Model compression as constrained optimization, with application to neural nets

Miguel Á. Carreira-Perpiñán
Electrical Engineering and Computer Science
University of California, Merced
`http://faculty.ucmerced.edu/mcarreira-perpinan`

work with **Yerlan Idelbayev** and **Arman Zharmagambetov**

Outline

- Machine learning and the Internet-of-Things:
  the need for model compression
- Direct compression via standard signal processing algorithms
- Model compression as constrained optimization
- The “learning-compression” (LC) algorithm
- Three examples of deep neural net compression:
  - Quantization
  - Pruning
  - Low-rank
- Beyond compression:
  - Optimizing runtime, energy, etc. instead of memory;
    hardware retargeting: “compiling” a neural net
  - Model compression as structure learning

The rise of machine learning

Machine learning, and deep learning in particular, has recently become practical and widespread, significantly pushing the state of the art in many applications:

- Computer vision: object recognition, tracking, etc.
- Image and signal processing: super-resolution, style transfer, etc.
- Speech processing: speech recognition, speech synthesis, etc.
- NLP: dialog systems, image captioning, etc.
- Information retrieval in large databases of image, audio, text, etc.

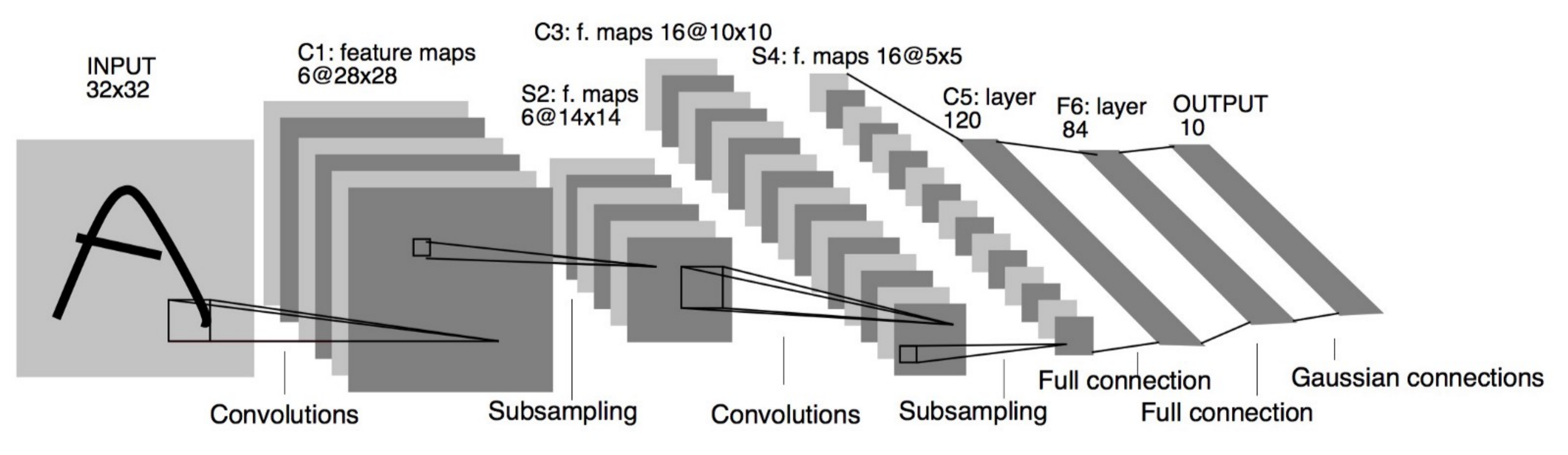
Reasons:

- much data available publicly or privately
- sophisticated models & algorithms in the statistics & ML literature
- computing power provided by GPUs
- deep learning toolboxes accessible to non-experts. Tensorflow, PyTorch…

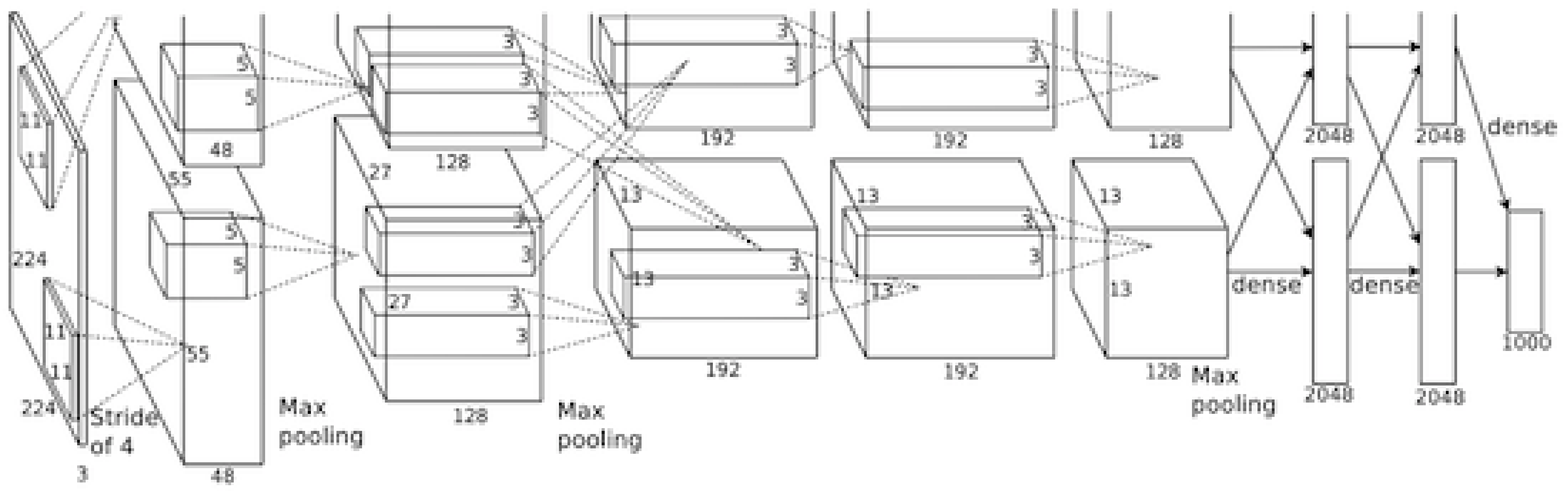
The rise of machine learning (cont.)

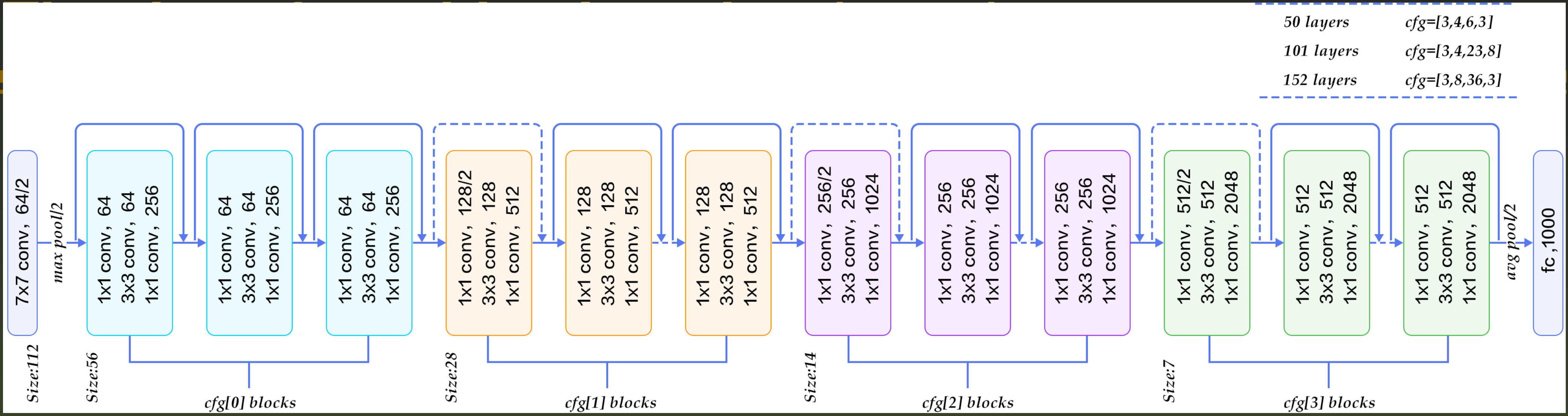
Typical deep net architectures: convolutional, residual nets, etc.

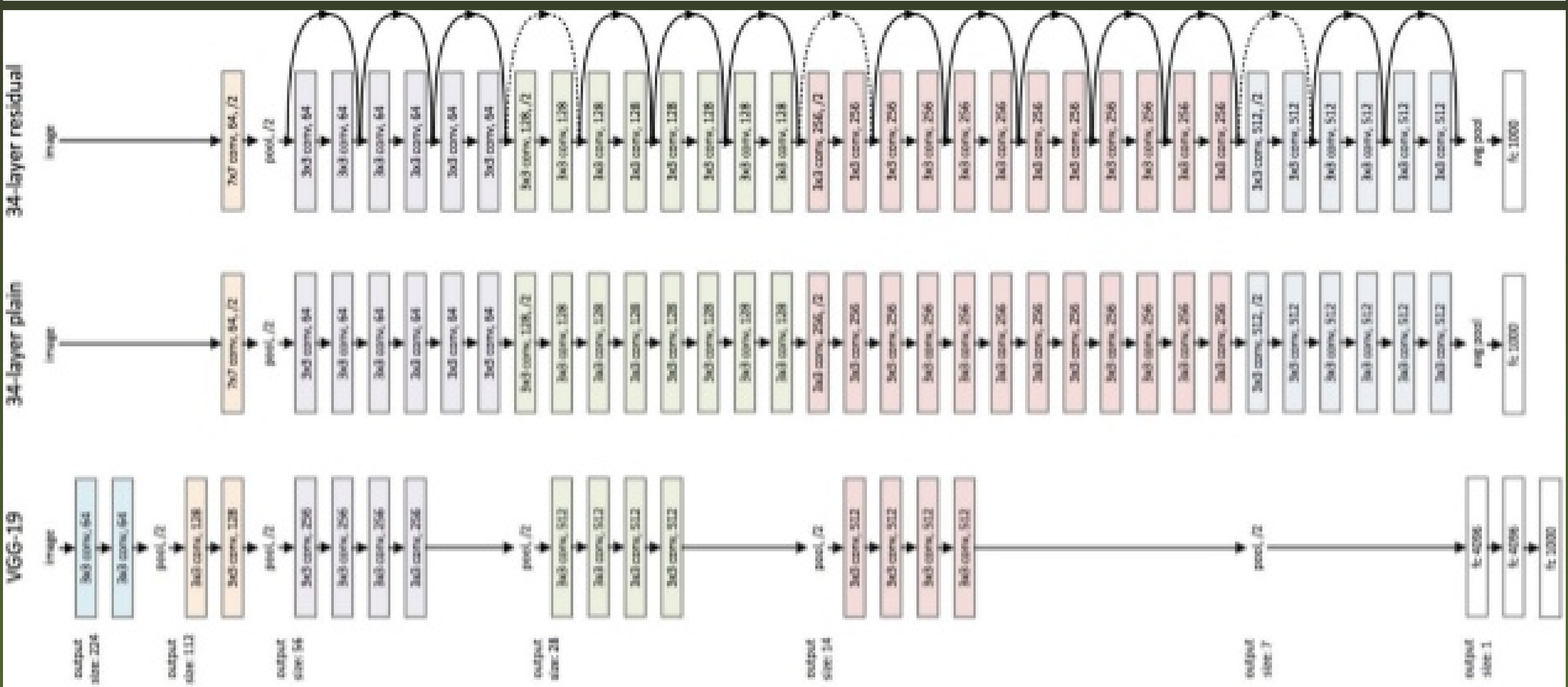
LeNet



AlexNet

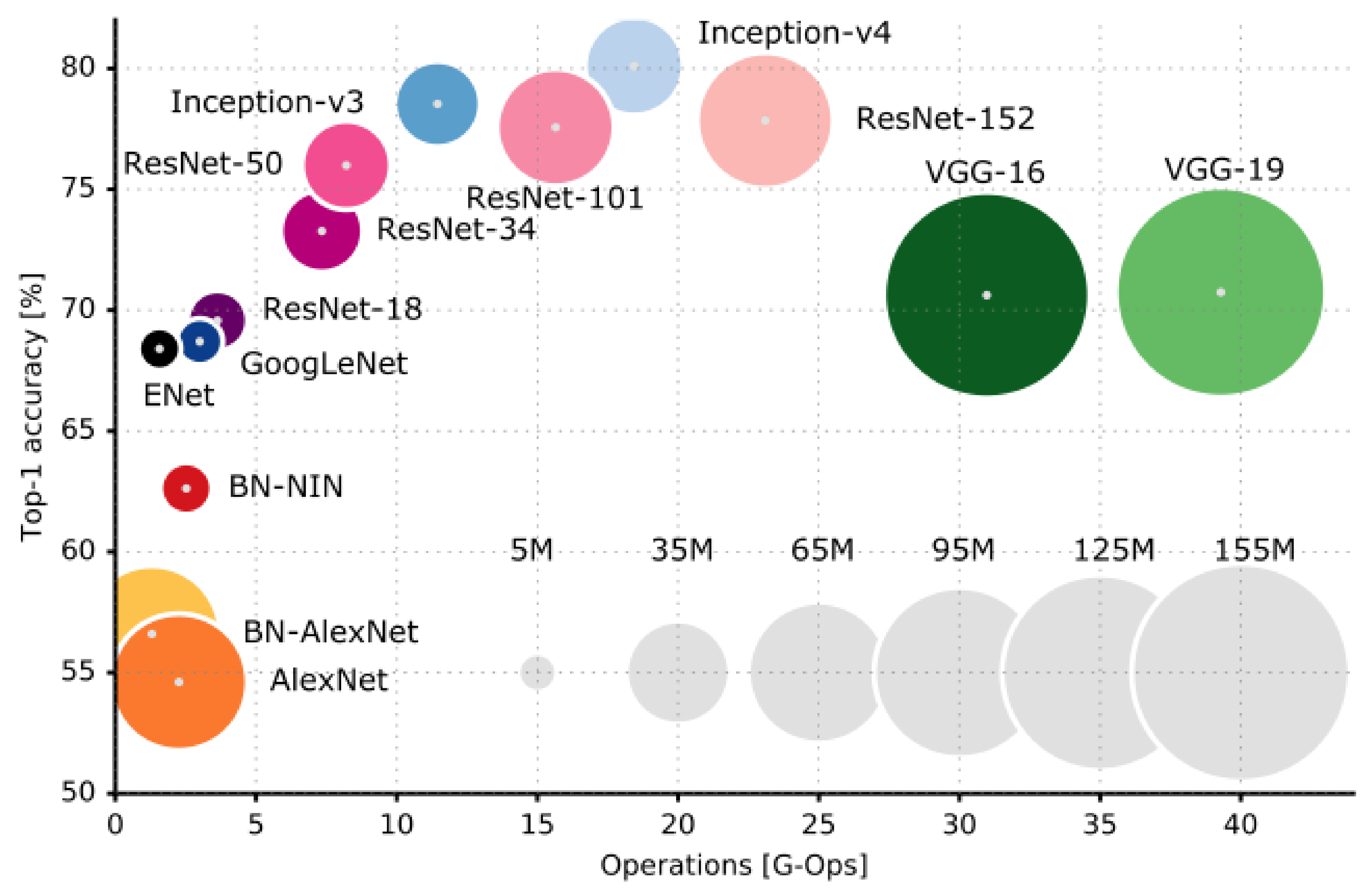



50 layers cfg=[3,4,6,3]
101 layers cfg=[3,4,23,8]
152 layers cfg=[3,8,36,3]
7x7 conv，64/2
max pool/2
1x1 conv，64
3x3 conv，64
1x1 conv，256
1x1 conv，64
3x3 conv，64
1x1 conv，256
1x1 conv，64
3x3 conv，64
1x1 conv，256
1x1 conv，128/2
3x3 conv，128
1x1 conv，512
1x1 conv，128
3x3 conv，128
1x1 conv，512
1x1 conv，128
3x3 conv，128
1x1 conv，512
1x1 conv，256/2
3x3 conv，256
1x1 conv，1024
1x1 conv，256
3x3 conv，256
1x1 conv，1024
1x1 conv，256
3x3 conv，256
1x1 conv，1024
1x1 conv，512/2
3x3 conv，512
1x1 conv，2048
1x1 conv，512
3x3 conv，512
1x1 conv，2048
1x1 conv，512
3x3 conv，512
1x1 conv，2048
avg pool/2
fc，1000
Size:112
Size:56
Size:28
Size:14
Size:7
cfg[0] blocks
cfg[1] blocks
cfg[2] blocks
cfg[3] blocks


34-layer residual
34-layer plain
VGG-19
image
7x7 conv, 64, /2
pool, /2
3x3 conv, 64
3x3 conv, 128, /2
3x3 conv, 128
3x3 conv, 256, /2
3x3 conv, 256
3x3 conv, 512, /2
3x3 conv, 512
avg pool
fc 1000
image
3x3 conv, 64
pool, /2
3x3 conv, 128
pool, /2
3x3 conv, 256
pool, /2
3x3 conv, 512
pool, /2
3x3 conv, 512
pool, /2
fc 4096
fc 4096
fc 1000
output size: 224
output size: 112
output size: 56
output size: 28
output size: 14
output size: 7
output size: 1

The rise of machine learning (cont.)

Downside: very good classification accuracy needs training large, deep neural nets on large datasets, with several GPUs over several days.



Canziani et al. 2017, arXiv:1605.07678

The rise of the Internet-of-Things (IoT)

Internet-of-Things (IoT) and other mobile devices have become pervasive in daily life and this trend is going to continue growing:

- smart phones
- smart cameras
- mobile robots
- wearables
- personal assistants, tablets
- sensors in medicine, ecology, smart buildings. . .

Their functionality can be significantly enhanced with intelligent, adaptive software for tasks in computer vision, speech, NLP, etc.

The rise of the Internet-of-Things (cont.)

Downside: IoT devices have stringent computing limitations:

- CPU speed
- memory
- bandwidth and connectivity
- battery life
- energy consumption
- other processes running in the device beyond the neural net.

Actual specs vary widely depending on the device and intended application, but are generally very limited compared to a workstation.

This places a limit in, say, the size or the energy consumption of the deep nets that can be deployed, and motivates the problem of model compression: taking a well-trained, reference model and compressing it into one that fits in the device (in memory, energy consumption, etc.).

Model compression and statistics

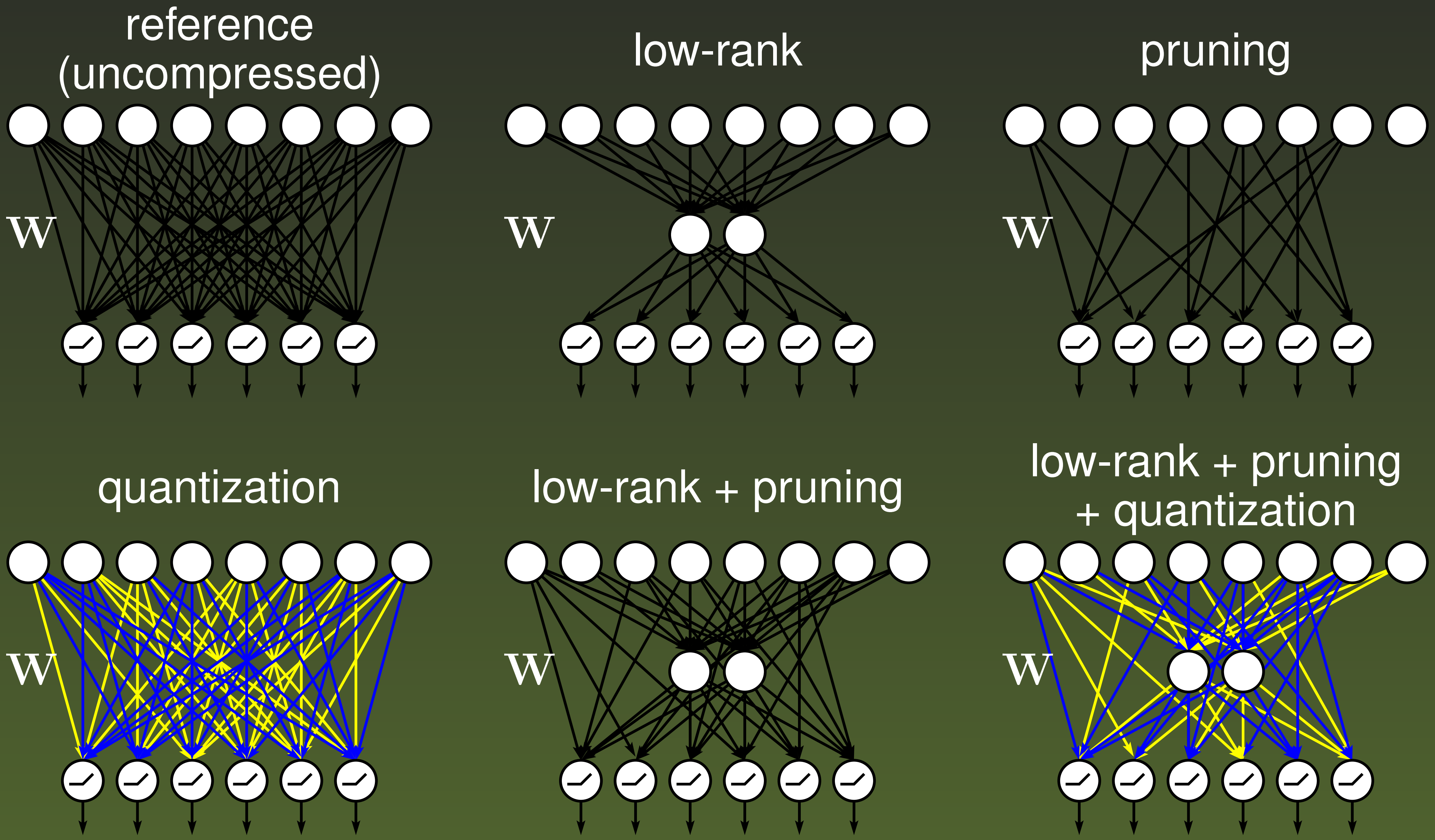
Although the model compression problem is motivated by a practical, engineering need, it is related to deeper questions about model selection, structure learning, effective number of parameters, regularization, etc.

Some forms of compression have a long history in statistics & ML, usually applied to linear models of the form $\mathbf{y} = \mathbf{W}\mathbf{x}$:

- pruning = sparse learning (certain parameters w_{ij} are zero)
  LASSO, sparse regression, etc.
- low-rank compression = parameter matrix $\mathbf{W}$ is low-rank
  reduced-rank regression (RRR)
- quantization = learnable parameter sharing (certain groups of parameters w_{ij} take the same value)

In general, compression is a sophisticated form of regularization. Compressed models are less complex and may generalize better.

Model compression: examples



reference (uncompressed)

low-rank

pruning

quantization

low-rank + pruning

low-rank + pruning + quantization

Neural net compression

Why not train a small model in the first place? Certainly possible, but:

- The desired size depends on the target device, so the training (which is not easy) must be repeated for each new device type.
- Training well a model without restrictions (the reference model) gives us a baseline: the best possible model for the available training data.
- For reasons not yet well understood, in practice it seems easier to achieve state-of-the-art performance by using a larger-than-needed model rather than a model of the right size: over-parameterization.

  It seems that, for deep nets: 1) over-parameterized nets (more parameters than points, even without regularization) don't overfit; 2) training an over-parameterized net with SGD tends to minima with zero training error that generalize well; 3) the optimization is easier if the space has excess dimensions that allow us to take shortcuts towards a minimizer.

  This leads to reference nets with many layers and many units per layer that can be significantly compressed.

A common approach at present is to train a good reference model and then compress it as needed for a particular device.

Neural net compression: related work

Many papers in the last 3–4 years, although neural net compression was already studied in the 1980–90s. Some common approaches:

- Direct compression: train a reference net, then compress its weights using standard signal compression techniques.

  Quantization (binarization, low precision), pruning weights/neurons, low-rank, etc.

  Simple and fast, but it ignores the loss function, so the accuracy deteriorates a lot for high compression rates.
- Embedding the compression mechanism in the SGD training.
  - remove neuron/weight values with low magnitude on the fly
  - binarize or round weight values in the backpropagation forward pass

  Often heuristic, with no convergence guarantees.
- Other algorithms proposed for specific compression techniques.
- Multiple compression techniques can be combined.

Over-parameterized nets can be considerably compressed even with simple techniques, but we seek optimal compression.

How to train the reference model

In machine learning, we typically learn by solving an optimization problem. We define and objective function (loss) $L(\mathbf{w})$, possibly s.t. constraints, given by the task, model and training set, e.g.:

- classification: cross-entropy $L(\mathbf{w}) = -\sum_{n=1}^{N} \sum_{k=1}^{K} y_{nk} f_k(\mathbf{x}_n; \mathbf{w})$
- regression: least-squares error $L(\mathbf{w}) = \sum_{n=1}^{N} \|\mathbf{y}_n - \mathbf{f}(\mathbf{x}_n; \mathbf{w})\|^2$
- maximum likelihood $L(\mathbf{w}) = -\sum_{n=1}^{N} \log p(\mathbf{x}_n; \mathbf{w})$
- etc.

where $\{(\mathbf{x}_n, \mathbf{y}_n)\}_{n=1}^{N}$ is the labeled training set and $\mathbf{f}(\mathbf{x}; \mathbf{w})$ is the predictive function to be learnt (say, a neural net), with parameters $\mathbf{w}$.

We then find a (local) minimum:

$$\min_{\mathbf{w}} L(\mathbf{w})$$

with a suitable optimization algorithm.

How to train the reference model (cont.)

For deep nets in particular:

- $\mathbf{w}$ = weights & biases in all layers of the net; the loss is nonconvex.
- Minimizing $L(\mathbf{w}) = \sum_{n=1}^{N} L_n(\mathbf{w})$ is typically achieved by stochastic gradient descent (SGD): $\mathbf{w}_{k+1} = \mathbf{w}_k - \eta_k \sum_{n \in \mathcal{B}_k} \nabla L_n(\mathbf{w}_k),\ k = 0, 1 \dots$

  Gradient step on a minibatch $\mathcal{B}_k \subset \{1, \dots, N\}$, typically with momentum. For convergence, the step sizes $\{\eta_k\}$ must satisfy the Robbins-Monro conditions (tend to zero at a certain rate).

  Practically, the user must carefully select the hyperparameters to ensure relatively fast convergence (step sizes, momentum rate, minibatch size, possibly others), and stop training when a low enough loss is achieved. This is tricky.

  Other algorithms exist but generally SGD with well-tuned hyperparameters is best.
- GPUs are currently the best computing platform.

  Other platforms (e.g. CPUs) have not been as successful.

  Training times of many hours or days for large problems.
- Deep learning toolboxes have made this process accessible to non-experts. Tensorflow, Theano, PyTorch, MXNet, Caffe…

Direct compression (via signal processing algorithms)

- Simply take the reference model parameters and compress them using off-the-shelf compression algorithms, just as is done with compression of text, image or other signals:
  - Lossless: LZW, run-length / Huffman / arithmetic codes, etc.

    Generally won't achieve much compression.
  - Lossy: DCT (JPEG), wavelet, etc. (see below).

    Generally will achieve much more compression, depending on the error tolerated.
- Simple and practical, because we use existing algorithms with well-developed code:
  - variable-length codes: run Huffman's algorithm
  - pruning (zeroing): zero all weights smaller than a threshold
  - binarization into $\{-1,+1\}$: replace each weight w_i with $\operatorname{sgn}(w_i)$
  - quantization: run k-means on the weights $\mathbf{w} = \{w_1,\dots,w_P\} \subset \mathbb{R}$
  - low-rank: compute the singular value decomposition (SVD).

Direct compression (cont.)

The resulting compressed model $\mathbf{w}^{\text{DC}}$ is optimal wrt the uncompressed, reference model $\overline{\mathbf{w}}$…

That is, among all possible compressed models, $\mathbf{w}^{\text{DC}}$ has the lowest compression error (e.g. quadratic distortion $\left\|\mathbf{w}^{\text{DC}} - \overline{\mathbf{w}}\right\|^2$ for k-means or SVD).

…but it is not optimal wrt the loss $L(\mathbf{w})$.

That is, among all possible compressed models, $\mathbf{w}^{\text{DC}}$ does not generally have the smallest loss.

This introduces a radical difference with the traditional view of signal compression. We still want to compress, but to minimize the loss, not the compression error.

How well does direct compression work?

- Quite well as long as we don't compress much, but the loss blows up as we compress more. Surprisingly high compression rates are often possible because deep nets are vastly over-parameterized in practice.
- In practice, we want to compress as much as possible, so we need an optimal way to do this that achieves the lowest loss for a given compression rate.

Traditional compression vs machine learning

How do we combine the following notions?

1. The traditional signal compression view:
   - extensively studied
   - many existing forms of compression
   - usually carried out with an algorithm that is (near-)optimal wrt the compression error

     quantization by k-means, low-rank by SVD, etc.

2. The traditional machine learning view:
   - a model with a given architecture, e.g. a deep net
   - a loss defined given a training set (e.g. cross-entropy) that we need to minimize over the model parameters
   - many optimization algorithms specific for each loss and model

     gradient descent, SGD, variations of Newton's method, (L-)BFGS, EM algorithm. . .

Desiderata for a good solution to the problem

Firstly, we need a precise, general mathematical definition of the problem of model compression that is amenable to numerical optimization techniques. Intuitively, we want to achieve the lowest loss possible for a given compression technique and compression rate.

Then, we would like an algorithm that:

- is generic, applicable to many losses and compression techniques

  so the user can easily try different types of compression on a given loss function without having to use a specialized algorithm for each case
- has optimality guarantees inasmuch as possible
- is easy to integrate in existing deep learning toolboxes
- is efficient in training
- works on the native discrete space and can make use of effective discrete optimization solvers rather than relaxing/applying soft priors & truncating, for compression involving discrete variables: quantization, pruning, learning ranks.

An illustrative example: low-rank compression

Consider a linear regression problem with weight matrix $\mathbf{W}_{d \times D}$:

$$\min_{\mathbf{W}} L(\mathbf{W}) = \sum_{n=1}^{N} \left\| \mathbf{y}_n - \mathbf{W}\mathbf{x}_n \right\|^2 \qquad \text{(the loss)}$$

We want $\operatorname{rank}(\mathbf{W}) \leq r \Leftrightarrow \mathbf{W} = \mathbf{U}\mathbf{V}^T$ with $\mathbf{U}_{d \times r}$, $\mathbf{V}_{D \times r}$, $r < \min(d, D)$ (the compression type). This results in $(D + d)r$ parameters in $(\mathbf{U}, \mathbf{V})$, which is smaller than the Dd parameters in $\mathbf{W}$ if r is small enough.

Hence, the problem (called reduced-rank regression (RRR) in statistics) is:

$$\min_{\mathbf{U},\mathbf{V}} L(\mathbf{U}, \mathbf{V}) = \sum_{n=1}^{N} \left\| \mathbf{y}_n - \mathbf{U}\mathbf{V}^T\mathbf{x}_n \right\|^2 \quad \text{s.t.} \quad \mathbf{U}^T\mathbf{U} = \mathbf{I}$$

This could be solved in various ways, more or less convenient

- using gradient-based optimization, since the problem is differentiable
- using alternating optimization over $\mathbf{U}$ and $\mathbf{V}$
- in fact, there is a closed-form solution as an eigenproblem.

However, all these solutions are specific to the form of the RRR problem, and do not apply generally.

An illustrative example: low-rank compression (cont.)

For example, imagine that we want to compress the matrix $\mathbf{W}$ so that, rather than having rank r, its elements are quantized to K values $\mathcal{C} = \{c_1, \dots, c_K\} \subset \mathbb{R}$. The problem would then be (quantization):

$$\min_{\mathbf{W},\mathcal{C}} L(\mathbf{W},\mathcal{C}) = \sum\nolimits_{n=1}^{N} \left\| \mathbf{y}_n - \mathbf{W}\mathbf{x}_n \right\|^2 \quad \text{s.t.} \quad \{w_{ij}\}_{i,j=1}^{d,D} \subset \mathcal{C}$$

Or, if we want to compress $\mathbf{W}$ so it uses binary values (binarization):

$$\min_{\mathbf{W}} L(\mathbf{W},\mathcal{C}) = \sum\nolimits_{n=1}^{N} \left\| \mathbf{y}_n - \mathbf{W}\mathbf{x}_n \right\|^2 \quad \text{s.t.} \quad \{w_{ij}\}_{i,j=1}^{d,D} \subset \{-1,+1\}$$

None of the common approaches for RRR mentioned above work in these other cases; the objective function is not even differentiable.

An illustrative example: low-rank compression (cont.)

Rather than trying to design a different optimization algorithm for each combination of loss L and compression technique, we advocate the use of a single algorithm that can handle, with small adjustments, each combination.

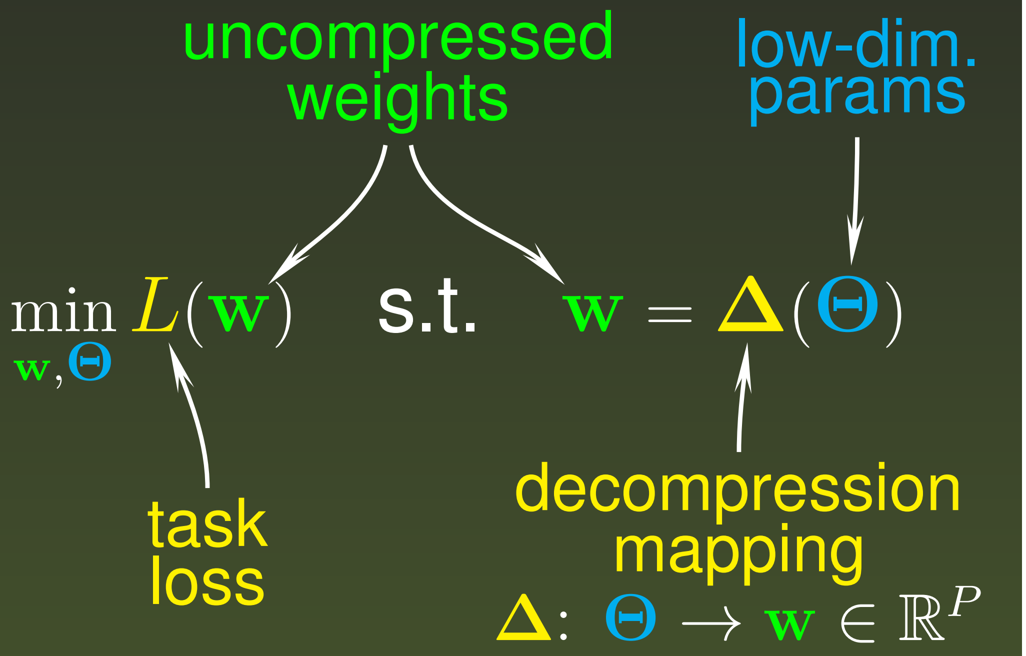
Idea (for low-rank): introduce auxiliary variables $\mathbf{W} = \mathbf{U}\mathbf{V}^T$ and define the model compression problem as a constrained problem:

$$\min_{\mathbf{W},\mathbf{U},\mathbf{V}} \underbrace{L(\mathbf{W}) = \sum_{n=1}^{N} \left\|\mathbf{y}_n - \mathbf{W}\mathbf{x}_n\right\|^2}_{\text{loss part}} \quad \text{s.t.} \quad \underbrace{\mathbf{W} = \mathbf{U}\mathbf{V}^T, \quad \mathbf{U}^T\mathbf{U} = \mathbf{I}}_{\text{compression part}}$$

This separates the loss part from the compression part. The resulting "learning-compression" (LC) algorithm alternates training a regularized loss with compressing $\mathbf{W}$ via the SVD.

Model compression as constrained optimization

General formulation:



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

uncompressed weights

low-dim. params

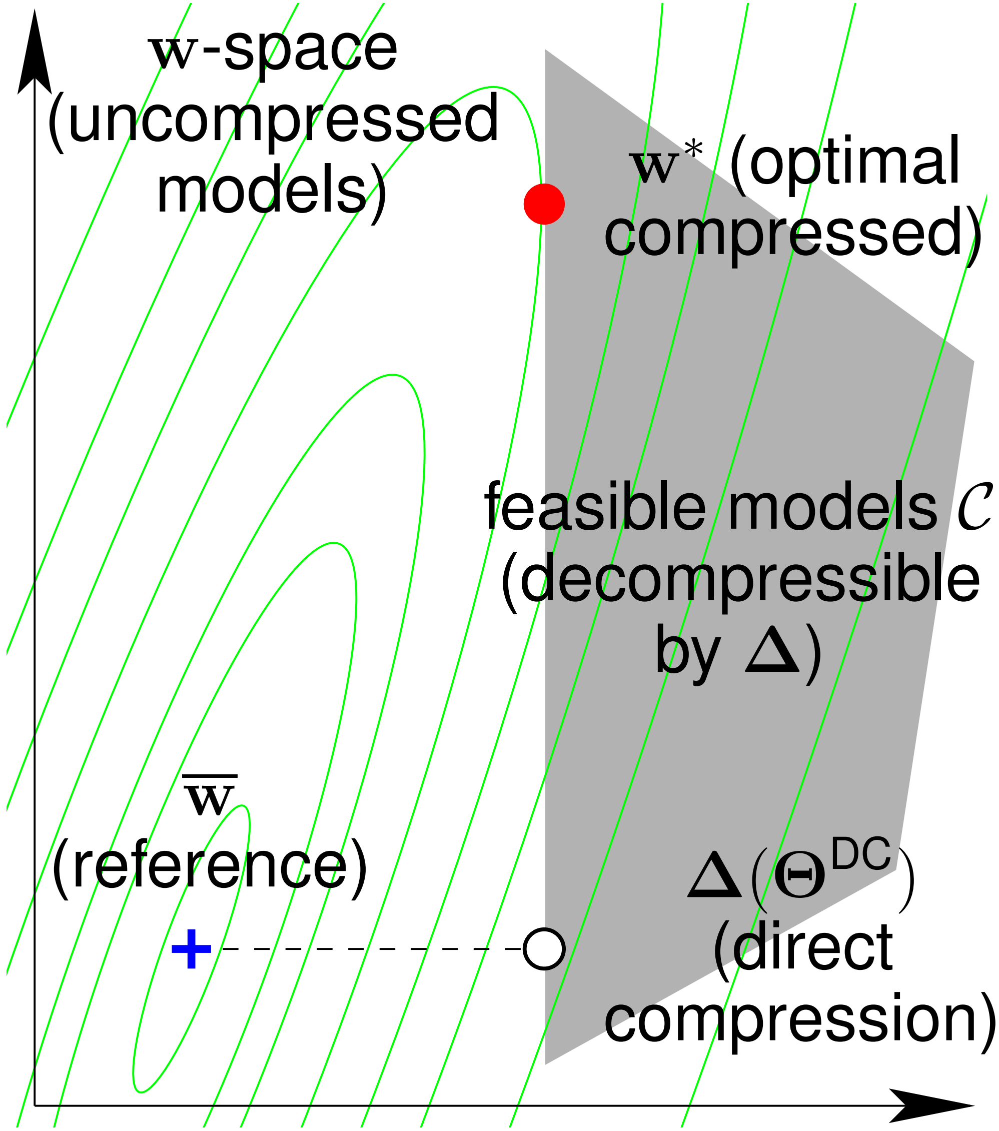
task loss

decompression mapping

$\boldsymbol{\Delta}\colon\ \boldsymbol{\Theta} \rightarrow \mathbf{w} \in \mathbb{R}^P$

Compression and decompression are usually seen as algorithms, but here we regard them as mathematical mappings in parameter space.

The details of the compression technique are abstracted in $\boldsymbol{\Delta}(\boldsymbol{\Theta})$.

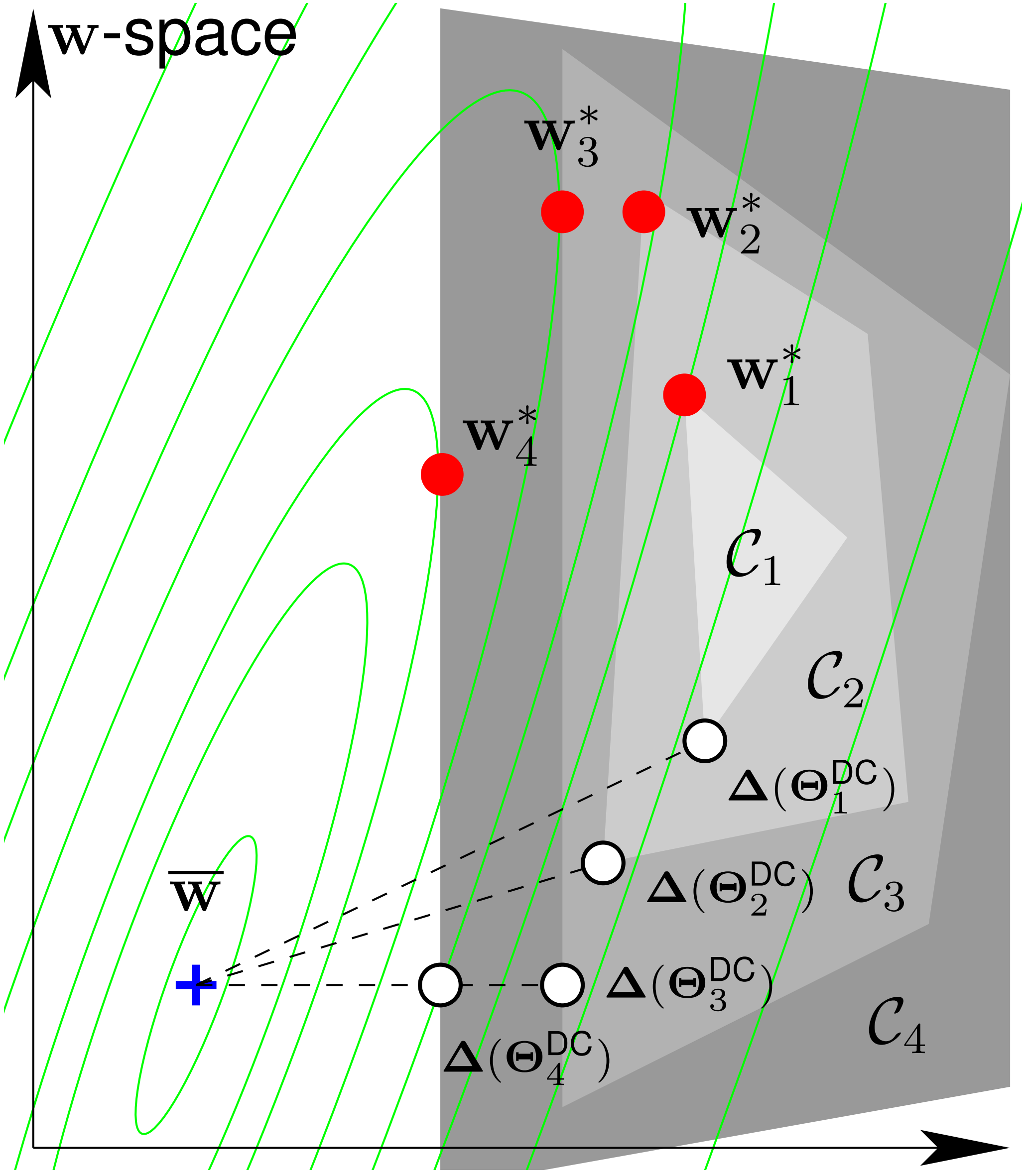


$\mathbf{w}$-space (uncompressed models)
$\mathbf{w}^*$ (optimal compressed)
feasible models $\mathcal{C}$ (decompressible by $\boldsymbol{\Delta}$)
$\overline{\mathbf{w}}$ (reference)
$\boldsymbol{\Delta}(\boldsymbol{\Theta}^{\text{DC}})$ (direct compression)

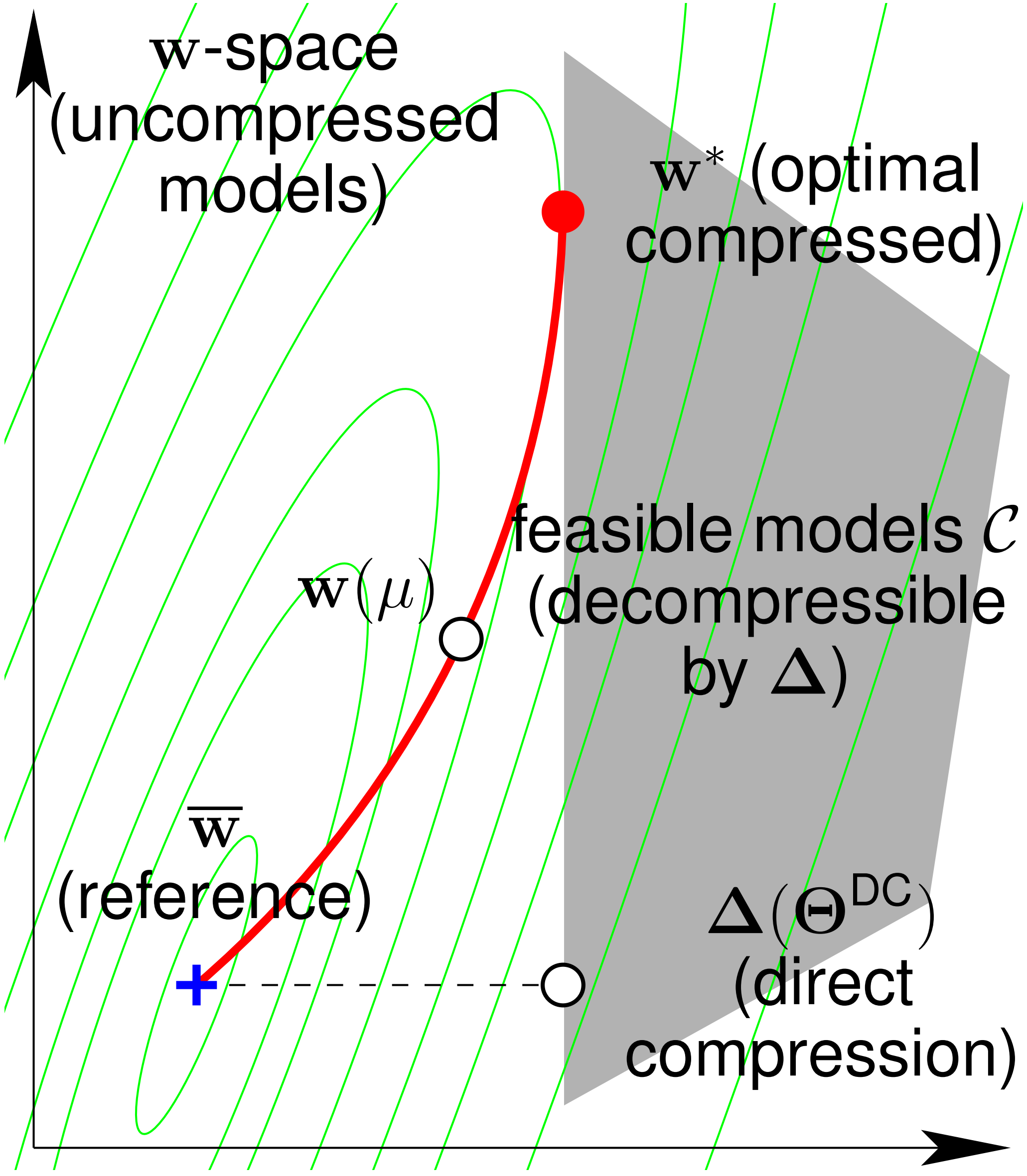
feasible set

$\mathcal{C} = \{\mathbf{w} \in \mathbb{R}^P\colon\ \mathbf{w} = \boldsymbol{\Delta}(\boldsymbol{\Theta}) \text{ for } \boldsymbol{\Theta} \in \mathbb{R}^Q\}$

Model compression as constrained optimization (cont.)


$\mathbf{w}$-space
$\mathbf{w}_3^*$
$\mathbf{w}_2^*$
$\mathbf{w}_1^*$
$\mathbf{w}_4^*$
$\mathcal{C}_1$
$\mathcal{C}_2$
$\boldsymbol{\Delta}(\boldsymbol{\Theta}_1^{\text{DC}})$
$\boldsymbol{\Delta}(\boldsymbol{\Theta}_2^{\text{DC}})$
$\mathcal{C}_3$
$\overline{\mathbf{w}}$
$\boldsymbol{\Delta}(\boldsymbol{\Theta}_3^{\text{DC}})$
$\mathcal{C}_4$
$\boldsymbol{\Delta}(\boldsymbol{\Theta}_4^{\text{DC}})$

feasible sets corresponding to different compression levels ($\mathcal{C}_1$ is most compression)


$\mathbf{w}$-space (uncompressed models)
$\mathbf{w}^*$ (optimal compressed)
feasible models $\mathcal{C}$ (decompressible by $\boldsymbol{\Delta}$)
$\mathbf{w}(\mu)$
$\overline{\mathbf{w}}$ (reference)
$\boldsymbol{\Delta}(\boldsymbol{\Theta}^{\text{DC}})$ (direct compression)

path in $\mathbf{w}$-space of LC algorithm iterates $\mathbf{w}(\mu)$ for $\mu > 0$

The "learning-compression" (LC) algorithm

Use a penalty method (e.g. quadratic penalty, augmented Lagrangian). Minimize:

$$Q(\mathbf{w},\boldsymbol{\Theta};\mu) = L(\mathbf{w}) + \frac{\mu}{2}\|\mathbf{w}-\boldsymbol{\Delta}(\boldsymbol{\Theta})\|^2$$

as $\mu \to \infty$, using alternating optimization over $\mathbf{w}$ and $\boldsymbol{\Theta}$:

- L ("learning") step: $\min_{\mathbf{w}} L(\mathbf{w}) + \frac{\mu}{2}\|\mathbf{w}-\boldsymbol{\Delta}(\boldsymbol{\Theta})\|^2$ independent of the compression technique
- C ("compression") step: $\min_{\boldsymbol{\Theta}} \|\mathbf{w}-\boldsymbol{\Delta}(\boldsymbol{\Theta})\|^2$ independent of the loss and dataset.

Generic: one algorithm, many compression techniques.

To change the compression type, we just need to change the C step.

The minimizers of Q trace a path $(\mathbf{w}(\mu),\boldsymbol{\Theta}(\mu))$ for $\mu \geq 0$:

- Beginning of the path ($\mu \to 0^+$): direct compression $(\overline{\mathbf{w}},\boldsymbol{\Theta}^{\text{DC}})$ (training the reference model and then compressing its weights).
- End of the path ($\mu \to \infty$): local solution of the constrained problem.

LC algorithm, augmented Lagrangian version

```
input training data and model with parameters (weights) $\mathbf{w}$
$\mathbf{w} \leftarrow \overline{\mathbf{w}} = \arg\min_{\mathbf{w}} L(\mathbf{w})$    reference model
$\boldsymbol{\Theta} \leftarrow \boldsymbol{\Theta}^{\text{DC}} = \boldsymbol{\Pi}(\overline{\mathbf{w}}) = \arg\min_{\boldsymbol{\Theta}} \|\overline{\mathbf{w}} - \boldsymbol{\Delta}(\boldsymbol{\Theta})\|^2$    compress reference model
$\boldsymbol{\lambda} \leftarrow 0$
for $\mu = \mu_0 < \mu_1 < \cdots < \infty$
    $\mathbf{w} \leftarrow \arg\min_{\mathbf{w}} L(\mathbf{w}) + \frac{\mu}{2}\|\mathbf{w} - \boldsymbol{\Delta}(\boldsymbol{\Theta}) - \frac{1}{\mu}\boldsymbol{\lambda}\|^2$    L step: learn model
    $\boldsymbol{\Theta} \leftarrow \boldsymbol{\Pi}(\mathbf{w} - \frac{1}{\mu}\boldsymbol{\lambda}) = \arg\min_{\boldsymbol{\Theta}} \|\mathbf{w} - \frac{1}{\mu}\boldsymbol{\lambda} - \boldsymbol{\Delta}(\boldsymbol{\Theta})\|^2$    C step: compress model
    $\boldsymbol{\lambda} \leftarrow \boldsymbol{\lambda} - \mu(\mathbf{w} - \boldsymbol{\Delta}(\boldsymbol{\Theta}))$    Lagrange multipliers
    if $\|\mathrm{w} - \boldsymbol{\Delta}(\boldsymbol{\Theta})\|$ is small enough then exit the loop
return $\mathrm{w}$, $\boldsymbol{\Theta}$
```

LC algorithm: convergence

The “model compression as constrained optimization” framework includes a large variety of problems, usually nonconvex, whose nature can be continuous (e.g. low-rank, pruning with ℓ_1) or combinatorial (e.g. quantization, pruning with ℓ_0).

Convergence of the LC algorithm to a (local) minimizer of the constrained problem can be established for the easier cases (smooth convex) but not in general for the harder cases (nonconvex, discrete). These are the more interesting ones in practice, and the ones the LC algorithm is designed to handle.

We suspect that the partial separability of Q over $\mathbf{w}$ and $\mathbf{\Theta}$, and the fact that the C (compression) step is often exactly solvable, may afford stronger results, but this is an open question.

LC algorithm: convergence (cont.)

Smooth case: cont. differentiable loss & decompression mapping.

Theorem 1 (for the QP $Q(\mathbf{w},\mathbf{\Theta};\mu)$). Given a positive increasing sequence $(\mu_k) \rightarrow \infty$, a nonnegative sequence $(\tau_k) \rightarrow 0$, and a starting point $(\mathbf{w}^0,\mathbf{\Theta}^0)$, suppose the QP method finds an approximate minimizer $(\mathbf{w}^k,\mathbf{\Theta}^k)$ of $Q(\mathbf{w}^k,\mathbf{\Theta}^k;\mu_k)$ that satisfies $\|\nabla_{\mathbf{w},\mathbf{\Theta}}\, Q(\mathbf{w}^k,\mathbf{\Theta}^k;\mu_k)\| \leq \tau_k$ for $k = 1,2,\dots$ Then, $\lim_{k\rightarrow\infty}\left((\mathbf{w}^k,\mathbf{\Theta}^k)\right) = (\mathbf{w}^*,\mathbf{\Theta}^*)$, which is a KKT point for the constrained problem, and its Lagrange multiplier vector has elements $\boldsymbol{\lambda}_i^* = \lim_{k\rightarrow\infty}\left(-\mu_k\left(\mathbf{w}_i^k - \mathbf{\Delta}(\mathbf{\Theta}_i^k)\right)\right)$, $i = 1,\dots,P$.

That is, there exists a continuous path $(\mathbf{w}(\mu),\mathbf{\Theta}(\mu))$ that converges to a local stationary point (typically a minimizer) of the constrained problem and it can be loosely followed by approximately (but accurately enough) optimizing Q as $\mu \rightarrow \infty$.

Does alternating optimization of Q for fixed μ over $\mathbf{w}$ and $\mathbf{\Theta}$ in the LC algorithm converge?

- If Q is convex, yes quite generally.
- If Q is nonconvex: convergence results are complex and more restrictive. One simple case where convergence occurs is if there is a unique minimizer along each block ($\mathbf{w}$ and $\mathbf{\Theta}$).

LC algorithm: convergence (cont.)

Nonsmooth or discrete case.

- These often are NP-complete problems, e.g. quantization (with adaptive assignments), binarization and ℓ_0 pruning.
- Convergence results are an open problem.
- With mixed continuous-discrete problems, the LC algorithm has the flavor of k-means; it stops at a "local optimum" in the sense that no further progress is possible.
- Empirically, the LC algorithm is competitive with specialized approaches for quantization, pruning or low-rank compression.
- It seems able to explore the discrete space (of weight assignments to codebook entries, or of the weight subset to be pruned) while driving down the loss.

LC algorithm: solving the L step

The L step always takes the same form regardless of the compression technique:

$$\min_{\mathbf{w}} L(\mathbf{w}) + \frac{\mu}{2}\|\mathbf{w} - \boldsymbol{\Delta}(\boldsymbol{\Theta})\|^2$$

- This is the original loss on the uncompressed weights $\mathbf{w}$, regularized with a quadratic term on the weights (which "decay" towards a validly compressed model $\boldsymbol{\Delta}(\boldsymbol{\Theta})$).
- It can be optimized in the same way as the reference net.

  Simply add $\mu(\mathbf{w} - \boldsymbol{\Delta}(\boldsymbol{\Theta}))$ to the gradient.
- With large datasets we typically use SGD.

  We clip the learning rates so they never exceed $\frac{1}{\mu}$ to avoid oscillations as $\mu \to \infty$.
- Straightforward to integrate into existing deep learning frameworks.

The L step behaves like a regular, machine-learning training of the original model. The compression technique appears only in the C step.

LC algorithm: solving the C step

Solving the C step means optimally compressing the current weights $\mathbf{w}$:

$$\min_{\boldsymbol{\Theta}} \|\mathbf{w} - \boldsymbol{\Delta}(\boldsymbol{\Theta})\|^2$$

- Fast: it does not require the dataset (which appears in $L(\mathbf{w})$).

  Runtime usually negligible compared to the L step.

- Equivalent to orthogonal projection $\boldsymbol{\Theta} = \boldsymbol{\Pi}(\mathbf{w})$ on the feasible set.

  Find the closest compressed model $\mathbf{v} = \boldsymbol{\Pi}(\mathbf{w})$ to $\mathbf{w}$: $\min_{\boldsymbol{\Theta},\mathbf{v}} \|\mathbf{w} - \mathbf{v}\|^2$ s.t. $\mathbf{v} = \boldsymbol{\Delta}(\boldsymbol{\Theta})$.

- The solution depends on the choice of the decompression mapping $\boldsymbol{\Delta}$, and is known for many common compression techniques:
  - low rank: SVD + truncation
  - binarization: binarize each weight
  - quantization in general: k-means
  - pruning: zero all but top weights
  - … and combinations thereof.

The C step behaves like a regular compression of a signal and simply requires calling the corresponding compression subroutine as a black box. The loss and training set appear only in the L step.

Quantization with adaptive codebook: C step

In quantization, each real valued weight w_i must equal one of the K values in a codebook $\mathcal{C} = \{c_1, \dots, c_K\} \subset \mathbb{R}$.

- Compressed net: codebook (K floats) + $\lceil \log_2 K \rceil$ bits per weight.
- The decompression mapping $\mathbf{\Delta}$ is a table lookup $w_i = c_{\kappa(i)}$ where $\kappa\colon \{1, \dots, P\} \rightarrow \{1, \dots, K\}$ is a discrete mapping that assigns each weight to one codebook entry.
- Rewriting $\kappa()$ using binary assignment variables $\mathbf{Z} \in \{0,1\}^{P \times K}$:

$$\min_{\mathcal{C},\kappa} \sum_{p=1}^{P} \left\| w_i - c_{\kappa(i)} \right\|^2 \Leftrightarrow \min_{\mathcal{C},\mathbf{Z}} \sum_{i,k=1}^{P,K} z_{ik} \| w_i - c_k \|^2 \text{ s.t. } \begin{cases} \sum_{k=1}^{K} z_{ik} = 1 \\ i = 1, \dots, P. \end{cases}$$

- So $\mathbf{\Theta} = \{\mathcal{C}, \mathbf{Z}\}$, and the decompression mapping is $w_i = \sum_{k=1}^{K} z_{ik} c_k$.
- This is the squared distortion problem. With scalars, it is solvable exactly in polynomial time. Or, approximate solution by k-means.

  Alternating optimization over $\mathcal{C}$ and $\mathbf{Z}$, converges to a local minimum in a finite number of iterations.

Quantization with fixed codebook: C step

Sometimes (e.g. because of fast hardware arithmetic), it is convenient to use a predetermined codebook, often implicit (no need to store it), so $\boldsymbol{\Theta} = \mathbf{Z}$, or equivalently the decompression mapping is $w_i = \theta_i$ s.t. $\theta_i \in \mathcal{C}$:

- Binarization: $\mathcal{C} = \{-1, +1\}$.
- Ternarization: $\mathcal{C} = \{-1, 0, +1\}$.
- Powers of 2: $\mathcal{C} = \{0, \pm 1, \pm 2^{-1}, \dots, \pm 2^{-C}\}$.

The exact solution of the C step is to assign each weight to its closest codebook entry (e.g. binarization: $w_i \rightarrow \operatorname{sgn}(w_i)$).

A partly adaptive codebook can be obtained by learning a scale factor $a > 0$, e.g. $\mathcal{C} = \{-a, +a\}$ or $\{-a, 0, +a\}$. The C step for these can also be solved exactly (e.g. binarization: $a = \frac{1}{P} \sum_{i=1}^{P} |w_i|$, $w_i \rightarrow a \operatorname{sgn}(w_i)$).

However, all these variations are generally worse, in terms of the compression-loss tradeoff, than using an adaptive codebook.

Quantization with fixed assignments: C step

The constraints have the form $w_i = c_{\kappa(i)}$ for each weight w_i, where the assignment mapping $\kappa()$ is fixed, so $\boldsymbol{\Theta} = \mathcal{C}$. Equivalently, the weights associated with the kth centroid must take the same value:

$$w_i = c_k \qquad \forall i \in \{1, \dots, P\}\colon \ \kappa(i) = k$$

The C step has an exact solution where c_k equals the mean of the weights associated with it.

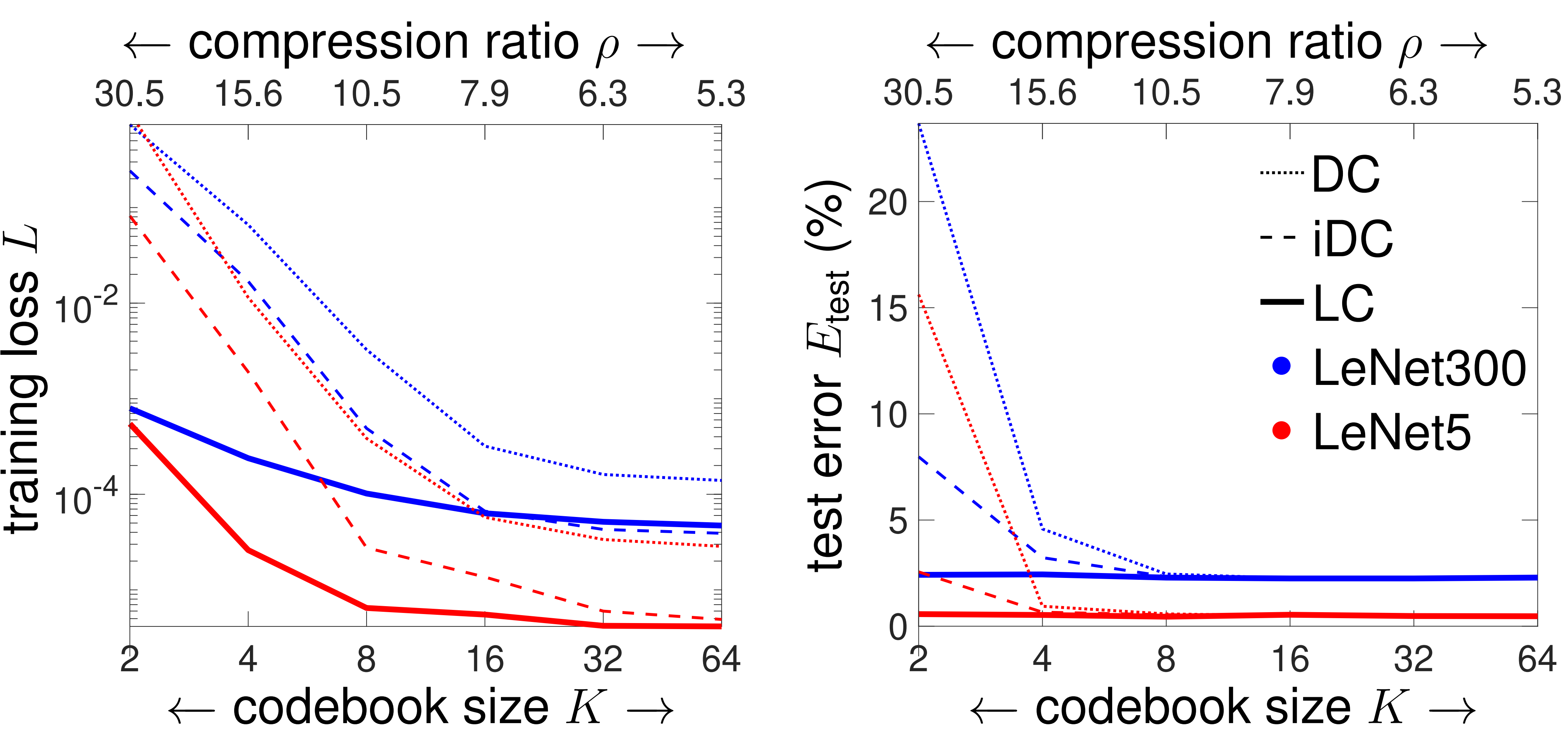
This is the same as weight sharing, a widely successful technique in convolutional neural nets, where we have a priori information about what good assignments should be (homogeneous filters).

Computationally, it is more efficient to eliminate the constraints and apply the chain rule to minimize the loss $L(\mathcal{C})$ directly over $\mathcal{C}$.

Quantization with adaptive codebook: experiments

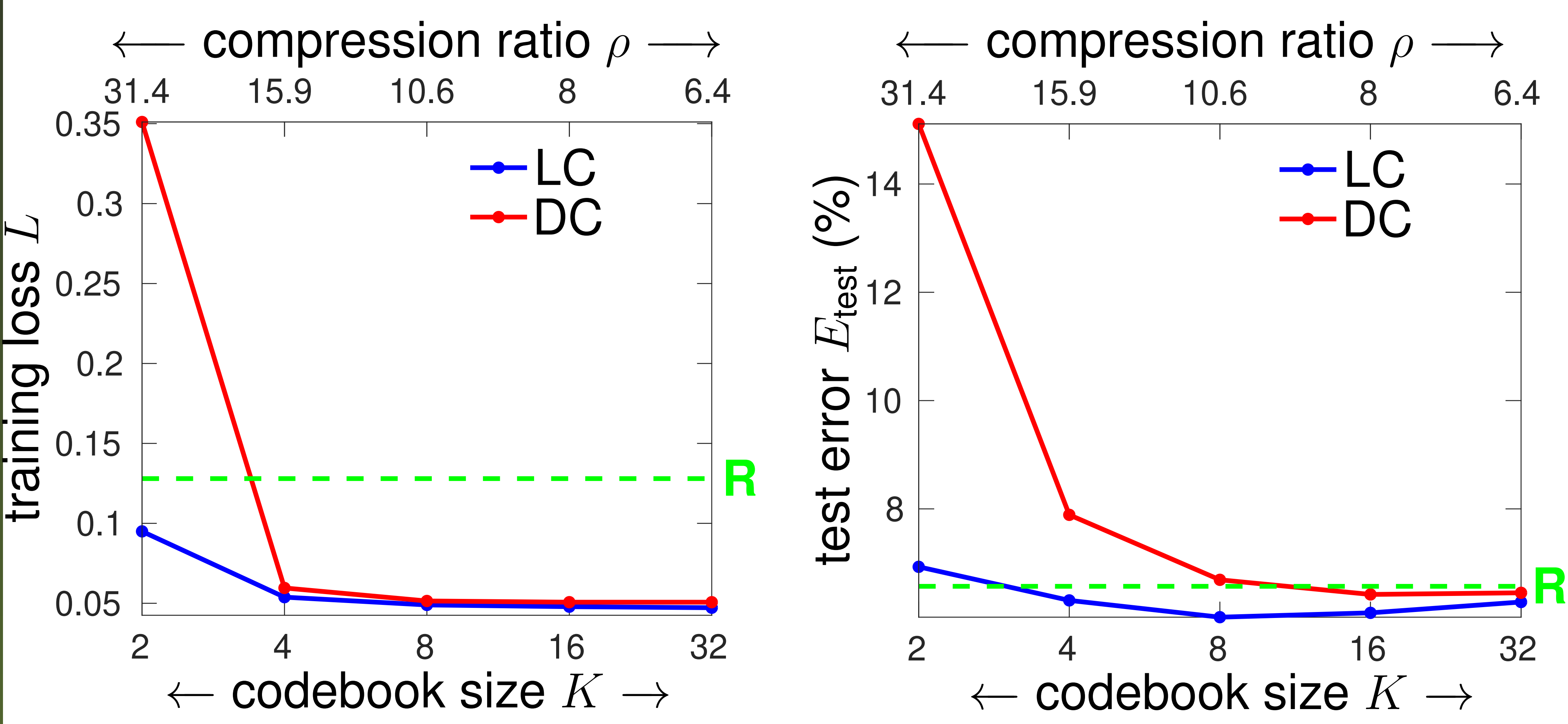
More compression for the same target loss than other algorithms.

LeNet neural nets on MNIST dataset: nearly no error degradation using $K = 2$ (1 bit/weight, compression ratio ×30.5). Error-compression tradeoff curves:



Quantization w/adaptive codebook: experiments (cont.)

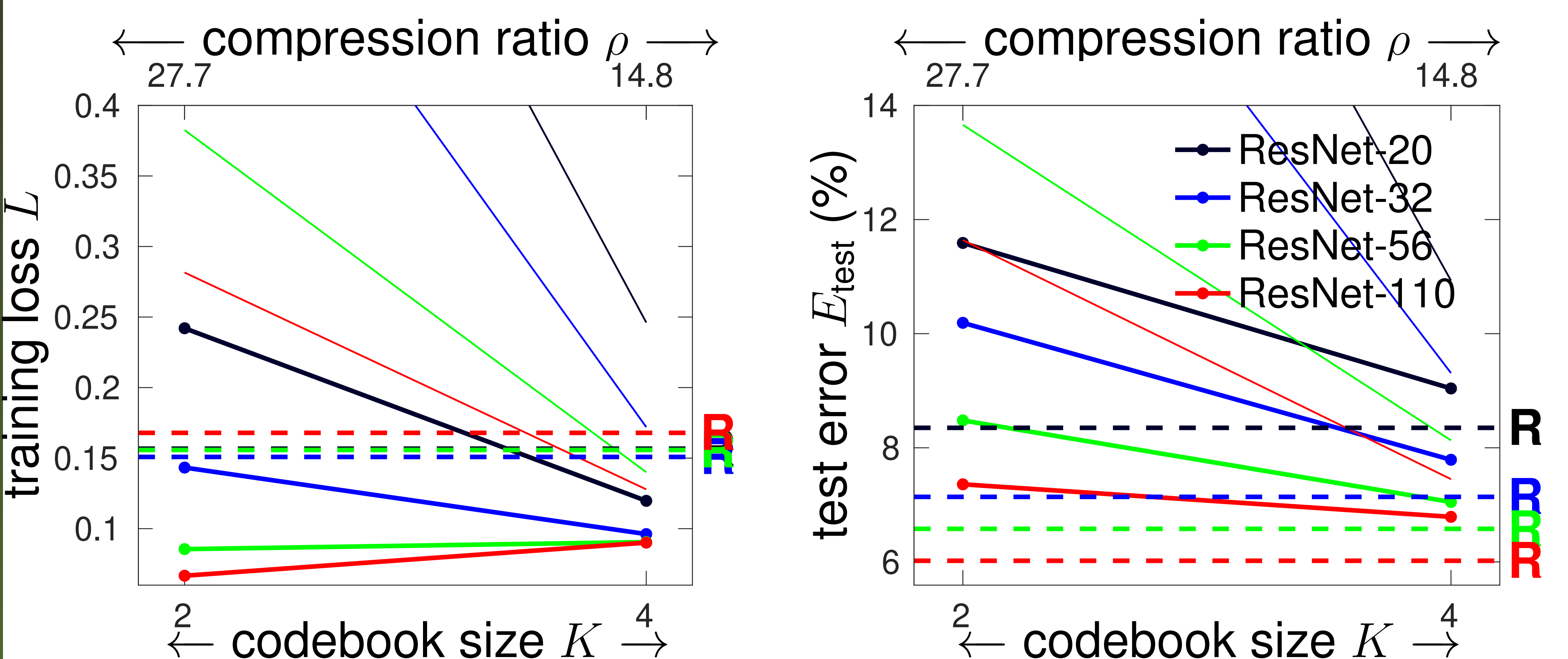
VGG-16 neural net on CIFAR10 dataset (16 layers, 15M weights, 57MB storage): nearly no error degradation using $K = 2$ (1 bit/weight, compression ratio ×30.5), where it would take only 1.85MB and fit on the L2 cache of modern processors.



Quantization w/adaptive codebook: experiments (cont.)

ResNet neural nets on CIFAR10 dataset (20, 32, 56, 110 layers and 0.27M, 0.46M, 0.85M, 1.7M weights, resp.): nearly no error degradation using $K = 4$, some degradation using $K = 2$ (1 bit/weight, compression ratio ×26). ResNets are much leaner models (achieving state-of-the-art classification with a much smaller number of weights), and harder to compress.

LC algorithm: **—**, DC: —, reference net (**R**): - - -.



Pruning: C step

In pruning, the P weights are real-valued but we can have at most κ nonzero weights.

- Compressed net: store nonzero weights (and their indices).
- Decompression mapping: $\boldsymbol{\Delta}(\boldsymbol{\theta}) = \boldsymbol{\theta}$ s.t. $\|\boldsymbol{\theta}\|_0 \leq \kappa$.

  Other formulations possible: penalty instead of a constraint, or other sparsifying norm such as ℓ_1.
- Solution: leave as is the top-κ weights (in magnitude), zero the rest.

  In general, some king of thresholding.
- With a deep net, this automatically finds the optimal number of weights to prune in each layer.

  We need not set a separate pruning parameter for each layer.

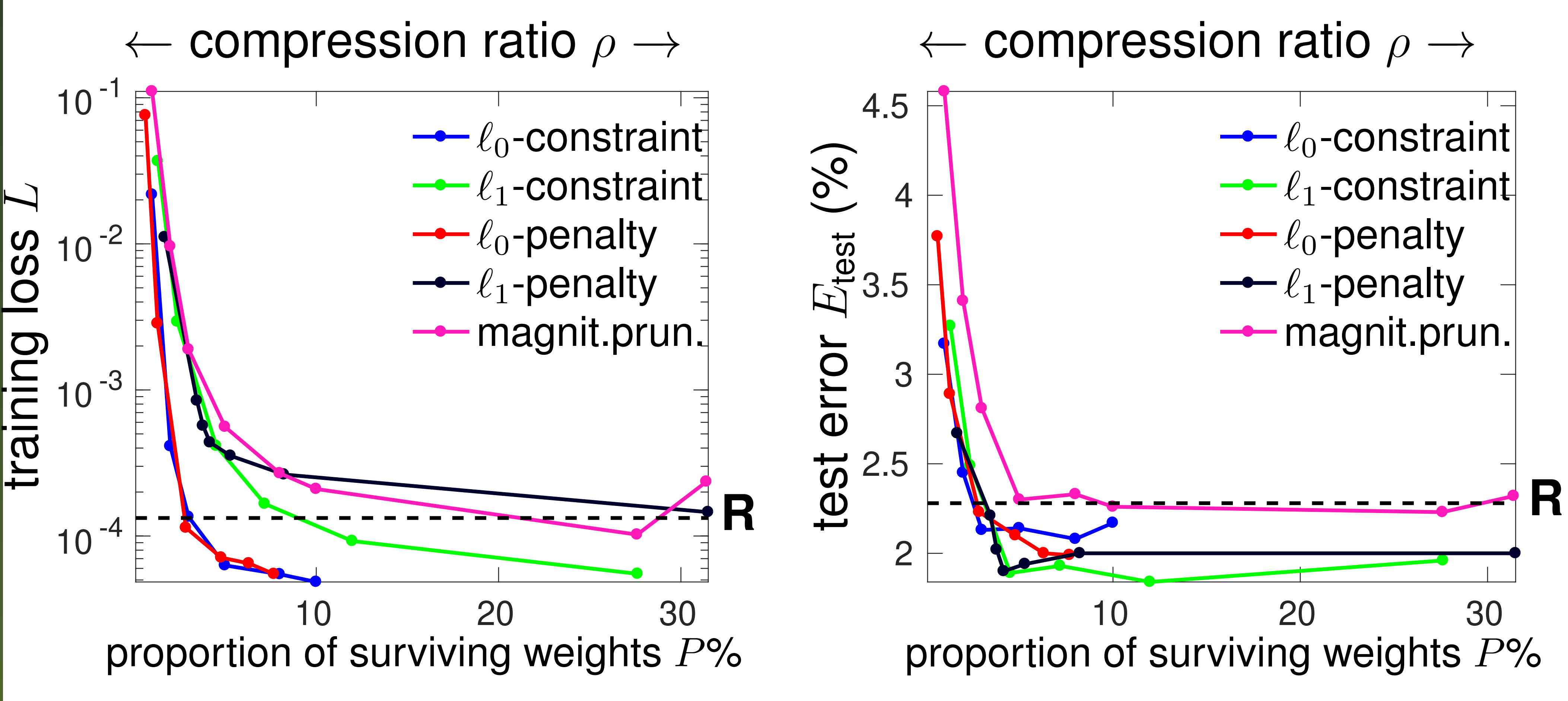
This is like magnitude pruning, but weights are not irrevocably removed. They are marked as currently pruned in the C step (via $\boldsymbol{\theta}$), and this is interlaced with the L step (which drives $\mathbf{w}$ towards $\boldsymbol{\theta}$). So the set of pruned weights changes during LC iterations as we converge.

Magnitude pruning is a simple approach where we remove all but the top-κ weights (in magnitude) of the reference net and retrain the remaining κ weights.

Pruning: experiments

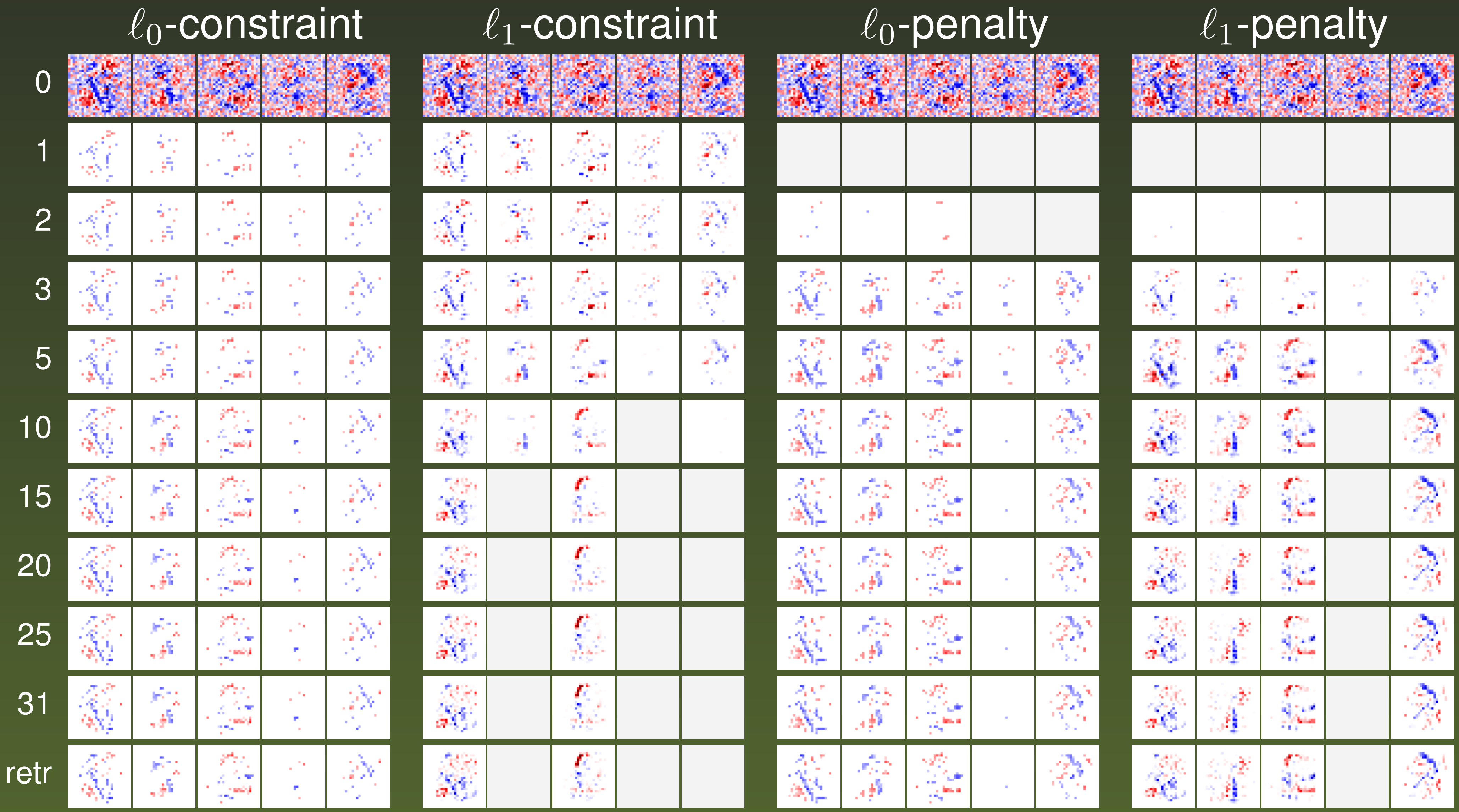
More compression for the same target loss than other algorithms.

LeNet300 neural net on MNIST dataset (3 layers, 266k weights): nearly no error degradation using $P = 2\%$ of the weights. Magnitude pruning removes all but the largest $P\%$ weights and retrains them.



Pruning: experiments (cont.)

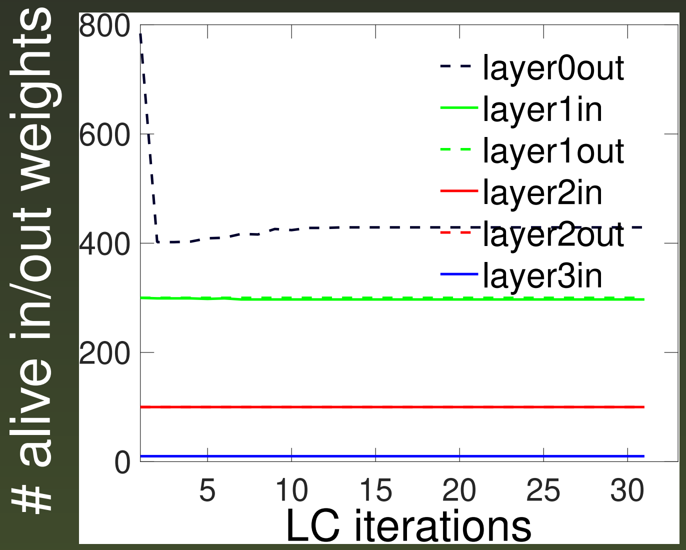
Weight vector of selected first-layer neurons for LeNet300 with $P \approx 5\%$ over iterations (0 = reference, retr = after retraining).



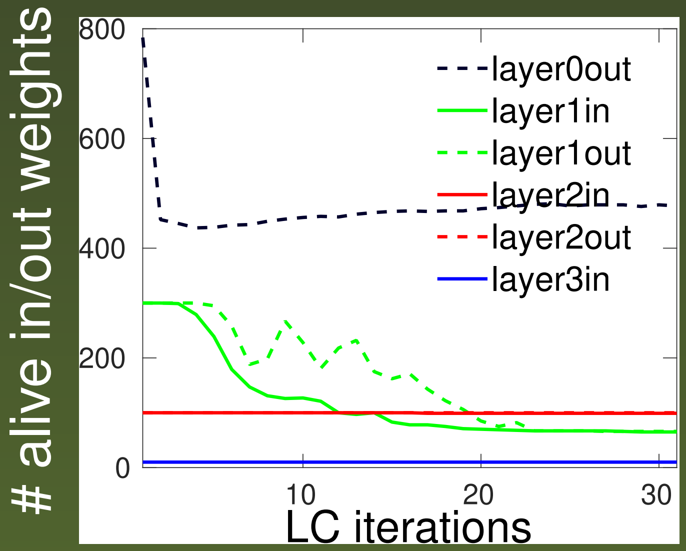
Pruning: experiments (cont.)

- 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.
- The algorithm prunes weights, not neurons, but we observe aggressive neuron pruning, particularly in the first layer. The original LeNet300 architecture 784–300–100–10 becomes 400–64–99–10 with similar error (for the ℓ_1-constraint).
- Hence, the algorithm might be useful to do feature selection and determine the optimal number of neurons in each layer automatically (structure learning).

ℓ_0-constraint



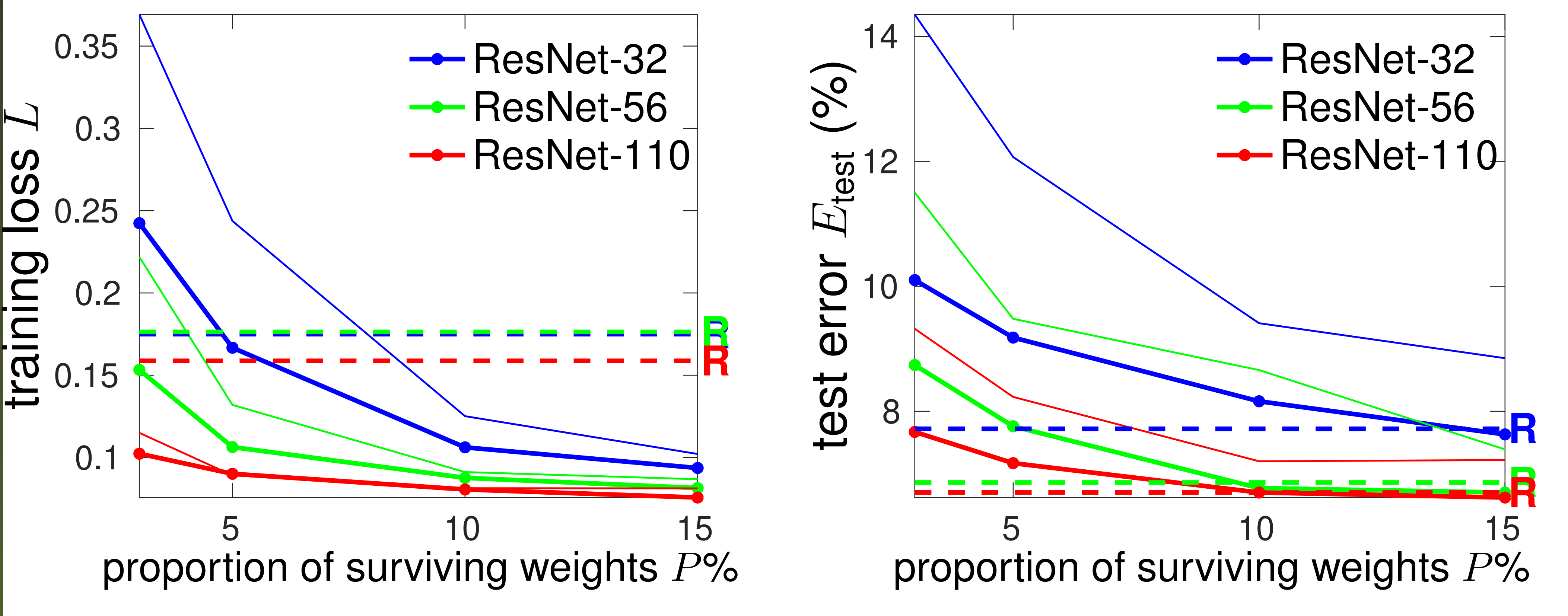
ℓ_1-constraint



Pruning: experiments (cont.)

ResNet neural nets on CIFAR10 dataset (32, 56, 110 layers and 0.46M, 0.85M, 1.7M weights, resp.): we can reach $P \approx$ 5–10% surviving weights with no error degradation.

LC algorithm (ℓ_0-constraint): **—**, magnitude pruning: —, reference net (**R**): - - -.



Quantization vs pruning

Our common framework (LC algorithm) allows us to compare different compression techniques.

Assume:

- A neural net with P_1 multiplicative weights and P_0 biases.
- We don't compress biases since they are more critical and $P_0 \ll P_1$.
- A float takes $b = 32$ bits storage.

Compression ratio $\rho = \frac{\text{\#bits(reference)}}{\text{\#bits(compressed)}} = \frac{(P_1+P_0)b}{\text{\#bits(compressed)}}$.

Quantization (codebook of K entries):

$$\text{\#bits(compressed)} = P_1 \lceil \log_2 K \rceil + (P_0 + K)b \Rightarrow \rho(K) \approx \frac{b}{\log_2 K}.$$

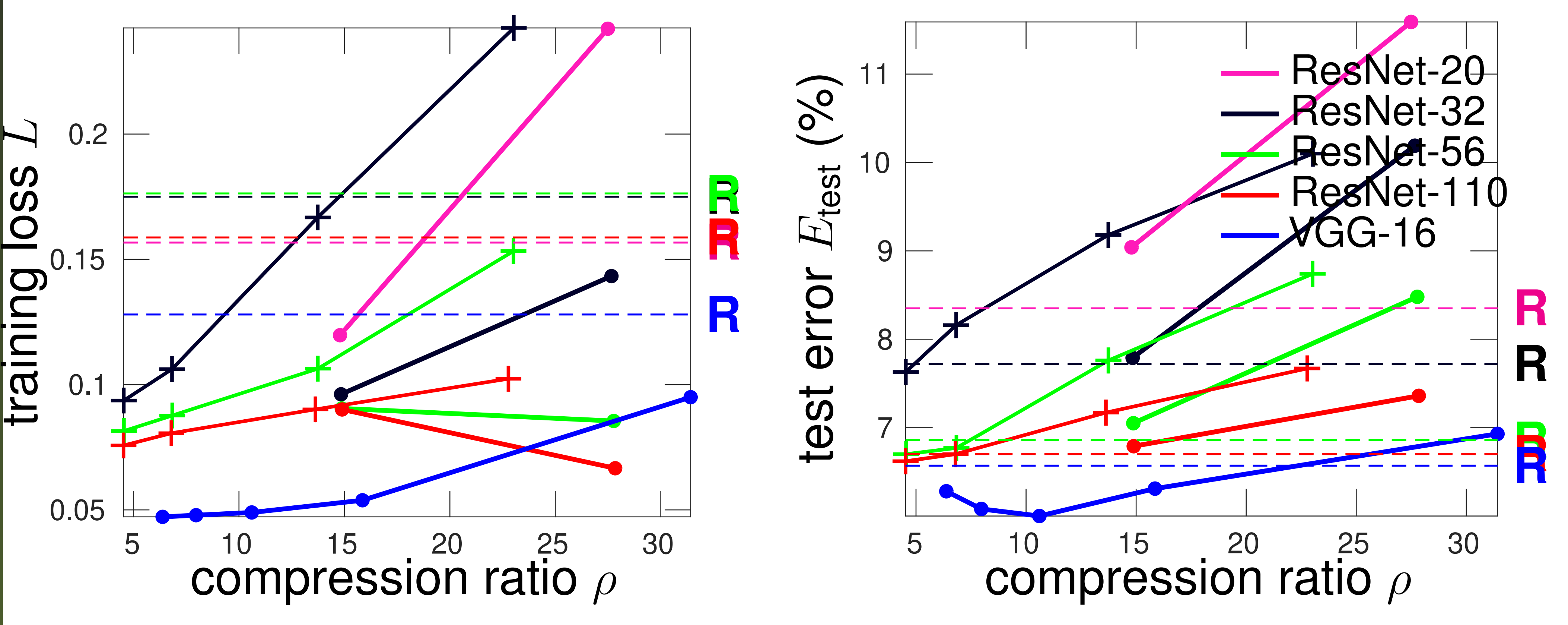
Pruning (P% surviving weights) in coordinate list format (which stores (row,column,value) per nonzero weight) and $b_{\text{idx}} = \text{\#bits(row,column)}$:

$$\text{\#bits(compressed)} = (PP_1 + P_0)(b + b_{\text{idx}}) \Rightarrow \rho(P) \approx \frac{1}{P} \frac{b}{(b+b_{\text{idx}})}.$$

Quantization vs pruning (cont.)

Quantization beats pruning in terms of storage.

Quantization: •, pruning: +, reference net (**R**): - - -.
All results using the LC algorithm.



Beyond compression

- Optimizing runtime, energy, etc. instead of memory
  Hardware retargeting: “compiling” a neural net
- Model compression as structure learning

Optimizing runtime, energy, etc. instead of memory

We can make the “model compression as constrained optimization” framework more flexible by adding a penalty term:

$$\min_{\mathbf{w},\boldsymbol{\Theta}} L(\mathbf{w}) + \lambda\, C(\boldsymbol{\Theta}) \qquad \text{s.t.} \qquad \mathbf{w} = \boldsymbol{\Delta}(\boldsymbol{\Theta})$$

For example, $C(\boldsymbol{\Theta})$ can be the runtime or energy of the inference pass in a neural net (based on the number of arithmetic and data movement operations). Application: learning the rank of each layer automatically.

Hardware retargeting: deploy a given reference model on a target device making optimal use of its system architecture, e.g.:

- optimal layout (e.g. storing $\boldsymbol{\Theta}$ vs $\boldsymbol{\Theta}^T$)
- optimal use of memory system (registers, caches, etc.)
- optimal scheduling of threads

These can be encoded as constraints or penalties in the framework, and have the LC algorithm search over combinations of parameters to find a (local) optimum (if we can solve the C step): a “neural net compiler”.

Model compression as structure learning

Compression is a sophisticated form of regularization:

$$\min_{\mathbf{w},\boldsymbol{\Theta}} L(\mathbf{w}) \qquad \text{s.t.} \qquad \mathbf{w} = \boldsymbol{\Delta}(\boldsymbol{\Theta})$$

$$\min Q(\mathbf{w},\boldsymbol{\Theta};\mu) = L(\mathbf{w}) + \tfrac{\mu}{2}\left\|\mathbf{w} - \boldsymbol{\Delta}(\boldsymbol{\Theta})\right\|^2$$

- The regularization term "$\left\|\mathbf{w} - \boldsymbol{\Delta}(\boldsymbol{\Theta})\right\|^2$" is itself parameterized.

  Cf. weight decay / ridge regression.
- The parameters $\mathbf{w}$ must take a particular structure which is itself learned (which weights are nonzero, which weights are equal, etc.).

We could use this to achieve better generalization rather than (just) compression. In fact, a better reference model (lower test error) is often achieved by applying a small amount of compression to the original reference (slightly smaller K, P, r).

So we should always use compression as a postprocessing step to get a better model (not necessarily for actual compression).

But we can take this further. . .

Model compression as structure learning (cont.)

Traditionally, the structure of the model is designed by hand and its size optimized via model selection. The framework above suggests a different approach:

1. Define a “blank slate” model, large and over-parameterized.

   Ex: a neural net with many layers & units.

2. Learn its structure and parameter values using the LC algorithm.

   - removing connections (pruning)
   - forcing subsets of parameters to be equal (quantization)
   - thinning layers (low-rank)

3. The “size” (effective number of parameters) is given by the compression level (K, P, r), which could be cross-validated.

Indeed, particular cases of these are well-known in statistics (usually applied with linear models):

- Sparsity: LASSO and related models.
- Low-rank: reduced-rank regression (RRR).

Conclusion

- A precise mathematical formulation of neural net compression as a constrained optimization problem.
- A generic way to solve it, the learning-compression (LC) algorithm.

  A single algorithm handles any combination of compression technique and loss function.

  - The compression technique appears as a black-box subroutine in the C step, independent of the loss and dataset.

    We can try a different type of compression by simply calling its subroutine.
  - The loss and neural net training by SGD appear in the L step, independent of the compression technique.

    As if we were training the original, reference net with a weight decay.
- Easy to implement in deep learning toolboxes.
- Convergence to local optimum under some assumptions.
- Very effective in practice: not much slower than training the reference model, and achieves more compression for the same target loss than direct compression and other approaches.
- Extensions: optimal runtime, energy or hardware retargeting.

References and code

http://faculty.ucmerced.edu/mcarreira-perpinan

General framework:

- M. Á. Carreira-Perpiñán: *Model compression as constrained optimization, with application to neural nets. Part I: general framework*.
  arXiv:1707.01209, Jul. 5, 2017.

We are developing this framework for number of compression types:

- C-P and Y. Idelbayev: *Model compression as constrained optimization, with application to neural nets. Part II: quantization*.
  arXiv:1707.04319, Jul. 13, 2017.
- C-P and Y. Idelbayev: *Model compression as constrained optimization, with application to neural nets. Part III: pruning*.
  arXiv, 2018.
- C-P and Y. Idelbayev: *Model compression as constrained optimization, with application to neural nets. Part IV: low-rank*.
  arXiv, 2018.
- C-P and Y. Idelbayev: *"Learning-Compression" algorithms for neural net pruning*.
  CVPR, 2018.
- More to come.

Partly supported by NSF award IIS–1423515, NVIDIA GPU donations and MERCED cluster (NSF grant ACI–1429783).