# Supplementary material for: “Learning-Compression” algorithms for neural net pruning

Miguel Á. Carreira-Perpiñán        Yerlan Idelbayev
Electrical Engineering and Computer Science, University of California, Merced
`http://eecs.ucmerced.edu`

March 28, 2018

**Abstract**

This is an extended version of the CVPR paper. Specific supplementary materials pointed to in the latter are as follows. 1) Pseudocode for the augmented Lagrangian (AL) version of the LC algorithm (fig. 2). 2) Proofs for the C step solution, both for the constraint version (appendix B.1) and the penalty version (appendix B.2). 3) Theoretical estimate for a good initial value $\mu_0$ for the penalty parameter, and analysis of the beginning of the path (appendix C). 4) Detailed parameters (optimization, etc.) for the experiments with LeNet and ResNet neural nets, and experiments with LeNet5 (section 8).

## 1 Introduction

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 (see related work below). The 2010s have shown that neural nets achieve state-of-the-art performance in applications such as computer vision, speech or natural language processing 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 or embedded sensors, having small memories, slow CPUs and limited battery life. This has brought a renewed interest in pruning and generally 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 (Reed, 1993).

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 (Carreira-Perpiñán, 2017) that formulates neural net compression in a general way via constrained optimization and shows how this leads to a powerful weight quantization algorithm (Carreira-Perpiñán and Idelbayev, 2017). 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: although these are one of the best approaches for large-scale constrained optimization problems whose feasible set is the nonnegative orthant, 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[1], 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 $\ell_0$ or $\ell_1$, 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, i.e., all but the largest weights in magnitude are zeroed. This gives support to using magnitude as a measure of weight saliency (as opposed to, say, curvature of the loss locally). 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 (section 3) and develop it for its constraint and penalty forms (sections 4–5). Although we focus on $\ell_0$, $\ell_1$ and $\ell_2^2$, we emphasize our framework applies generally to other costs. Then we describe how to learn the amount of pruning per layer automatically (section 6) and discuss the algorithm's behavior (section 7). Our experiments (section 8) 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.

## 2 Related work

Pruning was recognized as an important problem since the 1980s; see reviews (Reed, 1993) and (Bishop, 1995, ch. 9.5). Most methods can be classified into two types: 1) *saliency ranking methods* use a criterion to estimate the importance of each weight in the net, remove less important weights and retrain the rest (similarly to "filter" methods in feature selection; Kohavi and John, 1997); and 2) *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.

For saliency ranking, many saliency criteria were used. The difference in loss between training with and without $w_i$ (equivalent to a backward selection algorithm) provides a direct measure of importance for $w_i$, but is too slow to compute with large nets and datasets. The magnitude $|w_i|$ is fast to compute and would work well if removing very small weights (which would have a neglible effect on the loss). Unfortunately, the weights' distribution is widely and continuously spread out, with no clear separation between small and large weights; and magnitude pruning implicitly assumes that the inputs of all neurons have the same scale. Approximating the loss quadratically leads to a curvature criterion using the diagonal Hessian entries (LeCun et al., 1990) or all the Hessian entries (Hassibi and Stork, 1993; Hassibi et al., 1994). Other criteria involved the sensitivity of the loss to removing $w_i$ (defined or estimated in various ways), and the variance of the weights into a neuron (for pruning neurons). While most saliency methods are simple and fast, their performance is limited for several reasons: 1) they are *local*, i.e., the saliency estimate for each $w_i$ is valid at the reference net $\overline{\mathbf{w}}$ but not away from it at $\overline{\mathbf{w}} + \Delta\mathbf{w}$ (and removing a set of weights will send us far away from $\overline{\mathbf{w}}$). 2) They are *greedy*: weights are pruned irrevocably, with no backtracking. And 3), *individual* weight saliency (however defined) 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 doing the pruning/retraining procedure in stages, where at each stage we prune only a few weights and retrain the rest. However, practical application of this places a burden on the network designer, who has to find by expensive trial-and-error how many stages to run and how many weights to prune at each one (so the designer effectively becomes part of the algorithm). Magnitude pruning in stages has been revived recently (Yu et al., 2012; Han et al., 2015), resulting in nets with a considerable proportion of weights being pruned, however this is partly due to the fact that many neural nets are enormously overparameterized.

Penalty methods were mostly based on weight decay and variations of it. Weight decay penalizes $\alpha \sum_i w_i^2$ for $\alpha > 0$ and can indeed help avoid overfitting and prune weights. However, it favors many small weights rather than a few large ones. Variations of it encourage weights to be either large or small,

[1]An actual implementation of a pruned network in a target hardware also needs to consider how the indices of the nonzero weights will be encoded. How to do this optimally depends on the specific hardware (CPU, GPU, FPGA. . . ) and is beyond the scope of this paper. One work studying this is Han et al. (2016a).

such as $\alpha \sum_i w_i^2/(A+w_i^2)$ for $A>0$ (Hanson and Pratt, 1989; Weigend et al., 1991). The basic problem of these penalties is that they are not sparsifying: upon convergence, generally none of the weights are zero, and one has to draw an arbitrary line below which weights are pruned, just as with saliency-based methods. Sparsifying penalties such as $\ell_0$ or $\ell_1$ seem not to have been investigated until recently, presumably because backpropagation cannot handle their nonsmoothness. Group LASSO penalties (to prune entire filters of a net) have been recently considered by Liu et al. (2015); Wen et al. (2016). 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 (Yu et al., 2012, section 3.1).

Finally, pruning can be combined with other compression techniques to reduce the size of the net further, such as weight quantization (Gong et al., 2015; Han et al., 2015; Carreira-Perpiñán and Idelbayev, 2017), low-rank decomposition of weight matrices (Sainath et al., 2013; Denil et al., 2013; Jaderberg et al., 2014; Denton et al., 2014; Novikov et al., 2015), hashing (Chen et al., 2015), lossless compression such as Huffman codes (Han et al., 2016b), 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.*

## 3 Neural network pruning as an optimization problem[2]

We define a *pruning cost* as a function $C\colon \mathbb{R}^n \rightarrow \mathbb{R}^+$ satisfying $C(\mathbf{0}) = 0$ and $C(\mathbf{w}) > 0$ if $\mathbf{w} \neq \mathbf{0}$. This applies to vectors. We say that $C$ is separable if $C(\mathbf{w}) = \sum_{i=1}^n c(w_i)$ where $c\colon \mathbb{R} \rightarrow \mathbb{R}^+$ is a scalar pruning cost, i.e., $c(0) = 0$ and $c(w) > 0$ if $w \neq 0$. $C$ should be designed such that it penalizes nonzeros in the vector $\mathbf{w}$. The *pruning cost function* $C$ and the *pruning operators* $\Pi_C^+$, $\Pi_C^{\leq}$ defined later are central concepts in our framework.

We study three important examples, all of which are separable:

- $C(\mathbf{w}) = \|\mathbf{w}\|_0 = \sum_{i=1}^n \|w_i\|_0 = \sum_{i=1}^n \delta_{w_i}$ (the number of nonzero elements of $\mathbf{w}$).
- $C(\mathbf{w}) = \|\mathbf{w}\|_1 = \sum_{i=1}^n |w_i|$.
- $C(\mathbf{w}) = \|\mathbf{w}\|_2^2 = \sum_{i=1}^n w_i^2$.

While other costs could be studied that are of practical interest, these three are representative of what can be achieved in our framework and illustrate the issues of sparsification and shrinkage. $\ell_0$ is arguably the most natural definition of the problem of pruning, as it is equivalent to finding the best subset of pruned weights, but it is hard combinatorial problem. $\ell_2^2$ corresponds to regular weight decay and can be optimized directly by gradient-based 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 using a neural net with weights $\mathbf{w}$):

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

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

Both naturally aim at learning an optimal model, by minimizing the data-dependent loss $L(\mathbf{w})$, but subject to or encouraged to having many zero weights, as given by the *pruning parameters* $\kappa \geq 0$ and $\alpha \geq 0$, for the constraint and penalty form, respectively. 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* (Carreira-Perpiñán, 2017) 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 alternating optimization over

[2]Notation. Norms $\|\cdot\|$ are Euclidean norms $\|\cdot\|_2$ by default. See also appendix A about the $\ell_p$ and $\ell_0$ "norms". We use the indicator function $I(x) = 1$ if $x$ is true and 0 otherwise, and the sign function $\operatorname{sgn}(x) = -1$ if $x < 0$, 0 if $x = 0$ and $+1$ if $x > 0$. "Largest" element or weight means largest in magnitude (absolute value).

$\mathbf{w}$ and $\boldsymbol{\theta}$. We describe the mathematical development for the constraint form first (section 4) and for the penalty form next (section 5).

Before proceeding, note that the penalty and the constraint forms define problems that are equivalent for appropriate choices of $\kappa$ and $\alpha$. 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 $\ell_0$

## 4 Constraint form for the pruning cost

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

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

This problem is in the *model compression as constrained optimization* form of Carreira-Perpiñán (2017):

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

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 (note we only apply the QP or AL to the equality constraint, not to the inequality):

$$Q(\mathbf{w},\boldsymbol{\theta};\mu) = L(\mathbf{w}) + \frac{\mu}{2}\left\|\mathbf{w}-\boldsymbol{\theta}\right\|^2 \text{ s.t. } C(\boldsymbol{\theta}) \leq \kappa \tag{4}$$

$$\mathcal{L}_A(\mathbf{w},\boldsymbol{\theta},\boldsymbol{\lambda};\mu) = L(\mathbf{w}) + \frac{\mu}{2}\left\|\mathbf{w}-\boldsymbol{\theta}\right\|^2 - \boldsymbol{\lambda}^T(\mathbf{w}-\boldsymbol{\theta}) \text{ s.t. } C(\boldsymbol{\theta}) \leq \kappa \tag{5}$$

$$= L(\mathbf{w}) + \frac{\mu}{2}\left\|\mathbf{w}-\boldsymbol{\theta}-\frac{1}{\mu}\boldsymbol{\lambda}\right\|^2 - \frac{1}{2\mu}\left\|\boldsymbol{\lambda}\right\|^2 \text{ s.t. } C(\boldsymbol{\theta}) \leq \kappa. \tag{6}$$

For the QP, we optimize $Q$ over $(\mathbf{w},\boldsymbol{\theta})$ while driving $\mu \to \infty$, 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 $\mu \to \infty$. Note the term $-\frac{1}{2\mu}\left\|\boldsymbol{\lambda}\right\|^2$ is constant and can be ignored during the optimization of $\mathcal{L}_A$, so the AL function looks like a QP function with a penalty offset by $\frac{1}{\mu}\boldsymbol{\lambda}$. The Lagrange multiplier estimates $\boldsymbol{\lambda}$ make the iterates $(\mathbf{w},\boldsymbol{\theta})$ be closer to the solution for the same value of $\mu$, so generally AL is preferable.

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

**Learning (L) step (over w)**

$$\min_{\mathbf{w}} L(\mathbf{w}) + \frac{\mu}{2}\left\|\mathbf{w}-\boldsymbol{\theta}\right\|^2. \tag{7}$$

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 $\boldsymbol{\theta}$)**

$$\Pi_C^{\leq}(\mathbf{w};\kappa) \quad = \quad \arg\min_{\boldsymbol{\theta}} \left\|\mathbf{w}-\boldsymbol{\theta}\right\|^2 \quad \text{s.t.} \quad C(\boldsymbol{\theta}) \leq \kappa \tag{8}$$

(where the "$^{\leq}$" superindex refers to the constraint form, to differentiate it from the "$^{+}$" superindex for the penalty form introduced in section 5). This has the form of a proximal operator, which we call *pruning operator*. It can be solved exactly for several useful $C$, including $\ell_0$, $\ell_1$ and $\ell_2^2$. In our context, "compression" is understood as "weight pruning".

We describe both steps 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. Fig. 2 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 $\theta_i$ to 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 each $w_i \xrightarrow[\mu\to\infty]{} 0$ and thus is actually pruned.

## 4.1 L step: learning the weights

The L step of eq. (7) has the same general form given by Carreira-Perpiñán (2017). It does not depend on the fact that we are pruning the neural net (as opposed to compressing it in some other way, such as quantizing its weights). The values in $\boldsymbol{\theta}$ are constant within the L step. It does depend on the training set and the neural net loss $L(\mathbf{w})$.

The optimization of the L step is very similar to that of the reference net, which minimizes $L(\mathbf{w})$ alone. With deep nets, we minimize $L(\mathbf{w})$ with stochastic gradient descent (SGD). Likewise, we use SGD in the L step to optimize over $\mathbf{w}$ either $Q$ in eq. (4) or $\mathcal{L}_A$ in eq. (6). To make SGD more robust, we use a learning rate $\eta'_t = \min\left(\eta_t, \frac{1}{\mu}\right)$ at epoch $t$ where $(\eta_t)$ is a Robbins-Monro schedule for the SGD learning rates for $L(\mathbf{w})$. This clipped schedule avoids erratic weight updates as $\mu$ becomes large, while still guaranteeing convergence, since $(\eta'_t)$ is a Robbins-Monro schedule (Carreira-Perpiñán, 2017).

## 4.2 C step: marking weights for pruning

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 (i.e., $\Pi_C^{\leq}(\mathbf{w};\kappa) = \mathbf{w}$) if $C(\mathbf{w}) \leq \kappa$. Otherwise, the resulting $\boldsymbol{\theta}$ satisfies $C(\boldsymbol{\theta}) \leq \kappa < C(\mathbf{w})$, which implies that $\boldsymbol{\theta}$ is "smaller" than $\mathbf{w}$ in the sense of the cost $C$. Indeed, this will result (depending on the cost $C$) in sparsification, where many weights $\theta_i$ are exactly zero, and/or shrinkage, where many weights individually satisfy $|\theta_i| < |w_i|$ (although some weights may stay or increase in magnitude).

The solution for several costs $C$ corresponding to projection on $\ell_p$ balls is well known in optimization and is given in fig. 1 (see formal statements and proofs in appendix B.1). When $\mathbf{w}$ is in the ball, $\boldsymbol{\theta} = \mathbf{w}$ for all $\ell_p$ norms, i.e., no change in the weights. Otherwise, the pruning behavior depends on the norm, and can present two important characteristics: *sparsification*, in which weights within some interval become exactly zero; and *shrinkage*, in which weights that do not become zero become smaller anyway. Specifically:

- $\ell_0$ leaves unchanged the top-$\kappa$ weights (in magnitude) and prunes the rest, that is, it sparsifies but does not shrink.
- $\ell_1$ shrinks on average the top-$k$ (not top-$\kappa$) weights, where $k$ depends on $\mathbf{w}$ and $\kappa$, and prunes the rest, that is, it sparsifies and shrinks.
- $\ell_2^2$ normalizes $\mathbf{w}$ (dividing it by $\|\mathbf{w}\|$), that is, it shrinks all weights but does not sparsify.

The solution for $\ell_0$ and $\ell_1$ involves thresholding the weights. That is, weights with magnitude above a certain threshold value survive and weights below it are pruned. Hence, the surviving weights are the largest weights. The value of the threshold can always be taken as the value of the $k$th largest weight (in magnitude), where $k$ is different for each norm. For $\ell_0$, $k = \kappa + 1$. For $\ell_1$, $k$ depends on $\mathbf{w}$ and $\kappa$ in a more complicated way, and can be obtained by scanning the weights in decreasing order of magnitude (see appendix B.1).

Computationally, a simple algorithm for $\ell_0$ and $\ell_1$ involves first sorting the elements of $\mathbf{w}$ in magnitude (at a runtime $\mathcal{O}(n \log n)$ if $\mathbf{w}$ has $n$ elements), and then scanning this in $\mathcal{O}(n)$ to find the threshold $\eta$ and return the nonzeros. But both $\ell_0$ and $\ell_1$ 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; Cormen et al., 2009, ch. 9). For $\ell_0$, this is obvious. For $\ell_1$, see Condat (2016). Since the number of weights in a deep net, which is our driving application for model compression, is large (upwards of millions in practice), using selection instead of sorting matters. That said, the L step dominates the C step by far.

With $\ell_0$ there may be multiple optima if there are ties in the top-$\kappa$ weights. In that case, we prefer the elements that were in the nonzero set in the previous iteration (this avoids arbitrary oscillations in the set of nonzeros over iterations). This can be achieved by using a stable sort (Cormen et al., 2009) or a stable selection.

## 5 Penalty form for the pruning cost

Although the penalty form does not have the *model compression as constrained optimization* form of Carreira-Perpiñán (2017), we can apply the same technique to arrive at a convenient LC algorithm. First we duplicate $\mathbf{w}$ via an auxiliary variable $\boldsymbol{\theta}$:

$$\min_{\mathbf{w},\boldsymbol{\theta}} L(\mathbf{w}) + \alpha\, C(\boldsymbol{\theta}) \quad \text{s.t.} \quad \mathbf{w} = \boldsymbol{\theta}. \tag{9}$$

Then we optimize this via QP or AL:

$$Q(\mathbf{w},\boldsymbol{\theta};\mu) = L(\mathbf{w}) + \frac{\mu}{2}\|\mathbf{w}-\boldsymbol{\theta}\|^2 + \alpha\, C(\boldsymbol{\theta}) \tag{10}$$

$$\mathcal{L}_A(\mathbf{w},\boldsymbol{\theta},\boldsymbol{\lambda};\mu) = L(\mathbf{w}) + \frac{\mu}{2}\|\mathbf{w}-\boldsymbol{\theta}\|^2 - \boldsymbol{\lambda}^T(\mathbf{w}-\boldsymbol{\theta}) + \alpha\, C(\boldsymbol{\theta}) \tag{11}$$

$$= L(\mathbf{w}) + \frac{\mu}{2}\left\|\mathbf{w}-\boldsymbol{\theta}-\frac{1}{\mu}\boldsymbol{\lambda}\right\|^2 + \alpha\, C(\boldsymbol{\theta}) - \frac{1}{2\mu}\|\boldsymbol{\lambda}\|^2. \tag{12}$$

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 (eq. (7)), and a **C step (over $\boldsymbol{\theta}$)** with the following form for the QP (for the AL, replace $\mathbf{w}$ with $\mathbf{w} - \frac{1}{\mu}\boldsymbol{\lambda}$):

$$\Pi^+_C\Big(\mathbf{w};\frac{2\alpha}{\mu}\Big) \quad = \quad \arg\min_{\boldsymbol{\theta}} \; \|\mathbf{w}-\boldsymbol{\theta}\|^2 + \frac{2\alpha}{\mu}C(\boldsymbol{\theta}) \tag{13}$$

(where the "+" superindex refers to the penalty form). This is again a proximal operator that can often be solved exactly, as shown in the next section. Fig. 2 gives the algorithm pseudocode for the AL.

### 5.1 C step: marking weights for pruning

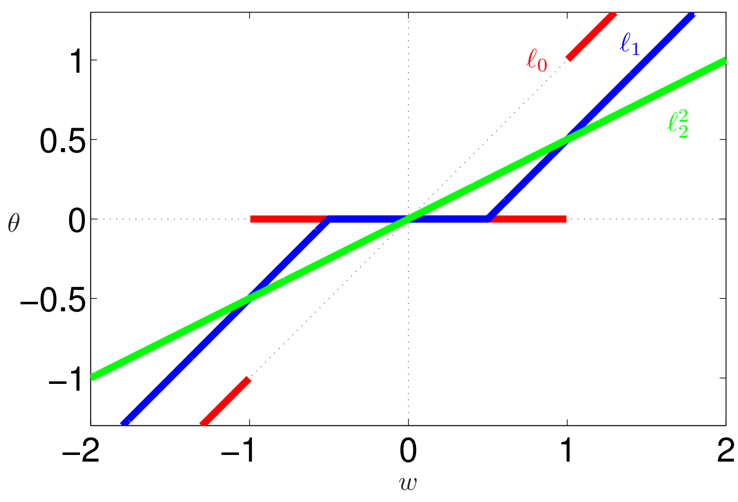
If the pruning cost separates, $C(\boldsymbol{\theta}) = \sum_{i=1}^n c(\theta_i)$, so does the C step objective function, which can then be solved elementwise (separately for each weight in the net) and takes the form (for the QP)

$$\Pi^+_c\Big(w;\frac{2\alpha}{\mu}\Big) \quad = \quad \arg\min_{\theta\in\mathbb{R}} \; (w-\theta)^2 + \frac{2\alpha}{\mu}c(\theta) \tag{14}$$

with scalars $w,\theta \in \mathbb{R}$. Firstly, we show that $\Pi^+_C\big(w;\frac{2\alpha}{\mu}\big)$ makes $w$ smaller or equal with the same sign.

**Theorem 5.1.** *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* $\operatorname{sgn}(\theta^*) = \operatorname{sgn}(w)$ *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 $\theta$ and $w$. Second, let $w\in\mathbb{R}$. Since $(w-\theta)^2$ is smaller when $\theta$ has the same sign as $w$ than when it has the opposite sign (for the same magnitude of $\theta$), and $c(\theta)=c(-\theta)$, then $F$ is smaller also. Hence, $\theta^*$ 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 the fact that $F(\theta^*;w) \leq F(\theta;w)$ $\forall\theta\geq 0$ since $\theta^*$ is the global minimizer. □



| $C(\mathbf{w})$ | Constraint form[†] $\boldsymbol{\theta} = \Pi_C^{\leq}(\mathbf{w};\kappa)$ | Penalty form $\boldsymbol{\theta} = \Pi_C^{+}\big(\mathbf{w};\frac{2\alpha}{\mu}\big)$ |
|---|---|---|
| $\ell_0$: $\lVert\mathbf{w}\rVert_0$ | $w_i \cdot I\big(\lvert w_i\rvert > \eta_0\big)$ | $w_i \cdot I\Big(\lvert w_i\rvert > \sqrt{\frac{2\alpha}{\mu}}\Big)$ |
| $\ell_1$: $\lVert\mathbf{w}\rVert_1$ | $(w_i - \operatorname{sgn}(w_i)\,\eta_1)\cdot I(\lvert w_i\rvert > \eta_1)$ | $\big(w_i - \operatorname{sgn}(w_i)\,\frac{\alpha}{\mu}\big)\cdot I\big(\lvert w_i\rvert > \frac{\alpha}{\mu}\big)$ |
| $\ell_2^2$: $\lVert\mathbf{w}\rVert_2^2$ | $\sqrt{\kappa}\,w_i/\lVert\mathbf{w}\rVert_2$ | $w_i/\big(1+\frac{2\alpha}{\mu}\big)$ |

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

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 $\Pi_C^{+}\big(\mathbf{w};\frac{2\alpha}{\mu}\big)$ (penalty form). All cases result in an elementwise operator that computes $\theta_i$ from $w_i$ for each weight. In the constraint form, there are two thresholds $\eta_0$ and $\eta_1$ that depend on all the weights $\mathbf{w} \in \mathbb{R}^n$: for $\ell_0$, the threshold $\eta_0$ equals the magnitude of the $(\kappa+1)$th largest weight; for $\ell_1$, the threshold $\eta_1$ can be obtained by scanning $w_i$ in decreasing order of magnitude (see main text). In the penalty form, the thresholds are independent of the weights. The top graph plots the elementwise pruning operator for the penalty form.

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 formal statements and proofs in appendix B.2). As in the constraint form, we observe two types of behavior: *sparsification* and *shrinkage*. Specifically, $\ell_0$ sparsifies but does not shrink: $w$ is either pruned ($\theta = 0$) or left as is ($\theta = w$). $\ell_1$ sparsifies and shrinks: $w$ is either pruned ($\theta = 0$) or shifted towards zero ($\theta = w - \operatorname{sgn}(w)\,\frac{\alpha}{\mu}$). And $\ell_2^2$ does not sparsify but shrinks: $w$ is divided by a number bigger than 1.

## 6 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. The behavior of the penalty and constraint forms when using a single pruning parameter ("global sparsity") vs per-layer pruning parameters ("local sparsity") is quite different and we study it here. Consider a net with $K$ layers of weights $\mathbf{w} = (\mathbf{w}_1, \dots, \mathbf{w}_K)$ and assume the pruning cost $C$ is separable.

In the penalty form, the global sparsity uses $\alpha\, C(\mathbf{w})$ and the local one uses $\alpha_1\, C(\mathbf{w}_1) + \dots + \alpha_K\, C(\mathbf{w}_K)$, where $\alpha \geq 0$ and $\alpha_1, \dots, \alpha_K \geq 0$ are the user-set pruning parameters. Clearly, the global sparsity is a particular case of the local one with $\alpha_1 = \dots = \alpha_K = \alpha$, so the local sparsity will produce better nets (lower loss for the same number of weights), or at least equally good, than the global sparsity. However, in practice the local sparsity requires the user to select $K$ pruning parameters ahead of time, by trial and error, which is a costly model selection problem.

In the constraint form, the global sparsity uses $C(\mathbf{w}) \leq \kappa$ (a single pruning parameter $\kappa \geq 0$ that applies to all weights in the net) and the local one uses $C(\mathbf{w}_1) \leq \kappa_1, \dots, C(\mathbf{w}_K) \leq \kappa_K$ (a separate pruning parameter $\kappa_i \geq 0$ for each layer in the net). The situation is the opposite to that of the penalty form, as shown by theorem 6.1, which applies to norms $\ell_0$ and $\ell_1$: global sparsity is guaranteed to produce equal or better nets than local sparsity. This is remarkable because we achieve the best of both worlds: ease of use (only one pruning parameter to select) and best results. *In the constraint form with a single pruning parameter $\kappa$, the LC algorithm automatically determines the best number of weights for each layer.*

**Theorem 6.1.** *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 $\mathcal{S}_l = \{\mathbf{w} = (\mathbf{w}_1, \dots, \mathbf{w}_K) \in \mathbb{R}^n\colon\ C(\mathbf{w}_1) \leq \kappa_1, \dots, C(\mathbf{w}_K) \leq \kappa_K\}$ and $\mathcal{S}_g = \{\mathbf{w} \in \mathbb{R}^n\colon\ C(\mathbf{w}) \leq \kappa\}$. Then $\mathcal{S}_l \subset \mathcal{S}_g$.*

*Proof.* Let $\mathbf{w} \in \mathcal{S}_\text{l}$. 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 $\mathbf{w} \in \mathcal{S}_\text{g}$. $\square$

Simply stated, this means the following. Consider the $\ell_0$ case with $\kappa_1 + \dots + \kappa_K = \kappa$ set by the user. Global sparsity defines a feasible set of nets with at most $\kappa$ weights total. Local sparsity defines a feasible set of nets with at most $\kappa_i$ weights in layer $i$, for $i = 1, \dots, K$ (with a total of $\kappa_1 + \dots + \kappa_K = \kappa$ weights). So the global sparsity searches over more possible nets (all combinations of $(\kappa_1, \dots, \kappa_K)$ values) automatically and saves the user the considerable trouble of solving this exponentially costly model selection. It is possible (though not necessary) that for particular values satisfying $\kappa < \kappa_1 + \dots + \kappa_K$ the global sparsity solution may be worse than the local sparsity one. But by scanning the domain $\kappa \in [0, \infty)$ of the global sparsity we will find all the optimal solutions for both the global and local spaces. For $\ell_0$, the domain of $\kappa$ is simply $[0, n]$ where $n$ is the total number of weights.

Happily, our optimization mechanism in the C step applies equally easily to both global and local sparsity. For example, for the $\ell_0$ case with global sparsity, the top-$\kappa$ 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. With the local sparsity, the top-$\kappa_i$ weights in layer $i$ stay, for $i = 1, \dots, K$. Our experiments show that the simpler, global sparsity setting indeed produces a lower loss than the local sparsity for the same total number of weights. The pseudocode in fig. 2 gives a global version for both the constraint (single $\kappa$) or penalty form (single $\alpha$ parameter).

Finally, we may want to apply pruning only to a subset of parameters of the net. For example, we usually do not prune the biases, only the multiplicative weights (there are very few biases in comparison, but they are more critical). This is achieved by having the cost $C(\mathbf{w})$ apply only to the parameters that pruning applies to.

**Some subtle points** Sometimes (for some values of the pruning parameters $\kappa$ and $\alpha$) the penalty form may appear to prune more aggressively than the constraint form. To see this, imagine the loss $L(\mathbf{w})$ is constant. Then, since the value of $L(\mathbf{w})$ in eqs. (1) can be ignored, the constraint form will accept as optimal any net with cost $C(\mathbf{w}) = \kappa$ (or smaller), while the penalty form will pick the smallest possible cost $C(\mathbf{w}) = 0$ for any $\alpha > 0$. Hence, the penalty form would lead to more pruning than the constraint form (with equal loss value). In practice the loss is not constant, but since the optimization may not be exact, we may find in practice the following situation: both the constraint and penalty forms produce nets with a similar loss (for some setting of $\kappa$ and $\alpha$) but the penalty form achieves more pruning. This does not mean that the penalty form is more effective than the constraint one, for the same reason as above: we have to consider the nets obtained over the domain $\kappa \in [0, \infty)$. In the previous example, reducing $\kappa$ will force the constraint form to prune more weights and find a solution as good as any penalty-based one.

The discussion about global sparsity in the constraint form shows that the LC algorithm automatically determines how many weights to prune in each layer. One may wonder: could an entire layer be pruned at some iteration of the LC algorithm? Obviously, this would be problematic, because the loss would be very large (the neural net would ignore its input). The answer to this question is: yes, it can happen that $\boldsymbol{\theta} = \mathbf{0}$ for an entire layer, but that does not mean the layer is pruned. Let us see this. A layer can indeed get $\boldsymbol{\theta} = \mathbf{0}$ because the range of variation of the weights depends on their layer in the net, e.g. Carreira-Perpiñán and Idelbayev (2017) (fig. 13) shows that for LeNet300 the weights are roughly in $[-0.4, 0.4]$ for layer 1 and in $[-2, 2]$ for layer 3. Hence, it is possible that a layer with relatively small weights get $\theta_i = 0$ for every weight $w_i$ if we require a high sparsity (small $\kappa$). Further, for the penalty form, we always have $\boldsymbol{\theta} = \mathbf{0}$ for the entire net in the first C step. So $\boldsymbol{\theta} = \mathbf{0}$ can perfectly occur for a whole layer. But setting $\theta_i = 0$ for a weight $w_i$ does not mean that $w_i$ is actually pruned: it is simply *currently marked to be pruned*. The L step, which takes into account the loss, will ensure that some weights remain in each layer as needed to achieve a low loss upon convergence.

We can take this discussion further and connect it with magnitude-based pruning, the simple algorithm where we prune all but the top $\kappa$ weights (in magnitude) in the reference net and retrain these $\kappa$ weights to minimize the loss. Theorem 6.1 applies just as well to magnitude-based pruning, hence this algorithm also determines automatically the best number of weights for each layer—but this is optimal *given the reference net weights*. Magnitude-based pruning irrevocably commits to selecting the top $\kappa$ weights in the reference, while our LC algorithm explores different subsets of weights. The following example makes this clear: if $\kappa$ is very small, some layers may be *entirely pruned* by magnitude-based pruning, and this will blow up the loss, as mentioned before. The LC algorithm will recover from this initial situation because the L step will increase some weights in those layers to reduce the loss, as we just described in the previous paragraph. The key is that the C step *marks weights to be currently pruned* but does not actually prune them.

## 7 Behavior, convergence and practicalities of the LC algorithm

### 7.1 Behavior as $\mu$ increases from 0 to $\infty$

QP and AL belong to the family of homotopy (path-following) algorithms, where the minima $(\mathbf{w}(\mu), \boldsymbol{\theta}(\mu))$ of $Q(\mathbf{w}, \boldsymbol{\theta}; \mu)$ or $\mathcal{L}_A(\mathbf{w}, \boldsymbol{\theta}, \boldsymbol{\lambda}; \mu)$ define a path for $\mu \geq 0$ and the solution we want is at $\mu \to \infty$. Computationally, it is better to approach the solution by following the path from small $\mu$, because $Q$ or $\mathcal{L}_A$ become progressively ill-conditioned as $\mu \to \infty$. With convex problems, there is a unique path leading to the solution. With nonconvex problems, there are multiple paths, each leading to a local optimum.

**Beginning of the path ($\mu \to 0^+$) and initialization of the LC algorithm** For AL we always initialize $\boldsymbol{\lambda} = \mathbf{0}$. What happens at the beginning of the path provides an interesting perspective on pruning/retraining algorithms previously proposed (e.g. Han et al., 2015). We describe this for the constraint form; the penalty form is more complicated to analyze, but is qualitatively similar (see appendix C). Taking the limit $\mu \to 0^+$ results in $(\mathbf{w}(0), \boldsymbol{\theta}(0)) = (\overline{\mathbf{w}}, \boldsymbol{\theta}^{\mathrm{DC}})$, as follows. Minimizing $Q$ gives the same result as minimizing $\mathcal{L}_A$ since $\boldsymbol{\lambda} = \mathbf{0}$ at the beginning. This corresponds to minimizing over $\mathbf{w}$ with $\mu = 0$ and then minimizing over $\boldsymbol{\theta}$ given that $\mathbf{w}$. Minimizing over $\mathbf{w}$ with $\mu = 0$ is an exact L step $\overline{\mathbf{w}} = \arg\min_{\mathbf{w}} L(\mathbf{w})$, or equivalently, $\overline{\mathbf{w}}$ *is a well-trained, reference model*. Minimizing over $\boldsymbol{\theta}$ given $\mathbf{w} = \overline{\mathbf{w}}$ is an exact C step whose solution is $\boldsymbol{\theta}^{\mathrm{DC}} = \Pi_C^{\leq}(\overline{\mathbf{w}}; \kappa)$. This was called *direct compression* (here, direct pruning) in Carreira-Perpiñán (2017), as it corresponds to *pruning the reference weights independently of the loss*. The weights $\boldsymbol{\theta}^{\mathrm{DC}}$ result from magnitude pruning of the reference weights $\overline{\mathbf{w}}$ and they produce a large loss, which can be reduced by retraining the nonzero weights in $\boldsymbol{\theta}^{\mathrm{DC}}$. This *pruning/retraining approach* is perhaps the most widespread pruning method for deep nets; we discuss it further in section 9.

**End of the path ($\mu \to \infty$) and termination of the LC algorithm** For large enough $\mu$ the LC algorithm will identify the final set of weights that are pruned, i.e., which elements in $\boldsymbol{\theta}$ are zero. The values of $\mathbf{w}$ and the nonzeros in $\boldsymbol{\theta}$ continue changing until $\mathbf{w} = \boldsymbol{\theta}$ as $\mu \to \infty$. If we knew when the final set is identified, we could stop there, prune the weights and retrain the unpruned ones. In practice we stop when $\|\mathbf{w} - \boldsymbol{\theta}\|$ is smaller than a set tolerance and retrain the unpruned weights. Also, note that for the constraint form (which defines the feasible set $C(\boldsymbol{\theta}) \leq \kappa$) the iterate $\boldsymbol{\theta}$ is always feasible but the iterate $\mathbf{w}$ is always infeasible during training (otherwise the algorithm would stop with a $\mathbf{w} = \boldsymbol{\theta}$ that optimizes $L(\mathbf{w})$ and is feasible).

### 7.2 Convergence

Theorem 2.1 in Carreira-Perpiñán (2017) gave general conditions for convergence of the LC algorithm to a local solution of the constrained compression problem (3). Essentially, it states that if we follow the path closely enough (by minimizing $Q$ or $\mathcal{L}_A$ for each $\mu$ via sufficiently many L and C steps), then we reach a local solution of problem (9) in the limit $\mu \to \infty$ (strictly, a local stationary point, but in practice typically a minimizer). That theorem applies to our penalty form (9) without modification and it should be easy to extend it to our constraint form (2) (which contains the extra constraint $C(\boldsymbol{\theta}) \le \kappa$). However, that theorem assumes smooth (though not necessarily convex) $L(\mathbf{w})$ and $C(\boldsymbol{\theta})$. This holds for $\ell_2^2$ but not for the more interesting, sparsifying costs $\ell_0$ and $\ell_1$, which are nonsmooth. Guarantees for $\ell_0$ are likely hard to come by because it defines an NP-complete problem. Guarantees for $\ell_1$ may be easier to state. Indeed, with underdetermined linear systems (i.e., when $L(\mathbf{w})$ is quadratic positive semidefinite and the overall problem is convex), much recent progress has been made in the literature of sparse coding and compressed sensing (Donoho, 2006; Candès and Tao, 2010). It may be possible to extend some of those results to the case where $L(\mathbf{w})$ is a deep net.

We can prove that the LC algorithm does solve the problem if we assume that the set of elements that are zero in $\boldsymbol{\theta}$ is the optimal set of zeros and that it does not change over iterations. Given a set of indices $\mathcal{Z} \in \{1,\dots,n\}$, let us denote as $\mathbf{w}_{\mathcal{Z}}$ the subset of elements of $\mathbf{w} \in \mathbb{R}^n$ with indices in $\mathcal{Z}$, and $\overline{\mathcal{Z}} = \{1,\dots,n\} \setminus \mathcal{Z}$. Assume that the global optimum of problem (1a) (constraint form) occurs at a point $\mathbf{w}$ having $\kappa$ zeros whose indices are given by $\mathcal{Z}^* \in \{1,\dots,n\}$. That is, the solution of the pruning problem is the global optimum of the problem

$$\min_{\mathbf{w}} L(\mathbf{w}) \quad \text{s.t.} \quad \mathbf{w}_{\mathcal{Z}^*} = \mathbf{0}. \tag{15}$$

Consider the LC algorithm assuming that the C step always picks $\mathcal{Z}^*$ as the pruned weights. Then, the LC algorithm optimizes (for the QP):

$$Q(\mathbf{w};\mu) = L(\mathbf{w}) + \frac{\mu}{2}\left\|\mathbf{w}_{\overline{\mathcal{Z}}^*} - \boldsymbol{\theta}_{\overline{\mathcal{Z}}^*}\right\|^2 + \frac{\mu}{2}\left\|\mathbf{w}_{\mathcal{Z}^*}\right\|^2 \quad \text{s.t.} \quad C(\boldsymbol{\theta}) \le \kappa. \tag{16}$$

- For the $\ell_0$ case, $C(\boldsymbol{\theta}) = \|\boldsymbol{\theta}\|_0 = \|\boldsymbol{\theta}_{\mathcal{Z}^*}\|_0 + \|\boldsymbol{\theta}_{\overline{\mathcal{Z}}^*}\|_0 = \|\boldsymbol{\theta}_{\overline{\mathcal{Z}}^*}\|_0 \le \kappa$, so the constraint becomes ineffective. Hence, $Q$ in eq. (16) is the quadratic-penalty function of the problem

$$\min_{\mathbf{w}} L(\mathbf{w}) \quad \text{s.t.} \quad \mathbf{w}_{\overline{\mathcal{Z}}^*} = \boldsymbol{\theta}_{\overline{\mathcal{Z}}^*},\ \mathbf{w}_{\mathcal{Z}^*} = \mathbf{0} \tag{17}$$

  which is equivalent to solving problem (15) and setting $\boldsymbol{\theta}_{\overline{\mathcal{Z}}^*} = \mathbf{w}_{\overline{\mathcal{Z}}^*}$. Theorem 2.1 in Carreira-Perpiñán (2017) applies to (17) if $L$ is smooth, so the LC algorithm will solve problem (15) (in the sense of finding a local stationary point with the correct zeros).

- For the $\ell_1$ case, this is not true: $\|\boldsymbol{\theta}\|_1 = \|\boldsymbol{\theta}_{\mathcal{Z}^*}\|_1 + \|\boldsymbol{\theta}_{\overline{\mathcal{Z}}^*}\|_1 = \|\boldsymbol{\theta}_{\overline{\mathcal{Z}}^*}\|_1$, but this need not be smaller or equal than $\kappa$. Hence, $Q$ in eq. (16) is the quadratic-penalty function of the problem

$$\min_{\mathbf{w}} L(\mathbf{w}) \quad \text{s.t.} \quad \mathbf{w}_{\overline{\mathcal{Z}}^*} = \boldsymbol{\theta}_{\overline{\mathcal{Z}}^*},\ \mathbf{w}_{\mathcal{Z}^*} = \mathbf{0},\ \|\boldsymbol{\theta}_{\overline{\mathcal{Z}}^*}\|_1 \le \kappa. \tag{18}$$

  This will result in weights $\mathbf{w}_{\overline{\mathcal{Z}}^*}$ that need not be optimal for problem (15); they will be shrunk. However, retraining the net over $\mathbf{w}_{\overline{\mathcal{Z}}^*}$ (i.e., solving (18) without the inequality) will find the correct optimum.

What is the relevance of this? The previous argument shows that, from the moment the set of zeros $\mathcal{Z}$ does not change, the LC algorithm correctly but ineffiently solves the problem of minimizing over the surviving weights, with the pruned weights being zero. In this case, i.e., knowing what $\mathcal{Z}$ is, it would be more efficient to set $\mathbf{w}_{\mathcal{Z}}$ to zero and optimize over $\mathbf{w}_{\overline{\mathcal{Z}}}$ directly. But, of course, in practice we do not know what the optimal $\mathcal{Z}$ is (or when $\mathcal{Z}$ is going to stop changing), which is the key difficulty in the pruning problem, and therein is the advantage of the LC algorithm. Algorithms that apply a criterion to prune a set of weights and retrain the remaining ones are solving (17) based on a guess at $\mathcal{Z}^*$, and their success depends on how good this guess is. In the LC algorithm, $\mathcal{Z}$ is reestimated at each C step, and the alternation of L and C steps allows it to explore the space of subsets $\mathcal{Z}$ while reducing the loss value, making it more likely that it finds a better net.

```
input $\kappa > 0$ (constraint form) or $\alpha > 0$ (penalty form),
      training data and neural net architecture with weights $\mathbf{w}$
$\mathbf{w} \leftarrow \overline{\mathbf{w}} = \arg\min_{\mathbf{w}} L(\mathbf{w})$                reference net
$\boldsymbol{\theta} \leftarrow \begin{cases} \mathbf{0}, & \text{penalty form} \\ \Pi^{\leq}_{C}(\mathbf{w};\kappa), & \text{constraint form} \end{cases}$     mark weights for pruning
$\boldsymbol{\lambda} \leftarrow \mathbf{0}$
for $\mu = \mu_0 < \mu_1 < \dots < \infty$
   $\mathbf{w} \leftarrow \arg\min_{\mathbf{w}} L(\mathbf{w}) + \frac{\mu}{2}\|\mathbf{w} - \boldsymbol{\theta} - \frac{1}{\mu}\boldsymbol{\lambda}\|^2$     L step: learn net
   $\boldsymbol{\theta} \leftarrow \begin{cases} \Pi^{+}_{C}(\mathbf{w} - \frac{1}{\mu}\boldsymbol{\lambda}; \frac{2\alpha}{\mu}), & \text{penalty form} \\ \Pi^{\leq}_{C}(\mathbf{w} - \frac{1}{\mu}\boldsymbol{\lambda}; \kappa), & \text{constraint form} \end{cases}$     C step: mark weights for pruning
   $\boldsymbol{\lambda} \leftarrow \boldsymbol{\lambda} - \mu(\mathbf{w} - \boldsymbol{\theta})$     Lagrange multipliers
   if $\|\mathbf{w} - \boldsymbol{\theta}\|$ is small enough then exit the loop
$\mathbf{w} \leftarrow \boldsymbol{\theta}$
Retrain nonzero elements of $\mathbf{w}$ to minimize $L(\mathbf{w})$
return $\mathbf{w}$
```

Figure 2: Pseudocode for the pruning LC algorithm, augmented-Lagrangian version.

### 7.3 Practicalities

Fig. 2 gives the pseudocode for the LC algorithm using the augmented-Lagrangian version (for the quadratic-penalty version, ignore or set to zero $\boldsymbol{\lambda}$ everywhere). The algorithm starts by minimizing $L(\mathbf{w})$ in order to start from a well-trained, reference model (although it is generally neither necessary not practical to minimize $L$ exactly). One can use any optimization algorithm, but with large datasets and deep nets the best choice will be SGD (or any of its variations). This same algorithm is used in the L step (where SGD should used a clipped learning rate schedule as described in section 4.1). In the C step, we use the exact pruning operators of sections 4.2 and 5.1. In the AL algorithm one can run multiple L and C steps before updating $\boldsymbol{\lambda}$, in effect better minimizing $\mathcal{L}_A(\mathbf{w}, \boldsymbol{\theta}, \boldsymbol{\lambda}; \mu)$ for fixed $\mu$ and $\boldsymbol{\lambda}$. But since the C step and the $\boldsymbol{\lambda}$ update are computationally negligible compared to the L step, it is simpler to do a single L and C step (as in the pseudocode) and increase $\mu$ more slowly.

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 $\mu_0$ is given in appendix C 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). The AL or QP minimization should generally be stopped when $\|\mathbf{w} - \boldsymbol{\theta}\|$ is smaller than a set tolerance, which will happen when $\mu$ is large enough. The retraining step is unnecessary in theory for the $\ell_0$ cost, but in practice with deep nets (which are notoriously hard to optimize accurately) running it will improve a bit the result. For the $\ell_1$ cost retraining is necessary and will significantly improve the result (and increase on average the weights' magnitude).

## 8 Experiments[3]

We evaluate our LC algorithm for pruning on classification neural nets of different sizes in the MNIST and CIFAR10 datasets and compare with magnitude-based pruning. 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 $\kappa$ or $\alpha$ and a single stage of pruning. The total runtime of our LC algorithm is roughly given by the number of L steps; in our experiments, we found it is to be no more than 1.5 times the runtime of the reference net.

We report results using the augmented Lagrangian version of the LC algorithm. We used the Theano (Theano Development Team, 2016) and Lasagne (Dieleman et al., 2015) libraries to train deep nets. We initialize all algorithms from a reasonably (but not necessarily perfectly) well-trained reference model. The

[3]Code and data are available from the authors.

initial LC iteration ($\mu = 0$, or more precisely $\mu \to 0^+$) for the constraint form gives the magnitude-based pruning solution (the "direct compression" of section 7.1). 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 $P$ (%) of surviving weights (i.e., not pruned), total and per layer.

When comparing different methods it is often desirable to fix the value of $P$ and then compare the values of the loss and error. However, this is cumbersome: while the hyperparameters $\kappa$ or $\alpha$ do determine the resulting $P$ value, they do so indirectly (except for the $\ell_0$-constraint form, where $\kappa$ equals the number of surviving weights). So achieving, say, $P = 5\%$ requires a search for the corresponding value of $\kappa$ (for $\ell_1$) or $\alpha$ (for both norms), which is computationally costly. To simplify this in our experiments we report results that scan the interesting range of the corresponding hyperparameter.

## 8.1 Classification on MNIST with LeNet300 and LeNet5

The LeNet models (LeCun et al., 1998) are a widely used benchmark that allows for comparison with published work. We randomly split the MNIST training dataset (60k grayscale images of $28 \times 28$, 10 digit classes) into training (90%) and validation (10%) sets. We normalize the pixel grayscales to [0,1] and then substract the mean.

The loss is the average cross-entropy. To train a good reference model, we use Nesterov's accelerated gradient method (Nesterov, 1983) with momentum 0.95 for 100k minibatches, with a carefully fine-tuned learning rate $0.02 \cdot 0.99^j$ 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 (LeNet300) or 4k (LeNet5) SGD iterations with momentum 0.95 and learning rate $0.1 \cdot 0.98^j$. We retrain the surviving weights as follows: for our LC algorithm, learning rate $0.005 \cdot 0.99^j$ running 2k iterations for every $j$, total 50k iterations, momentum 0.95; for magnitude-based pruning, learning rate $0.02 \cdot 0.98^j$ running 2k iterations for every $j$, total 100k iterations, momentum 0.95.

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 originally a 60k-parameter convolutional net. We use a larger variation from Caffe[4] for comparison with Han et al. (2015), with ReLU activations (Nair and Hinton, 2010) and softmax outputs, total 431k trainable parameters, and the following network structure:

| input | layer 1, convolutional | layer 2, convolutional | layer 3, dense | output, dense |
|---|---|---|---|---|
| $28 \times 28$ | 20@$\{5 \times 5\}$, stride=1<br>maxpool $\{2 \times 2\}$, stride=2 | 50@$\{5 \times 5\}$, stride=1<br>maxpool $\{2 \times 2\}$, stride=2 | 500 | 10 |

**Results** Firstly, table 1 (for LeNet300 with $\ell_0$) confirms that, in the constraint form, using a single, global $\kappa$ pruning parameter consistently beats using a separate $\kappa_i$ parameter per layer: the loss/error for both training and test is better with the global $\kappa$, particularly as the sparsity level increases. Note also how the amount of pruning per layer organizes differently: the first fully connected layer (which accounts for most of the weights) is pruned more with the global parameter than the local ones (for the same total number of pruned weights). In the rest of the paper we use always a single, global parameter ($\kappa$ for the constraint form $\alpha$ for the penalty form).

Fig. 3 shows the loss and error curves over LC iterations (where the initial net is the reference net and the final one corresponds to retraining the pruned net). The LC training loss need not decrease monotonically because the augmented Lagrangian minimizes eq. (6) or (12) for each $\mu$, not the actual loss (though it does approach a local optimum in the limit when $\mu \to \infty$).

Tables 2–3 report the results. Generally, the constraint form does better than the penalty form, and the $\ell_0$ cost does better than the $\ell_1$ cost, but not significantly. Retraining the pruned net has a large effect for the $\ell_1$ cost, as expected, because $\ell_1$ shrinks the surviving weights: the loss decreases and the weights' magnitude increases on average. Retraining has barely any effect for the $\ell_0$ cost, which does not shrink the weights.

We did not try to find the very best parameter settings (for the pruning cost $\kappa$ or $\alpha$, or for the SGD and LC optimization parameters), instead we sample what can be achieved. We can prune $\sim$ 98–99% of the weights with about the same loss/error as the reference. We can go beyond 99% with a minor degradation.

[4]https://github.com/BVLC/caffe/blob/master/examples/mnist/lenet.prototxt

| | | | before retraining | | | after retraining | | | |
|---|---|---|---|---|---|---|---|---|---|
| | $P\%$ | $\kappa$ | $\log L$ | $E_{\text{train}}$ | $E_{\text{test}}$ | $\log L$ | $E_{\text{train}}$ | $E_{\text{test}}$ | $\Delta\mathbf{w}\%$ |
| **R** | 100=[100 100 100] | | -3.87 | 0 | 2.28 | | | | |
| local | 10=[10 10 10] | 26620 | -4.07 | 0 | 2.14 | -4.10 | 0 | 2.15 | 0.20 |
| | 8=[8 8 8] | 21296 | -4.07 | 0 | 2.11 | -4.11 | 0 | 2.11 | 0.16 |
| | 5=[5 5 5] | 13310 | -3.94 | 0 | 2.30 | -3.98 | 0 | 2.27 | 0.12 |
| | 3=[3 3 3] | 7986 | -3.66 | 0 | 2.37 | -3.73 | 0 | 2.37 | 0.07 |
| | 2=[2 2 2] | 5324 | -2.85 | 0 | 3.00 | -3.12 | 0 | 2.85 | 0.08 |
| global | 10=[6.2 37 86] | 26620 | -4.28 | 0 | 2.19 | -4.31 | 0 | 2.17 | 0.15 |
| | 8=[4.8 31 84] | 21296 | -4.23 | 0 | 2.08 | -4.26 | 0 | 2.08 | 0.17 |
| | 5=[2.9 19 81] | 13310 | -4.16 | 0 | 2.14 | -4.20 | 0 | 2.14 | 0.12 |
| | 3=[1.6 11 76] | 7986 | -3.81 | 0 | 2.27 | -3.86 | 0 | 2.13 | 0.22 |
| | 2=[1.1 6.6 72] | 5324 | -3.15 | 0 | 2.46 | -3.38 | 0 | 2.45 | 0.50 |

Table 1: Global ($C(\boldsymbol{\theta}) \leq \kappa$) vs local ($C(\boldsymbol{\theta}_i) \leq \kappa$) pruning parameter in the $\ell_0$-constraint form, on LeNet300.

This outperforms nearly all published work we have seen: magnitude pruning done in stages in Han et al. (2015) achieves 92% (a little better than the single-stage magnitude pruning we show) and Wen et al. (2016) is much worse. Only Guo et al. (2016) 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 $\ell_0$ constraint form is trivial to set: $\kappa$ equals the number of surviving weights.*

As long as the sparsity level is not extreme, our pruned nets have a lower training and/or test error than the reference. One reason why this happens is that the reference was close but not equal to a local optimum, due to the long training times required by SGD-type algorithms. Since the pruned nets keep (re)training, they gain some accuracy over the reference.

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. 4 shows the weight vectors $\boldsymbol{\theta}$ 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 $\ell_0/\ell_1$ and constraint/penalty). As the LC algorithm iterates, $\boldsymbol{\theta}$ marks weights for pruning and $\mathbf{w}$ approaches $\boldsymbol{\theta}$ until $\mathbf{w} = \boldsymbol{\theta}$ upon convergence. Each weight vector can be shown as a $28 \times 28$ 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, as predicted by our theoretical analysis. After that, different weights move in and out of the marked subset. The evolution of weights and neurons can be seen dramatically in a supplementary animation, in particular how for $\ell_1$ 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 $\ell_1$ cost changes weights more than the $\ell_0$ 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 $\approx 120$ weights); see fig. 5(row 1). 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). Indeed, the original LeNet300 architecture 784–300–100–10 becomes 400–64–99–10 with similar or even better loss (for the $\ell_1$-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

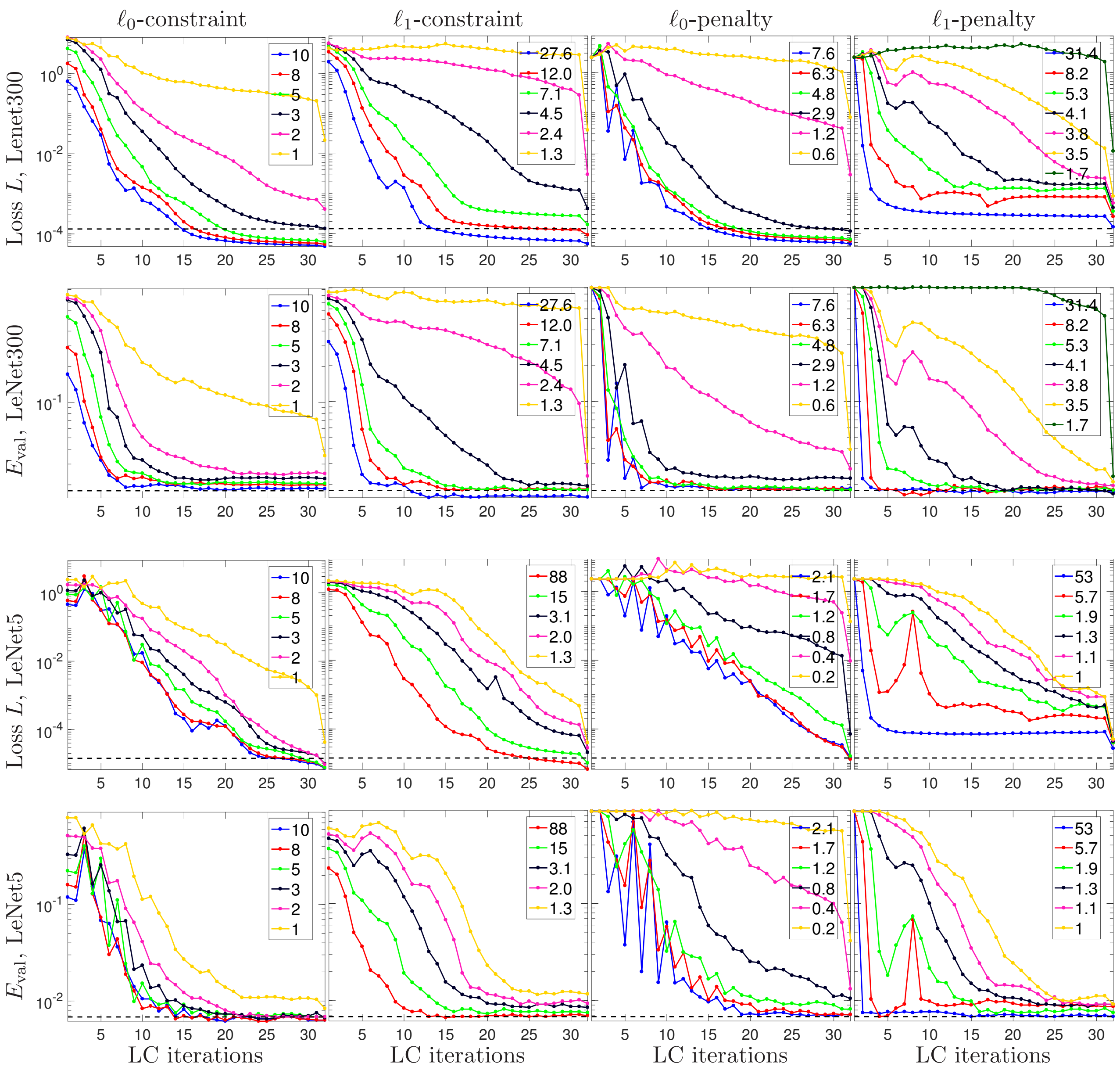


Figure 3: Learning curves (training loss and test error) for different problem forms and pruning costs, for LeNet300 (top 3 rows) and LeNet5 (bottom 2 rows). Each curve corresponds to a sparsity level (proportion $P\%$ of surviving weights). The black dashed line is the reference net. The last (32nd) iteration corresponds to retraining.

disappear first, followed by the output ones. $\ell_0$ 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 $\ell_1$, no such neurons remain. This is visible in the green curves in fig. 5(row 1), corresponding to the fan-in and fan-out of layer 1: for $\ell_1$ they both go down (fan-in first, then fan-out) and join upon convergence, for $\ell_0$ this happens partially. However, this is not a problem with $\ell_0$: such neurons can be safely removed in a postprocessing step.

Fig. 5 rows 2–3 show 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 $\ell_0$ the gap between the last two is much wider than for $\ell_1$.

| | | $\kappa$ or | before retraining | | | after retraining | | | |
|---|---|---|---|---|---|---|---|---|---|
| | $P\%$ | $\log\alpha$ | $\log L$ | $E_{\text{train}}$ | $E_{\text{test}}$ | $\log L$ | $E_{\text{train}}$ | $E_{\text{test}}$ | $\Delta\mathbf{w}\%$ |
| **R** | 100=[100 100 100] | | -3.87 | 0 | 2.28 | | | | |
| $\ell_0$-constraint | 10=[6.2 37 86] | 26620 | -4.28 | 0 | 2.19 | -4.31 | 0 | 2.17 | 0.15 |
| | 8=[4.8 31 84] | 21296 | -4.23 | 0 | 2.08 | -4.26 | 0 | 2.08 | 0.17 |
| | 5=[2.9 19 81] | 13310 | -4.16 | 0 | 2.14 | -4.20 | 0 | 2.14 | 0.12 |
| | 3=[1.6 11 76] | 7986 | -3.81 | 0 | 2.27 | -3.86 | 0 | 2.13 | 0.22 |
| | 2=[1.1 6.6 72] | 5324 | -3.15 | 0 | 2.46 | -3.38 | 0 | 2.45 | 0.50 |
| | 1=[0.5 2.7 65] | 2662 | -0.68 | 5.64 | 6.90 | -1.66 | 0.59 | 3.17 | -1.99 |
| $\ell_1$-constraint | 27.6=[22 63 96] | 3500 | -4.18 | 0 | 1.94 | -4.25 | 0 | 1.96 | 1.18 |
| | 12.0=[9.8 26 83] | 2500 | -3.90 | 0 | 2.03 | -4.03 | 0 | 1.84 | 2.15 |
| | 7.1=[5.9 15 70] | 2000 | -3.56 | 0 | 1.93 | -3.77 | 0 | 1.93 | 4.06 |
| | 4.5=[3.9 8.2 49] | 1500 | -2.92 | 0 | 1.86 | -3.38 | 0 | 1.89 | 10.00 |
| | 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 |
| $\ell_0$-penalty | 7.6=[4.9 27 85] | -6.30 | -4.22 | 0 | 2.00 | -4.26 | 0 | 1.99 | 0.23 |
| | 6.3=[4.1 21 83] | -6.15 | -4.15 | 0 | 1.95 | -4.18 | 0 | 2.00 | 0.26 |
| | 4.8=[3.1 15 80] | -6.00 | -4.11 | 0 | 2.10 | -4.14 | 0 | 2.10 | 0.25 |
| | 2.9=[1.8 8.2 72] | -5.69 | -3.89 | 0 | 2.26 | -3.94 | 0 | 2.23 | 0.26 |
| | 1.2=[0.8 3.4 58] | -5.30 | -1.39 | 1.34 | 3.63 | -2.54 | 0 | 2.89 | 1.57 |
| | 0.6=[0.3 1.7 44] | -5.00 | 0.01 | 25.12 | 23.82 | -1.12 | 2.42 | 3.77 | -4.22 |
| $\ell_1$-penalty | 31.4=[26 68 95] | -6.00 | -3.57 | 0 | 2.02 | -3.83 | 0 | 2.00 | 3.52 |
| | 8.2=[6.6 19 77] | -5.30 | -3.09 | 0 | 2.11 | -3.57 | 0 | 2.00 | 8.39 |
| | 5.3=[4.3 10 63] | -5.00 | -2.87 | 0 | 1.86 | -3.45 | 0 | 1.94 | 10.88 |
| | 4.1=[3.5 7.7 51] | -4.82 | -2.76 | 0 | 1.90 | -3.35 | 0 | 1.90 | 12.18 |
| | 3.8=[3.3 6.2 43] | -4.69 | -2.63 | 0 | 2.03 | -3.24 | 0 | 2.02 | 13.23 |
| | 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 |
| magnitude pruning | 31.4=[27 67 91] | | -2.11 | 0.22 | 2.82 | -3.62 | 0 | 2.32 | 1.95 |
| | 27.6=[23 59 90] | | -1.92 | 0.38 | 2.99 | -3.99 | 0 | 2.23 | 5.19 |
| | 10=[5.7 40 87] | | -0.19 | 16.11 | 16.13 | -3.67 | 0 | 2.26 | 23.21 |
| | 8=[4.1 36 85] | | 0.24 | 28.64 | 28.68 | -3.57 | 0 | 2.33 | 31.00 |
| | 5=[1.9 26 82] | | 0.62 | 53.02 | 52.26 | -3.25 | 0 | 2.30 | 51.33 |
| | 3=[0.8 17 80] | | 0.84 | 72.09 | 72.09 | -2.72 | 0 | 2.81 | 84.13 |
| | 2=[0.4 12 78] | | 0.90 | 74.78 | 74.29 | -2.01 | 0.44 | 3.41 | 112.37 |
| | 1=[0.1 5.5 74] | | 0.88 | 80.91 | 80.97 | -0.95 | 3.40 | 4.58 | 93.00 |

Table 2: LeNet300 results for the reference net, and for pruning (before and after retraining the surviving weights): magnitude-based and our LC algorithm ($\ell_0$ and $\ell_1$ cost, $\kappa$-constraint and $\alpha$-penalty forms). We report: proportion of surviving weights $P$ (%) in total and per-layer; training loss $\log L$ and training and test classification error $E_{\text{train}}$ and $E_{\text{test}}$ (%); and the average relative increase in weight magnitude after retraining $\Delta\mathbf{w} = \|\mathbf{w}_{\text{after}}\|/\|\mathbf{w}_{\text{before}}\| - 1$ (as %). All logarithms are base 10.

## 8.2 Classification on CIFAR10 with ResNets

The ResNet models (He et al., 2016) 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 (Li et al., 2017).

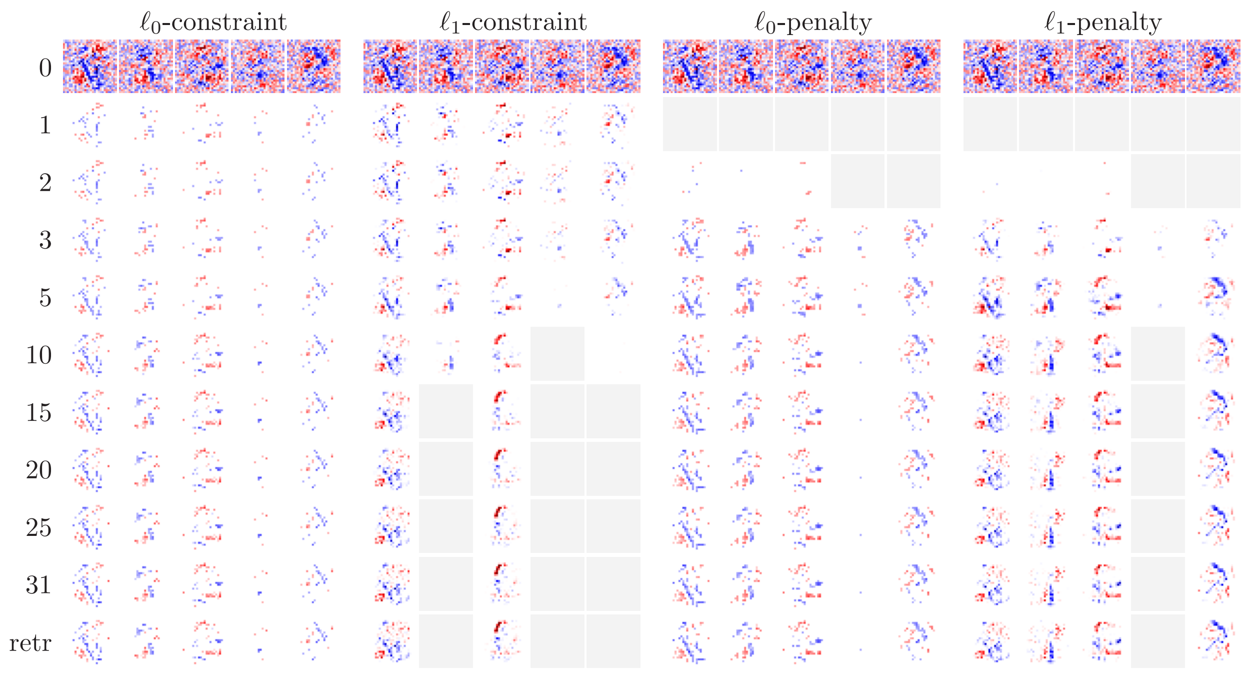
We train ResNets of depth 32, 56 and 110 layers (0.46M, 0.85M and 1.7M parameters, respectively) on the CIFAR10 dataset using the same setup as in He et al. (2016). We randomly split the dataset (50k

|  |  | $\kappa$ or | before retraining |  |  | after retraining |  |  |  |
|---|---|---|---|---|---|---|---|---|---|
|  | $P\%$ | $\log\alpha$ | $\log L$ | $E_{\text{train}}$ | $E_{\text{test}}$ | $\log L$ | $E_{\text{train}}$ | $E_{\text{test}}$ | $\Delta\mathbf{w}\%$ |
| **R** | 100=[100 100 100] |  | -4.84 | 0 | 0.80 |  |  |  |  |
| $\ell_0$-constraint | 10=[84 24 8.4 59] | 43050 | -5.01 | 0 | 0.86 | -5.13 | 0 | 0.82 | 0.41 |
|  | 8=[83 22 6.4 56] | 34440 | -4.98 | 0 | 0.84 | -5.12 | 0 | 0.86 | 0.42 |
|  | 5=[81 17 3.6 49] | 21525 | -4.96 | 0 | 0.85 | -5.11 | 0 | 0.85 | 0.41 |
|  | 3=[78 12 1.9 40] | 12915 | -4.76 | 0 | 0.85 | -5.01 | 0 | 0.82 | 0.53 |
|  | 2=[74 8.5 1.1 32] | 8610 | -4.75 | 0 | 0.86 | -5.02 | 0 | 0.88 | 0.40 |
|  | 1=[69 4.3 0.4 23] | 4305 | -2.99 | 0.018 | 1.53 | -2.99 | 0 | 1.06 | 1.58 |
| $\ell_1$-constraint | 88=[99 93 87 95] | 1500 | -5.00 | 0 | 0.92 | -5.17 | 0 | 0.84 | 1.94 |
|  | 15=[80 28 14 37] | 1000 | -4.73 | 0 | 1.02 | -4.99 | 0 | 0.96 | 3.35 |
|  | 3.1=[58 9.1 2.4 18] | 700 | -4.19 | 0 | 1.04 | -4.68 | 0 | 1.04 | 5.32 |
|  | 2.0=[48 6.0 1.6 14] | 600 | -3.87 | 0 | 1.14 | -4.55 | 0 | 1.12 | 4.42 |
|  | 1.3=[38 3.7 1.0 9.7] | 500 | -3.32 | 0 | 1.28 | -4.40 | 0 | 1.18 | 10.8 |
| $\ell_0$-penalty | 2.1=[78 9.5 1.1 36] | -6.30 | -4.47 | 0 | 0.94 | -4.83 | 0 | 0.90 | 0.65 |
|  | 1.7=[74 7.4 0.8 31] | -6.15 | -4.52 | 0 | 0.89 | -4.88 | 0 | 0.89 | 0.52 |
|  | 1.2=[70 5.4 0.6 24] | -6.00 | -3.89 | 0 | 0.94 | -4.79 | 0 | 0.94 | 0.62 |
|  | 0.8=[63 3.1 0.3 16.2] | -5.69 | -1.87 | 0.4 | 1.32 | -4.15 | 0 | 1.10 | 2.52 |
|  | 0.4=[48 1.6 0.1 6.9] | -5.30 | -0.62 | 5.84 | 6.38 | -2.02 | 0.25 | 1.64 | 7.67 |
|  | 0.2=[34 1.1 0.02 3.3] | -5.00 | 0.41 | 55.8 | 56.93 | -0.88 | 3.92 | 4.29 | -4.08 |
| $\ell_1$-penalty | 53.7=[96 61 52 82] | -6.00 | -4.08 | 0 | 0.89 | -4.56 | 0 | 0.89 | 1.79 |
|  | 5.7=[82 33 3.3 41] | -5.30 | -3.69 | 0 | 0.89 | -4.40 | 0 | 1.10 | 4.36 |
|  | 1.9=[63 6.7 1.3 17] | -5.00 | -3.36 | 0 | 1.20 | -4.40 | 0 | 1.05 | 11.43 |
|  | 1.3=[48 4.6 0.9 11] | -4.82 | -3.31 | 0 | 1.12 | -4.36 | 0 | 1.09 | 11.74 |
|  | 1.1=[40 2.4 0.8 9.1] | -4.60 | -3.07 | 0 | 1.16 | -4.32 | 0 | 1.23 | 13.76 |
|  | 1=[34 2.6 0.7 8.0] | -4.69 | -3.06 | 0 | 1.36 | -4.29 | 0 | 1.21 | 14.40 |
| magnitude pruning | 88=[98 88 88 94] |  | -4.75 | 0 | 0.86 | -5.42 | 0 | 0.86 | 0.38 |
|  | 53.7=[91 54 53 78] |  | -2.29 | 0.17 | 1.14 | -5.36 | 0 | 0.85 | 1.20 |
|  | 15=[83 21 14 60] |  | -0.28 | 11.85 | 11.40 | -4.97 | 0 | 0.96 | 6.91 |
|  | 10=[82 18 8.7 57] |  | -0.34 | 12.01 | 11.43 | -4.90 | 0 | 0.99 | 11.30 |
|  | 8=[81 17 6.7 56] |  | -0.23 | 15.97 | 14.92 | -4.87 | 0 | 1.02 | 14.86 |
|  | 5=[80 14 3.7 53] |  | -0.04 | 21.37 | 20.03 | -4.79 | 0 | 0.98 | 26.10 |
|  | 3=[78 12 1.7 50] |  | 0.06 | 32.74 | 31.66 | -4.69 | 0 | 1.05 | 45.61 |
|  | 2=[76 9.7 0.9 47] |  | 0.22 | 50.75 | 50.49 | -4.57 | 0 | 1.05 | 68.21 |
|  | 1=[72 5.9 0.1 39] |  | 0.37 | 78.12 | 78.35 | -3.95 | 0 | 1.57 | 124.43 |

Table 3: LeNet5 results, as in table 2.

RGB images of $32 \times 32$, 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 $32 \times 32$ image). For test we use the original images without augmentation. We select the net having smallest validation error during training. The loss is the cross-entropy.

We train reference nets, and nets compressed in two ways: with our LC algorithm ($\ell_0$-constraint version), and with magnitude-based pruning, both followed by retraining the surviving weights. The reference nets are trained with SGD with momentum of 0.9 on minibatches of size 128. The loss is the average cross entropy with weight decay of $10^{-4}$. The weights are initialized as in He et al. (2015). The network is trained for 60k minibatch iterations with an initial learning rate of 0.1, which is divided by 10 after 32k and 48k iterations. The LC algorithm is run for 35 LC iterations, with $\mu = 10^{-3} \times 1.1^k$ at the $k$th iteration. Each L step is performed by Nesterov's accelerated gradient method (Nesterov, 1983) with momentum 0.95 and run for 2k minibatches, except for first L step, which is run for 10k minibatches, to ensure we start close to the path



| problem | $\kappa$ or $\alpha$ | $P\%$ | $L \times 10^{-5}$ | $E_{\text{train}}$, % | $E_{\text{test}}$, % |
|---|---|---|---|---|---|
| reference | | 100=[100 100 100] | 13.22 | 0 | 2.28 |
| $\ell_0$-constraint | 13310 | 5.0=[5 5 5] | 6.29 | 0 | 2.14 |
| $\ell_1$-constraint | 1500 | 4.6=[4 9 49] | 41.41 | 0 | 1.89 |
| $\ell_0$-penalty | $10^{-6}$ | 4.8=[4 6 81] | 7.11 | 0 | 2.10 |
| $\ell_1$-penalty | $1.5 \cdot 10^{-5}$ | 4.2=[4 8 51] | 43.69 | 0 | 1.90 |

Figure 4: Weight vector of selected first-layer neurons over iterations (0 = reference, retr = after retraining) for the nets described in the table (having $P \approx 5\%$), for LeNet300.

over $\mu$. For each L step, the learning rate decays exponentially from 0.01 to 0.001 and is adjusted after every minibatch. Retraining is performed with Nesterov's SGD with momentum 0.95 for 60k minibatches, with exponential learning rate decay from $5 \times 10^{-3}$ to $10^{-4}$. For magnitude-based pruning, retraining is performed using Nesterov's SGD with momentum 0.95, and run for 60k minibatches, with exponential learning rate decay from 0.01 to $5 \times 10^{-4}$.

Fig. 6 shows the results. With the LC algorithm we are able to achieve considerable pruning of even $P = 3\%$ surviving weights with a better training loss than the reference (in nearly in all cases), and consistently and significantly better than using magnitude-based pruning. This shows that the LC algorithm is indeed effective at finding a compressed model with a low loss. The test errors tell a similar story, but they can overfit a bit compared to the reference net if going below $P = 15\%$; this could be controlled by early stopping. Compared to work in the literature that prunes ResNets, we found one published comparison point: Li et al. (2017) remove filters from convolutional layers for ResNet56/110 and achieve $P = 67.6\%$ for ResNet110 (error 6.70%) and $P = 86.3\%$ ResNet56 (error 6.94%). Our LC algorithm achieves a much stronger pruning with the same error: $P = 10\%$ for ResNet110 (error 6.70%) and ResNet56 (error 6.77%).

## 9 Discussion

### 9.1 Comparison with previous pruning approaches

Most pruning approaches are based on the idea of permanently removing a subset of weights ("irrevocable pruning") based on some criterion that measures the importance of each weight (such as magnitude or

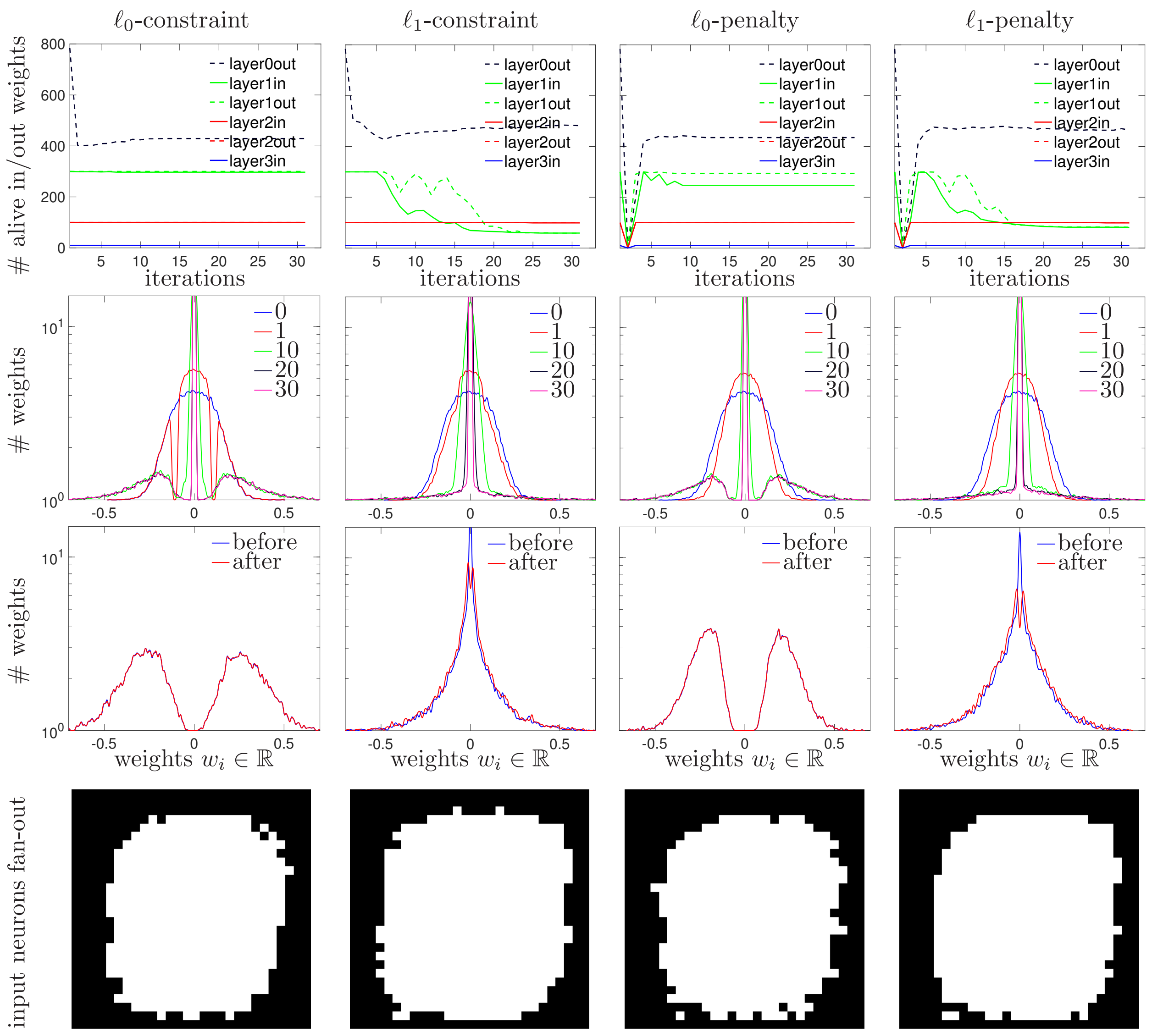


Figure 5: LeNet300 connectivity statistics. *Row 1*: number of neurons having nonzero weights (at their input or output, as indicated, for each layer). *Row 2*: distribution of weights for selected iterations for layer 1. *Row 3*: distribution of weights just before and after retraining for layer 1. *Row 4*: input layer neurons (pixels of MNIST image) having no output weights (black) vs having at least one output weight (white).

curvature), and then retraining 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 (especially, 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 $\ell_0$ and $\ell_1$. 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

| | | | LC algorithm | | | | | | | mag.-based pruning | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | $\kappa$ (= # | before retraining | | | after retraining | | | | before retr. | after retr. |
| | $P$% | weights) | $\log L$ | $E_{\text{train}}$ | $E_{\text{test}}$ | $\log L$ | $E_{\text{train}}$ | $E_{\text{test}}$ | $\Delta\mathbf{w}$% | $E_{\text{test}}$ | $E_{\text{test}}$ |
| 32-layers | **R** | 0.46M | -0.76 | 0.02 | 7.72 | | | | | | |
| | 15 | 69281 | -0.90 | 0.42 | 7.61 | -1.03 | 0.05 | 7.63 | -13.55 | 86.68 | 8.85 |
| | 10 | 46188 | -0.76 | 2.39 | 8.35 | -0.97 | 0.40 | 8.16 | -13.56 | 89.85 | 9.41 |
| | 5 | 23094 | -0.61 | 5.32 | 9.93 | -0.78 | 2.89 | 9.18 | -13.41 | 88.78 | 12.07 |
| | 3 | 13857 | -0.37 | 11.08 | 14.07 | -0.62 | 5.80 | 10.10 | -13.07 | 90.00 | 14.35 |
| 56-layers | **R** | 0.85M | -0.75 | 0.02 | 6.86 | | | | | | |
| | 15 | 127 342 | -0.97 | 0.06 | 6.94 | -1.09 | 0 | 6.70 | -13.76 | 67.06 | 7.39 |
| | 10 | 84 895 | -0.93 | 0.21 | 6.77 | -1.06 | 0.01 | 6.77 | -13.74 | 83.41 | 8.66 |
| | 5 | 42 448 | -0.76 | 2.34 | 8.37 | -0.97 | 0.48 | 7.76 | -13.80 | 89.06 | 9.48 |
| | 3 | 25 469 | -0.47 | 8.41 | 10.29 | -0.81 | 2.32 | 8.74 | -13.77 | 89.07 | 11.50 |
| 110-layers | **R** | 1.7M | -0.80 | 0 | 6.70 | | | | | | |
| | 15 | 257 979 | -1.00 | 0.01 | 6.58 | -1.12 | 0 | 6.62 | -13.79 | 32.50 | 7.22 |
| | 10 | 171 986 | -0.98 | 0.03 | 6.56 | -1.09 | 0 | 6.70 | -13.77 | 68.46 | 7.20 |
| | 5 | 85 993 | -0.91 | 0.35 | 7.55 | -1.05 | 0.02 | 7.17 | -13.80 | 75.63 | 8.23 |
| | 3 | 51 596 | -0.71 | 2.97 | 8.58 | -0.99 | 0.27 | 7.67 | -13.83 | 89.82 | 9.32 |

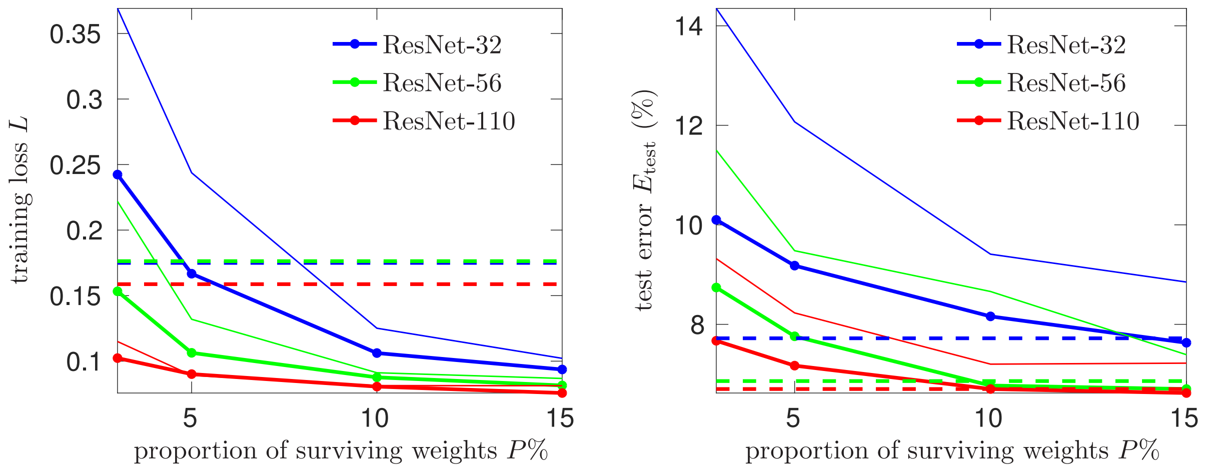


Figure 6: ResNet results for the reference net **R**, and for pruning (before and after retraining the surviving weights): magnitude-based pruning and our LC algorithm ($\ell_0$-constraint form). *Top*: table with detailed results. We report: proportion of surviving weights $P$ (%); $\kappa$ hyperparameter for LC (equal to the number of surviving weights); training loss $\log L$ and training and test classification error $E_{\text{train}}$ and $E_{\text{test}}$ (%); and the average relative increase in weight magnitude after retraining $\Delta\mathbf{w} = \|\mathbf{w}_{\text{after}}\|/\|\mathbf{w}_{\text{before}}\| - 1$ (as %). All logarithms are base 10. The last column is for the magnitude-based pruning test classification error, to be compared with the LC algorithm. *Bottom*: tradeoff curves of training loss error and test error vs sparsity. Thick lines: LC algorithm, thin lines: magnitude-based pruning, horizontal dashed lines: reference nets.

irrevocably to any set ("tentative pruning"). All the weights are there throughout training, but some are marked as currently pruned (zeros in $\boldsymbol{\theta}$). Compared to irrevocable pruning, this helps find a better subset and hence prune more weights with no or little loss degradation. Furthermore, the constraint form with a global pruning parameter can automatically learn the number of pruned weights for each layer. This further improves the solution and simplifies the parameter search by the user.

### 9.2 What version of the LC algorithm is best?

We have analyzed in theory and experiment several options: form of optimization problem (constraint, penalty); choice of pruning cost ($\ell_0$, $\ell_1$); global or local pruning parameter; QP or AL optimization algorithm. *We recommend using the $\ell_0$ or $\ell_1$ constraint form with a global $\kappa$ pruning parameter.* Below we summarize our reasons.

**Optimization problem: constraint or penalty** Both forms achieve similarly pruned nets. They both start by marking the smaller weights for pruning. The L step is identical. The C step is a thresholding for both, but the value of the threshold differs: for the penalty form it is constant (independent of the current weights) while for the constraint form it is adaptive (roughly speaking, it picks the top-$\kappa$ weights). Computationally, the constraint form is slightly more costly, although the difference this makes is negligible compared to the runtime of the L step. The solutions achieved by scanning each form's parameter ($\kappa$ or $\alpha$) over its range are similar. The main difference is in user friendliness: to achieve a desired sparsity level (say, 95% of the weights pruned), we can set $\kappa = 0.05$ times the total number of weights in the constraint form with $\ell_0$, but we require a search over $\alpha$ in the penalty form or over $\kappa$ in the constraint form for $\ell_1$. However, best compression of a net requires trying different sparsity levels anyway, so this is not crucial.

**Pruning cost: $\ell_0$ or $\ell_1$** The cost we would really like to optimize is $\ell_0$, since it gives the best subset of pruned weights, but it is a difficult NP-complete problem and prone to bad local optima. Optimizing $\ell_1$ (followed by retraining unpruned weights to unshrink them) is not guaranteed to find the $\ell_0$ optimum but it does sparsify and is easier to optimize (if $L(\mathbf{w})$ was convex the pruning problem would also be convex). Setting $\kappa$ is easier for $\ell_0$ than for $\ell_1$ (see above). Experimentally we find both costs give comparably good solutions: $\ell_0$ gives a lower training loss and $\ell_1$ gives a lower test error, but the difference may not be significant.

**Pruning parameter: global or local** By using a single pruning parameter $\kappa$ in the constraint form and scanning its domain $\kappa \geq 0$ we can learn automatically the best pruning level for each layer of the net, without the user having to do an exponentially costly search over per-layer parameters. In the penalty form, good results can be achieved with a single parameter $\alpha$, but best results do require per-layer parameters and hence an exponential search.

**Optimization: quadratic-penalty or augmented-Lagrangian** We find AL to be much better than QP, in agreement with past optimization work. It adds nearly no overhead to the LC algorithm but is more effective in moving closer to the solution for the same $\mu$ schedule and hence converging faster. It is also more robust in setting the SGD hyperparameters (learning rate, momentum, minibatch size).

## 10 Conclusion

We have revisited weight pruning, an old problem in neural net learning, and formulated it from a constrained optimization point of view. Pruning weights is effected via a cost function that either constrains some subset of weights to be zero or penalizes weights that are nonzero. We have given a "learning-compression" (LC) algorithm that can solve that problem (in the sense of finding a local optimum). The LC algorithm alternates an SGD-based learning step over the weights that optimizes the loss while pulling the weights towards the current pruning markup; and a thresholding-based compression step that updates the pruning markup. We have given detailed derivations for a few cost functions, including $\ell_0$ and $\ell_1$ norms, but others are possible and our general treatment should apply.

In a sense, our algorithm vindicates magnitude as a measure of the relevance of a weight (as opposed to other measures, such as curvature of the loss locally). This is because the C step takes the form of a thresholding (i.e., pruning all but the largest weights) if using sparsifying costs such as $\ell_0$ and $\ell_1$. The weights initially marked for pruning thus coincide with magnitude-based pruning. Crucially, however, the LC algorithm handles pruning in a tentative, iterative way. It explores different weight sets in search of a good set rather than committing greedily to the largest weights in the reference model.

The LC algorithm 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. So the LC algorithm scales to deep nets as large as desired, as long as one can train the reference net. An important advantage of the algorithm is that it automatically determines the best number of weights to prune in each layer, even though one needs to set the value of just a single hyperparameter, the pruning parameter. This avoids an exponentially costly search over per-layer pruning parameters and vastly simplifies the network designer's job.

Although the algorithm is designed to prune weights, not neurons, a neuron is effectively pruned when its input and output weights are pruned. This can result in many neurons being pruned (unlike with magnitude-based pruning, which generally prunes no neurons). Hence, the LC algorithm may be useful to do feature selection and determine the optimal number of neurons in each layer automatically, to some extent learning the neural net architecture itself. Related to this, although we have focused on model compression in this paper, the LC algorithm may be generally used during training to achieve good generalization and a small net.

## A Norms

In this paper, following a usage that has become common, we refer to the $\ell_p$ function $\|\mathbf{x}\|_p$ as a norm, in particular to the $\ell_0$ norm $\|\mathbf{x}\|_0$. This is an abuse of notation. We clarify it here.

A norm over the real field is defined as follows (Horn and Johnson, 2013). Let $V$ be a vector space over the field $\mathbb{R}$. A function $\|\cdot\|\colon V \rightarrow \mathbb{R}$ is a norm if, for all $\mathbf{x}, \mathbf{y} \in V$ and all $c \in \mathbb{R}$,

(1) $\|\mathbf{x}\| \geq 0$ (nonnegativity)

(1a) $\|\mathbf{x}\| = 0$ if and only if $\mathbf{x} = \mathbf{0}$ (positivity)

(2) $\|c\,\mathbf{x}\| = |c|\|\mathbf{x}\|$ (homogeneity)

(3) $\|\mathbf{x} + \mathbf{y}\| \leq \|\mathbf{x}\| + \|\mathbf{y}\|$ (triangle inequality)

Consider $V \subset \mathbb{R}^n$. The function $\|\mathbf{x}\|_p = (x_1^p + \dots + x_n^p)^{\frac{1}{p}}$ where $p \geq 0$ is usually called "$\ell_p$ norm". However, while $\ell_p$ is a norm for $p \geq 1$, it is not for $p < 1$ because it violates the triangle inequality.

Consider the cardinality function of a vector $\mathbf{x} \in \mathbb{R}^n$, i.e., the number of nonzero elements of $\mathbf{x}$. This is usually called "$\ell_0$ norm" and written "$\|\mathbf{x}\|_0$". However, the limit $p \rightarrow 0$ of the $\ell_p$ function does not exist, while the limit $p \rightarrow 0$ of the $\ell_p^p$ function does exist and equals the above cardinality function; so it would be more correct to write it as "$\|\mathbf{x}\|_0^0$". Also, the cardinality function is not a norm, because it violates the homogeneity property.

## B C step solution: proofs

The problems of projecting a point in a ball or solving a proximal operator are well known for the case of $\ell_p$ norms. In the paper, we gave detailed results about the cases $\ell_0$, $\ell_1$ and $\ell_2^2$. For reference, we give short proofs or point to the literature for these results. The vector pruning cost function $C(\mathbf{w})$ and scalar pruning cost function $c(w)$ are defined as in section 3.

### B.1 Constraint form

Consider the optimization problem with $\mathbf{w}, \boldsymbol{\theta} \in \mathbb{R}^n$ and $\kappa > 0$:

$$\Pi_C^{\leq}(\mathbf{w};\kappa) \quad = \quad \arg\min_{\boldsymbol{\theta}} \ \|\mathbf{w} - \boldsymbol{\theta}\|_2^2 \quad \text{s.t.} \quad C(\boldsymbol{\theta}) \leq \kappa. \tag{19}$$

The solution for the $\ell_0$, $\ell_1$ and $\ell_2^2$ cases is as follows.

**Theorem B.1** ($\ell_0$ constraint)**.** *Let $C(\boldsymbol{\theta}) = \|\boldsymbol{\theta}\|_0$. Assume $\mathbf{w}$ has distinct elements in magnitude and $\kappa$ is integer. Then*

$$\Pi_C^{\leq}(\mathbf{w};\kappa) = \begin{cases} \mathbf{w}, & C(\mathbf{w}) \leq \kappa \\ \max\left(|w_i| - \eta, 0\right) \cdot \operatorname{sgn}\left(w_i\right), \;\; i = 1, \dots, n \textit{ (elementwise)}, & \textit{otherwise} \end{cases} \tag{20}$$

*where the threshold $\eta$ equals the $(\kappa+1)$-th largest weight in $\mathbf{w}$ (in magnitude).*

*Proof.* If $C(\boldsymbol{\theta}) \leq \kappa$ then $\boldsymbol{\theta}$ contains at most $\kappa$ nonzeros. Since $\|\mathbf{w} - \boldsymbol{\theta}\|_2^2 = \sum_{i=1}^n (w_i - \theta_i)^2$, then the optimal set of nonzeros corresponds to the top-$\kappa$ elements of $\mathbf{w}$ in magnitude. Hence, we set $\theta_i = w_i$ for the top-$\kappa$ elements of $\mathbf{w}$ and $\theta_i = 0$ for the rest, from which the solution follows. □

*Remark* B.2. If $\kappa$ is not integer, then the solution is the same as for $\lfloor \kappa \rfloor$. The assumption that $\mathbf{w}$ has distinct elements in magnitude is not critical. If there are multiple elements of $\mathbf{w}$ with magnitude equal to the $(\kappa+1)$-th largest magnitude and picking them all in the solution contains $\kappa_1 > \kappa$ nonzeros, then there are multiple ties, all with the same optimal value. Leaving out any subset of $\kappa_1 - \kappa$ of those elements is a global optimum.

**Theorem B.3** ($\ell_1$ constraint)**.** *Let $C(\boldsymbol{\theta}) = \|\boldsymbol{\theta}\|_1$. Then*

$$\Pi_C^{\leq}(\mathbf{w};\kappa) = \begin{cases} \mathbf{w}, & C(\mathbf{w}) \leq \kappa \\ \max\left(|w_i| - \eta, 0\right) \cdot \operatorname{sgn}\left(w_i\right), \;\; i = 1, \dots, n \textit{ (elementwise)}, & \textit{otherwise} \end{cases} \tag{21}$$

*where the threshold $\eta$ is obtained as follows:*

1. *Sort the elementwise magnitudes of $\mathbf{w}$ into $\mathbf{u}$: $u_1 \geq u_2 \geq \cdots \geq u_n$.*
2. *Call $\eta_i = \frac{1}{i}\left(\sum_{j=1}^i u_i - \kappa\right)$ and find $k = \arg\max_{1 \leq i \leq n} i$ s.t. $\eta_i < u_i$. Set $\eta = \eta_k$.*

*Proof.* See Condat (2016) and references therein, which also describe faster ways to solve the problem for large dimension $n$. □

*Remark* B.4. In theorem B.3, $k$ is the number of nonzeros in the solution, so one can equivalently use a threshold $\eta = u_{k+1} = (k+1)$-th largest weight in $\mathbf{w}$ (in magnitude).

**Theorem B.5** ($\ell_2^2$ constraint)**.** *Let $C(\boldsymbol{\theta}) = \|\boldsymbol{\theta}\|_2^2$. Then*

$$\Pi_C^{\leq}(\mathbf{w};\kappa) = \begin{cases} \mathbf{w}, & C(\mathbf{w}) \leq \kappa \\ \sqrt{\kappa}\,\mathbf{w}/\|\mathbf{w}\|_2, & \textit{otherwise.} \end{cases} \tag{22}$$

*Proof.* Trivial. □

## B.2 Penalty form

Consider the scalar optimization problem with $w, \theta \in \mathbb{R}$ and $\alpha, \mu > 0$:

$$\Pi_c^+\left(w; \tfrac{2\alpha}{\mu}\right) = \arg\min_\theta F(\theta; w) \quad \text{with} \quad F(\theta; w) = (w - \theta)^2 + \tfrac{2\alpha}{\mu} c(\theta). \tag{23}$$

The solution for the $\ell_0$, $\ell_1$ and $\ell_2^2$ cases is as follows. We give expressions using two-branch functions which are equivalent to the expressions using the indicator function in fig. 1.

**Theorem B.6** ($\ell_0$ penalty)**.** *Let $c(\theta) = \|\theta\|_0$. Then* $\Pi_c^+\left(w; \frac{2\alpha}{\mu}\right) = \begin{cases} 0, & w^2 \leq \frac{2\alpha}{\mu} \\ w, & \textit{otherwise.} \end{cases}$

*Proof.* Since $c(\theta) = \|\theta\|_0 = 0$ if $\theta = 0$ and 1 otherwise, we have two options: choose $\theta = 0$, with $F(\theta; w) = w^2$, or choose $\theta = w \neq 0$, with $F(\theta; w) = \frac{2\alpha}{\mu}$. The smaller of these two gives the solution. □

**Theorem B.7** ($\ell_1$ penalty)**.** *Let* $c(\theta) = \|\theta\|_1$. *Then* $\Pi^+_c\big(w; \frac{2\alpha}{\mu}\big) = \begin{cases} 0, & |w| \le \frac{\alpha}{\mu} \\ w - \operatorname{sgn}(w)\,\frac{\alpha}{\mu}, & \textit{otherwise.} \end{cases}$

*Proof.* Taking the derivative of $F$ wrt $\theta$ and equating to zero we obtain:

$$0 = \frac{\partial F}{\partial \theta} = \begin{cases} 2(\theta - w + \frac{\alpha}{\mu}), & \theta > 0 \\ 2(\theta - w - \frac{\alpha}{\mu}), & \theta < 0 \end{cases} \quad \Leftrightarrow \quad \theta = \begin{cases} w - \frac{\alpha}{\mu}, & \theta > 0 \Leftrightarrow w > \frac{\alpha}{\mu} \\ w + \frac{\alpha}{\mu}, & \theta < 0 \Leftrightarrow w < -\frac{\alpha}{\mu} \end{cases}$$

which gives the solution for $|w| > \frac{\alpha}{\mu}$. If $|w| \le \frac{\alpha}{\mu}$ then $\frac{\alpha}{\mu} \pm w \ge 0$ and we have:

$$F(\theta; w) = (\theta - w)^2 + \frac{2\alpha}{\mu}|\theta| = (\theta - w)^2 + \begin{cases} \frac{2\alpha}{\mu}, & \theta \ge 0 \\ -\frac{2\alpha}{\mu}, & \theta \le 0 \end{cases} = w^2 + \theta^2 + \begin{cases} 2\theta(\frac{\alpha}{\mu} - w), & \theta \ge 0 \\ -2\theta(\frac{\alpha}{\mu} + w), & \theta \le 0 \end{cases} \ge w^2 + \theta^2$$

which is minimized at $\theta = 0$. □

**Theorem B.8** ($\ell_2^2$ penalty)**.** *Let* $c(\theta) = \|\theta\|_2^2$. *Then* $\Pi^+_c\big(w; \frac{2\alpha}{\mu}\big) = w/\big(1 + \frac{2\alpha}{\mu}\big)$.

*Proof.* It follows from equating the derivative of $F(\theta; w)$ wrt $\theta$ to zero and solving for $\theta$. □

## C Analysis of the beginning of the path for the penalty form

Section 7 described the beginning of the solution path, $(\mathbf{w}(\mu), \boldsymbol{\theta}(\mu))$ for $\mu \to 0^+$, for the constraint form. The result is that $\mathbf{w}(0) = \overline{\mathbf{w}}$ is a reference model (trained to minimize the loss $L(\mathbf{w})$ over all weights) and $\boldsymbol{\theta}(0) = \Pi^{\le}_C(\overline{\mathbf{w}}; \kappa)$, i.e., pruning $\overline{\mathbf{w}}$ by magnitude so that only the largest weights are nonzero. For the penalty form, the behavior is qualitatively similar but harder to analyze. We describe this in detail next.

As in the constraint form, taking $\mu \to 0^+$ and minimizing either the QP or AL results in first minimizing over $\mathbf{w}$ for $\mu = 0$ (an L step) and then minimizing over $\boldsymbol{\theta}$ given that $\mathbf{w}$ (a C step). For $\mathbf{w}$ the result is the same: $\mathbf{w}(0) = \overline{\mathbf{w}} = \arg\min_{\mathbf{w}} L(\mathbf{w})$ (the reference model). For $\boldsymbol{\theta}$, the result is very different: $\boldsymbol{\theta}(0) = \mathbf{0}$. This is because for $\mu \to 0^+$ the C step corresponds to minimizing $C(\boldsymbol{\theta})$, which achieves a global minimum at zero. *Hence, in the penalty form and for any pruning cost $C(\mathbf{w})$, the LC algorithm starts by marking all weights as pruned.*

A second surprise is that, *for the $\ell_0$ and $\ell_1$ costs, $\boldsymbol{\theta}$ remains equal to zero for $\mu \in [0, \mu_0]$*, where the value $\mu_0 > 0$ is determined below. As $\mu > 0$ increases, all the elements $w_i$ of $\mathbf{w}$ change continuously with $\mu$ as soon as $\mu > 0$, as seen from the form of the L step. This is also true for $\boldsymbol{\theta}$ with the $\ell_2^2$ cost but not with $\ell_1$ (which is continuous but nondifferentiable) and $\ell_0$ (which is discontinuous and is really a combinatorial problem). As long as $w_i^2$ does not exceed a certain value ($\frac{2\alpha}{\mu}$ for $\ell_0$, $\frac{\alpha}{\mu}$ for $\ell_1$), $\theta_i$ stays put at 0 even as $\mu$ increases and $w_i$ changes. *So $\theta_i$ becomes nonzero only when $\mu$ reaches a certain value $\mu_i > 0$, where $\mu_i = 2\alpha/w_i^2(\mu_i)$ for $\ell_0$ and $\mu_i = \alpha/w_i^2(\mu_i)$ for $\ell_1$.* Of particular interest is the smallest value $\mu_0$ at which the first $\theta_i$ becomes nonzero. Computing this exactly is difficult, since $\mathbf{w}$ changes as soon as $\mu > 0$, but a good approximation results from approximating the weights at $\mu_0$ with the initial weights: $\mathbf{w}(\mu_0) \approx \mathbf{w}(0) = \overline{\mathbf{w}}$. Then, we obtain:

$$\text{For the QP: } \mu_0 \approx \begin{cases} 2\alpha/\max_i(\overline{w}_i^2), & \text{for the } \ell_0 \text{ cost} \\ \alpha/\max_i(\overline{w}_i^2), & \text{for the } \ell_1 \text{ cost} \end{cases} \tag{24}$$

where $\max_i(\overline{w}_i^2)$ is the largest squared weight of the entire reference net[5]. We can also estimate $\mu_0$ for the AL. The update $\boldsymbol{\lambda} \leftarrow \boldsymbol{\lambda} - \mu(\mathbf{w} - \boldsymbol{\theta})$ becomes $\boldsymbol{\lambda} = -\mu_0 \mathbf{w}(\mu_0) \approx -\mu_0 \overline{\mathbf{w}}$ if we do the first update for $\mu = \mu_0$ (since $\boldsymbol{\theta} = \mathbf{0}$ for $\mu \le \mu_0$ and we initialize $\boldsymbol{\lambda} = \mathbf{0}$). The C step for AL is like for the QP but using $\mathbf{w} - \frac{1}{\mu}\boldsymbol{\lambda}$ instead of $\mathbf{w}$, hence $\mathbf{w}(\mu_0) - \frac{1}{\mu_0}\boldsymbol{\lambda} \approx 2\overline{\mathbf{w}}$. Hence, *the AL estimate for $\mu_0$ is 4 times smaller than for QP*:

$$\text{For the AL: } \mu_0 \approx \begin{cases} \frac{1}{2}\alpha/\max_i(\overline{w}_i^2), & \text{for the } \ell_0 \text{ cost} \\ \frac{1}{4}\alpha/\max_i(\overline{w}_i^2), & \text{for the } \ell_1 \text{ cost.} \end{cases} \tag{25}$$

Some final remarks about our $\mu_0$ estimate:

[5]In this section, the weights $w_i$ are those parameters in the neural net that are subject to pruning. Typically, these are only the multiplicative weights, not the biases. Hence, in the formulas (24) and (25), the expression "$\max_i(\overline{w}_i^2)$" includes only the multiplicative weights of the reference model. This is important because the biases are usually quite larger than the multiplicative weights.

- $\mathbf{w}(\mu_0)$ will be slightly smaller than $\overline{\mathbf{w}}$ because the former is obtained by weight decay, hence our estimate of $\mu_0$ will likely be slightly smaller than the real $\mu_0$. Underestimating $\mu_0$ is better than overestimating it, so that we do not miss the first changes to $\boldsymbol{\theta}$.
- Since in practice with deep nets the weight values are not far from 1 in magnitude, $\mu_0$ is not far from $\alpha$ (at least at an order-of-magnitude estimate). Hence, a very simple, rough estimate is $\mu_0 \approx \alpha$.
- The above argument applies more generally to the case where each weight $w_i$ has a different $\alpha_i$ value (this is useful if using a different $\alpha$ in each layer). In the equations above, we replace the expression "$\alpha/\max_i(\overline{w}_i^2)$" with "$\min_i(\alpha_i/\overline{w}_i^2)$".
- We can use directly the expression $\mu_i = 2\alpha/w_i^2(\mu_i)$ to estimate the $\mu$ value at which a particular weight $w_i$ will revive (in other words, we simply remove the "$\max_i$" expression in eqs. (24)–(25)). For example, $\mu_i \approx 2\alpha/\overline{w}_i^2$ (for QP, $\ell_0$). Again, this assumes $w_i(\mu_i) \approx \overline{w}_i$, i.e., that the real-valued weight did not change much from the reference. This approximation will become increasingly worse as $\mu$ increases and smaller weights are revived.

At the end of the day, what this means in practice is that we start following the path at the above value $\mu_0 > 0$. We initialize $\mathbf{w} = \overline{\mathbf{w}}$, run an L step $\mathbf{w} \leftarrow \arg\min_{\mathbf{w}} L(\mathbf{w}) + \frac{\mu_0}{2}\|\mathbf{w}\|^2$ (i.e., optimize the loss $L$ with weight decay of parameter $\mu_0$), set $\boldsymbol{\theta} = \mathbf{0}$ and (for AL) set $\boldsymbol{\lambda} = -\mu_0\overline{\mathbf{w}}$. This places us close to the path point $(\mathbf{w}(\mu_0), \boldsymbol{\theta}(\mu_0))$. We then run L and C steps as usual[6]. The first $\theta_i$ to become nonzero corresponds to the largest weight. As $\mu$ keeps increasing, a second $\theta_i$ becomes nonzero, corresponding to the second largest weight, and so on. One by one in single file the $\theta_i$ become nonzero, in an order approximately equal to that of the reference weights $\overline{\mathbf{w}}$ in decreasing magnitude order. How a $\theta_i$ leaves zero depends on the cost $C(\boldsymbol{\theta})$: for $\ell_0$, $\theta_i$ changes discontinuously from 0 to $w_i$; for $\ell_1$, it increases continuously from 0. This is because the pruning operator is discontinuous for $\ell_0$ but continuous for $\ell_1$ (fig. 1). Also, since $\mu_0 \propto \alpha$, the more weights we prune (the larger $\alpha$) the larger $\mu_0$ is. We observe this behavior clearly in our experiments.

For the constraint form, as noted in section 7, the nonzeros in $\boldsymbol{\theta}$ are already formed at $\mu = 0$ and follow the magnitude order of $\overline{\mathbf{w}}$. As with the penalty form, in practice we start following the path at the $\mu_0$ estimate above, since we observe this gives good results.

Being able to estimate $\mu_0$ is practically helpful, as it avoids a trial-and-error search for the initial $\mu$ value, and because many of the important changes to the set of nonzeros (particularly for $\ell_0$) happen early in the path, so it is important not to miss them.

## References

C. M. Bishop. *Neural Networks for Pattern Recognition.* Oxford University Press, New York, Oxford, 1995.

E. J. Candès and T. Tao. The power of convex relaxation: Near-optimal matrix completion. *IEEE Trans. Information Theory*, 56(5):2053–2080, Apr. 2010.

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.

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.

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.

L. Condat. Fast projection onto the simplex and the $\ell_1$ ball. *Math. Prog.*, 158(1):575–585, July 2016.

T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms.* MIT Press, Cambridge, MA, third edition, 2009.

[6]We do not add these refinements to the pseudocode of fig. 2 to keep the latter simple.

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.

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.

S. Dieleman, J. Schlüter, C. Raffel, E. Olson, S. K. Sœnderby, 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. Degrave. Lasagne: First release, Aug. 2015.

D. L. Donoho. Compressed sensing. *IEEE Trans. Information Theory*, 52(4):1289–1306, Apr. 2006.

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.

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.

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.

S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, and W. J. Dally. EIE: Efficient inference engine on compressed deep neural network. In *Proc. 43rd Int. Symposium on Computer Architecture (ISCA 2016)*, pages 243–254, Seoul, Korea, June 18–22 2016a.

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 2016b.

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.

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.

B. Hassibi, D. G. Stork, G. Wolff, and T. Watanabe. Optimal brain surgeon: Extensions and performance considerations. In J. D. Cowan, G. Tesauro, and J. Alspector, editors, *Advances in Neural Information Processing Systems (NIPS)*, volume 6, pages 263–270. Morgan Kaufmann, San Mateo, CA, 1994.

K. He, X. Zhang, S. Ren, and J. Sun. Delving deep into rectifiers: Surpassing human-level performance on ImageNet classification. In *Proc. 15th Int. Conf. Computer Vision (ICCV'15)*, pages 1026–1034, Santiago, Chile, Dec. 11–18 2015.

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.

R. A. Horn and C. R. Johnson. *Matrix Analysis.* Cambridge University Press, second edition, 2013.

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.

R. Kohavi and G. H. John. Wrappers for feature subset selection. *Artificial Intelligence*, 97(1–2):273–324, Dec. 1997.

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.

Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proc. IEEE*, 86(11):2278–2324, Nov. 1998.

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.

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.

V. Nair and G. E. Hinton. Rectified linear units improve restricted Boltzmann machines. In J. Fürnkranz and T. Joachims, editors, *Proc. of the 27th Int. Conf. Machine Learning (ICML 2010)*, Haifa, Israel, June 21–25 2010.

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.

A. Novikov, D. Podoprikhin, 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.

R. Reed. Pruning algorithms—a survey. *IEEE Trans. Neural Networks*, 4(5):740–747, Sept. 1993.

T. N. Sainath, B. Kingsbury, V. Sindhwani, E. Arısoy, 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.

Theano Development Team. Theano: A Python framework for fast computation of mathematical expressions. arXiv:1605.02688, May 9 2016.

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.

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.

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.