

“Learning-Compression” Algorithms for Neural Net Pruning

Miguel Á. Carreira-Perpiñán
 EECS, University of California, Merced
<http://eecs.ucmerced.edu>

Yerlan Idelbayev
 EECS, University of California, Merced
<http://eecs.ucmerced.edu>

Abstract

Pruning a neural net consists of removing weights without degrading its performance. This is an old problem of renewed interest because of the need to compress ever larger nets so they can run in mobile devices. Pruning has been traditionally done by ranking or penalizing weights according to some criterion (such as magnitude), removing low-ranked weights and retraining the remaining ones. We formulate pruning as an optimization problem of finding the weights that minimize the loss while satisfying a pruning cost condition. We give a generic algorithm to solve this which alternates “learning” steps that optimize a regularized, data-dependent loss and “compression” steps that mark weights for pruning in a data-independent way. Magnitude thresholding arises naturally in the compression step, but unlike existing magnitude pruning approaches, our algorithm explores subsets of weights rather than committing irrevocably to a specific subset from the beginning. It is also able to learn automatically the best number of weights to prune in each layer of the net without incurring an exponentially costly model selection. Using a single pruning-level user parameter, we achieve state-of-the-art pruning in LeNet and ResNets of various sizes.

Pruning neural nets is an old problem that has been revived in recent years. It consists of removing weights and/or neurons with the goal of reducing the size of the net without hurting its accuracy, learning automatically the right number of neurons and weights, or avoiding overfitting. Work in the 90s produced various algorithms that generally operate by using some criterion or penalty to detect unimportant weights or neurons, removing them and retraining the remaining ones, possibly on the fly while training the net. The 2010s have shown that neural nets achieve state-of-the-art performance in various applications if trained on large data sets using GPUs and using a large net, having many layers, neurons and weights. The large size of these nets (upwards of millions of weights) make them difficult to deploy in limited-computation devices such as mobile phones. This has brought a renewed interest in pruning and gener-

ally in neural net compression, so that one can obtain small yet accurate nets. Pruning and compression are possible because these large nets are hugely overparameterized, and empirical evidence suggests it is easier to train a large net and compress it than to train a smaller net from start [28].

Although pruning can be seen as a way to find the right architecture for a net or to improve generalization, here we focus on pruning as a way to compress a well-trained, reference net with little accuracy loss (the reference net is also helpful in telling us what is the best that we can achieve, up to local optima). Much pruning work uses a heuristic modification of the usual neural net training so that one removes weights on the fly via some criterion. While this can succeed in practice, it is not clear whether the resulting pruned net is optimal and in what sense. We take a top-down optimization view: we define the problem mathematically as an optimization over the net weights that incorporates our conflicting desires of minimizing the loss (e.g. classification error) and minimizing the number of weights. Specifically, we are inspired by recent work [2, 3] that formulates neural net compression in a general way via constrained optimization and shows how this leads to a powerful weight quantization algorithm. Here, we develop and extend this approach for the problem of pruning a deep net. In addition, we seek algorithms that are able to identify *exactly* which weights should be zero. An example that does not satisfy this are interior-point methods (one of the best approaches for large-scale problems), since their iterates are nonzero throughout training and only converge to exact zeros in the limit. The reason to identify the zeros exactly is that, with many weights, optimizing to high accuracy is impractical, and this introduces uncertainty about which nonzero weights should really be zero based on their value.

We consider pruning as a form of compression, where unpruned weights $\mathbf{w} \in \mathbb{R}^n$ are compressed into sparse weights $\boldsymbol{\theta} \in \mathbb{R}^n$ satisfying a condition dependent on a pruning cost function $C(\boldsymbol{\theta})$ which promotes sparsity in the weight vector $\boldsymbol{\theta}$, such as ℓ_0 or ℓ_1 , and is mathematically expressed as either a constraint or a penalty. The result is a “learning-compression” (LC) algorithm that alternates a learning step that optimizes the data-dependent loss over

the real-valued weights \mathbf{w} with a compression (pruning) step that compresses \mathbf{w} into $\boldsymbol{\theta}$, independently of the loss and data. Interestingly, for certain pruning costs this compression step naturally has the form of magnitude pruning, which gives support to using magnitude as a measure of weight saliency (as opposed to, say, curvature). However, our algorithm does not prune permanently: weights move in and out of the set of pruned weights during training until we converge on a final set. We first describe our general approach and develop it for its constraint and penalty forms. Although we focus on ℓ_0 , ℓ_1 and ℓ_2^2 , we emphasize our framework applies to other costs. Then we describe how to learn the amount of pruning per layer automatically and discuss the algorithm's behavior. Our experiments with LeNet and ResNets show that our LC algorithm achieves larger amounts of pruning with no loss degradation, and show the peculiar structure of the pruned net that arises.

Related work Pruning was recognized as an important problem since the 1980s; see reviews [28] and [1, ch. 9.5]. Most methods can be classified into two types: *saliency ranking methods* use a criterion to estimate the importance of each weight in the net, remove less important weights and retrain the rest; and *penalty methods* minimize the loss $L(\mathbf{w})$ plus a penalty term $\alpha C(\mathbf{w})$ that penalizes nonzero weights, remove small weights upon convergence and retrain the rest. Many saliency criteria exist, such as magnitude $|w_i|$, curvature using the diagonal [23] or all the Hessian entries [17, 18], and sensitivity of the loss to removing w_i . While most saliency methods are simple and fast, their performance is limited: they are local (the saliency estimate for each w_i is valid at the reference net but not away from it), greedy (weights are pruned irrevocably, with no backtracking), and individual weight saliency is after all a heuristic estimate for the effect on the loss of the set of weights to be pruned. This can be partly improved by applying the pruning/retraining in stages (where only a few weights are pruned at a time, e.g. [33, 15]), but this is time-consuming in practice. Penalty methods were mostly based on weight decay and variations of it, penalizing $\alpha \sum_i w_i^2$ or variations such as $\alpha \sum_i w_i^2 / (A + w_i^2)$ [16, 31] that encourage weights to be either large or small. Weight decay helps avoid overfitting and prune weights, but is not sparsifying: upon convergence, none of the weights are zero, and truncation is somewhat arbitrary just as with saliency methods. Sparsifying penalties such as ℓ_0 or ℓ_1 seem not to have been investigated, presumably because backpropagation cannot handle their nonsmoothness. Group LASSO penalties (to prune entire filters of a net) have been recently considered in [25, 32]. Their SGD optimization adds a heuristic thresholding step to zero values below 10^{-4} . However, online methods such as SGD have trouble deciding whether a given weight should be pruned or not based on a minibatch [33, section 3.1]. Finally, pruning can be com-

bined with other compression techniques, such as weight quantization [12, 15, 3], low-rank decomposition of weight matrices [29, 8, 21, 9, 27], hashing [5], lossless compression such as Huffman codes [14], etc. Here we focus on pruning alone. *Interestingly, although saliency and penalty methods appear very different, we will show they are related in our LC algorithm: an iterative form of magnitude-based pruning arises in a principled way from the use of sparsifying penalties on the loss.*

1. Neural network pruning as an optimization problem¹

We define a *pruning cost* as a function $C: \mathbb{R}^n \rightarrow \mathbb{R}^+$ satisfying $C(\mathbf{0}) = 0$ and $C(\mathbf{w}) > 0$ if $\mathbf{w} \neq \mathbf{0}$. We say that C is separable if $C(\mathbf{w}) = \sum_{i=1}^n c(w_i)$ where $c: \mathbb{R} \rightarrow \mathbb{R}^+$ is a scalar pruning cost. C should be designed such that it penalizes nonzeros in \mathbf{w} . The *pruning cost function* C and the *pruning operators* Π_C^+ , Π_C^- defined later are central concepts in our framework. We study three important examples of $C(\mathbf{w})$: $\|\mathbf{w}\|_0$ (the number of nonzero elements of \mathbf{w}), $\|\mathbf{w}\|_1$, and $\|\mathbf{w}\|_2^2$, all of which are separable. While other costs could be studied that are of practical interest, these are representative of what can be achieved in our framework and illustrate the issues of sparsification and shrinkage. ℓ_0 is arguably the most natural definition of pruning, as it is equivalent to finding the best subset of pruned weights, but it is a hard combinatorial problem. ℓ_2^2 corresponds to regular weight decay and can be optimized directly by descent methods, but it is instructive in our discussion.

Consider then the following general formulation for learning an optimally pruned network, where $L(\mathbf{w})$ is a loss function of interest (such as the classification or regression error on a training set):

$$\text{Constraint form: } \min_{\mathbf{w}} L(\mathbf{w}) \quad \text{s.t.} \quad C(\mathbf{w}) \leq \kappa \quad (1a)$$

$$\text{Penalty form: } \min_{\mathbf{w}} L(\mathbf{w}) + \alpha C(\mathbf{w}). \quad (1b)$$

Both naturally aim at learning an optimal model, by minimizing the data-dependent loss $L(\mathbf{w})$, but subject to having many zero weights, as given by the *pruning parameters* $\kappa \geq 0$ and $\alpha \geq 0$. Although optimizing the above could be done in different ways, here we focus on a common mechanism that results in a very simple yet effective *learning-compression (LC) algorithm* [2] for both forms. This alternates a data-dependent step that updates the “uncompressed parameters” (here, all the weights in the net) with a data-independent step that compresses the parameters (here, *prunes* the weights). The idea is to decouple the pruning term on C from the learning term on L via an auxiliary variable $\boldsymbol{\theta}$, a quadratic-penalty function and an alter-

¹Notation. Norms $\|\cdot\|$ are Euclidean norms $\|\cdot\|_2$ by default. We use the indicator function $I(x) = 1$ if x is true and 0 otherwise, and the sign function $\text{sgn}(x) = -1$ if $x < 0$, 0 if $x = 0$ and $+1$ if $x > 0$.

nating optimization over \mathbf{w} and $\boldsymbol{\theta}$. We describe the mathematical development for the constraint form first and for the penalty form next.

Before proceeding, note that the penalty and the constraint forms define problems that are equivalent for appropriate choices of κ and α . However, algorithmically they differ, and one form may be preferable over the other depending on the case; see our discussions later of factors such as computational cost, global vs local sparsity, or user friendliness of hyperparameter setting. This is particularly true with nonconvex problems, having local optima, and nonsmooth or combinatorial functions such as ℓ_0 .

1.1. Constraint form for the pruning cost

Let us introduce an auxiliary variable $\boldsymbol{\theta}$ in eq. (1) that duplicates \mathbf{w} :

$$\min_{\mathbf{w}, \boldsymbol{\theta}} L(\mathbf{w}) \quad \text{s.t.} \quad C(\boldsymbol{\theta}) \leq \kappa, \mathbf{w} = \boldsymbol{\theta}. \quad (2)$$

This problem is in the “model compression as constrained optimization” form $\min_{\mathbf{w}, \boldsymbol{\theta}} L(\mathbf{w})$ s.t. $\mathbf{w} = \boldsymbol{\Delta}(\boldsymbol{\theta})$ of [2], where the “decompression mapping” $\mathbf{w} = \boldsymbol{\Delta}(\boldsymbol{\theta})$, which recovers the uncompressed model parameters from their compressed version, takes a very simple form: $\mathbf{w} = \boldsymbol{\theta}$ but satisfying $C(\boldsymbol{\theta}) \leq \kappa$, i.e., having few nonzeros. We now optimize this constrained problem via either the quadratic-penalty (QP) or augmented-Lagrangian (AL) method (applied only to the equality constraint, not to the inequality):

$$\begin{aligned} Q(\mathbf{w}, \boldsymbol{\theta}; \mu) &= L(\mathbf{w}) + \frac{\mu}{2} \|\mathbf{w} - \boldsymbol{\theta}\|^2 \quad \text{s.t.} \quad C(\boldsymbol{\theta}) \leq \kappa \\ \mathcal{L}_A(\mathbf{w}, \boldsymbol{\theta}, \boldsymbol{\lambda}; \mu) &= L(\mathbf{w}) + \frac{\mu}{2} \|\mathbf{w} - \boldsymbol{\theta}\|^2 \\ &\quad - \boldsymbol{\lambda}^T (\mathbf{w} - \boldsymbol{\theta}) \quad \text{s.t.} \quad C(\boldsymbol{\theta}) \leq \kappa. \end{aligned}$$

For the QP, we optimize Q over $(\mathbf{w}, \boldsymbol{\theta})$ while driving $\mu \rightarrow \infty$, so the equality constraints are satisfied in the limit. For the AL, we alternate optimizing \mathcal{L}_A over $(\mathbf{w}, \boldsymbol{\theta})$ with updating $\boldsymbol{\lambda} \leftarrow \boldsymbol{\lambda} - \mu(\mathbf{w} - \boldsymbol{\theta})$ while driving $\mu \rightarrow \infty$. The Lagrange multiplier estimates $\boldsymbol{\lambda}$ make the iterates $(\mathbf{w}, \boldsymbol{\theta})$ be closer to the solution for the same value of μ , so the AL is preferable.

Finally, in order to optimize the QP or AL functions over the variables $(\mathbf{w}, \boldsymbol{\theta})$, we apply alternating optimization. This results in the following steps for the QP:

Learning (L) step (over \mathbf{w}) $\min_{\mathbf{w}} L(\mathbf{w}) + \frac{\mu}{2} \|\mathbf{w} - \boldsymbol{\theta}\|^2$.

This has the form of a usual neural net learning but with a quadratic regularizer that pulls some weights to zero (since $\boldsymbol{\theta}$ will usually contain some exactly zero elements) and the rest to some other nonzero value.

Compression (C) step (over $\boldsymbol{\theta}$)

$\Pi_C^{\leq}(\mathbf{w}; \kappa) = \arg \min_{\boldsymbol{\theta}} \|\mathbf{w} - \boldsymbol{\theta}\|^2 \quad \text{s.t.} \quad C(\boldsymbol{\theta}) \leq \kappa$ (where the “ \leq ” superindex refers to the constraint form). This has the form of a proximal operator, which we call *pruning operator*. It can be solved exactly for several useful costs C , including ℓ_0 , ℓ_1 and ℓ_2^2 . In our context, “compression” means “weight pruning”.

We describe the C step in more detail later for the QP. For the AL, replace $\boldsymbol{\theta}$ by $\boldsymbol{\theta} + \frac{1}{\mu} \boldsymbol{\lambda}$ in the L step, and \mathbf{w} by $\mathbf{w} - \frac{1}{\mu} \boldsymbol{\lambda}$ in the C step. The suppl. mat. gives the algorithm pseudocode for the AL.

Note that the C step does not actually prune weights, it simply “marks” weights to be pruned (by setting their $\theta_i = 0$); the w_i values stay as nonzero. The L step is the one that actually updates the real-valued weights w_i taking into account both the loss and the markup. As the LC algorithm alternates both steps, it explores different sets of marked weights, eventually converging to a specific set for which $w_i \xrightarrow[\mu \rightarrow \infty]{} 0$ and thus is actually pruned.

C step The proximal operator $\boldsymbol{\theta} = \Pi_C^{\leq}(\mathbf{w}; \kappa) = \arg \min_{\boldsymbol{\theta}} \|\mathbf{w} - \boldsymbol{\theta}\|^2$ s.t. $C(\boldsymbol{\theta}) \leq \kappa$ maps a real-valued weight vector \mathbf{w} to another real-valued vector $\boldsymbol{\theta}$ of the same dimension containing a certain number of zero elements, so $\boldsymbol{\theta}$ is a pruned version of \mathbf{w} (and, as we will see, it possibly shrinks its nonzero values). It has the form of a *projection*, or nearest point $\boldsymbol{\theta}$ to \mathbf{w} (in Euclidean distance) that lies in the feasible set $C(\boldsymbol{\theta}) \leq \kappa$. The projection operator leaves all weights unchanged if $C(\mathbf{w}) \leq \kappa$ (i.e., $\Pi_C^{\leq}(\mathbf{w}; \kappa) = \mathbf{w}$). Otherwise, the resulting $\boldsymbol{\theta}$ has $C(\boldsymbol{\theta}) \leq \kappa < C(\mathbf{w})$, which implies that $\boldsymbol{\theta}$ is “smaller” than \mathbf{w} , and indeed many weights will individually satisfy $|\theta_i| < |w_i|$, but some may stay or increase in magnitude.

The solution for several costs C corresponding to projection on ℓ_p balls is well known and is given in fig. 1 (see proofs in the suppl. mat.). When \mathbf{w} is in the ball, $\boldsymbol{\theta} = \mathbf{w}$. Otherwise, ℓ_0 leaves the top- κ weights unchanged and prunes the rest; ℓ_1 shrinks on average the top- k weights (where k depends on \mathbf{w} and κ) and prunes the rest; and ℓ_2^2 shrinks all weights (normalizes \mathbf{w}).

Computationally, a simple algorithm for ℓ_0 and ℓ_1 involves sorting the elements of \mathbf{w} in magnitude (at a runtime $\mathcal{O}(n \log n)$ if \mathbf{w} has n elements), and scanning this in $\mathcal{O}(n)$ to find the threshold η and return the nonzeros. But both ℓ_0 and ℓ_1 can be solved in $\mathcal{O}(n)$ worst case runtime by using selection to find the k th value in $\mathcal{O}(n)$ (this can be achieved with a partial quicksort; [7, ch. 9]). For ℓ_0 , this is obvious. For ℓ_1 , see [6]. Since the number of weights in a deep net, which is our driving application, is large (upwards of millions in practice), using selection instead of sorting matters. That said, the L step dominates the C step by far.

1.2. Penalty form for the pruning cost

Although the penalty form does not have the *model compression as constrained optimization* form of [2], we can apply the same technique to arrive at a convenient LC algorithm. First we duplicate \mathbf{w} via an auxiliary variable $\boldsymbol{\theta}$:

$$\min_{\mathbf{w}, \boldsymbol{\theta}} L(\mathbf{w}) + \alpha C(\boldsymbol{\theta}) \quad \text{s.t.} \quad \mathbf{w} = \boldsymbol{\theta}. \quad (3)$$

Then we optimize this via QP or AL:

$$\begin{aligned} Q(\mathbf{w}, \boldsymbol{\theta}; \mu) &= L(\mathbf{w}) + \frac{\mu}{2} \|\mathbf{w} - \boldsymbol{\theta}\|^2 + \alpha C(\boldsymbol{\theta}) \\ \mathcal{L}_A(\mathbf{w}, \boldsymbol{\theta}, \boldsymbol{\lambda}; \mu) &= L(\mathbf{w}) + \frac{\mu}{2} \|\mathbf{w} - \boldsymbol{\theta}\|^2 \\ &\quad - \boldsymbol{\lambda}^T (\mathbf{w} - \boldsymbol{\theta}) + \alpha C(\boldsymbol{\theta}). \end{aligned}$$

Finally, we apply alternating optimization over $(\mathbf{w}, \boldsymbol{\theta})$. This results in an **L step (over \mathbf{w})** identical to that of the constraint form, and a **C step (over $\boldsymbol{\theta}$)** $\Pi_C^+(\mathbf{w}; \frac{2\alpha}{\mu}) = \arg \min_{\boldsymbol{\theta}} \|\mathbf{w} - \boldsymbol{\theta}\|^2 + \frac{2\alpha}{\mu} C(\boldsymbol{\theta})$ (where the “+” superindex refers to the penalty form). This is again a proximal operator that can often be solved exactly, as shown next.

C step If the pruning cost separates, $C(\boldsymbol{\theta}) = \sum_{i=1}^n c(\theta_i)$, so does the C step objective, which can then be solved elementwise (separately for each weight in the net) and takes the form $\Pi_C^+(w; \frac{2\alpha}{\mu}) = \arg \min_{\theta} (w - \theta)^2 + \frac{2\alpha}{\mu} c(\theta)$ with scalars $w, \theta \in \mathbb{R}$. Firstly, we show that $\Pi_C^+(w; \frac{2\alpha}{\mu})$ makes w smaller or equal with the same sign.

Theorem 1.1. Call $F(\theta; w) = (w - \theta)^2 + \frac{2\alpha}{\mu} c(\theta)$ where $c(0) = 0$, $c(\theta) > 0$ if $\theta \neq 0$, $c(\theta) = c(-\theta)$ and $c(\theta)$ is nondecreasing for $\theta \geq 0$. Assume $\theta^* = \arg \min_{\theta} F(\theta; w)$ is the unique global minimizer. Then $\text{sgn}(\theta^*) = \text{sgn}(w)$ and $|\theta^*| \leq |w|$.

Proof. First, $F(-\theta; -w) = (-w + \theta)^2 + \frac{2\alpha}{\mu} c(-\theta) = (w - \theta)^2 + \frac{2\alpha}{\mu} c(\theta) = F(\theta; w)$, so F is invariant to negating θ and w . Second, let $w \in \mathbb{R}$. Since $(w - \theta)^2$ is smaller when θ has the same sign as w than when it has the opposite sign (for the same magnitude of θ), and $c(\theta) = c(-\theta)$, then F is smaller also. Hence, θ^* has the same sign as w . Finally, we prove by contradiction that $\theta^* \leq w$ for the case $w \geq 0$ w.l.o.g. Suppose $\theta^* > w$, then $F(\theta^*; w) = (w - \theta^*)^2 + \frac{2\alpha}{\mu} c(\theta^*) \geq (w - \theta^*)^2 + \frac{2\alpha}{\mu} c(w) > \frac{2\alpha}{\mu} c(w) = F(w; w)$, which contradicts $F(\theta^*; w) \leq F(\theta; w) \forall \theta \geq 0$. \square

This means that the pruning operator drives w to zero, as one would expect, but how this happens depends on the pruning cost C . Fig. 1 gives explicitly the pruning operator for several costs (see proofs in the suppl. mat.). We observe two types of behavior: *sparsification*, in which weights within some interval become exactly zero; and *shrinkage*, in which weights that do not become zero become smaller anyway. ℓ_0 sparsifies but does not shrink: w is either pruned ($\theta = 0$) or left as is ($\theta = w$). ℓ_1 sparsifies and shrinks: w is either pruned ($\theta = 0$) or shifted towards zero ($\theta = w - \text{sgn}(w) \frac{\alpha}{\mu}$). ℓ_2^2 does not sparsify but shrinks: w is divided by a number bigger than 1.

1.3. Global vs local sparsity

In a neural net, a fully-connected layer has many more weights than a convolutional layer and can be pruned more aggressively. Hence, allowing a different sparsity level for

each layer (say, 5% unpruned weights for the convolutional layer and 1% for the fully-connected one) will result in networks with lower loss for the same total number of weights. In the penalty form, this unfortunately requires an exponentially costly selection for the per-layer penalties (“local” sparsity). However, in the constraint form we can prove that using a single κ parameter for the whole net (“global” sparsity) can find the optimal per-layer sparsities. This is remarkable because we achieve the best of both worlds: ease of use (only one pruning parameter to select) and best results. *The LC algorithm automatically determines the best number of weights for each layer.*

Theorem 1.2. Let C be a separable pruning cost; $\kappa_1, \dots, \kappa_K, \kappa \in \mathbb{R}^+$ with $\kappa_1 + \dots + \kappa_K \leq \kappa$; and $\mathcal{S}_l = \{\mathbf{w} = (\mathbf{w}_1, \dots, \mathbf{w}_K) \in \mathbb{R}^n: C(\mathbf{w}_1) \leq \kappa_1, \dots, C(\mathbf{w}_K) \leq \kappa_K\}$ and $\mathcal{S}_g = \{\mathbf{w} \in \mathbb{R}^n: C(\mathbf{w}) \leq \kappa\}$. Then $\mathcal{S}_l \subset \mathcal{S}_g$.

Proof. Let $\mathbf{w} \in \mathcal{S}_l$. Then $C(\mathbf{w}_1) \leq \kappa_1, \dots, C(\mathbf{w}_K) \leq \kappa_K$, so $C(\mathbf{w}) = C(\mathbf{w}_1) + \dots + C(\mathbf{w}_K) \leq \kappa_1 + \dots + \kappa_K \leq \kappa$ and $\mathbf{w} \in \mathcal{S}_g$. \square

Our optimization applies equally easily to both global and local sparsity. For example, for the ℓ_0 case with global sparsity, the top- κ weights throughout the entire net stay and the rest are pruned; how many weights are pruned in each layer in the C step arises automatically and optimally.

1.4. Behavior, convergence and practicalities of the LC algorithm

To follow the path over $\mu \geq 0$ numerically, we use a multiplicative schedule $\mu_k = \mu_0 a^k$, $k = 0, 1, 2, \dots$, where μ_0 is given in the suppl. mat. and $a > 1$ is determined by trial and error (using a smaller a follows the path more slowly and generally gives a better solution, but is computationally slower). We stop when $\|\mathbf{w} - \boldsymbol{\theta}\|$ is smaller than a set tolerance and retrain the unpruned weights. This is unnecessary in theory for the ℓ_0 cost, but in practice with deep nets (which are notoriously hard to optimize accurately) retraining it will improve a bit the result. For the ℓ_1 cost retraining is necessary and will significantly improve the result (and increase on average the weights’ magnitude).

For large enough μ the LC algorithm will identify the final set of weights that are pruned, i.e., which elements in $\boldsymbol{\theta}$ are zero. We can analyze what happens at the beginning of the path (see suppl. mat.), which provides an interesting perspective on pruning/retraining algorithms (e.g. [15]). Essentially, for the constraint form we start at a point $(\mathbf{w}(0), \boldsymbol{\theta}(0)) = (\overline{\mathbf{w}}, \boldsymbol{\theta}_{\text{DC}})$ where $\overline{\mathbf{w}}$ is a well-trained, reference model, and $\boldsymbol{\theta}_{\text{DC}} = \Pi_C^{\leq}(\overline{\mathbf{w}}; \kappa)$. This was called *direct compression* (here, direct pruning) in [2], as it corresponds to *pruning the reference weights independently of the loss*. The weights $\boldsymbol{\theta}_{\text{DC}}$ result from magnitude pruning of the reference weights $\overline{\mathbf{w}}$ and they produce a large loss,

$C(\mathbf{w})$	Constraint form [†] $\theta = \Pi_C^{\leq}(\mathbf{w}; \kappa)$	Penalty form $\theta = \Pi_C^+(\mathbf{w}; \frac{2\alpha}{\mu})$
$\ \mathbf{w}\ _0$	$w_i \cdot I(w_i > \eta_0)$	$w_i \cdot I(w_i > \sqrt{\frac{2\alpha}{\mu}})$
$\ \mathbf{w}\ _1$	$(w_i - \text{sgn}(w_i) \eta_1) \cdot I(w_i > \eta_1)$	$(w_i - \text{sgn}(w_i) \frac{\alpha}{\mu}) \cdot I(w_i > \frac{\alpha}{\mu})$
$\ \mathbf{w}\ _2^2$	$\sqrt{\kappa} w_i / \ \mathbf{w}\ _2$	$w_i / (1 + \frac{2\alpha}{\mu})$

[†]Each formula applies if $C(\mathbf{w}) > \kappa$, otherwise $\theta = \mathbf{w}$.

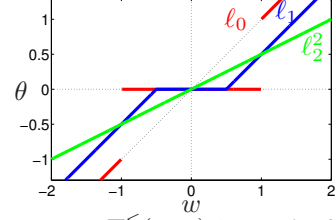


Figure 1. C step solution: selected pruning cost functions $C(\mathbf{w})$ and their corresponding pruning operators $\Pi_C^{\leq}(\mathbf{w}; \kappa)$ (constraint form) and $\Pi_C^+(\mathbf{w}; \frac{2\alpha}{\mu})$ (penalty form). All cases result in an elementwise operator that computes θ_i from w_i for each weight. The threshold η_0 equals the magnitude of the $(\kappa + 1)$ th largest weight. The threshold η_1 can be obtained by scanning w_i in decreasing order of magnitude (see main text). The graph plots the elementwise pruning operator for the penalty form.

which can be reduced by retraining the nonzero weights in θ_{DC} . This *pruning/retraining approach* is perhaps the most widespread pruning method for deep nets; we discuss it further in section 3.

Theorem 2.1 in [2] applies to our penalty form (3) without modification and should be easy to extend to our constraint form (2) (which contains the extra constraint $C(\theta) \leq \kappa$). Essentially, it states that if we follow the path closely enough (by minimizing Q or \mathcal{L}_A for each μ via sufficiently many L and C steps), then we reach a local solution in the limit $\mu \rightarrow \infty$. However, that theorem assumes smooth (though not necessarily convex) $L(\mathbf{w})$ and $C(\theta)$. This holds for ℓ_2^2 but not for the more interesting, sparsifying costs ℓ_0 and ℓ_1 . Guarantees for ℓ_0 are likely hard to come by because it defines an NP-complete problem. Guarantees for ℓ_1 may be easier to state.

1.5. Relation with compressed sensing and Lasso

All four variations of our LC algorithm (ℓ_0/ℓ_1 , constraint/penalty) arise from a common methodology: to duplicate variables $\mathbf{w} = \theta$, apply a penalty method and optimize alternately over \mathbf{w} and θ . Although our focus is on deep neural nets, these algorithms also apply when the loss $L(\mathbf{w})$ is quadratic and the model is linear. There is a relation with some fundamental algorithms in compressed sensing and sparse learning, specifically Lasso regression. These minimize over \mathbf{w} objectives of the type $\|\mathbf{y} - \mathbf{A}\mathbf{w}\|^2$, where \mathbf{w} is sparse. The alternating direction method of multipliers (ADMM) for the Lasso [19] uses an ℓ_1 penalty and the resulting algorithm is the same as our ℓ_1 -penalty variation. Iterative hard thresholding (IHT) [11] seeks an (approximate) ℓ_0 -constraint solution, but its algorithm is somewhat different from our ℓ_0 -constraint variation, involving a gradient step and a hard thresholding step. Those algorithms have been found very effective in practice and enjoy certain theoretical guarantees of finding the global minimum of the ℓ_0 problem (under assumptions such as restricted isometry or incoherence on the matrix \mathbf{A} and vectors \mathbf{w} , \mathbf{y}). It is unclear whether any of those guarantees may carry over to our case of interest, neural net pruning, but it is encouraging that such guarantees may hold in the linear case.

2. Experiments

We evaluate our LC algorithm for pruning on classification neural nets of different sizes in the MNIST (LeNet) and CIFAR10 (ResNets) datasets and compare with magnitude-based pruning, that is, pruning all but the largest magnitude weights of the reference net and retraining them. We exceed or are comparable in both training and test error with any published results we know of, at any pruning level, even though we use a single user parameter κ or α and a single round of pruning. Code/data are available from the authors.

We report the augmented Lagrangian results and use always a single, global parameter (κ for the constraint form, α for the penalty form). We used the Theano [30] and Lasagne [10] libraries. We initialize all algorithms from a reasonably (but not necessarily perfectly) well-trained reference model. The initial LC iteration ($\mu = 0$) for the constraint form gives the magnitude-based pruning solution. We only prune the multiplicative weights in the net, not the biases. We report the loss and classification error in training and test, and the proportion (%) of pruned weights (total and per layer).

The optimization parameters are as follows throughout our experiments with minor exceptions (see suppl. mat. and [4]). We use Nesterov’s accelerated gradient method [26] with momentum 0.95 for around 100k minibatches, with a learning rate of the form $\eta \cdot 0.99^j$ (where η is between 0.02 and 0.1), running 2k iterations for every j (each a minibatch of 512 points). Our LC algorithm uses $\mu_j = \mu_0 a^j$ with $\mu_0 = 9.76 \cdot 10^{-5}$ and $a = 1.1$, for $0 \leq j \leq 30$. The j th L step runs 2k SGD iterations. We retrain the surviving weights with SGD for our LC algorithm and for magnitude-based pruning. The total runtime of our LC algorithm is roughly given by the number of L steps; we found it is to be no more than 1.5 times the runtime of the reference net.

Classification on MNIST with LeNet300 and LeNet5

The LeNet models [22] are a widely used benchmark that allows for comparison with published work. We randomly split the MNIST training dataset (60k grayscale images of 28×28 , 10 digit classes) into training (90%) and validation (10%) sets. We normalize the pixel grayscale to [0,1]


and then subtract the mean. The loss is the average cross-entropy. LeNet300 is a 3-layer fully-connected feedforward net 784–300–100–10 with tanh activations and softmax outputs, total 266 610 learnable parameters. LeNet5 is a convolutional net with ReLU activations and softmax outputs, total 431k trainable parameters. We report results mostly for LeNet300; see full details in the suppl. mat.

Table 1 gives a subset of pruning results for LeNet300. Generally, the constraint form does better than the penalty form, and the ℓ_0 cost does better than the ℓ_1 cost, but not significantly. Retraining the pruned net has a large effect for the ℓ_1 cost, as expected, because ℓ_1 shrinks the surviving weights: the loss decreases and the weights’ magnitude increases on average. Retraining has barely any effect for the ℓ_0 cost, which does not shrink the weights.

We did not try to find the very best parameter settings (for the pruning cost κ or α , or for the SGD and LC optimization parameters), instead we sample what can be achieved. (Note that for ℓ_0 -constraint the proportion $P\%$ of surviving weights is directly given by κ , which is convenient for the user; but for ℓ_1 -constraint and the penalty forms, achieving a desired $P\%$ requires trial and error of κ or α , which is cumbersome.) We can prune ~ 98 – 99% of the weights with about the same loss/error as the reference. We can go beyond 99% with a minor degradation. This outperforms nearly all published work we have seen: magnitude pruning done in stages in [15] achieves 92% (a little better than the single-stage magnitude pruning we show) and [32] is much worse. Only [13] is comparable to us, however their results are not reproducible based on the information in the paper, which neglects to disclose even the per-layer pruning parameters they used. Besides, tuning by hand the pruning parameter for each layer or the stages of pruning makes the network designer effectively part of the algorithm, painstakingly so. *We reiterate we simply select a single pruning parameter, which for the ℓ_0 constraint form is trivial to set: κ equals the number of surviving weights.*

Now we analyze which weights and neurons get pruned and how this changes over LC iterations, as the final connectivity structure is very interesting. Fig. 2 shows the weight vectors θ over LC iterations for 5 selected neurons in the first layer of LeNet300, for pruning around 95% weights (the same neurons for each combination of ℓ_0/ℓ_1 and constraint/penalty). As the LC algorithm iterates, θ marks weights for pruning and \mathbf{w} approaches θ until $\mathbf{w} = \theta$ upon convergence. Each weight vector can be shown as a 28×28 color image (red: positive, blue: negative, white: zero, gray: neuron pruned). The initial weights appear random and cover the entire image area. For the constraint form, the first iteration prunes all weights except the largest ones. For the penalty form, the first iteration prunes all weights, but when $\mu \approx \mu_0$ the largest weights revive (see theoretical analysis in suppl. mat.). After that, different weights

move in and out of the marked subset. The evolution of weights and neurons can be seen dramatically in an animation (suppl. mat.), in particular how for ℓ_1 the “weight mass” of a pruned neuron is captured by weights in other neurons. Although the weights change during training, the final weights resemble the initially pruned ones to some extent. The ℓ_1 cost changes weights more than the ℓ_0 one, and results in more neurons being pruned. *The final weight vectors often segment the image into negative and positive regions reminiscent of center-surround receptive fields, but these regions are sparse rather than compact.* Presumably this is because neighboring pixels are correlated and it suffices to sample a few to capture a good feature.

Although our algorithm prunes weights, not neurons, we observe an aggressive neuron pruning in the first layer, much more than would be expected if weights were pruned uniformly at random. Even though there are 5% of $784 \approx 39$ surviving input weights per first-layer neuron, in fact up to 3/4 of the neurons are pruned (which hence have ≈ 120 weights); see fig. 3 (# alive weights). Likewise, about half of the input neurons (pixels) have all output weights pruned and so are pruned (mostly around the image boundaries, which are constant in MNIST; for ℓ_0 -constraint, it looks like this: ). Indeed, the original LeNet300 architecture 784–300–100–10 becomes 400–64–99–10 with similar or even better loss (for the ℓ_1 -constraint). Hence, our pruning algorithm might be useful to do feature selection and determine the optimal number of neurons in each layer automatically.

A neuron is pruned when all its input and output weights are pruned. We observe the input weights disappear first, followed by the output ones. ℓ_0 is slightly less effective in pruning neurons: upon convergence we often find a few neurons each having no input weights and only a few output weights. With ℓ_1 , no such neurons remain. This is visible in the green curves in fig. 3 (# alive weights), corresponding to the fan-in and fan-out of layer 1: for ℓ_1 they both go down (fan-in first, then fan-out) and join upon convergence; for ℓ_0 this happens partially (which is not a problem since such neurons can be safely removed in a postprocessing step).

Fig. 3 (right subplots) shows the weight distribution in layer 1. It starts as a zero-mean Gaussian (from the reference net). Then it becomes trimodal, with a peak at zero (pruned weights) and two skewed distributions for negative and positive weights. For ℓ_0 the gap between the last two is much wider than for ℓ_1 .

Classification on CIFAR10 with ResNets The ResNet models [20] are one of the best performing deep nets in recent literature, and they are also very lean, achieving state-of-the-art classification error with a much smaller number of weights than other nets such as AlexNet or VGG. This makes them harder to prune, and indeed we are aware of only one other work on pruning ResNets on CIFAR10 [24].

	$P\%$	κ or $\log \alpha$	Before retraining			After retraining			$\Delta \mathbf{w}\%$
R	100=[100 100 100]		$\log L$	E_{train}	E_{test}	$\log L$	E_{train}	E_{test}	
ℓ_0 -c	3=[1.6 11 76]	7986	-3.81	0	2.27	-3.86	0	2.13	0.22
	1=[0.5 2.7 65]	2662	-0.68	5.64	6.90	-1.66	0.59	3.17	-1.99
ℓ_1 -c	2.4=[1.9 4.5 33]	1000	-0.56	9.19	10.16	-2.53	0.003	2.49	38.2
	1.3=[0.9 4.2 20]	500	0.43	61.49	60.30	-1.43	1.04	3.27	123.00
ℓ_0 -p	2.9=[1.8 8.2 72]	-5.69	-3.89	0	2.26	-3.94	0	2.23	0.26
	0.6=[0.3 1.7 44]	-5.00	0.01	25.12	23.82	-1.12	2.42	3.77	-4.22
ℓ_1 -p	3.5=[3.0 5.6 38]	-4.60	-1.88	0	2.82	-3.07	0	2.21	17.52
	1.7=[1.4 3.5 6.1]	-4.00	0.26	52.19	52.04	-1.95	0.12	2.67	118.74
mag	10=[5.7 40 87]		-0.19	16.11	16.13	-3.67	0	2.26	23.21
	1=[0.1 5.5 74]		0.88	80.91	80.97	-0.95	3.40	4.58	93.00

Table 1. Representative pruning results for the LeNet300 reference net **R**, and before and after retraining the surviving weights: magnitude-based and our LC algorithm (ℓ_0 and ℓ_1 cost, κ -constraint and α -penalty forms). We report: proportion of surviving weights P (%) in total and per-layer; training loss $\log L$ and training and test classification error E_{train} and E_{test} (%); and the average relative increase in weight magnitude after retraining $\Delta \mathbf{w} = \|\mathbf{w}_{\text{after}}\|/\|\mathbf{w}_{\text{before}}\| - 1$ (as %).

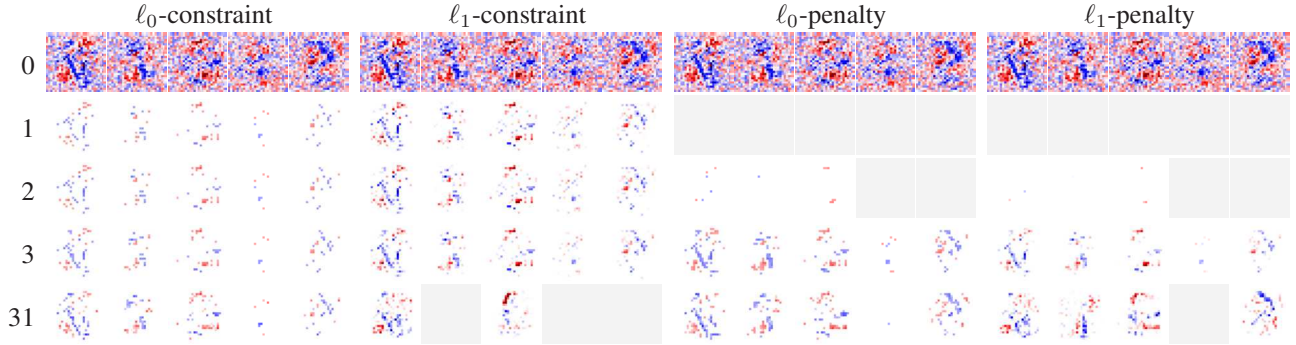


Figure 2. Weight vector of selected first-layer neurons over iterations (0 = reference), $P \approx 5\%$. Zoom in to see details.

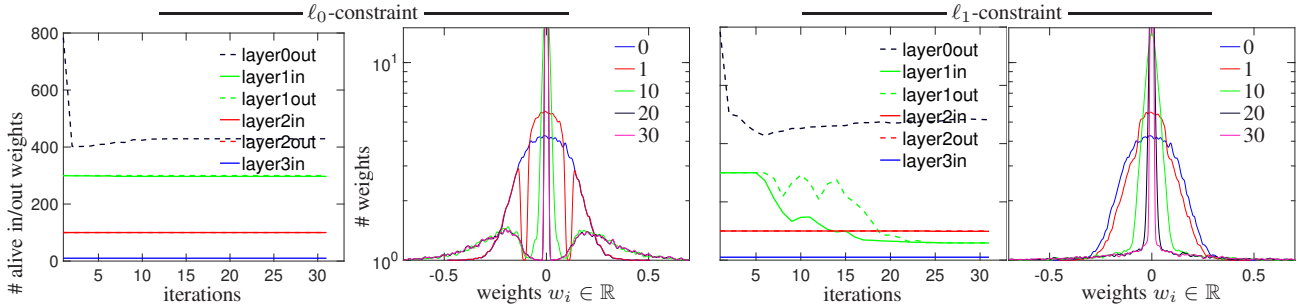


Figure 3. Connectivity statistics: left subplot, # neurons having nonzero weights (input or output, as indicated, for each layer); right subplot, distribution of weights for selected iterations for layer 1.

We train ResNets of depth 32, 56 and 110 layers (0.46M, 0.85M and 1.7M parameters, respectively) on the CIFAR10 dataset using the same setup as in [20]. We randomly split the dataset (50k RGB images of 32×32 , 10 object classes) into 90% training and 10% validation, and report results on the CIFAR10 test portion having 10k RGB images of the same sizes. For training, we subtract the pixel mean and use

simple augmentation (random horizontal flip, zero pad with 4 pixels on each side and randomly crop a 32×32 image). For test we use the original images without augmentation. We select the net having smallest validation error during training. We train reference nets, nets compressed with our LC algorithm (ℓ_0 -constraint version) and with magnitude-based pruning, followed by retraining the surviving weights

	$P\%$	κ (# weights)	E_{test} LC	E_{test} mag.
32-layers	R	0.46M	7.72	7.72
	15	69 281	7.32	8.85
	10	46 188	7.88	9.41
	5	23 094	9.26	12.07
	3	13 857	10.74	14.35
56-layers	R	0.85M	6.86	6.86
	15	127 342	6.92	7.39
	10	84 895	6.67	8.66
	5	42 448	7.51	9.48
	3	25 469	8.21	11.50
110-layers	R	1.7M	6.70	6.70
	15	257 979	6.30	7.22
	10	171 986	6.50	7.20
	5	85 993	6.93	8.23
	3	51 596	7.61	9.32

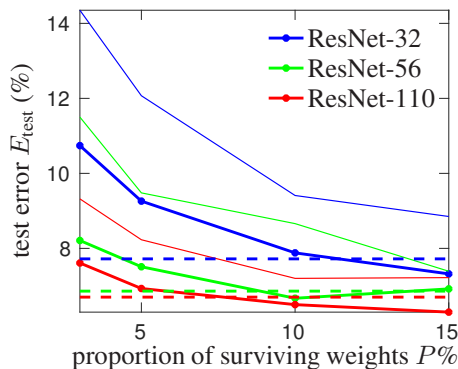


Figure 4. ResNet results for the reference net **R**, and for pruning after retraining the surviving weights: magnitude-based and our LC algorithm (ℓ_0 -constraint). We report: proportion of surviving weights P (%), κ hyperparameter for LC (equal to the number of surviving weights), and the test error for LC and magnitude-based pruning, in tabular form (above) and as tradeoff curves of error vs sparsity (below). Thick lines: LC, thin lines: magnitude-based pruning, horizontal dashed lines: reference nets.

(see details in the suppl. mat.).

Fig. 4 shows the results. We are able to achieve considerable pruning of $P \approx 5$ –10% surviving weights with about the same test error as the reference. The LC errors are always much lower than those of magnitude-based pruning. We found one published comparison point: [24] remove filters from convolutional layers for ResNet56/110 and achieve $P = 67.6\%$ for ResNet110 (error 6.7%) and $P = 86.3\%$ ResNet56 (error 6.94%). Our LC algorithm achieves a much stronger pruning $P = 10\%$ for ResNet110 (error 6.50%) and ResNet56 (error 6.67%).

3. Discussion

Most pruning approaches are based on the idea of permanently removing a subset of weights (“hard pruning”) based on some criterion that measures the importance of

each weight (such as magnitude or curvature), and then re-training the remaining weights. This approach is successful: it can prune many weights with no or little loss degradation. However, it is heuristic, lacking a theoretical understanding of how good these criteria are, and greedy: its success depends on choosing the right subset to prune among all possible subsets of weights, since there is no backtracking. The pruning/retraining process may be repeated several times, each time removing a small subset of the weights. By trial-and-error, one can make this improve over choosing a single large subset, but this effectively shifts the effort of searching over solutions to the user and is not practical (taking into account the long training times required for a deep net). An important consequence of our optimization-based approach is that *magnitude pruning arises naturally* both in the constraint and penalty forms with ℓ_0 and ℓ_1 . Hence, our LC algorithm gives theoretical support to the use of magnitude as criterion. But it differs from previous methods in that it uses it gradually, by exploring possible sets of pruned weights while optimizing the loss over all weights, without committing irrevocably to any set (“soft pruning”). All the weights are there throughout training, but some are marked as currently pruned (zeros in θ). Compared to hard pruning, this helps find a better subset and hence prune more weights with no or little loss degradation.

Finally, we recommend using our LC algorithm with the ℓ_0 or ℓ_1 constraint form and a global κ pruning parameter, because it learns automatically the optimal pruning level for each layer. Also, setting κ for ℓ_0 is very simple: it equals the desired number of surviving weights.

4. Conclusion

Our algorithm vindicates magnitude-based pruning but in a soft, iterative way. It explores many weight sets in search of a good one rather than committing greedily to the largest weights in the reference model. It is easy to implement: most of the runtime is spent training the reference model with a quadratic regularization term, with fast, periodic updates to the set of weights to be pruned and to the Lagrange multipliers. A crucial advantage is that it automatically determines the best number of weights to prune in each layer even though it uses a single user parameter. This avoids an exponentially costly search over per-layer pruning parameters and vastly simplifies the network designer’s job. Although we focused on model compression, our algorithm may be generally used during training to achieve good generalization and a small net.

Acknowledgments

Work supported by NSF award IIS-1423515, by a UC Merced Faculty Research Grant, by a Titan X Pascal GPU donated by the NVIDIA Corporation, and by computing time in the MERCED cluster (NSF grant ACI-1429783).

References

- [1] C. M. Bishop. *Neural Networks for Pattern Recognition*. Oxford University Press, New York, Oxford, 1995.
- [2] M. Á. Carreira-Perpiñán. Model compression as constrained optimization, with application to neural nets. Part I: General framework. arXiv:1707.01209 [cs.LG], July 5 2017.
- [3] M. Á. Carreira-Perpiñán and Y. Idelbayev. Model compression as constrained optimization, with application to neural nets. Part II: Quantization. arXiv:1707.04319 [cs.LG], July 13 2017.
- [4] M. Á. Carreira-Perpiñán and Y. Idelbayev. Model compression as constrained optimization, with application to neural nets. Part III: Pruning. arXiv, 2018.
- [5] W. Chen, J. Wilson, S. Tyree, K. Weinberger, and Y. Chen. Compressing neural networks with the hashing trick. In F. Bach and D. Blei, editors, *Proc. of the 32nd Int. Conf. Machine Learning (ICML 2015)*, pages 2285–2294, Lille, France, July 6–11 2015.
- [6] L. Condat. Fast projection onto the simplex and the ℓ_1 ball. *Math. Prog.*, 158(1):575–585, July 2016.
- [7] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, Cambridge, MA, third edition, 2009.
- [8] M. Denil, B. Shakibi, L. Dinh, M. Ranzato, and N. de Freitas. Predicting parameters in deep learning. In C. J. C. Burges, L. Bottou, M. Welling, Z. Ghahramani, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems (NIPS)*, volume 26, pages 2148–2156. MIT Press, Cambridge, MA, 2013.
- [9] E. L. Denton, W. Zaremba, J. Bruna, Y. LeCun, and R. Fergus. Exploiting linear structure within convolutional networks for efficient evaluation. In Z. Ghahramani, M. Welling, C. Cortes, N. D. Lawrence, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems (NIPS)*, volume 27, pages 1269–1277. MIT Press, Cambridge, MA, 2014.
- [10] S. Dieleman, J. Schlüter, C. Raffel, E. Olson, S. K. Sønderby, D. Nouri, D. Maturana, M. Thoma, E. Battenberg, J. Kelly, J. D. Fauw, M. Heilman, D. M. de Almeida, B. McFee, H. Weideman, G. Takács, P. de Rivaz, J. Crall, G. Sanders, K. Rasul, C. Liu, G. French, and J. Degraeve. Lasagne: First release, Aug. 2015.
- [11] Y. C. Eldar and G. Kutyniok, editors. *Compressed Sensing: Theory and Applications*. Cambridge University Press, 2012.
- [12] Y. Gong, L. Liu, M. Yang, and L. Bourdev. Compressing deep convolutional networks using vector quantization. In *Proc. of the 3rd Int. Conf. Learning Representations (ICLR 2015)*, San Diego, CA, May 7–9 2015.
- [13] Y. Guo, A. Yao, and Y. Chen. Dynamic network surgery for efficient DNNs. In D. D. Lee, M. Sugiyama, U. von Luxburg, I. Guyon, and R. Garnett, editors, *Advances in Neural Information Processing Systems (NIPS)*, volume 29, pages 1379–1387. MIT Press, Cambridge, MA, 2016.
- [14] S. Han, H. Mao, and W. J. Dally. Deep compression: Compressing deep neural networks with pruning, trained quantization and Huffman coding. In *Proc. of the 4th Int. Conf. Learning Representations (ICLR 2016)*, San Juan, Puerto Rico, May 2–4 2016.
- [15] S. Han, J. Pool, J. Tran, and W. Dally. Learning both weights and connections for efficient neural network. In C. Cortes, N. D. Lawrence, D. D. Lee, M. Sugiyama, and R. Garnett, editors, *Advances in Neural Information Processing Systems (NIPS)*, volume 28, pages 1135–1143. MIT Press, Cambridge, MA, 2015.
- [16] S. J. Hanson and L. Y. Pratt. Comparing biases for minimal network construction with back-propagation. In D. S. Touretzky, editor, *Advances in Neural Information Processing Systems (NIPS)*, volume 1, pages 177–185. Morgan Kaufmann, San Mateo, CA, 1989.
- [17] B. Hassibi and D. G. Stork. Second order derivatives for network pruning: Optimal brain surgeon. In S. J. Hanson, J. D. Cowan, and C. L. Giles, editors, *Advances in Neural Information Processing Systems (NIPS)*, volume 5, pages 164–171. Morgan Kaufmann, San Mateo, CA, 1993.
- [18] B. Hassibi, D. G. Stork, G. Wolff, and T. Watanabe. Optimal brain surgeon: Extensions and performance considerations. In J. D. Cowan, G. Tesauero, and J. Alspector, editors, *Advances in Neural Information Processing Systems (NIPS)*, volume 6, pages 263–270. Morgan Kaufmann, San Mateo, CA, 1994.
- [19] T. Hastie, R. Tibshirani, and M. Wainwright. *Statistical Learning with Sparsity: The Lasso and Generalizations*. Monographs on Statistics and Applied Probability. Chapman & Hall/CRC, 2015.
- [20] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. In *Proc. of the 2016 IEEE Computer Society Conf. Computer Vision and Pattern Recognition (CVPR’16)*, pages 770–778, Las Vegas, NV, June 26 – July 1 2016.
- [21] M. Jaderberg, A. Vedaldi, and A. Zisserman. Speeding up convolutional neural networks with low rank expansions. In M. Valstar, A. French, and T. Pridmore, editors, *Proc. of the 25th British Machine Vision Conference (BMVC 2014)*, Nottingham, UK, Sept. 1–5 2014.
- [22] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proc. IEEE*, 86(11):2278–2324, Nov. 1998.
- [23] Y. LeCun, J. S. Denker, and S. A. Solla. Optimal brain damage. In D. S. Touretzky, editor, *Advances in Neural Information Processing Systems (NIPS)*, volume 2, pages 598–605. Morgan Kaufmann, San Mateo, CA, 1990.
- [24] H. Li, A. Kadav, I. Durdanovic, and H. P. Graf. Pruning filters for efficient ConvNets. In *Proc. of the 5th Int. Conf. Learning Representations (ICLR 2017)*, Toulon, France, Apr. 24–26 2017.
- [25] B. Liu, M. Wan, H. Foroosh, M. Tappen, and M. Pensky. Sparse convolutional neural networks. In *Proc. of the 2015 IEEE Computer Society Conf. Computer Vision and Pattern Recognition (CVPR’15)*, pages 806–814, Boston, MA, June 7–12 2015.
- [26] Y. Nesterov. A method of solving a convex programming problem with convergence rate $\mathcal{O}(1/k^2)$. *Soviet Math. Dokl.*, 27(2):372–376, 1983.

- [27] A. Novikov, D. Podoprikin, A. Osokin, and D. P. Vetrov. Tensorizing neural networks. In C. Cortes, N. D. Lawrence, D. D. Lee, M. Sugiyama, and R. Garnett, editors, *Advances in Neural Information Processing Systems (NIPS)*, volume 28, pages 442–450. MIT Press, Cambridge, MA, 2015.
- [28] R. Reed. Pruning algorithms—a survey. *IEEE Trans. Neural Networks*, 4(5):740–747, Sept. 1993.
- [29] T. N. Sainath, B. Kingsbury, V. Sindhwani, E. Arisoy, and B. Ramabhadran. Low-rank matrix factorization for deep neural network training with high-dimensional output targets. In *Proc. of the IEEE Int. Conf. Acoustics, Speech and Sig. Proc. (ICASSP'13)*, pages 6655–6659, Vancouver, Canada, Mar. 26–30 2013.
- [30] Theano Development Team. Theano: A Python framework for fast computation of mathematical expressions. arXiv:1605.02688, May 9 2016.
- [31] A. S. Weigend, D. E. Rumelhart, and B. A. Huberman. Generalization by weight-elimination with application to forecasting. In R. P. Lippmann, J. E. Moody, and D. S. Touretzky, editors, *Advances in Neural Information Processing Systems (NIPS)*, volume 3, pages 875–882. Morgan Kaufmann, San Mateo, CA, 1991.
- [32] W. Wen, C. Wu, Y. Wang, Y. Chen, and H. Li. Learning structured sparsity in deep neural networks. In D. D. Lee, M. Sugiyama, U. von Luxburg, I. Guyon, and R. Garnett, editors, *Advances in Neural Information Processing Systems (NIPS)*, volume 29, pages 2074–2082. MIT Press, Cambridge, MA, 2016.
- [33] D. Yu, F. Seide, G. Li, and L. Deng. Exploiting sparseness in deep neural networks for large vocabulary speech recognition. In *Proc. of the IEEE Int. Conf. Acoustics, Speech and Sig. Proc. (ICASSP'12)*, pages 4409–4412, Kyoto, Japan, Mar. 25–30 2012.