Over-parameterized Networks that Provably Generalize on Linearly Separable Data. (2017). arXiv:cs.LG/1710.10174

P. Burrascano. 1993. A pruning technique maximizing generalization. In *Proceedings of 1993 International Conference on Neural Networks (IJCNN-93-Nagoya, Japan)*, Vol. 1. 347–350 vol.1. https://doi.org/10.1109/IJCNN.1993.713928

M. A. Carreira-Perpinan and Y. Idelbayev. 2018. "Learning-Compression" Algorithms for Neural Net Pruning. In *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 8532–8541. https://doi.org/10.1109/CVPR.2018.00890

Giovanna Castellano and Anna Maria Fanelli. 2000. Variable selection using neural-network models. *Neurocomputing* 31, 1-4 (2000), 1–13.

G. Castellano, A. M. Fanelli, and M. Pelillo. 1997. An iterative pruning algorithm for feedforward neural networks. *IEEE Transactions on Neural Networks* 8, 3 (1997), 519–531. https://doi.org/10.1109/72.572092

Hema Chandrasekaran, Hung-Han Chen, and Michael T. Manry. 2000. Pruning of basis functions in nonlinear approximators. *Neurocomputing* 34, 1 (2000), 29 – 53. https://doi.org/10.1016/S0925-2312(00)00311-8

Soravit Changpinyo, Mark Sandler, and Andrey Zhmoginov. 2017. The Power of Sparsity in Convolutional Neural Networks. (2017). arXiv:cs.CV/1702.06257

Shih-Kang Chao, Zhanyu Wang, Yue Xing, and Guang Cheng. 2020. Directional Pruning of Deep Neural Networks. (2020). arXiv:cs.LG/2006.09358

Yves Chauvin. 1989. *A Back-Propagation Algorithm with Optimal Use of Hidden Units*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 519–526.

Kumar Chellapilla, Sidd Puri, and Patrice Simard. 2006. High performance convolutional neural networks for document processing.

Chia-Yu Chen, Jungwook Choi, Daniel Brand, Ankur Agrawal, Wei Zhang, and Kailash Gopalakrishnan. 2017. AdaComp: Adaptive residual gradient compression for data-parallel distributed training. In *32nd AAAI Conference on Artificial Intelligence*. 2827–2835. arXiv:cs.LG/1712.02679

Jianda Chen, Shangyu Chen, and Sinno Jialin Pan. 2020. Storage Efficient and Dynamic Flexible Runtime Channel Pruning via Deep Reinforcement Learning. *Advances in Neural Information Processing Systems* 33 (2020).

Tianlong Chen, Jonathan Frankle, Shiyu Chang, Sijia Liu, Yang Zhang, Zhangyang Wang, and Michael Carbin. 2020. The Lottery Ticket Hypothesis for Pre-trained BERT Networks. (2020). arXiv:cs.LG/2007.12223

Y. Chen, T. Krishna, J. S. Emer, and V. Sze. 2017. Eyeriss: An Energy-Efficient Reconfigurable Accelerator for Deep Convolutional Neural Networks. *IEEE Journal of Solid-State Circuits* 52, 1 (2017), 127–138. https://doi.org/10.1109/JSSC.2016.2616357

Yu-Hsin Chen, Tien-Ju Yang, Joel Emer, and Vivienne Sze. 2019. Eyeriss v2: A Flexible Accelerator for Emerging Deep Neural Networks on Mobile Devices. (2019). arXiv:cs.DC/1807.07928

Yu Cheng, Duo Wang, Pan Zhou, and Tao Zhang. 2020. A Survey of Model Compression and Acceleration for Deep Neural Networks. (2020). arXiv:cs.LG/1710.09282

Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. 2014. cuDNN: Efficient primitives for deep learning. (2014). arXiv:cs.NE/1410.0759

Rewon Child, Scott Gray, Alec Radford, and Ilya Sutskever. 2019. Generating Long Sequences with Sparse Transformers. (2019). arXiv:cs.LG/1904.10509

Minsu Cho, Ameya Joshi, and Chinmay Hegde. 2020. ESPN: Extremely Sparse Pruned Networks. (2020). arXiv:cs.LG/2006.15741

Tejalal Choudhary, Vipul Mishra, Anurag Goswami, and Jagannathan Sarangapani. 2020. A comprehensive survey on model compression and acceleration. *Artificial Intelligence Review* (2020), 1–43.

Tautvydas Cibas, Françoise Fogelman Soulié, Patrick Gallinari, and Sarunas Raudys. 1996. Variable selection with neural networks. *Neurocomputing* 12, 2 (1996), 223 – 248. https://doi.org/10.1016/0925-2312(95)00121-2 Current European Neurocomputing Research.

Joseph Paul Cohen, Henry Z. Lo, and Wei Ding. 2017. RandomOut: Using a convolutional gradient norm to rescue convolutional filters. (2017). arXiv:cs.CV/1602.05931

Maxwell D. Collins and Pushmeet Kohli. 2014. Memory Bounded Deep Convolutional Networks. *CoRR* abs/1412.1442 (2014). arXiv:1412.1442 http://arxiv.org/abs/1412.1442

Gonçalo M Correia, Vlad Niculae, and André FT Martins. 2019. Adaptively sparse transformers. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*. arXiv:cs.CL/1909.00015

Justin Cosentino, Federico Zaiter, Dan Pei, and Jun Zhu. 2019. The Search for Sparse, Robust Neural Networks. (2019). arXiv:cs.LG/1912.02386

Baiyun Cui, Yingming Li, Ming Chen, and Zhongfei Zhang. 2019. Fine-tune BERT with Sparse Self-Attention Mechanism. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*. 3539–3544.

Bin Dai, Chen Zhu, and David Wipf. 2018b. Compressing Neural Networks using the Variational Information Bottleneck. (2018). arXiv:cs.CV/1802.10399

Xiaoliang Dai, Hongxu Yin, and Niraj K. Jha. 2018a. NeST: A Neural Network Synthesis Tool Based on a Grow-and-Prune Paradigm. (2018). arXiv:cs.NE/1711.02017

Shail Dave, Riyadh Baghdadi, Tony Nowatzki, Sasikanth Avancha, Aviral Shrivastava, and Baoxin Li. 2020. Hardware Acceleration of Sparse and Irregular Tensor Computations of ML Models: A Survey and Insights. (2020). arXiv:cs.AR/2007.00864

Peter Davies, Vijaykrishna Gurunathan, Niusha Moshrefi, Saleh Ashkboos, and Dan Alistarh. 2020. Distributed Variance Reduction with Optimal Communication. (2020). arXiv:cs.LG/2002.09268

Pau de Jorge, Amartya Sanyal, Harkirat S. Behl, Philip H. S. Torr, Gregory Rogez, and Puneet K. Dokania. 2020. Progressive Skeletonization: Trimming more fat from a network at initialization. (2020). arXiv:cs.CV/2006.09081

Luisa De Vivo, Michele Bellesi, William Marshall, Eric A Bushong, Mark H Ellisman, Giulio Tononi, and Chiara Cirelli. 2017. Ultrastructural evidence for synaptic scaling across the wake/sleep cycle. *Science* 355, 6324 (2017), 507–510.

L. Deng, G. Li, S. Han, L. Shi, and Y. Xie. 2020. Model Compression and Hardware Acceleration for Neural Networks: A Comprehensive Survey. *Proc. IEEE* 108, 4 (2020), 485–532. https://doi.org/10.1109/JPROC.2020.2976475

Misha Denil, Babak Shakibi, Laurent Dinh, Marc'Aurelio Ranzato, and Nando de Freitas. 2014. Predicting Parameters in Deep Learning. (2014). arXiv:cs.LG/1306.0543

Tim Dettmers and Luke Zettlemoyer. 2019. Sparse Networks from Scratch: Faster Training without Losing Performance. (2019). arXiv:cs.LG/1907.04840

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the*

Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers). 4171–4186.

S. Dey, K. Huang, P. A. Beerel, and K. M. Chugg. 2019. Pre-Defined Sparse Neural Networks With Hardware Acceleration. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems* 9, 2 (2019), 332–345. https://doi.org/10.1109/JETCAS.2019.2910864

Graham H Diering, Raja S Nirujogi, Richard H Roth, Paul F Worley, Akhilesh Pandey, and Richard L Huganir. 2017. Homer1a drives homeostatic scaling-down of excitatory synapses during sleep. *Science* 355, 6324 (2017), 511–515.

Xiaohan Ding, Guiguang Ding, Yuchen Guo, and Jungong Han. 2019a. Centripetal SGD for Pruning Very Deep Convolutional Networks with Complicated Structure. (2019). arXiv:cs.LG/1904.03837

Xiaohan Ding, Guiguang Ding, Xiangxin Zhou, Yuchen Guo, Jungong Han, and Ji Liu. 2019b. Global Sparse Momentum SGD for Pruning Very Deep Neural Networks. (2019). arXiv:cs.LG/1909.12778

William B Dolan and Chris Brockett. 2005. Automatically constructing a corpus of sentential paraphrases. In *Proceedings of the Third International Workshop on Paraphrasing (IWP2005)*.

Pedro Domingos. 2020. Every Model Learned by Gradient Descent Is Approximately a Kernel Machine. (2020). arXiv:cs.LG/2012.00152

Xin Dong, Shangyu Chen, and Sinno Jialin Pan. 2017. Learning to Prune Deep Neural Networks via Layer-wise Optimal Brain Surgeon. (2017). arXiv:cs.NE/1705.07565

Xiao Dong, Lei Liu, Guangli Li, Jiansong Li, Peng Zhao, Xueying Wang, and Xiaobing Feng. 2019. Exploiting the input sparsity to accelerate deep neural networks: poster. In *Proceedings of the 24th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2019, Washington, DC, USA, February 16-20, 2019*. 401–402. https://doi.org/10.1145/3293883.3295713

Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, et al. 2021. An image is worth 16x16 words: Transformers for image recognition at scale. In *Proceedings of the Ninth International Conference on Learning Representations*. arXiv:cs.CV/2010.11929

Nikoli Dryden, Tim Moon, Sam Ade Jacobs, and Brian Van Essen. 2016. Communication quantization for data-parallel training of deep neural networks. In *2nd Workshop on Machine Learning in HPC Environments (MLHPC)*. 1–8.

Simon S. Du, Xiyu Zhai, Barnabas Poczos, and Aarti Singh. 2019. Gradient Descent Provably Optimizes Over-parameterized Neural Networks. (2019). arXiv:cs.LG/1810.02054

Aritra Dutta, El Houcine Bergou, Ahmed M Abdelmoniem, Chen-Yu Ho, Atal Narayan Sahu, Marco Canini, and Panos Kalnis. 2020. On the discrepancy between the theoretical analysis and practical implementations of compressed communication for distributed deep learning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 34. 3817–3824. arXiv:cs.DC/1911.08250

Erich Elsen, Marat Dukhan, Trevor Gale, and Karen Simonyan. 2019. Fast Sparse ConvNets. (2019). arXiv:cs.CV/1911.09723

Thomas Elsken, Jan Hendrik Metzen, and Frank Hutter. 2019. Neural Architecture Search: A Survey. (2019). arXiv:stat.ML/1808.05377

A. P. Engelbrecht. 2001. A new pruning heuristic based on variance analysis of sensitivity information. *IEEE Transactions on Neural Networks* 12, 6 (2001), 1386–1399. https://doi.org/10.1109/72.963775

A. P. Engelbrecht and I. Cloete. 1996. A sensitivity analysis algorithm for pruning feedforward neural networks. In *Proceedings of International Conference on Neural Networks (ICNN'96)*, Vol. 2. 1274–1278 vol.2. https://doi.org/10.1109/ICNN.1996.549081

Andries Petrus Engelbrecht, Ian Cloete, and Jacek M Zurada. 1995. Determining the significance of input parameters using sensitivity analysis. In *International Workshop on Artificial Neural Networks*. Springer, 382–388.

Utku Evci, Trevor Gale, Jacob Menick, Pablo Samuel Castro, and Erich Elsen. 2020. Rigging the Lottery: Making All Tickets Winners. (2020). arXiv:cs.LG/1911.11134

Angela Fan, Edouard Grave, and Armand Joulin. 2020. Reducing transformer depth on demand with structured dropout. In *Proceedings of the Eighth International Conference on Learning Representations*. arXiv:cs.LG/1909.11556

William Fedus, Barret Zoph, and Noam Shazeer. 2021. Switch Transformers: Scaling to trillion parameter models with simple and efficient sparsity. (2021). arXiv:cs.LG/2101.03961

William Finnoff, Ferdinand Hergert, and Hans Georg Zimmermann. 1993. Improving model selection by nonconvergent methods. *Neural Networks* 6, 6 (1993), 771–783.

L. Fletcher, V. Katkovnik, F. E. Steffens, and A. P. Engelbrecht. 1998. Optimizing the number of hidden nodes of a feedforward artificial neural network. In *1998 IEEE International Joint Conference on Neural Networks Proceedings. IEEE World Congress on Computational Intelligence (Cat. No.98CH36227)*, Vol. 2. 1608–1612 vol.2. https://doi.org/10.1109/IJCNN.1998.686018

Jonathan Frankle and Michael Carbin. 2019. The Lottery Ticket Hypothesis: Finding Sparse, Trainable Neural Networks. (2019). arXiv:cs.LG/1803.03635

Jonathan Frankle, Gintare Karolina Dziugaite, Daniel M. Roy, and Michael Carbin. 2020a. Linear Mode Connectivity and the Lottery Ticket Hypothesis. (2020). arXiv:cs.LG/1912.05671

Jonathan Frankle, Gintare Karolina Dziugaite, Daniel M. Roy, and Michael Carbin. 2020b. Stabilizing the Lottery Ticket Hypothesis. (2020). arXiv:cs.LG/1903.01611

Jonathan Frankle, David J. Schwab, and Ari S. Morcos. 2020c. The Early Phase of Neural Network Training. (2020). arXiv:cs.LG/2002.10365

J. Friedman, T. Hastie, and R. Tibshirani. 2010. A note on the group lasso and a sparse group lasso. (2010). arXiv:math.ST/1001.0736

K.J. Friston. 2008. Hierarchical Models in the Brain. *PLOS Computational Biology* 4, 11 (2008), e1000211. https://doi.org/10.1371/journal.pcbi.1000211

Yarin Gal and Zoubin Ghahramani. 2016. Dropout as a Bayesian Approximation: Representing Model Uncertainty in Deep Learning. In *Proceedings of The 33rd International Conference on Machine Learning (Proceedings of Machine Learning Research)*, Maria Florina Balcan and Kilian Q. Weinberger (Eds.), Vol. 48. PMLR, New York, New York, USA, 1050–1059. http://proceedings.mlr.press/v48/gal16.html

Yarin Gal, Jiri Hron, and Alex Kendall. 2017. Concrete Dropout. In *Advances in Neural Information Processing Systems*, I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett (Eds.), Vol. 30. Curran Associates, Inc., 3581–3590. https://proceedings.neurips.cc/paper/2017/file/84ddfb34126fc3a48ee38d7044e87276-Paper.pdf

Trevor Gale, Erich Elsen, and Sara Hooker. 2019. The State of Sparsity in Deep Neural Networks. (2019). arXiv:cs.LG/1902.09574

Trevor Gale, Matei Zaharia, Cliff Young, and Erich Elsen. 2020. Sparse GPU Kernels for Deep Learning. (2020). arXiv:cs.LG/2006.10901

Prakhar Ganesh, Yao Chen, Xin Lou, Mohammad Ali Khan, Yin Yang, Deming Chen, Marianne Winslett, Hassan Sajjad, and Preslav Nakov. 2020. Compressing large-scale transformer-based models: A case study on BERT. (2020). arXiv:cs.LG/2002.11985

Dongdong Ge, Xiaoye Jiang, and Yinyu Ye. 2011. A note on the complexity of L p minimization. *Mathematical programming* 129, 2 (2011), 285–299.

Georgios Georgiadis. 2019. Accelerating Convolutional Neural Networks via Activation Map Compression. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 7085–7095.

Golnaz Ghiasi, Tsung-Yi Lin, and Quoc V Le. 2018. DropBlock: A regularization method for convolutional networks. In *Advances in Neural Information Processing Systems*, S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett (Eds.), Vol. 31. Curran Associates, Inc., 10727–10737. https://proceedings.neurips.cc/paper/2018/file/7edcfb2d8f6a659ef4cd1e6c9b6d7079-Paper.pdf

Joydeep Ghosh and Kagan Tumer. 1994. Structural Adaptation and Generalization in Supervised Feed-Forward Networks. *J. Artif. Neural Netw.* 1, 4 (Nov. 1994), 431–458.

Xavier Glorot, Antoine Bordes, and Yoshua Bengio. 2011a. Deep sparse rectifier neural networks. In *Proceedings of the fourteenth international conference on artificial intelligence and statistics*. 315–323.

Xavier Glorot, Antoine Bordes, and Yoshua Bengio. 2011b. Deep sparse rectifier neural networks. In *Proceedings of the fourteenth international conference on artificial intelligence and statistics*. 315–323.

Maximilian Golub, Guy Lemieux, and Mieszko Lis. 2019. Full deep neural network training on a pruned weight budget. (2019). arXiv:cs.LG/1806.06949

Aidan N. Gomez, Ivan Zhang, Siddhartha Rao Kamalakara, Divyam Madaan, Kevin Swersky, Yarin Gal, and Geoffrey E. Hinton. 2019. Learning Sparse Networks Using Targeted Dropout. (2019). arXiv:cs.LG/1905.13678

Ashish Gondimalla, Noah Chesnut, Mithuna Thottethodi, and T. N. Vijaykumar. 2019. SparTen: A Sparse Tensor Accelerator for Convolutional Neural Networks. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '52)*. Association for Computing Machinery, New York, NY, USA, 151–165. https://doi.org/10.1145/3352460.3358291

Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. 2014a. Generative adversarial nets. In *Advances in neural information processing systems*. 2672–2680. arXiv:stat.ML/1406.2661

Ian J. Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. 2014b. Generative Adversarial Networks. (2014). arXiv:stat.ML/1406.2661

Soorya Gopalakrishnan, Zhinus Marzi, Upamanyu Madhow, and Ramtin Pedarsani. 2018. Combating Adversarial Attacks Using Sparse Representations. (2018). arXiv:stat.ML/1803.03880

Ariel Gordon, Elad Eban, Ofir Nachum, Bo Chen, Hao Wu, Tien-Ju Yang, and Edward Choi. 2018. Morphnet: Fast & simple resource-constrained structure learning of deep networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 1586–1595.

Mitchell A. Gordon, Kevin Duh, and Nicholas Andrews. 2020. Compressing BERT: Studying the Effects of Weight Pruning on Transfer Learning. In *Proceedings of the 5th Workshop on Representation Learning for NLP*. 143–155. arXiv:cs.CL/2002.08307

Peter Grönquist, Chengyuan Yao, Tal Ben-Nun, Nikoli Dryden, Peter Dueben, Shigang Li, and Torsten Hoefler. 2020. Deep Learning for Post-Processing Ensemble Weather Forecasts. (2020). arXiv:cs.LG/2005.08748

William Gropp, Torsten Hoefler, Rajeev Thakur, and E. Lusk. 2014. *Using Advanced MPI: Modern Features of the Message-Passing Interface*. MIT Press.

William Gropp, Torsten Hoefler, Rajeev Thakur, and Jesper Larsson Träff. 2011. Performance Expectations and Guidelines for MPI Derived Datatypes. In *Recent Advances in the Message Passing Interface (EuroMPI'11)*, Vol. 6960. Springer, 150–159.

Peter D Grünwald. 2007. *The minimum description length principle*. MIT press.

Denis Gudovskiy, Alec Hodgkinson, and Luca Rigazio. 2018. DNN Feature Map Compression using Learned Representation over GF (2). In *Proceedings of the European Conference on Computer Vision (ECCV)*. 0–0.

Luis Guerra, Bohan Zhuang, Ian Reid, and Tom Drummond. 2020. Automatic Pruning for Quantized Neural Networks. (2020). arXiv:cs.CV/2002.00523

Fu-Ming Guo, Sijia Liu, Finlay S Mungall, Xue Lin, and Yanzhi Wang. 2019a. Reweighted proximal pruning for large-scale language representation. (2019). arXiv:cs.LG/1909.12486

Qipeng Guo, Xipeng Qiu, Pengfei Liu, Yunfan Shao, Xiangyang Xue, and Zheng Zhang. 2019b. Star-Transformer. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*. 1315–1325. arXiv:cs.CL/1902.09113

Yiwen Guo, Anbang Yao, and Yurong Chen. 2016. Dynamic Network Surgery for Efficient DNNs. (2016). arXiv:cs.NE/1608.04493

Yiwen Guo, Chao Zhang, Changshui Zhang, and Yurong Chen. 2018. Sparse dnns with improved adversarial robustness. In *Advances in neural information processing systems*. 242–251.

Manish Gupta and Puneet Agrawal. 2020. Compression of Deep Learning Models for Text: A Survey. (2020). arXiv:cs.CL/2008.05221

Udit Gupta, Brandon Reagen, Lillian Pentecost, Marco Donato, Thierry Tambe, Alexander M. Rush, Gu-Yeon Wei, and David Brooks. 2019. MASR: A Modular Accelerator for Sparse RNNs. (2019). arXiv:eess.SP/1908.08976

Masafumi Hagiwara. 1993. Removal of hidden units and weights for back propagation networks. In *Proceedings of 1993 International Conference on Neural Networks (IJCNN-93-Nagoya, Japan)*, Vol. 1. IEEE, 351–354.

Masafumi Hagiwara. 1994. A simple and effective method for removal of hidden units and weights. *Neurocomputing* 6, 2 (1994), 207 – 218. https://doi.org/10.1016/0925-2312(94)90055-8 Backpropagation, Part IV.

Hong-Gui Han and Jun-Fei Qiao. 2013. A structure optimisation algorithm for feedforward neural network construction. *Neurocomputing* 99 (2013), 347–357.

Song Han, Junlong Kang, Huizi Mao, Yiming Hu, Xin Li, Yubin Li, Dongliang Xie, Hong Luo, Song Yao, Yu Wang, Huazhong Yang, and William J. Dally. 2017. ESE: Efficient Speech Recognition Engine with Sparse LSTM on FPGA. (2017). arXiv:cs.CL/1612.00694

Song Han, Xingyu Liu, Huizi Mao, Jing Pu, Ardavan Pedram, Mark A. Horowitz, and William J. Dally. 2016a. EIE: Efficient Inference Engine on Compressed Deep Neural Network. (2016). arXiv:cs.CV/1602.01528

Song Han, Huizi Mao, and William J. Dally. 2016b. Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding. (2016). arXiv:cs.CV/1510.00149

Song Han, Jeff Pool, Sharan Narang, Huizi Mao, Enhao Gong, Shijian Tang, Erich Elsen, Peter Vajda, Manohar Paluri, John Tran, Bryan Catanzaro, and William J. Dally. 2017. DSD: Dense-Sparse-Dense Training for Deep Neural Networks. (2017). arXiv:cs.CV/1607.04381

Lars Kai Hansen et al. 1994. Controlled growth of cascade correlation nets. In *International Conference on Artificial Neural Networks*. Springer, 797–800.

Stephen Hanson and Lorien Pratt. 1989. Comparing Biases for Minimal Network Construction with Back-Propagation. In *Advances in Neural Information Processing Systems*, D. Touretzky (Ed.), Vol. 1. Morgan-Kaufmann, 177–185. https://proceedings.neurips.cc/paper/1988/file/1c9ac0159c94d8d0cbedc973445af2da-Paper.pdf

Babak Hassibi and David G. Stork. 1992. Second Order Derivatives for Network Pruning: Optimal Brain Surgeon. In *Advances in Neural Information Processing Systems 5, [NIPS Conference]*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 164–171.

J. Hawkins. 2017. Special report : Can we copy the brain? - What intelligent machines need to learn from the Neocortex. *IEEE Spectrum* 54, 6 (2017), 34–71. https://doi.org/10.1109/MSPEC.2017.7934229

Soufiane Hayou, Jean-Francois Ton, Arnaud Doucet, and Yee Whye Teh. 2020. Pruning untrained neural networks: Principles and Analysis. (2020). arXiv:stat.ML/2002.08797

K. He, G. Gkioxari, P. Dollár, and R. Girshick. 2017. Mask R-CNN. In *2017 IEEE International Conference on Computer Vision (ICCV)*. 2980–2988. https://doi.org/10.1109/ICCV.2017.322

K. He, X. Zhang, S. Ren, and J. Sun. 2016. Deep Residual Learning for Image Recognition. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 770–778.

Yihui He, Ji Lin, Zhijian Liu, Hanrui Wang, Li-Jia Li, and Song Han. 2019a. AMC: AutoML for Model Compression and Acceleration on Mobile Devices. (2019). arXiv:cs.CV/1802.03494

Yang He, Ping Liu, Ziwei Wang, Zhilan Hu, and Yi Yang. 2019b. Filter Pruning via Geometric Median for Deep Convolutional Neural Networks Acceleration. (2019). arXiv:cs.CV/1811.00250

Yihui He, Xiangyu Zhang, and Jian Sun. 2017. Channel Pruning for Accelerating Very Deep Neural Networks. (2017). arXiv:cs.CV/1707.06168

Donald O. Hebb. 1949. *The organization of behavior: A neuropsychological theory*. Wiley, New York.

Kartik Hegde, Hadi Asghari-Moghaddam, Michael Pellauer, Neal Crago, Aamer Jaleel, Edgar Solomonik, Joel Emer, and Christopher W. Fletcher. 2019. ExTensor: An Accelerator for Sparse Tensor Algebra. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '52)*. Association for Computing Machinery, New York, NY, USA, 319–333. https://doi.org/10.1145/3352460.3358275

Dan Hendrycks and Thomas Dietterich. 2019. Benchmarking neural network robustness to common corruptions and perturbations. In *Proceedings of the Seventh International Conference on Learning Representations*. arXiv:cs.LG/1903.12261

Dan Hendrycks, Kevin Zhao, Steven Basart, Jacob Steinhardt, and Dawn Song. 2019. Natural adversarial examples. (2019). arXiv:cs.LG/1907.07174

Suzana Herculano-Houzel, Bruno Mota, Peiyan Wong, and Jon H. Kaas. 2010. Connectivity-driven white matter scaling and folding in primate cerebral cortex. *Proceedings of the National Academy of Sciences* 107, 44 (2010), 19008–19013. https://doi.org/10.1073/pnas.1012590107 arXiv:https://www.pnas.org/content/107/44/19008.full.pdf

P. Hill, A. Jain, M. Hill, B. Zamirai, C. Hsu, M. A. Laurenzano, S. Mahlke, L. Tang, and J. Mars. 2017. DeftNN: Addressing Bottlenecks for DNN Execution on GPUs via Synapse Vector Elimination and Near-compute Data Fission. In *2017 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 786–799.

Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. 2015. Distilling the Knowledge in a Neural Network. (2015). arXiv:stat.ML/1503.02531

Geoffrey E. Hinton, Nitish Srivastava, Alex Krizhevsky, Ilya Sutskever, and Ruslan R. Salakhutdinov. 2012. Improving neural networks by preventing co-adaptation of feature detectors. (2012). arXiv:cs.NE/1207.0580

Geoffrey E Hinton and Drew Van Camp. 1993. Keeping the neural networks simple by minimizing the description length of the weights. In *Proceedings of the sixth annual conference on Computational learning theory*. 5–13.

Torsten Hoefler and Roberto Belli. 2015. Scientific Benchmarking of Parallel Computing Systems. ACM, 73:1–73:12. Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC15).

Sara Hooker, Aaron Courville, Gregory Clark, Yann Dauphin, and Andrea Frome. 2019. What Do Compressed Deep Neural Networks Forget? (2019). arXiv:cs.LG/1911.05248

Sara Hooker, Nyalleng Moorosi, Gregory Clark, Samy Bengio, and Emily Denton. 2020. Characterising bias in compressed models. (2020). arXiv:cs.LG/2010.03058

Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. 2017. MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications. (2017). arXiv:cs.CV/1704.04861

Patrik O Hoyer. 2004. Non-negative matrix factorization with sparseness constraints. *Journal of machine learning research* 5, Nov (2004), 1457–1469.

Hengyuan Hu, Rui Peng, Yu-Wing Tai, and Chi-Keung Tang. 2016. Network Trimming: A Data-Driven Neuron Pruning Approach towards Efficient Deep Architectures. (2016). arXiv:cs.NE/1607.03250

Gao Huang, Yu Sun, Zhuang Liu, Daniel Sedra, and Kilian Q. Weinberger. 2016. Deep Networks with Stochastic Depth. In *Computer Vision – ECCV 2016*, Bastian Leibe, Jiri Matas, Nicu Sebe, and Max Welling (Eds.). Springer International Publishing, Cham, 646–661.

Zehao Huang and Naiyan Wang. 2018. Data-Driven Sparse Structure Selection for Deep Neural Networks. (2018). arXiv:cs.CV/1707.01213

Ziyue Huang, Wang Yilei, Ke Yi, et al. 2019. Optimal Sparsity-Sensitive Bounds for Distributed Mean Estimation. In *Advances in Neural Information Processing Systems*. 6371–6381.

Itay Hubara, Matthieu Courbariaux, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. 2016. Binarized Neural Networks. In *Proceedings of the 30th International Conference on Neural Information Processing Systems (NIPS'16)*. Curran Associates Inc., Red Hook, NY, USA, 4114–4122.

Forrest N. Iandola, Song Han, Matthew W. Moskewicz, Khalid Ashraf, William J. Dally, and Kurt Keutzer. 2016. SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and <0.5MB model size. (2016). arXiv:cs.CV/1602.07360

Andrei Ivanov, Nikoli Dryden, Tal Ben-Nun, Shigang Li, and Torsten Hoefler. 2020. Data Movement Is All You Need: A Case Study on Optimizing Transformers. (2020). arXiv:cs.LG/2007.00072

Nikita Ivkin, Daniel Rothchild, Enayat Ullah, Ion Stoica, Raman Arora, et al. 2019. Communication-efficient distributed SGD with sketching. In *Advances in Neural Information Processing Systems*. 13144–13154. arXiv:cs.LG/1903.04488

Robert A Jacobs, Michael I Jordan, Steven J Nowlan, and Geoffrey E Hinton. 1991. Adaptive mixtures of local experts. *Neural computation* 3, 1 (1991), 79–87.

Niehues Jan, Roldano Cattoni, Stuker Sebastian, Matteo Negri, Marco Turchi, Salesky Elizabeth, Sanabria Ramon, Barrault Loic, Specia Lucia, and Marcello Federico. 2019. The IWSLT 2019 evaluation campaign. In *16th International Workshop on Spoken Language Translation 2019*.

Steven A Janowsky. 1989. Pruning versus clipping in neural networks. *Physical Review A* 39, 12 (1989), 6600.

Siddhant Jayakumar, Razvan Pascanu, Jack Rae, Simon Osindero, and Erich Elsen. 2020. Top-KAST: Top-K Always Sparse Training. *Advances in Neural Information Processing Systems* 33 (2020).

Peng Jiang and Gagan Agrawal. 2018. A linear speedup analysis of distributed deep learning with sparse and quantized communication. In *Advances in Neural Information Processing Systems*. 2525–2536.

Sian Jin, Sheng Di, Xin Liang, Jiannan Tian, Dingwen Tao, and Franck Cappello. 2019. DeepSZ: A Novel Framework to Compress Deep Neural Networks by Using Error-Bounded Lossy Compression. In *Proceedings of the 28th International Symposium on High-Performance Parallel and Distributed Computing (HPDC '19)*. Association for Computing Machinery, New York, NY, USA, 159–170. https://doi.org/10.1145/3307681.3326608

Xiaojie Jin, Xiaotong Yuan, Jiashi Feng, and Shuicheng Yan. 2016. Training Skinny Deep Neural Networks with Iterative Hard Thresholding Methods. (2016). arXiv:cs.CV/1607.05423

Sari Jones, Lars Nyberg, Johan Sandblom, Anna Stigsdotter Neely, Martin Ingvar, Karl Magnus Petersson, and Lars Bäckman. 2006. Cognitive and neural plasticity in aging: general and task-specific limitations. *Neuroscience & Biobehavioral Reviews* 30, 6 (2006), 864–871.

Michael I Jordan and Robert A Jacobs. 1994. Hierarchical mixtures of experts and the EM algorithm. *Neural computation* 6, 2 (1994), 181–214.

K. Kameyama and Y. Kosugi. 1991. Automatic fusion and splitting of artificial neural elements in optimizing the network size. In *Conference Proceedings 1991 IEEE International Conference on Systems, Man, and Cybernetics*. 1633–1638 vol.3. https://doi.org/10.1109/ICSMC.1991.169926

Minsoo Kang and Bohyung Han. 2020. Operation-Aware Soft Channel Pruning using Differentiable Masks. (2020). arXiv:cs.LG/2007.03938

P. P. Kanjilal, P. K. Dey, and D. N. Banerjee. 1993. Reduced-size neural networks through singular value decomposition and subset selection. *Electronics Letters* 29, 17 (1993), 1516–1518. https://doi.org/10.1049/el:19931010

Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B. Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. 2020. Scaling Laws for Neural Language Models. (2020). arXiv:cs.LG/2001.08361

Sai Praneeth Karimireddy, Quentin Rebjock, Sebastian U Stich, and Martin Jaggi. 2019. Error feedback fixes SignSGD and other gradient compression schemes. In *Proceedings of the Thirty-sixth International Conference on Machine Learning*. 3252–3261. arXiv:cs.LG/1901.09847

E. D. Karnin. 1990. A simple procedure for pruning back-propagation trained neural networks. *IEEE Transactions on Neural Networks* 1, 2 (1990), 239–242. https://doi.org/10.1109/72.80236

Jason N. D. Kerr, David Greenberg, and Fritjof Helmchen. 2005. Imaging input and output of neocortical networks in vivo. *Proceedings of the National Academy of Sciences* 102, 39 (2005), 14063–14068. https://doi.org/10.1073/pnas.0506029102 arXiv:https://www.pnas.org/content/102/39/14063.full.pdf

D. Kim, J. Ahn, and S. Yoo. 2018. ZeNA: Zero-Aware Neural Network Accelerator. *IEEE Design Test* 35, 1 (2018), 39–46. https://doi.org/10.1109/MDAT.2017.2741463

Diederik P Kingma, Tim Salimans, and Max Welling. 2015. Variational Dropout and the Local Reparameterization Trick. In *Advances in Neural Information Processing Systems*, C. Cortes, N. Lawrence, D. Lee, M. Sugiyama, and R. Garnett (Eds.), Vol. 28. Curran Associates, Inc., 2575–2583. https://proceedings.neurips.cc/paper/2015/file/bc7316929fe1545bf0b98d114ee3ecb8-Paper.pdf

Diederik P Kingma and Max Welling. 2013. Auto-encoding variational bayes. (2013). arXiv:cs.LG/1312.6114

Maxim Kodryan, Artem Grachev, Dmitry Ignatov, and Dmitry Vetrov. 2019. Efficient Language Modeling with Automatic Relevance Determination in Recurrent Neural Networks. In *Proceedings of the 4th Workshop on Representation Learning for NLP (RepL4NLP-2019)*. 40–48.

Jakub Konečnỳ and Peter Richtárik. 2018. Randomized distributed mean estimation: Accuracy vs. communication. *Frontiers in Applied Mathematics and Statistics* 4 (2018), 62. arXiv:cs.DC/1611.07555

Anders Krogh and John A. Hertz. 1991. A Simple Weight Decay Can Improve Generalization. In *Proceedings of the 4th International Conference on Neural Information Processing Systems (NIPS'91)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 950–957.

David Krueger, Tegan Maharaj, János Kramár, Mohammad Pezeshki, Nicolas Ballas, Nan Rosemary Ke, Anirudh Goyal, Yoshua Bengio, Aaron Courville, and Chris Pal. 2017. Zoneout: Regularizing RNNs by Randomly Preserving Hidden Activations. *International Conference on Learning Representations (ICLR)* (2017).

H. T. Kung, Bradley McDanel, and Sai Qian Zhang. 2018. Packing Sparse Convolutional Neural Networks for Efficient Systolic Array Implementations: Column Combining Under Joint Optimization. (2018). arXiv:cs.LG/1811.04770

Frederik Kunstner, Philipp Hennig, and Lukas Balles. 2019. Limitations of the empirical Fisher approximation for natural gradient descent. In *Advances in Neural Information Processing Systems*. 4156–4167.

Mark Kurtz, Justin Kopinsky, Rati Gelashvili, Alexander Matveev, John Carr, Michael Goin, William Leiserson, Sage Moore, Nir Shavit, and Dan Alistarh. 2020. Inducing and Exploiting Activation Sparsity for Fast Inference on Deep Neural Networks. In *International Conference on Machine Learning*. PMLR, 5533–5543.

Aditya Kusupati, Vivek Ramanujan, Raghav Somani, Mitchell Wortsman, Prateek Jain, Sham Kakade, and Ali Farhadi. 2020. Soft Threshold Weight Reparameterization for Learnable Sparsity. (2020). arXiv:cs.LG/2002.03231

Andrey Kuzmin, Markus Nagel, Saurabh Pitre, Sandeep Pendyam, Tijmen Blankevoort, and Max Welling. 2019. Taxonomy and Evaluation of Structured Compression of Convolutional Neural Networks. (2019). arXiv:cs.LG/1912.09802

Tom Kwiatkowski, Jennimaria Palomaki, Olivia Redfield, Michael Collins, Ankur Parikh, Chris Alberti, Danielle Epstein, Illia Polosukhin, Jacob Devlin, Kenton Lee, et al. 2019. Natural questions: a benchmark for question answering research. *Transactions of the Association for Computational Linguistics* 7 (2019), 453–466.

Guillaume Lample, Alexandre Sablayrolles, Marc'Aurelio Ranzato, Ludovic Denoyer, and Hervé Jégou. 2019. Large Memory Layers with Product Keys. (2019). arXiv:cs.CL/1907.05242

Gustav Larsson, Michael Maire, and Gregory Shakhnarovich. 2017. FractalNet: Ultra-Deep Neural Networks without Residuals. *International Conference on Learning Representations (ICLR)* (2017).

Philippe Lauret, Eric Fock, and Thierry Alex Mara. 2006. A node pruning algorithm based on a Fourier amplitude sensitivity test method. *IEEE transactions on neural networks* 17, 2 (2006), 273–293.

A. Lavin and S. Gray. 2016. Fast Algorithms for Convolutional Neural Networks. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 4013–4021. https://doi.org/10.1109/CVPR.2016.435

Yann Le Cun, John S. Denker, and Sara A. Solla. 1990. *Optimal Brain Damage*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 598–605.

Vadim Lebedev and Victor Lempitsky. 2015. Fast ConvNets Using Group-wise Brain Damage. (2015). arXiv:cs.CV/1506.02515

Namhoon Lee, Thalaiyasingam Ajanthan, Stephen Gould, and Philip H. S. Torr. 2020a. A Signal Propagation Perspective for Pruning Neural Networks at Initialization. (2020). arXiv:cs.LG/1906.06307

Namhoon Lee, Thalaiyasingam Ajanthan, and Philip H. S. Torr. 2019. SNIP: Single-shot Network Pruning based on Connection Sensitivity. (2019). arXiv:cs.CV/1810.02340

Namhoon Lee, Thalaiyasingam Ajanthan, Philip H. S. Torr, and Martin Jaggi. 2020b. Understanding the Effects of Data Parallelism and Sparsity on Neural Network Training. (2020). arXiv:cs.LG/2003.11316

Dmitry Lepikhin, HyoukJoong Lee, Yuanzhong Xu, Dehao Chen, Orhan Firat, Yanping Huang, Maxim Krikun, Noam Shazeer, and Zhifeng Chen. 2020. GShard: Scaling Giant Models with Conditional Computation and Automatic Sharding. (2020). arXiv:cs.CL/2006.16668

Hao Li, Asim Kadav, Igor Durdanovic, Hanan Samet, and Hans Peter Graf. 2017. Pruning Filters for Efficient ConvNets. (2017). arXiv:cs.CV/1608.08710

J. Li, S. Jiang, S. Gong, J. Wu, J. Yan, G. Yan, and X. Li. 2019. SqueezeFlow: A Sparse CNN Accelerator Exploiting Concise Convolution Rules. *IEEE Trans. Comput.* 68, 11 (2019), 1663–1677. https://doi.org/10.1109/TC.2019.2924215

Xiaoya Li, Yuxian Meng, Mingxin Zhou, Qinghong Han, Fei Wu, and Jiwei Li. 2020. SAC: Accelerating and Structuring Self-Attention via Sparse Adaptive Connection. (2020). arXiv:cs.CL/2003.09833

Yunqiang Li, Silvia Laura Pintea, and Jan van Gemert. 2021. Less bits is more: How pruning deep binary networks increases weight capacity. (2021). https://openreview.net/forum?id=Hy8JM_Fvt5N

Yuanzhi Li, Colin Wei, and Tengyu Ma. 2020b. Towards Explaining the Regularization Effect of Initial Large Learning Rate in Training Neural Networks. (2020). arXiv:cs.LG/1907.04595

Zhuohan Li, Eric Wallace, Sheng Shen, Kevin Lin, Kurt Keutzer, Dan Klein, and Joseph E. Gonzalez. 2020a. Train Large, Then Compress: Rethinking Model Size for Efficient Training and Inference of Transformers. (2020). arXiv:cs.CL/2002.11794

Lucas Liebenwein, Cenk Baykal, Harry Lang, Dan Feldman, and Daniela Rus. 2020. Provable Filter Pruning for Efficient Neural Networks. (2020). arXiv:cs.LG/1911.07412

Timothy P. Lillicrap, Jonathan J. Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. 2019. Continuous control with deep reinforcement learning. (2019). arXiv:cs.LG/1509.02971

Timothy P Lillicrap, Adam Santoro, Luke Marris, Colin J Akerman, and Geoffrey Hinton. 2020. Backpropagation and the brain. *Nature Reviews Neuroscience* (2020), 1–12.

Hyeontaek Lim, David Andersen, and Michael Kaminsky. 2019. 3LC: Lightweight and Effective Traffic Compression for Distributed Machine Learning. In *Proceedings of the Conference on Systems and Machine Learning*. arXiv:cs.LG/1802.07389

Ji Lin, Yongming Rao, Jiwen Lu, and Jie Zhou. 2017. Runtime Neural Pruning. In *Advances in Neural Information Processing Systems*, I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett (Eds.), Vol. 30. Curran Associates, Inc., 2181–2191. https://proceedings.neurips.cc/paper/2017/file/a51fb975227d6640e4fe47854476d133-Paper.

pdf

Tao Lin, Sebastian U. Stich, Luis Barba, Daniil Dmitriev, and Martin Jaggi. 2020. Dynamic Model Pruning with Feedback. (2020). arXiv:cs.LG/2006.07253

Yujun Lin, Song Han, Huizi Mao, Yu Wang, and William J Dally. 2018. Deep gradient compression: Reducing the communication bandwidth for distributed training. In *Proceedings of the Sixth International Conference on Learning Representations*. arXiv:cs.CV/1712.01887

Zi Lin, Jeremiah Zhe Liu, Zi Yang, Nan Hua, and Dan Roth. 2020. Pruning Redundant Mappings in Transformer Models via Spectral-Normalized Identity Prior. In *Findings of the Association for Computational Linguistics: EMNLP 2020*. 719–730. arXiv:cs.CL/2010.01791

Pierre Lison, Jörg Tiedemann, Milen Kouylekov, et al. 2019. Open subtitles 2018: Statistical rescoring of sentence alignments in large, noisy parallel corpora. In *LREC 2018, Eleventh International Conference on Language Resources and Evaluation*. European Language Resources Association (ELRA).

Baoyuan Liu, Min Wang, H. Foroosh, M. Tappen, and M. Penksy. 2015b. Sparse Convolutional Neural Networks. In *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 806–814. https://doi.org/10.1109/CVPR.2015.7298681

Lanlan Liu and Jia Deng. 2018. Dynamic Deep Neural Networks: Optimizing Accuracy-Efficiency Trade-offs by Selective Execution. (2018). arXiv:cs.LG/1701.00299

Liu Liu, Lei Deng, Xing Hu, Maohua Zhu, Guoqi Li, Yufei Ding, and Yuan Xie. 2019. Dynamic Sparse Graph for Efficient Deep Learning. (2019). arXiv:cs.LG/1810.00859

Tianlin Liu and Friedemann Zenke. 2020. Finding trainable sparse networks through Neural Tangent Transfer. (2020). arXiv:cs.LG/2006.08228

Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. 2019a. RoBERTa: A robustly optimized BERT pretraining approach. (2019). arXiv:cs.CL/1907.11692

Zhuang Liu, Jianguo Li, Zhiqiang Shen, Gao Huang, Shoumeng Yan, and Changshui Zhang. 2017. Learning Efficient Convolutional Networks through Network Slimming. (2017). arXiv:cs.CV/1708.06519

Ziwei Liu, Ping Luo, Xiaogang Wang, and Xiaoou Tang. 2015a. Deep learning face attributes in the wild. In *Proceedings of the IEEE international conference on computer vision*. 3730–3738. arXiv:cs.CV/1411.7766

Zhuang Liu, Mingjie Sun, Tinghui Zhou, Gao Huang, and Trevor Darrell. 2019b. Rethinking the Value of Network Pruning. (2019). arXiv:1810.05270

Ekaterina Lobacheva, Nadezhda Chirkova, and Dmitry Vetrov. 2018. Bayesian sparsification of gated recurrent neural networks. (2018). arXiv:cs.LG/1812.05692

Ilya Loshchilov and Frank Hutter. 2019. Decoupled weight decay regularization. In *Proceedings of the Seventh International Conference on Learning Representations*. arXiv:1711.05101

Christos Louizos, Karen Ullrich, and Max Welling. 2017. Bayesian Compression for Deep Learning. (2017). arXiv:stat.ML/1705.08665

Christos Louizos, Max Welling, and Diederik P. Kingma. 2018. Learning Sparse Neural Networks through L_0 Regularization. (2018). arXiv:stat.ML/1712.01312

Jian-Hao Luo and Jianxin Wu. 2019. AutoPruner: An End-to-End Trainable Filter Pruning Method for Efficient Deep Model Inference. (2019). arXiv:cs.CV/1805.08941

Jian-Hao Luo, Jianxin Wu, and Weiyao Lin. 2017. ThiNet: A Filter Level Pruning Method for Deep Neural Network Compression. (2017). arXiv:cs.CV/1707.06342

Alexander Ly, Maarten Marsman, Josine Verhagen, Raoul Grasman, and Eric-Jan Wagenmakers. 2017. A Tutorial on Fisher Information. (2017). arXiv:math.ST/1705.01064

Sangkug Lym, Esha Choukse, Siavash Zangeneh, Wei Wen, Sujay Sanghavi, and Mattan Erez. 2019. PruneTrain. *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (Nov 2019). https://doi.org/10.1145/3295500.3356156

Divyam Madaan, Jinwoo Shin, and Sung Ju Hwang. 2020. Adversarial Neural Pruning with Latent Vulnerability Suppression. (2020). arXiv:cs.LG/1908.04355

Chris J. Maddison, Andriy Mnih, and Yee Whye Teh. 2017. The Concrete Distribution: A Continuous Relaxation of Discrete Random Variables. *International Conference on Learning Representations (ICLR)* (2017).

Alireza Makhzani and Brendan Frey. 2015. Winner-Take-All Autoencoders. (2015). arXiv:cs.LG/1409.2752

Eran Malach, Gilad Yehudai, Shai Shalev-Shwartz, and Ohad Shamir. 2020. Proving the Lottery Ticket Hypothesis: Pruning is All You Need. (2020). arXiv:cs.LG/2002.00585

Chaitanya Malaviya, Pedro Ferreira, and André FT Martins. 2018. Sparse and constrained attention for neural machine translation. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*. arXiv:cs.CL/1805.08241

Franco Manessi, Alessandro Rozza, Simone Bianco, Paolo Napoletano, and Raimondo Schettini. 2018. Automated Pruning for Deep Neural Network Compression. *2018 24th International Conference on Pattern Recognition (ICPR)* (Aug 2018).

https://doi.org/10.1109/icpr.2018.8546129

Huizi Mao, Song Han, Jeff Pool, Wenshuo Li, Xingyu Liu, Yu Wang, and William J. Dally. 2017. Exploring the Regularity of Sparse Structure in Convolutional Neural Networks. (2017). arXiv:cs.LG/1705.08922

Zelda Mariet and Suvrit Sra. 2017. Diversity Networks: Neural Network Compression Using Determinantal Point Processes. (2017). arXiv:cs.LG/1511.05077

James Martens and Roger Grosse. 2015. Optimizing Neural Networks with Kronecker-factored Approximate Curvature. (2015). arXiv:cs.LG/1503.05671

Andre Martins and Ramon Astudillo. 2016. From softmax to sparsemax: A sparse model of attention and multi-label classification. In *International Conference on Machine Learning*. 1614–1623. arXiv:cs.CL/1602.02068

Peter Mattson, Christine Cheng, Cody Coleman, Greg Diamos, Paulius Micikevicius, David Patterson, Hanlin Tang, Gu-Yeon Wei, Peter Bailis, Victor Bittorf, David Brooks, Dehao Chen, Debojyoti Dutta, Udit Gupta, Kim Hazelwood, Andrew Hock, Xinyuan Huang, Atsushi Ike, Bill Jia, Daniel Kang, David Kanter, Naveen Kumar, Jeffery Liao, Guokai Ma, Deepak Narayanan, Tayo Oguntebi, Gennady Pekhimenko, Lillian Pentecost, Vijay Janapa Reddi, Taylor Robie, Tom St. John, Tsuguchika Tabaru, Carole-Jean Wu, Lingjie Xu, Masafumi Yamazaki, Cliff Young, and Matei Zaharia. 2020. MLPerf Training Benchmark. (2020). arXiv:cs.LG/1910.01500

Sam McCandlish, Jared Kaplan, Dario Amodei, and OpenAI Dota Team. 2018. An Empirical Model of Large-Batch Training. (2018). arXiv:cs.LG/1812.06162

J. S. McCarley, Rishav Chakravarti, and Avirup Sil. 2020. Structured Pruning of a BERT-based Question Answering Model. (2020). arXiv:cs.CL/1910.06360

Rahul Mehta. 2019. Sparse Transfer Learning via Winning Lottery Tickets. (2019). arXiv:cs.LG/1905.07785

Fanxu Meng, Hao Cheng, Ke Li, Huixiang Luo, Xiaowei Guo, Guangming Lu, and Xing Sun. 2020. Pruning Filter in Filter. (2020). arXiv:cs.CV/2009.14410

Hrushikesh Mhaskar and Tomaso Poggio. 2016. Deep vs. shallow networks : An approximation theory perspective. (2016). arXiv:cs.LG/1608.03287

Paul Michel, Omer Levy, and Graham Neubig. 2019. Are Sixteen Heads Really Better than One? (2019). arXiv:cs.CL/1905.10650

Beren Millidge, Alexander Tschantz, and Christopher L. Buckley. 2020. Predictive Coding Approximates Backprop along Arbitrary Computation Graphs. (2020). arXiv:cs.LG/2006.04182

Asit K. Mishra, Eriko Nurvitadhi, Jeffrey J. Cook, and Debbie Marr. 2017. WRPN: Wide Reduced-Precision Networks. *CoRR* abs/1709.01134 (2017). arXiv:1709.01134 http://arxiv.org/abs/1709.01134

Deepak Mittal, Shweta Bhardwaj, Mitesh M. Khapra, and Balaraman Ravindran. 2018. Recovering from Random Pruning: On the Plasticity of Deep Convolutional Neural Networks. (2018). arXiv:cs.CV/1801.10447

Decebal Constantin Mocanu, Elena Mocanu, Peter Stone, Phuong H Nguyen, Madeleine Gibescu, and Antonio Liotta. 2018. Scalable training of artificial neural networks with adaptive sparse connectivity inspired by network science. *Nature communications* 9, 1 (2018), 1–12.

Dmitry Molchanov, Arsenii Ashukha, and Dmitry Vetrov. 2017. Variational Dropout Sparsifies Deep Neural Networks. (2017). arXiv:stat.ML/1701.05369

Pavlo Molchanov, Arun Mallya, Stephen Tyree, Iuri Frosio, and Jan Kautz. 2019. Importance Estimation for Neural Network Pruning. (2019). arXiv:cs.LG/1906.10771

Pavlo Molchanov, Stephen Tyree, Tero Karras, Timo Aila, and Jan Kautz. 2017. Pruning Convolutional Neural Networks for Resource Efficient Inference. (2017). arXiv:cs.LG/1611.06440

John E Moody. 1991. Note on generalization, regularization and architecture selection in nonlinear learning systems. In *Neural Networks for Signal Processing Proceedings of the 1991 IEEE Workshop*. IEEE, 1–10.

Ari S. Morcos, Haonan Yu, Michela Paganini, and Yuandong Tian. 2019. One ticket to win them all: generalizing lottery ticket initializations across datasets and optimizers. (2019). arXiv:stat.ML/1906.02773

Hesham Mostafa and Xin Wang. 2019. Parameter Efficient Training of Deep Convolutional Neural Networks by Dynamic Sparse Reparameterization. (2019). arXiv:cs.LG/1902.05967

Michael C Mozer and Paul Smolensky. 1988. Skeletonization: A technique for trimming the fat from a network via relevance assessment. *Advances in neural information processing systems* 1 (1988), 107–115.

Sayan Mukherjee, Partha Niyogi, Tomaso Poggio, and Ryan Rifkin. 2006. Learning theory: stability is sufficient for generalization and necessary and sufficient for consistency of empirical risk minimization. *Advances in Computational Mathematics* 25, 1-3 (2006), 161–193.

Ben Mussay, Daniel Feldman, Samson Zhou, Vladimir Braverman, and Margarita Osadchy. 2020. Data-Independent Structured Pruning of Neural Networks via Coresets. (2020). arXiv:cs.LG/2008.08316

Sharan Narang, Erich Elsen, Gregory Diamos, and Shubho Sengupta. 2017. Exploring Sparsity in Recurrent Neural Networks. (2017). arXiv:cs.LG/1704.05119

Pramod L. Narasimha, Walter H. Delashmit, Michael T. Manry, Jiang Li, and Francisco Maldonado. 2008. An integrated growing-pruning method for feedforward network training. *Neurocomputing* 71, 13 (2008), 2831 – 2847. https://doi.org/

10.1016/j.neucom.2007.08.026 Artificial Neural Networks (ICANN 2006) / Engineering of Intelligent Systems (ICEIS 2006).

Kirill Neklyudov, Dmitry Molchanov, Arsenii Ashukha, and Dmitry Vetrov. 2017. Structured Bayesian Pruning via Log-Normal Multiplicative Noise. (2017). arXiv:stat.ML/1705.07283

Behnam Neyshabur, Zhiyuan Li, Srinadh Bhojanapalli, Yann LeCun, and Nathan Srebro. 2018. Towards Understanding the Role of Over-Parametrization in Generalization of Neural Networks. (2018). arXiv:cs.LG/1805.12076

J. Ngiam, Z. Chen, D. Chia, P. W. Koh, Q. V. Le, and A. Y. Ng. 2010. Tiled convolutional neural networks. In *Advances in Neural Information Processing Systems 23*. 1279–1287.

Vlad Niculae and Mathieu Blondel. 2017. A regularized framework for sparse and structured neural attention. In *Advances in neural information processing systems*. 3338–3348. arXiv:stat.ML/1705.07704

Nils J Nilsson. 2009. *The quest for artificial intelligence: A history of ideas and achievements*. Cambridge University Press.

Yue Niu, Rajgopal Kannan, Ajitesh Srivastava, and Viktor Prasanna. 2020. Reuse Kernels or Activations? A Flexible Dataflow for Low-Latency Spectral CNN Acceleration. In *Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA '20)*. Association for Computing Machinery, New York, NY, USA, 266–276. https://doi.org/10.1145/3373087.3375302

Yue Niu, Hanqing Zeng, Ajitesh Srivastava, Kartik Lakhotia, Rajgopal Kannan, Yanzhi Wang, and Viktor Prasanna. 2019. SPEC2: SPECtral SParsE CNN Accelerator on FPGAs. (2019). arXiv:cs.CV/1910.11103

Hyeonwoo Noh, Seunghoon Hong, and Bohyung Han. 2015. Learning Deconvolution Network for Semantic Segmentation. (2015). arXiv:cs.CV/1505.04366

Steven J Nowlan and Geoffrey E Hinton. 1992. Simplifying neural networks by soft weight-sharing. *Neural computation* 4, 4 (1992), 473–493.

Nvidia. 2020. NVIDIA A100 Tensor Core GPU Architecture. (2020).

Bruno A Olshausen and David J Field. 1996. Emergence of simple-cell receptive field properties by learning a sparse code for natural images. *Nature* 381, 6583 (1996), 607–609.

Laurent Orseau, Marcus Hutter, and Omar Rivasplata. 2020. Logarithmic Pruning is All You Need. (2020). arXiv:cs.LG/2006.12156

Kazuki Osawa, Yohei Tsuji, Yuichiro Ueno, Akira Naruse, Rio Yokota, and Satoshi Matsuoka. 2019. Large-Scale Distributed Second-Order Optimization Using Kronecker-Factored Approximate Curvature for Deep Convolutional Neural Networks. *2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)* (Jun 2019). https://doi.org/10.1109/cvpr.2019.01264

Wei Pan, Hao Dong, and Yike Guo. 2016. DropNeuron: Simplifying the Structure of Deep Neural Networks. (2016). arXiv:cs.CV/1606.07326

Angshuman Parashar, Minsoo Rhu, Anurag Mukkara, Antonio Puglielli, Rangharajan Venkatesan, Brucek Khailany, Joel Emer, Stephen W. Keckler, and William J. Dally. 2017. SCNN: An Accelerator for Compressed-sparse Convolutional Neural Networks. (2017). arXiv:cs.NE/1708.04485

Jongsoo Park, Sheng Li, Wei Wen, Ping Tak Peter Tang, Hai Li, Yiran Chen, and Pradeep Dubey. 2017. Faster CNNs with Direct Sparse Convolutions and Guided Pruning. (2017). arXiv:cs.CV/1608.01409

Niki Parmar, Ashish Vaswani, Jakob Uszkoreit, Lukasz Kaiser, Noam Shazeer, Alexander Ku, and Dustin Tran. 2018. Image Transformer. In *International Conference on Machine Learning*. 4055–4064. arXiv:cs.CV/1802.05751

Morten Pedersen, Lars Hansen, and Jan Larsen. 1996. Pruning with generalization based weight saliencies: lambda OBD, lambda OBS. In *Advances in Neural Information Processing Systems*, D. Touretzky, M. C. Mozer, and M. Hasselmo (Eds.), Vol. 8. MIT Press, 521–527. https://proceedings.neurips.cc/paper/1995/file/3473decccb0509fb264818a7512a8b9b-Paper.pdf

Ankit Pensia, Shashank Rajput, Alliot Nagle, Harit Vishwakarma, and Dimitris Papailiopoulos. 2020. Optimal Lottery Tickets via SubsetSum: Logarithmic Over-Parameterization is Sufficient. (2020). arXiv:cs.LG/2006.07990

Bryan A. Plummer, Nikoli Dryden, Julius Frost, Torsten Hoefler, and Kate Saenko. 2020. Shapeshifter Networks: Cross-layer Parameter Sharing for Scalable and Effective Deep Learning. (2020). arXiv:cs.LG/2006.10598

A. Polyak and L. Wolf. 2015. Channel-level acceleration of deep face representations. *IEEE Access* 3 (2015), 2163–2175. https://doi.org/10.1109/ACCESS.2015.2494536

Udo W. Pooch and Al Nieder. 1973. A Survey of Indexing Techniques for Sparse Matrices. *ACM Comput. Surv.* 5, 2 (June 1973), 109–133. https://doi.org/10.1145/356616.356618

Ameya Prabhu, Girish Varma, and Anoop Namboodiri. 2018. Deep Expander Networks: Efficient Deep Networks from Graph Theory. (2018). arXiv:cs.CV/1711.08757

Sai Prasanna, Anna Rogers, and Anna Rumshisky. 2020. When BERT Plays the Lottery, All Tickets Are Winning. (2020). arXiv:cs.CL/2005.00561

Lutz Prechelt. 1997. Connection pruning with static and adaptive pruning schedules. *Neurocomputing* 16, 1 (1997), 49 – 61. https://doi.org/10.1016/S0925-2312(96)00054-9

E. Qin, A. Samajdar, H. Kwon, V. Nadella, S. Srinivasan, D. Das, B. Kaul, and T. Krishna. 2020. SIGMA: A Sparse and Irregular GEMM Accelerator with Flexible Interconnects for DNN Training. In *2020 IEEE International Symposium on*

High Performance Computer Architecture (HPCA). 58–70. https://doi.org/10.1109/HPCA47549.2020.00015

Md Aamir Raihan and Tor M. Aamodt. 2020. Sparse Weight Activation Training. (2020). arXiv:cs.LG/2001.01969

Adnan Siraj Rakin, Zhezhi He, Li Yang, Yanzhi Wang, Liqiang Wang, and Deliang Fan. 2020. Robust Sparse Regularization: Defending Adversarial Attacks Via Regularized Sparse Network. In *Proceedings of the 2020 on Great Lakes Symposium on VLSI (GLSVLSI '20)*. Association for Computing Machinery, New York, NY, USA, 125–130. https://doi.org/10.1145/3386263.3407651

Vivek Ramanujan, Mitchell Wortsman, Aniruddha Kembhavi, Ali Farhadi, and Mohammad Rastegari. 2020. What's Hidden in a Randomly Weighted Neural Network? (2020). arXiv:cs.CV/1911.13299

Carl Edward Rasmussen and Zoubin Ghahramani. 2001. Occam's razor. In *Advances in neural information processing systems*. 294–300.

B. Reagen, P. Whatmough, R. Adolf, S. Rama, H. Lee, S. K. Lee, J. M. Hernández-Lobato, G. Wei, and D. Brooks. 2016. Minerva: Enabling Low-Power, Highly-Accurate Deep Neural Network Accelerators. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*. 267–278. https://doi.org/10.1109/ISCA.2016.32

R. Reed. 1993. Pruning algorithms-a survey. *IEEE Transactions on Neural Networks* 4, 5 (1993), 740–747. https://doi.org/10.1109/72.248452

Alex Renda, Jonathan Frankle, and Michael Carbin. 2020. Comparing Rewinding and Fine-tuning in Neural Network Pruning. (2020). arXiv:cs.LG/2003.02389

Cèdric Renggli, Saleh Ashkboos, Mehdi Aghagolzadeh, Dan Alistarh, and Torsten Hoefler. 2019. SparCML: High-performance sparse communication for machine learning. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–15. arXiv:cs.DC/1802.08021

Albert Reuther, Peter Michaleas, Michael Jones, Vijay Gadepally, Siddharth Samsi, and Jeremy Kepner. 2020. Survey of Machine Learning Accelerators. (2020). arXiv:cs.DC/2009.00993

Danilo Jimenez Rezende, Shakir Mohamed, and Daan Wierstra. 2014. Stochastic backpropagation and variational inference in deep latent gaussian models. In *International Conference on Machine Learning*, Vol. 2.

Minsoo Rhu, Mike O'Connor, Niladrish Chatterjee, Jeff Pool, Youngeun Kwon, and Stephen W Keckler. 2018. Compressing DMA engine: Leveraging activation sparsity for training deep neural networks. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 78–91.

Anna Rogers, Olga Kovaleva, and Anna Rumshisky. 2021. A primer in BERTology: What we know about how bert works. *Transactions of the Association for Computational Linguistics* 8 (2021), 842–866. arXiv:cs.CL/2002.12327

Clemens Rosenbaum, Tim Klinger, and Matthew Riemer. 2017. Routing Networks: Adaptive Selection of Non-linear Functions for Multi-Task Learning. (2017). arXiv:cs.LG/1711.01239

Stuart Russell and Peter Norvig. 2020. *Artificial Intelligence: A Modern Approach* (4th ed.). Prentice Hall Press.

T. N. Sainath, B. Kingsbury, V. Sindhwani, E. Arisoy, and B. Ramabhadran. 2013. Low-rank matrix factorization for Deep Neural Network training with high-dimensional output targets. In *2013 IEEE International Conference on Acoustics, Speech and Signal Processing*. 6655–6659. https://doi.org/10.1109/ICASSP.2013.6638949

Victor Sanh, Thomas Wolf, and Alexander M. Rush. 2020. Movement Pruning: Adaptive Sparsity by Fine-Tuning. (2020). arXiv:cs.CL/2005.07683

Pedro Savarese, Hugo Silva, and Michael Maire. 2020. Winning the Lottery with Continuous Sparsification. (2020). arXiv:cs.LG/1912.04427

Simone Scardapane, Danilo Comminiello, Amir Hussain, and Aurelio Uncini. 2017. Group sparse regularization for deep neural networks. *Neurocomputing* 241 (2017), 81 – 89. https://doi.org/10.1016/j.neucom.2017.02.029

Paul Scheffler, Florian Zaruba, Fabian Schuiki, Torsten Hoefler, and Luca Benini. 2020. Indirection Stream Semantic Register Architecture for Efficient Sparse-Dense Linear Algebra. (2020). arXiv:cs.AR/2011.08070

Abigail See, Minh-Thang Luong, and Christopher D. Manning. 2016. Compression of Neural Machine Translation Models via Pruning. (2016). arXiv:cs.AI/1606.09274

Vikash Sehwag, Shiqi Wang, Prateek Mittal, and Suman Jana. 2020. HYDRA: Pruning Adversarially Robust Neural Networks. (2020). arXiv:cs.CV/2002.10509

Frank Seide, Hao Fu, Jasha Droppo, Gang Li, and Dong Yu. 2014. 1-bit stochastic gradient descent and its application to data-parallel distributed training of speech DNNs. In *Fifteenth Annual Conference of the International Speech Communication Association*.

Aditya Sharma, Nikolas Wolfe, and Bhiksha Raj. 2017. The Incredible Shrinking Neural Network: New Perspectives on Learning Representations Through The Lens of Pruning. (2017). arXiv:cs.NE/1701.04465

Noam Shazeer, Azalia Mirhoseini, Krzysztof Maziarz, Andy Davis, Quoc Le, Geoffrey Hinton, and Jeff Dean. 2017. Outrageously Large Neural Networks: The Sparsely-Gated Mixture-of-Experts Layer. (2017). arXiv:cs.LG/1701.06538

Shaohuai Shi, Qiang Wang, Kaiyong Zhao, Zhenheng Tang, Yuxin Wang, Xiang Huang, and Xiaowen Chu. 2019a. A distributed synchronous SGD algorithm with global Top-k sparsification for low bandwidth networks. In *2019 IEEE 39th International Conference on Distributed Computing Systems Workshop on Networks*. 2238–2247. arXiv:cs.DC/1901.04359

Shaohuai Shi, Kaiyong Zhao, Qiang Wang, Zhenheng Tang, and Xiaowen Chu. 2019b. A Convergence Analysis of Distributed SGD with Communication-Efficient Gradient Sparsification.. In *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence*. 3411–3417.

Reza Shokri and Vitaly Shmatikov. 2015. Privacy-preserving deep learning. In *Proceedings of the 22nd ACM SIGSAC conference on computer and communications security*. 1310–1321.

Ravid Shwartz-Ziv and Naftali Tishby. 2017. Opening the Black Box of Deep Neural Networks via Information. (2017). arXiv:cs.LG/1703.00810

Sietsma and Dow. 1988. Neural net pruning-why and how. In *IEEE 1988 International Conference on Neural Networks*. 325–333 vol.1. https://doi.org/10.1109/ICNN.1988.23864

Jocelyn Sietsma and Robert JF Dow. 1991. Creating artificial neural networks that generalize. *Neural networks* 4, 1 (1991), 67–79.

Laurent Sifre and Stéphane Mallat. 2014. *Rigid-motion scattering for image classification*. Ph.D. Dissertation. Ecole Polytechnique, CMAP.

Sidak Pal Singh and Dan Alistarh. 2020. WoodFisher: Efficient Second-Order Approximation for Neural Network Compression. (2020). arXiv:cs.LG/2004.14340

Samarth Sinha, Zhengli Zhao, Anirudh Goyal, Colin A Raffel, and Augustus Odena. 2020. Top-k Training of GANs: Improving GAN Performance by Throwing Away Bad Samples. In *Advances in Neural Information Processing Systems*. arXiv:stat.ML/2002.06224

Samuel L. Smith, Pieter-Jan Kindermans, Chris Ying, and Quoc V. Le. 2018. Don't Decay the Learning Rate, Increase the Batch Size. (2018). arXiv:cs.LG/1711.00489

Richard Socher, Alex Perelygin, Jean Wu, Jason Chuang, Christopher D Manning, Andrew Y Ng, and Christopher Potts. 2013. Recursive deep models for semantic compositionality over a sentiment treebank. In *Proceedings of the 2013 conference on empirical methods in natural language processing*. 1631–1642.

Suraj Srinivas and R. Venkatesh Babu. 2015. Data-free parameter pruning for Deep Neural Networks. (2015). arXiv:cs.CV/1507.06149

Suraj Srinivas and R. Venkatesh Babu. 2016. Learning Neural Network Architectures using Backpropagation. (2016). arXiv:cs.LG/1511.05497

Suraj Srinivas, Akshayvarun Subramanya, and R. Venkatesh Babu. 2016. Training Sparse Neural Networks. (2016). arXiv:cs.CV/1611.06694

Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. 2014a. Dropout: A Simple Way to Prevent Neural Networks from Overfitting. *Journal of Machine Learning Research* 15, 56 (2014), 1929–1958.

Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. 2014b. Dropout: A Simple Way to Prevent Neural Networks from Overfitting. *J. Mach. Learn. Res.* 15, 1 (Jan. 2014), 1929–1958.

Sebastian U Stich, Jean-Baptiste Cordonnier, and Martin Jaggi. 2018. Sparsified SGD with memory. In *Advances in Neural Information Processing Systems*. 4447–4458. arXiv:cs.LG/1809.07599

Nikko Ström. 1997. Sparse connection and pruning in large dynamic artificial neural networks. In *Fifth European Conference on Speech Communication and Technology*.

Nikko Strom. 2015. Scalable distributed DNN training using commodity GPU cloud computing. In *Sixteenth Annual Conference of the International Speech Communication Association*.

Jingtong Su, Yihang Chen, Tianle Cai, Tianhao Wu, Ruiqi Gao, Liwei Wang, and Jason D. Lee. 2020. Sanity-Checking Pruning Methods: Random Tickets can Win the Jackpot. (2020). arXiv:cs.LG/2009.11094

Xavier Suau, Luca Zappella, and Nicholas Apostoloff. 2019. Filter Distillation for Network Compression. (2019). arXiv:cs.CV/1807.10585

Haobo Sun, Yingxia Shao, Jiawei Jiang, Bin Cui, Kai Lei, Yu Xu, and Jiang Wang. 2019. Sparse gradient compression for distributed SGD. In *International Conference on Database Systems for Advanced Applications*. Springer, 139–155.

Xu Sun, Xuancheng Ren, Shuming Ma, and Houfeng Wang. 2017. meProp: Sparsified back propagation for accelerated deep learning with reduced overfitting. In *Proceedings of the Thirty-Fourth International Conference on Machine Learning*. arXiv:cs.LG/1706.06197

Yi Sun, Xiaogang Wang, and Xiaoou Tang. 2015. Sparsifying Neural Network Connections for Face Recognition. (2015). arXiv:cs.CV/1512.01891

Ananda Theertha Suresh, X Yu Felix, Sanjiv Kumar, and H Brendan McMahan. 2017. Distributed mean estimation with limited communication. In *International Conference on Machine Learning*. 3329–3337. arXiv:cs.LG/1611.00429

Kenji Suzuki, Isao Horiba, and Noboru Sugie. 2001. A simple neural network pruning algorithm with application to filter synthesis. In *Neural Processing Letters*. 43–53.

V. Sze, Y. Chen, T. Yang, and J. S. Emer. 2017. Efficient Processing of Deep Neural Networks: A Tutorial and Survey. *Proc. IEEE* 105, 12 (2017), 2295–2329. https://doi.org/10.1109/JPROC.2017.2761740

Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. 2015. Going Deeper with Convolutions. In *Computer Vision and Pattern Recognition (CVPR)*. http://arxiv.org/abs/1409.4842

C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna. 2016. Rethinking the Inception Architecture for Computer Vision. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. IEEE Computer Society, Los Alamitos, CA, USA, 2818–2826. https://doi.org/10.1109/CVPR.2016.308

S. Tamura, M. Tateishi, M. Matumoto, and S. Akita. 1993. Determination of the number of redundant hidden units in a three-layered feedforward neural network. In *Proceedings of 1993 International Conference on Neural Networks (IJCNN-93-Nagoya, Japan)*, Vol. 1. 335–338 vol.1. https://doi.org/10.1109/IJCNN.1993.713925

Chong Min John Tan and Mehul Motani. 2020. DropNet: Reducing Neural Network Complexity via Iterative Pruning. In *Proceedings of the 37th International Conference on Machine Learning (Proceedings of Machine Learning Research)*, Hal Daumé III and Aarti Singh (Eds.), Vol. 119. PMLR, 9356–9366. http://proceedings.mlr.press/v119/tan20a.html

Mingxing Tan, Bo Chen, Ruoming Pang, Vijay Vasudevan, Mark Sandler, Andrew Howard, and Quoc V. Le. 2019. MnasNet: Platform-Aware Neural Architecture Search for Mobile. (2019). arXiv:cs.CV/1807.11626

Mingxing Tan and Quoc V. Le. 2020. EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks. (2020). arXiv:cs.LG/1905.11946

Hidenori Tanaka, Daniel Kunin, Daniel L. K. Yamins, and Surya Ganguli. 2020. Pruning neural networks without any data by iteratively conserving synaptic flow. (2020). arXiv:cs.LG/2006.05467

Hanlin Tang, Chen Yu, Xiangru Lian, Tong Zhang, and Ji Liu. 2019. DoubleSqueeze: Parallel stochastic gradient descent with double-pass error-compensated compression. In *Proceedings of the Thirty-sixth International Conference on Machine Learning*. 6155–6165. arXiv:cs.DC/1905.05957

Yehui Tang, Yunhe Wang, Yixing Xu, Dacheng Tao, Chunjing Xu, Chao Xu, and Chang Xu. 2021. SCOP: Scientific Control for Reliable Neural Network Pruning. (2021). arXiv:cs.CV/2010.10732

Zhenheng Tang, Shaohuai Shi, Xiaowen Chu, Wei Wang, and Bo Li. 2020. Communication-efficient distributed deep learning: A comprehensive survey. (2020). arXiv:cs.DC/2003.06307

Enzo Tartaglione, Skjalg Lepsøy, Attilio Fiandrotti, and Gianluca Francini. 2018. Learning Sparse Neural Networks via Sensitivity-Driven Regularization. (2018). arXiv:cs.LG/1810.11764

Yi Tay, Mostafa Dehghani, Samira Abnar, Yikang Shen, Dara Bahri, Philip Pham, Jinfeng Rao, Liu Yang, Sebastian Ruder, and Donald Metzler. 2021. Long Range Arena: A Benchmark for Efficient Transformers. In *Proceedings of the Ninth International Conference on Learning Representations*. arXiv:cs.LG/2011.04006

Yi Tay, Mostafa Dehghani, Dara Bahri, and Donald Metzler. 2020. Efficient transformers: A survey. (2020). arXiv:cs.LG/2009.06732

Ian Tenney, Dipanjan Das, and Ellie Pavlick. 2019. BERT rediscovers the classical NLP pipeline. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*. 4593–4601. arXiv:cs.CL/1905.05950

Lucas Theis, Iryna Korshunova, Alykhan Tejani, and Ferenc Huszár. 2018. Faster gaze prediction with dense networks and Fisher pruning. (2018). arXiv:cs.CV/1801.05787

Georg Thimm and Emile Fiesler. 1995. Evaluating Pruning Methods. In *National Chiao-Tung University*. 2.

Robert Tibshirani. 1996. Regression shrinkage and selection via the lasso. *Journal of the Royal Statistical Society: Series B (Methodological)* 58, 1 (1996), 267–288.

Michael E Tipping. 2001. Sparse Bayesian learning and the relevance vector machine. *Journal of machine learning research* 1, Jun (2001), 211–244.

Jonathan Tompson, Ross Goroshin, Arjun Jain, Yann LeCun, and Christopher Bregler. 2015. Efficient Object Localization Using Convolutional Networks. (2015). arXiv:cs.CV/1411.4280

Yusuke Tsuzuku, Hiroto Imachi, and Takuya Akiba. 2018. Variance-based gradient compression for efficient distributed deep learning. In *Proceedings of the Sixth International Conference on Learning Representations, Workshop Track*. arXiv:cs.LG/1802.06058

Karen Ullrich, Edward Meeds, and Max Welling. 2017. Soft Weight-Sharing for Neural Network Compression. (2017). arXiv:stat.ML/1702.04008

Didem Unat, Anshu Dubey, Torsten Hoefler, John Shalf, Mark Abraham, Mauro Bianco, Bradford L. Chamberlain, Romain Cledat, H. Carter Edwards, Hal Finkel, Karl Fuerlinger, Frank Hannig, Emmanuel Jeannot, Amir Kamil, Jeff Keasler, Paul H J Kelly, Vitus Leung, Hatem Ltaief, Naoya Maruyama, Chris J. Newburn, , and Miquel Pericas. 2017. Trends in Data Locality Abstractions for HPC Systems. *IEEE Transactions on Parallel and Distributed Systems (TPDS)* 28, 10 (Oct. 2017).

Mart van Baalen, Christos Louizos, Markus Nagel, Rana Ali Amjad, Ying Wang, Tijmen Blankevoort, and Max Welling. 2020. Bayesian Bits: Unifying Quantization and Pruning. (2020). arXiv:cs.LG/2005.07093

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention Is All You Need. (2017). arXiv:cs.CL/1706.03762

Stijn Verdenius, Maarten Stol, and Patrick Forré. 2020. Pruning via Iterative Ranking of Sensitivity Statistics. (2020). arXiv:cs.LG/2006.00896

Elena Voita, David Talbot, Fedor Moiseev, Rico Sennrich, and Ivan Titov. 2019. Analyzing Multi-Head Self-Attention: Specialized Heads Do the Heavy Lifting, the Rest Can Be Pruned. (2019). arXiv:cs.CL/1905.09418

Li Wan, Matthew Zeiler, Sixin Zhang, Yann Le Cun, and Rob Fergus. 2013. Regularization of Neural Networks using DropConnect. In *Proceedings of the 30th International Conference on Machine Learning (Proceedings of Machine Learning Research)*, Sanjoy Dasgupta and David McAllester (Eds.), Vol. 28. PMLR, Atlanta, Georgia, USA, 1058–1066. http://proceedings.mlr.press/v28/wan13.html

Alex Wang, Amanpreet Singh, Julian Michael, Felix Hill, Omer Levy, and Samuel R Bowman. 2019. GLUE: A multi-task benchmark and analysis platform for natural language understanding. In *Proceedings of the Seventh International Conference on Learning Representations*. arXiv:cs.CL/1804.07461

Chaoqi Wang, Roger Grosse, Sanja Fidler, and Guodong Zhang. 2019. Eigendamage: Structured pruning in the kronecker-factored eigenbasis. (2019). arXiv:cs.LG/1905.05934

Hongyi Wang, Scott Sievert, Shengchao Liu, Zachary Charles, Dimitris Papailiopoulos, and Stephen Wright. 2018. ATOMO: Communication-efficient learning via atomic sparsification. In *Advances in Neural Information Processing Systems*. 9850–9861. arXiv:stat.ML/1806.04090

Linnan Wang, Wei Wu, Junyu Zhang, Hang Liu, George Bosilca, Maurice Herlihy, and Rodrigo Fonseca. 2020b. FFT-based Gradient Sparsification for the Distributed Training of Deep Neural Networks. In *Proceedings of the 29th International Symposium on High-Performance Parallel and Distributed Computing*. 113–124.

Ziheng Wang, Jeremy Wohlwend, and Tao Lei. 2020a. Structured pruning of large language models. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. 6151–6162. arXiv:cs.CL/1910.04732

Jianqiao Wangni, Jialei Wang, Ji Liu, and Tong Zhang. 2018. Gradient sparsification for communication-efficient distributed optimization. In *Advances in Neural Information Processing Systems*. 1299–1309. arXiv:cs.LG/1710.09854

Alex Warstadt, Amanpreet Singh, and Samuel R Bowman. 2019. Neural network acceptability judgments. *Transactions of the Association for Computational Linguistics* 7 (2019), 625–641. arXiv:cs.CL/1805.12471

Bingzhen Wei, Xu Sun, Xuancheng Ren, and Jingjing Xu. 2017. Minimal Effort Back Propagation for Convolutional Neural Networks. (2017). arXiv:cs.LG/1709.05804

Wei Wen, Chunpeng Wu, Yandan Wang, Yiran Chen, and Hai Li. 2016. Learning Structured Sparsity in Deep Neural Networks. (2016). arXiv:cs.NE/1608.03665

David White and Panos A. Ligomenides. 1993. GANNet: A Genetic Algorithm for Optimizing Topology and Weights in Neural Network Design. In *Proceedings of the International Workshop on Artificial Neural Networks: New Trends in Neural Computation (IWANN '93)*. Springer-Verlag, Berlin, Heidelberg, 322–327.

D. Whitley and C. Bogart. 1990. The Evolution of Connectivity: Pruning Neural Networks Using Genetic Algorithms. In *Proceedings of the International Joint Conference on Neural Networks* (Washington, DC). IEEE Press, 134–137.

Adina Williams, Nikita Nangia, and Samuel R Bowman. 2018. A broad-coverage challenge corpus for sentence understanding through inference. In *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*. arXiv:cs.CL/1704.05426

P. M. Williams. 1995. Bayesian Regularization and Pruning Using a Laplace Prior. *Neural Computation* 7, 1 (1995), 117–143. https://doi.org/10.1162/neco.1995.7.1.117

Mitchell Wortsman, Ali Farhadi, and Mohammad Rastegari. 2019. Discovering Neural Wirings. (2019). arXiv:cs.LG/1906.00586

Yuhuai Wu, Elman Mansimov, Roger B. Grosse, Shun Liao, and Jimmy Ba. 2017. Second-order Optimization for Deep Reinforcement Learning using Kronecker-factored Approximation. In *NIPS*. 5285–5294. http://papers.nips.cc/paper/7112-second-order-optimization-for-deep-reinforcement-learning-using-kronecker-factored-approximation

Xia Xiao, Zigeng Wang, and Sanguthevar Rajasekaran. 2019. AutoPrune: Automatic Network Pruning by Regularizing Auxiliary Parameters. In *Advances in Neural Information Processing Systems*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett (Eds.), Vol. 32. Curran Associates, Inc., 13681–13691. https://proceedings.neurips.cc/paper/2019/file/4efc9e02abdab6b6166251918570a307-Paper.pdf

Jinhua Xu and Daniel WC Ho. 2006. A new training and pruning algorithm based on node dependence and Jacobian rank deficiency. *Neurocomputing* 70, 1-3 (2006), 544–558.

Dingqing Yang, Amin Ghasemazar, Xiaowei Ren, Maximilian Golub, Guy Lemieux, and Mieszko Lis. 2020a. Procrustes: a Dataflow and Accelerator for Sparse Deep Neural Network Training. (2020). arXiv:cs.NE/2009.10976

Huanrui Yang, Wei Wen, and Hai Li. 2020b. DeepHoyer: Learning Sparser Neural Network with Differentiable Scale-Invariant Sparsity Measures. (2020). arXiv:cs.LG/1908.09979

Tien-Ju Yang, Yu-Hsin Chen, and Vivienne Sze. 2017. Designing Energy-Efficient Convolutional Neural Networks using Energy-Aware Pruning. (2017). arXiv:cs.CV/1611.05128

Jianbo Ye, Xin Lu, Zhe Lin, and James Z Wang. 2018. Rethinking the smaller-norm-less-informative assumption in channel pruning of convolution layers. (2018). arXiv:cs.LG/1802.00124

Mao Ye, Chengyue Gong, Lizhen Nie, Denny Zhou, Adam Klivans, and Qiang Liu. 2020. Good Subnetworks Provably Exist: Pruning via Greedy Forward Selection. (2020). arXiv:cs.LG/2003.01794

Penghang Yin, Jiancheng Lyu, Shuai Zhang, Stanley Osher, Yingyong Qi, and Jack Xin. 2019. Understanding Straight-Through Estimator in Training Activation Quantized Neural Nets. (2019). arXiv:cs.LG/1903.05662

Haoran You, Chaojian Li, Pengfei Xu, Yonggan Fu, Yue Wang, Xiaohan Chen, Richard G. Baraniuk, Zhangyang Wang, and Yingyan Lin. 2020. Drawing early-bird tickets: Towards more efficient training of deep networks. (2020). arXiv:cs.LG/1909.11957

Zhonghui You, Kun Yan, Jinmian Ye, Meng Ma, and Ping Wang. 2019. Gate Decorator: Global Filter Pruning Method for Accelerating Deep Convolutional Neural Networks. (2019). arXiv:cs.CV/1909.08174

D. Yu, F. Seide, G. Li, and L. Deng. 2012. Exploiting sparseness in deep neural networks for large vocabulary speech recognition. In *2012 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. 4409–4412. https://doi.org/10.1109/ICASSP.2012.6288897

Jiecao Yu, Andrew Lukefahr, David Palframan, Ganesh Dasika, Reetuparna Das, and Scott Mahlke. 2017. Scalpel: Customizing dnn pruning to the underlying hardware parallelism. *ACM SIGARCH Computer Architecture News* 45, 2 (2017), 548–560.

Ruichi Yu, Ang Li, Chun-Fu Chen, Jui-Hsin Lai, Vlad I. Morariu, Xintong Han, Mingfei Gao, Ching-Yung Lin, and Larry S. Davis. 2018. NISP: Pruning Networks using Neuron Importance Score Propagation. (2018). arXiv:cs.CV/1711.05908

Xin Yu, Zhiding Yu, and Srikumar Ramalingam. 2018. Learning strict identity mappings in deep residual networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 4432–4440. arXiv:cs.CV/1804.01661

Ming Yuan and Yi Lin. 2006. Model selection and estimation in regression with grouped variables. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)* 68, 1 (2006), 49–67.

Chulhee Yun, Yin-Wen Chang, Srinadh Bhojanapalli, Ankit Singh Rawat, Sashank J. Reddi, and Sanjiv Kumar. 2020. $O(n)$ Connections are Expressive Enough: Universal Approximability of Sparse Transformers. In *Advances in Neural Information Processing Systems*.

Manzil Zaheer, Guru Guruganesh, Avinava Dubey, Joshua Ainslie, Chris Alberti, Santiago Ontanon, Philip Pham, Anirudh Ravula, Qifan Wang, Li Yang, et al. 2020. Big bird: Transformers for longer sequences. In *Advances in Neural Information Processing Systems*. arXiv:cs.LG/2007.14062

Wenyuan Zeng and Raquel Urtasun. 2019. MLPrune: Multi-Layer Pruning for Automated Neural Network Compression. (2019). https://openreview.net/forum?id=r1g5b2RcKm

Xiaoqin Zeng and Daniel S Yeung. 2006. Hidden neuron pruning of multilayer perceptrons using a quantified sensitivity measure. *Neurocomputing* 69, 7-9 (2006), 825–837.

Chiyuan Zhang, Samy Bengio, Moritz Hardt, Benjamin Recht, and Oriol Vinyals. 2017. Understanding deep learning requires rethinking generalization. (2017). arXiv:cs.LG/1611.03530

Jiaqi Zhang, Xiangru Chen, Mingcong Song, and Tao Li. 2019. Eager Pruning: Algorithm and Architecture Support for Fast Training of Deep Neural Networks. In *Proceedings of the 46th International Symposium on Computer Architecture (ISCA '19)*. Association for Computing Machinery, New York, NY, USA, 292–303. https://doi.org/10.1145/3307650.3322263

Jie-Fang Zhang, Ching-En Lee, C. Liu, Y. Shao, Stephen W. Keckler, and Zhengya Zhang. 2019a. SNAP: A 1.67 21.55TOPS/W Sparse Neural Acceleration Processor for Unstructured Sparse Deep Neural Network Inference in 16nm CMOS. *2019 Symposium on VLSI Circuits* (2019), C306–C307.

Jeff (Jun) Zhang, Parul Raj, Shuayb Zarar, Amol Ambardekar, and Siddharth Garg. 2019b. CompAct: On-Chip ComPression of ActIvations for Low Power Systolic Array Based CNN Acceleration. *ACM Trans. Embed. Comput. Syst.* 18, 5s, Article 47 (Oct. 2019), 24 pages. https://doi.org/10.1145/3358178

S. Zhang, Z. Du, L. Zhang, H. Lan, S. Liu, L. Li, Q. Guo, T. Chen, and Y. Chen. 2016. Cambricon-X: An accelerator for sparse neural networks. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 1–12. https://doi.org/10.1109/MICRO.2016.7783723

Zhekai Zhang, Hanrui Wang, Song Han, and William J. Dally. 2020. SpArch: Efficient Architecture for Sparse Matrix Multiplication. (2020). arXiv:cs.AR/2002.08947

Guangxiang Zhao, Junyang Lin, Zhiyuan Zhang, Xuancheng Ren, Qi Su, and Xu Sun. 2019. Explicit Sparse Transformer: Concentrated Attention Through Explicit Selection. (2019). arXiv:cs.CL/1912.11637

Qibin Zhao, Masashi Sugiyama, and Andrzej Cichocki. 2017. Learning Efficient Tensor Representations with Ring Structure Networks. (2017). arXiv:cs.NA/1705.08286

Guian Zhou and Jennie Si. 1999. Subset-based training and pruning of sigmoid neural networks. *Neural networks* 12, 1 (1999), 79–89.

Hao Zhou, Jose M Alvarez, and Fatih Porikli. 2016. Less is more: Towards compact cnns. In *European Conference on Computer Vision*. Springer, 662–677.

Hattie Zhou, Janice Lan, Rosanne Liu, and Jason Yosinski. 2020. Deconstructing Lottery Tickets: Zeros, Signs, and the Supermask. (2020). arXiv:cs.LG/1905.01067

X. Zhou, Z. Du, Q. Guo, S. Liu, C. Liu, C. Wang, X. Zhou, L. Li, T. Chen, and Y. Chen. 2018. Cambricon-S: Addressing Irregularity in Sparse Neural Networks through A Cooperative Software/Hardware Approach. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 15–28. https://doi.org/10.1109/MICRO.2018.00011

Jingyang Zhu, Jingbo Jiang, Xizi Chen, and Chi-Ying Tsui. 2017. SparseNN: An Energy-Efficient Neural Network Accelerator Exploiting Input and Output Sparsity. (2017). arXiv:cs.LG/1711.01263

Jingyang Zhu, Zhiliang Qian, and Chi-Ying Tsui. 2016. LRADNN: High-throughput and energy-efficient Deep Neural Network accelerator using Low Rank Approximation. In *2016 21st Asia and South Pacific Design Automation Conference (ASP-DAC)*. 581–586. https://doi.org/10.1109/ASPDAC.2016.7428074

Michael Zhu and Suyog Gupta. 2017. To prune, or not to prune: exploring the efficacy of pruning for model compression. (2017). arXiv:stat.ML/1710.01878

Tao Zhuang, Zhixuan Zhang, Yuheng Huang, Xiaoyi Zeng, Kai Shuang, and Xiang Li. 2020. Neuron-level Structured Pruning using Polarization Regularizer. *Advances in Neural Information Processing Systems* 33 (2020).

Zhuangwei Zhuang, Mingkui Tan, Bohan Zhuang, Jing Liu, Yong Guo, Qingyao Wu, Junzhou Huang, and Jinhui Zhu. 2019. Discrimination-aware Channel Pruning for Deep Neural Networks. (2019). arXiv:cs.CV/1810.11809