LC: A Flexible, Extensible Open-Source Toolkit for Model Compression

Yerlan Idelbayev
yidelbayev@ucmerced.edu
University of California, Merced
Merced, CA, USA

Miguel Á. Carreira-Perpiñán
mcarreira-perpinan@ucmerced.edu
University of California, Merced
Merced, CA, USA

ABSTRACT

The continued increase in memory, runtime and energy consumption of deployed machine learning models on one side, and the trend to miniaturize intelligent devices and sensors on the other side, imply that model compression will remain a critical need for the foreseeable future. A scalable solution to this problem must be able to handle arbitrary choices of the reference model to be compressed (driven by the machine learning task), of the form of compression to use, and of the costs and constraints to obey (driven by the target device). We describe an open-source toolkit that is primarily designed to be flexible and extensible, but which is also efficient in compression time and achieves state-of-the-art accuracy-compression curves, as demonstrated empirically over a number of deep net architectures. Mathematically, this is achieved by formulating compression as a constrained optimization using auxiliary variables that facilitate separability, and solving it via a penalty method and alternating optimization, which results in a "learning-compression" (LC) algorithm. This alternates a "learning" step over the original model, independent of the compression, and a "compression" step over the compressed parameters, independent of the dataset and task. Each step can typically be solved by reusing well-known algorithms, such as SGD or EM in the learning step, or SVD or k-means in the compression step, and this makes the algorithm flexible and extensible. The toolkit is available at https://github.com/UCMerced-ML/LC-model-compression.

CCS CONCEPTS

• **Computing methodologies** → *Machine learning*; • **Software and its engineering** → **Software libraries and repositories**.

KEYWORDS

model compression framework; neural network compression

ACM Reference Format:
Yerlan Idelbayev and Miguel Á. Carreira-Perpiñán. 2021. LC: A Flexible, Extensible Open-Source Toolkit for Model Compression. In *Proceedings of the 30th ACM International Conference on Information and Knowledge Management (CIKM '21), November 1–5, 2021, Virtual Event, QLD, Australia.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3459637.3482005

1 INTRODUCTION

The last few years have seen model compression, in particular neural net compression, arise as a real-world need for practical deployment of machine learning (ML) models in limited-resource devices. On the one hand, very large and ever increasing models have been progressively improving their accuracy in computer vision, speech and natural language processing applications, but at the cost of a huge number of parameters (millions or billions), and a correspondingly large runtime and energy consumption during inference. On the other hand, we want intelligent devices (such as mobile phones, biomedical sensors and other IoT devices) that will run these models to become as small as possible, so they are portable, even implanted inside the body or deployed in remote areas, and consume as little energy as possible (so they require infrequent battery charging)—while remaining accurate and fast. And this can be expected to be a neverending arms race, with models becoming larger and devices smaller.

An open-source toolkit providing an effective solution to the problem of model compression will benefit a large user population. Any companies that deploy deep neural nets or other ML models in limited-computation devices will want to distribute the leanest possible model for each device. For example, web search and social network apps in mobile phones (from companies such as Google, Facebook, Tencent, Qualcomm, etc.); image processing and computer vision networks in mobile robots; biomedical sensors in portable devices; etc. Such a toolkit will also be useful for researchers and educators in neural net compression. Finally, model compression can be of independent interest beyond limited-computation devices, because compression can act as a form of regularization that improves generalization.

However, an effective model compression toolkit must address an important issue: the large amount of existing ML models, tasks, loss functions and corresponding training algorithms; and the many compression techniques available. For example, if we want to compress a ResNet trained with Nesterov accelerated gradient descent using weight quantization, we may (with some effort) design a specific algorithm to solve that. Such algorithm might involve, say, a modification of backprop to truncate the weights on the fly. But a very different algorithm would be necessary to sparsify the covariance matrices of a Gaussian mixture (perhaps some modification of EM to threshold the parameters), or to use low-rank factorization in linear regression (a problem called reduced-rank regression which can actually be solved in closed form; 41). In turn, while the ML model is guided by a task and loss function (say, classification via cross-entropy), the compression should be guided by the target device (e.g. memory size and hierarchy, number of GPUs or CPUs, etc.) and desirable costs (such as memory, runtime or energy).

This problem is reminiscent of that of translating a program in a high-level language (say, Quicksort in C) to a binary code in a target system (say an Intel CPU with Linux). Designing a specific translator from language X to system Y is possible, and potentially result in very efficient code, but does not scale when we have many languages and systems. Instead, modern compilers such as LLVM [31] solve this by having a front-end that translates the high-level program to an intermediate representation and a back-end that translates this to binary code in the target computer. While the resulting binary code may lose some performance, this is by far compensated by the gain in flexibility and extensibility, and in programmer productivity. It also facilitates modularity and reusability, which lead to scalable, reliable software engineering. The analogy is not perfect because traditional compilers must ensure the resulting binary code (the compressed model in our case) *correctly* implements the high-level program (the reference, uncompressed model), i.e., *lossless compression*. However, model compression allows trading off some accuracy for some compression, i.e., *lossy compression*. This tradeoff is controlled by a hyperparameter, as described later.

Hence, while solving specific model compression problems will remain an area of research interest, *we advocate a generic, scalable approach*. This is necessary if we want to use a compression technique Y on a model X and there are multiple possibilities for X and Y. Our proposed toolkit was designed with this as a primary goal. Mathematically, this is achieved by a (quite natural) constrained optimization formulation of the problem of model compression using auxiliary variables, which is solved using a penalty method and alternating optimization (see section 3). This results in a "learning-compression" (LC) algorithm, which iteratively alternates the solution of two well-defined problems. One step has the form of a standard ML problem using the original loss and dataset but with an additional regularization term, which can be solved using an existing algorithm to train the original model (the *learning (L) step*). The other step has the form of a standard signal processing problem using the squared distortion, involving the model parameters but not the loss or dataset, which can be solved using an existing algorithm for the chosen form of compression (the *compression (C) step*). Reusing existing algorithms means we capitalize on well-studied problems for which often efficient implementations are available. This saves time, makes the overall algorithm efficient, and facilitates debugging and maintenance. Adding a new ML model (and its corresponding training algorithm) to the toolkit makes it possible to apply any existing compression technique to this model. Conversely, adding a new compression technique to the toolkit makes it applicable to any model in the toolkit. The details of this are more clear in section 3, where we give explicitly the compression problem formulation (involving the ML task, loss function and dataset, and the cost function and compression constraints to obey), and the LC algorithm. Hence, our LC toolkit has the following important advantages:

- **Flexibility.** A user can select an arbitrary model/task/loss (e.g. neural nets, HMMs), cost function (e.g. inference time or energy) and compression technique (e.g. pruning, quantization, low-rank). They are all handled in a scalable way by the toolkit. Further, different compression techniques can be additively combined (such as low-rank plus sparse).
- **Extensibility.** New ML models (and their corresponding training algorithms) or compression techniques (and their corresponding algorithms) can be easily added.
- **Reusability of algorithms and code.** Algorithmic reusability happens because the problems defined in the L and C steps may be solved with different algorithms (each with its own pros and cons). For example, scalar square-distortion quantization can be solved exactly with dynamic programming and approximately with k-means. Code reusability for a given algorithm happens because of the availability of specialized libraries (such as BLAS or ATLAS for linear algebra computations) or of highly optimized systems-level accelerations for either specific ML models or compression techniques.
- **Modularity.** Each ML model (task and loss function) is a separate module, as is each compression type. Hence, changing the model and/or the compression simply involves calling the corresponding module's routine.
- **Usability.** In practice, one does not know what type of compression is best for a given model. Our toolkit offers, within the same framework of the LC algorithm, multiple models and multiple compression types (and more will be added) that a user can try or combine with minimum engineering effort. Or, if a user wants to try a specific choice of model and compression, all is needed is to pick a corresponding L step and C step. It is not necessary to create or look for a specific algorithm to handle that choice of model and compression. Further, the toolkit can produce a range of compressed models spanning an accuracy-compression tradeoff curve, from which the user can choose an optimal operating point.
- **Efficiency.** Empirically over many types of deep nets, we observe that compressing a model does not take much longer than training the uncompressed model in the first place. We also plan to offer a very fast (but less effective) compression that does not require access to the training set [7].
- **Theoretical guarantees.** The LC algorithm is based on solid optimization principles that guarantee that we find a local optimum of the (usually nonconvex, often nondifferentiable) compression problem. In some special cases, the LC algorithm is equivalent to an existing algorithm, e.g. for linear regression with ℓ_1 pruning it is equivalent to a leading Lasso algorithm [16, p. 122].
- **Unique characteristics.** The LC algorithm handles some unusual types of compressions effectively, such as automatically learning the optimal rank of each layer in a deep net when using low-rank compression (a combinatorial problem) [19], and additively combining or selecting different types of compression for each layer [22], so they best cooperate to compress the model. This makes it possible to automate the discovery of optimal compression types for, say, convolutional layers vs. fully-connected layers.
- **State-of-the-art compression performance.** As noted above, the choice of a common framework that handles all models and compressions means that we may lose performance compared to a custom algorithm for a specific model and compression. However, our by now extensive experiments show the LC algorithm is in fact comparable or better than alternative compression algorithms for specific cases proposed in the literature.

Besides the obvious practical applicability of our toolkit, one important insight that it makes possible is to understand what forms of compression are more suitable to what model, or even to what layer of a deep net, because they can be compared in an apples-to-apples way. For example, many deep nets have convolutional layers (with few parameters but lots of FLOPs) and fully-connected layers (with many parameters and few FLOPs). Is quantization better than pruning for the convolutional layers, say? As another example, in a hidden Markov model we have parameters of very different type: the transition matrix (nonnegative, stochastic), the Gaussian covariance matrix (positive definite, possibly shared across states), etc. One expects that each of these types of parameters will respond better to a specific compression strategy. Learning about this is facilitated by our common algorithmic framework, where the actual model training (in the L step) is decoupled from the parameter compression. For example, to compare compressions of a given neural net, we try different compressions (hence different C steps), but all decisions about training the net (for example, SGD hyperparameters such as learning and momentum rate and minibatch size) are confined to the L step and kept fixed. A first work exploring this is [21].

The LC algorithm was originally proposed in 2017 in its generic form [4]. Since then, we have developed it for a number of compression types, including several forms of pruning (sparsity), quantization and low-rank (or tensor) factorization, and evaluated it for several well-known deep net architectures (ResNets, VGG, AlexNet, LeNet, etc.), as described in a series of papers [4–7, 19–24]. Our code for it has continuously evolved, from a first version in Theano only for pruning [6] to our current version in Python and PyTorch, which we have made recently available in Github as open source under the BSD 3-clause license. The toolkit is an evolving work. In the immediate future, we plan on incorporating fast model compression based on a dataset-less approximation to the ML task [7], and models beyond neural nets (in particular softmax classifiers, Gaussian mixtures and HMMs) trained with algorithms such as SGD or the EM algorithm. We also welcome contributions from the community in the form of new neural net architectures and ML models, and new compression techniques. The toolkit can also be used to construct strong baselines with which to compare other compression algorithms. We also plan to provide such baseline results for many different models in the toolkit website.

In this paper, we report on the current form of the toolkit. We survey the LC framework (sections 3–4), discuss the software design and the functionality provided (section 5), and survey representative experimental results (section 8) in the context of the state-of-the-art in deep net compression.

2 RELATED WORK

The field of model compression has grown enormously in the recent years resulting in plethora of algorithmic approaches, research projects and software. At present, many ad-hoc solutions have been proposed that typically solve *only one specific type of compression*: quantization [5, 40, 59], pruning [15, 37, 46], low-rank decomposition [11, 12, 26, 34, 43, 44, 47, 52, 53, 57] or tensor factorizations [12, 32], and others. In this section we limit our attention to the software aspect of the neural network compression and overview the supported compression schemes among the software, available codes, and recently proposed compression frameworks.

Individual compressions. The majority of neural network compression research is available as individual projects and recipes tailored for a particular compression and model. Usually it is released as a companion code for published research paper, e.g. see [44, 45, 52]. Some repositories combine several compression recipes in a single place: e.g., Tensorpack[1] or the fork of the Caffe library by Wei Wen[2].

Out of many individual compressions proposed in the literature, the quantization aware training of Jacob et al. [25] has gained popularity and became a standard feature of major deep-learning frameworks. TensorFlow, Pytorch, MxNet and others independent projects like ADaPTION [39], Mayo [58], FINN-R[2] and TensorQuant [38] natively support both training of such quantized models and allow an efficient inference afterwards.

Efficient inference frameworks. Relatively mature software is available if the goal is not to compress the model (by changing the weights accordingly), but to run the model as efficiently as possible on a given hardware. Many frameworks target the mobile deployment regime and allow to convert (compile) already trained neural network to utilize the hardware-enabled fast computations: for instance, through usage of edge TPU-s on Pixel 4 (Pixel Neural Core) or Neural Engine on iPhones. Examples of such frameworks include Tensorflow Lite[3], PyTorch Mobile[4], Apple Core ML[5], Nvidia's TensorRT[6], Qualcomm's Neural Processing SDK[7], Xlinix's FINN[8], Facebook's QNNPack[9], and many others.

A generalization of this concept is to efficiently deploy and compile the dataflow of the inference/backward pass for an arbitrary set of hardware. Some examples include packages like Facebook's Glow[10], Google's XLA[11], or Apache TVM[12].

Compression frameworks. The diversity of compression mechanisms and limited support by deep learning frameworks led to the development of specialized software libraries such as Distiller [62], NCCF [29], and PocketFlow [49]. These frameworks gather multiple compression schemes and corresponding training algorithms into a single framework, and make it easier to apply the compressions to new models. Some of these frameworks allow to apply multiple compression simultaneously to disjoint parts of a single model, however most of the supported schemes can be applied with per-model granularity only. Additionally, the underlying compression algorithms do not share the same algorithmic base thus requiring a substantial understanding of many hyper-parameters for every compression-algorithm pair to efficiently tune the settings.

[1]https://github.com/tensorpack/tensorpack/tree/master/examples
[2]https://github.com/wenwei202/caffe
[3]https://www.tensorflow.org/lite
[4]https://pytorch.org/mobile/home/
[5]https://developer.apple.com/documentation/coreml
[6]https://developer.nvidia.com/tensorrt
[7]https://developer.qualcomm.com/software/qualcomm-neural-processing-sdk
[8]https://xilinx.github.io/finn/
[9]https://engineering.fb.com/ml-applications/qnnpack/
[10]https://ai.facebook.com/tools/glow
[11]https://tensorflow.google.cn/xla?hl=en
[12]https://tvm.apache.org/

There exists a third-party implementation (called Condensa) of our LC algorithm done by NVIDIA's Research Lab [27]. However, Condensa is restricted to pruning and quantization only, without the more advanced capabilities of our toolkit (mix-and-match combinations, large choice of compression techniques, extensibility).

3 MODEL COMPRESSION AS A CONSTRAINED OPTIMIZATION

In this section, we briefly overview the Learning-Compression (LC) framework, which is the backbone of our software. Let us begin by assuming we have a previously trained model with weights $\mathbf{w}$, which were obtained by minimizing some loss function $L(\mathbf{w})$. This is our *reference* model, which represents the best loss we can achieve without compression. Here we omitted the exact definition of the weights $\mathbf{w}$, but for now, let us assume it has P parameters. In the learning-compression framework, the compression is defined as finding a low-dimensional parameterization $\mathbf{\Delta}(\mathbf{\Theta})$ of the weights $\mathbf{w}$ in terms of Q-sized parameter $\mathbf{\Theta}$, with $Q < P$.

In the framework, the compression and decompression are regarded as mappings, while in the signal processing literature they are usually seen as algorithms, e.g., lossless compression algorithm of Ziv and Lempel [61]. Formally, the *decompression mapping* $\mathbf{\Delta}$ maps a low-dimensional parameters $\mathbf{\Theta}$ to the uncompressed model weights $\mathbf{w}$:

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

and the *compression mapping* behaves as its "inverse":

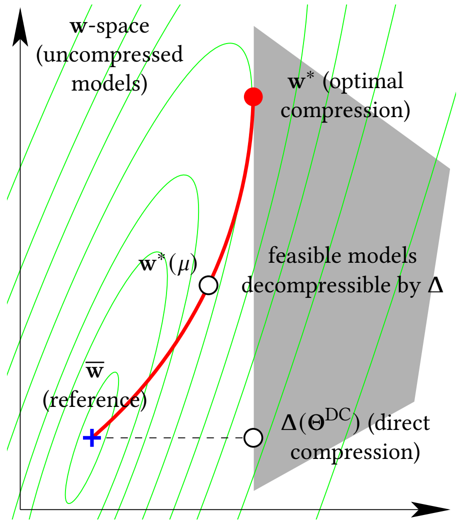
$$\mathbf{\Pi}(\mathbf{w}) = \arg\min_{\mathbf{\Theta}} \|\mathbf{w} - \mathbf{\Delta}(\mathbf{\Theta})\|^2.$$

The goal of model compression is to find such $\mathbf{\Theta}$ that its corresponding decompressed model has (locally) optimal loss for a cost of interest. Therefore the *model compression as a constrained optimization* problem is defined as:

$$\min_{\mathbf{w},\mathbf{\Theta}} L(\mathbf{w}) + \lambda C(\mathbf{\Theta}) \quad \text{s.t.} \quad \mathbf{w} = \mathbf{\Delta}(\mathbf{\Theta}). \tag{1}$$

Here, the term $\lambda C(\mathbf{\Theta})$ with $\lambda > 0$ is intended to represent the cost of the deployed compressed model in terms of quantities of interest: energy, size, compute, etc. The problem in equation (1) is constrained, nonlinear, and usually non-differentiable with respect to the compression parameters $\mathbf{\Theta}$ (e.g., when compression is binarization). To efficiently solve it, the LC-algorithm is obtained by converting this problem to an equivalent formulation using penalty methods (quadratic penalty or augmented Lagrangian) and employing an alternating optimization. This results in an algorithm that alternates two generic steps while slowly driving the penalty parameter $\mu \rightarrow \infty$:

- **L (learning) step:** $\min_{\mathbf{w}} L(\mathbf{w}) + \frac{\mu}{2}\|\mathbf{w} - \mathbf{\Delta}(\mathbf{\Theta})\|^2$.
  This is a regular training of the uncompressed model but with a quadratic regularization term. *The L step is independent from the form of chosen compression.*
- **C (compression) step:** $\min_{\mathbf{\Theta}} \|\mathbf{w} - \mathbf{\Delta}(\mathbf{\Theta})\|^2 + \lambda C(\mathbf{\Theta})$.
  When $\lambda = 0$ the solution of the C step is $\mathbf{\Theta} = \mathbf{\Pi}(\mathbf{w})$. It means finding the best lossy compression of the current uncompressed model weights $\mathbf{w}$ in the ℓ_2 sense: the solution is given by orthogonal projection on the feasible set. The C-step solution depends on the actual form of of the compression scheme $\mathbf{\Delta}(\mathbf{\Theta})$ and the structure of the cost $C(\mathbf{\Theta})$, *yet, it is independent of the model loss and does not require training dataset.*



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

Figure 1: The illustration of the model compression definition given by problem (1). The loss function $L(\mathrm{w})$ is defined over entire w-space, depicted with green contours, and has a minimum at point $\overline{\mathrm{w}}$. The space of decompressible models (given by the form of of Δ) is illustrated in gray. Directly compressing the pre-trained model by setting $\Theta^{\mathbf{DC}} = \Pi(\overline{\mathrm{w}})$ results in sub-optimal solution. To obtain the constrained minima of the problem (the point w^*), the LC algorithm alternates between L and C steps while driving parameter $\mu \rightarrow \infty$, which follows the path $\mathrm{w}^*(\mu)$.

We will be using the quadratic penalty (QP) formulation throughout this paper to make derivations easier. Yet, in practice, we implement the augmented Lagrangian (AL) version which has an additional vector of Lagrange multipliers $\boldsymbol{\beta}$, see Figure 2. The QP version can be obtained from the AL version by setting $\boldsymbol{\beta} = \mathbf{0}$ and skipping the multipliers update step. Figure 1 illustrates the idea of model compression as constrained optimization, and depicts the traced solution $\mathbf{w}^*(\mu)$ during the optimization.

Our software capitalizes on the separation of the L and C steps: to apply a new compression mechanism under the LC formulation, the software requires only a new C step corresponding to this mechanism. Indeed, the compression parameter $\mathbf{\Theta}$ enters the L step problem as a constant regardless of the chosen compression type. Therefore, all L steps for any combination of compressions have the same form. Once the L step has been implemented for a model, any possible compression (C steps) can be applied.

More importantly, this separation allows using the best tools available for each L and C steps. For modern neural networks, the L step optimization means performing iterations over the dataset (for SGD) and requires hardware accelerators. The formulation of the C step, on the other hand, is given by ℓ_2 minimization, and as

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

```python
class LCAlgorithm():
  # Housekeeping code ...
  # Pretrained model is provided by user at initialization
  def run(self):
    self.mu = 0
    self.c_step(step_number=0)
    for step_n, mu in enumerate(self.mu_schedule):
      self.mu = mu
      self.l_step(step_n) # call user-provided L step
      self.c_step(step_n) # resolve compression tasks
      self.multipliers_step()
```

Figure 2: *Left*: The pseudocode of the LC algorithm using the augmented Lagrangian formulation. *Right*: corresponding implementation in our software (located in `LCAlgorithm` class); the main running method is shown.

we will see in the next chapter, its solutions can be computed using efficient algorithms. In fact, for certain compression choices, the C-step problem is well studied and has a history of its usage on its own merit in the fields of data and signal compression. From the software engineering perspective, the separation of L and C steps makes code robust and allows us to thoroughly test and debug each component separately.

Our approach is based on solid optimization principles, with guarantees of convergence under standard assumptions. It formulates the problem of model compression in a way that is intuitive and amenable to efficient optimization. The form of the actual algorithm is obtained systematically by judiciously applying mathematical transformations to the objective function and constraints. For example, if one wants to optimize the cross-entropy over a certain type of neural net, and represent its weights via a quantized codebook, then the L and C steps necessarily take a specific form. If one wants instead to represent the weights via low-rank matrices, a different C step results, and so on. The resulting algorithm is not based on combining backpropagation training with heuristics, such as pruning weights on the fly, which may result in suboptimal results or even non-convergence. The user does not need to work out the form of individual L or C steps (unless so desired), as the toolkit already provides a range to choose from. For further details, we refer the reader to our original papers [4–7, 19–24].

4 SUPPORTED COMPRESSIONS

In this section, we describe some of the compression schemes supported by our library. The complete list of supported compressions is given in Table 1. We expect to add more compressions in the future.

4.1 Quantization

The quantization is the process of reducing the precision of the weights, and it is achieved by imposing a constraint on each weight $w_i \in \mathbf{w}$ to belong to a set of pre-defined or learned values C — the codebook. Depending on the allowed values in the codebook, the quantization schemes are known under different names: it is called binarization with $C = \{0, 1\}$ or $\{-1, 1\}$ and ternarization with $C = \{-1, 0, +1\}$.

Let us consider the general case when we compress the weights of the model with a learned codebook of size K, i.e., $C = \{c_1, \dots, c_K\}$. We will use an equivalent formulation of the quantization using

Type	Forms
Quantization	Adaptive Quantization into $\{c_1, c_2, \dots, c_K\}$
	Binarization into $\{-1, 1\}$ and $\{-c, c\}$
	Ternarization into $\{-c, 0, c\}$
Pruning	ℓ_0-constraint (s.t., $\|\mathbf{w}\|_0 \le \kappa$)
	ℓ_1-constraint (s.t., $\|\mathbf{w}\|_1 \le \kappa$)
	ℓ_0-penalty ($\alpha\|\mathbf{w}\|_0$)
	ℓ_1-penalty ($\alpha\|\mathbf{w}\|_1$)
Low-rank	Low-rank compression to a given rank
	Low-rank with *automatic* rank selection for FLOPs
	Low-rank with *automatic* rank selection for storage
Additive Combinations	Quantization + Pruning
	Quantization + Low-rank
	Pruning + Low-rank
	Quantization + Pruning + Low-rank

Table 1: Currently supported compression types, with their exact forms. These compression can be defined per one or multiple layers, and different compression can be applied to different parts of the model.

a binary assignment variables $\mathbf{z}_i$ such that $(\sum_k z_{ik} = 1)$ for each weight w_i. Then our compression goal can be written as:

$$\min_{\mathbf{w}, C, \mathbf{z}_1, \dots \mathbf{z}_P} \quad L(\mathbf{w}) \quad \text{s.t.} \quad w_i = \sum_{k=1}^{K} z_{ik} c_k, \quad \forall i = 1 \dots P.$$

This formulation immediately falls into the Learning-Compression form of (1) with $\mathbf{\Theta} = (C, \mathbf{z}_1, \dots \mathbf{z}_P)$ and $\lambda = 0$. The corresponding C-step problem of $\min_{\mathbf{\Theta}} \|\mathbf{w} - \Delta(\mathbf{\Theta})\|^2$ has the form of:

$$\min_{C, \mathbf{z}_1, \dots \mathbf{z}_P} \sum_{i=1}^{P} \sum_{k=1}^{K} z_{ik} (w_i - c_k)^2, \tag{2}$$

which has been thoroughly studied in the signal compression and unsupervised clustering literature. The solution to this problem is known as the k-means. The general k-means problem is NP hard [1, 10], however, this is a scalar version which has an efficient, globally optimal solution using dynamic programming [3, 50, 51]. Our software provides both k-means and dynamic programming solutions for the C step of adaptive quantization problem (2). Additionally, we provide solutions for fixed-codebook and scaled-codebook variants of quantization. See full list in Table 1.

4.2 Pruning

Pruning is the process of removing some of the weights of the model. One way of formulating this problem is by using the sparsifying norms (e.g., ℓ_0 or ℓ_1) as penalties or constraints, limiting the number of allowed non-zero weights. A particularly useful pruning scheme is ℓ_0-norm constrained pruning defined as:

$$\min_{\mathbf{w}} L(\mathbf{w}) \quad \text{s.t.} \quad \|\mathbf{w}\|_0 \le \kappa. \tag{3}$$

Since the ℓ_0-norm is the count of non-zero items in the vector, the formulation of (3) allows to precisely specify the number of remaining weights.

To bring it into the Learning-Compression form (1) we introduce a copy parameter $\boldsymbol{\theta}$ and obtain an equivalent optimization problem of:

$$\min_{\mathbf{w}} L(\mathbf{w}) \quad \text{s.t.} \quad \mathbf{w} = \boldsymbol{\theta}, \quad \|\boldsymbol{\theta}\|_0 \le \kappa,$$

for which the C step is given by solving:

$$\min_{\boldsymbol{\theta}} \|\mathbf{w} - \boldsymbol{\theta}\|^2 \quad \text{s.t.} \quad \|\boldsymbol{\theta}\|_0 \le \kappa. \tag{4}$$

The solution of (4) can be obtained by selecting all but top-κ weights (in magnitude) of $\mathbf{w}$ and zeroing remaining.

Using similar steps, we can obtain the C steps for ℓ_1 constrained formulation of pruning, and extend it to penalty based forms as $\min_{\mathbf{w}} L(\mathbf{w}) + \lambda\|\mathbf{w}\|_0$, see [6]. In our framework we provide the implementation for all combinations of ℓ_0 and ℓ_1-norms.

4.3 Low-rank compression

Our framework supports compressing the weight matrices of each layer to a given (preselected) target rank. This allows parametrizing the resulting compressed weight matrix $\mathbf{W}$ as a product of low-rank matrices, $\mathbf{UV}^T$. The challenge of such a compression scheme is a requirement to know the right choice of the ranks as it has a direct effect the error-compression tradeoff of the resulting model. To alleviate this issue, we include the implementation of the automatic rank selection from [19], which we describe next.

Assume we have a reference model with M layers and the weights $\mathbf{w} = \{\mathbf{W}_1, \dots, \mathbf{W}_M\}$, where $\mathbf{W}_m$ is the weight matrix of layer m. We want to optimize the following model selection problem over possible low-rank models:

$$\min_{\mathbf{w}} L(\mathbf{w}) + \lambda C(\mathbf{w}) \quad \text{s.t.} \quad \operatorname{rank}(\mathbf{W}_m) = r_1 \le R_m, \ \forall\, m = 1, \dots, M$$

here R_m is the maximum possible rank for matrix $\mathbf{W}_m$. The compression cost $C(\mathbf{w})$ is defined in terms of the ranks of the individual matrices:

$$C(\mathbf{w}) = \alpha_1 C(r_1) + \alpha_2 C(r_2) + \cdots + \alpha_M C(r_M),$$

and can capture both storage bits (to save space), total floating point operations (to speed up the model), and even device-targeted compression. To put it into the Learning-Compression form (eq. 1), we introduce the parameter $\boldsymbol{\Theta}_m$ for each layer, with constraint $\mathbf{W}_m = \boldsymbol{\Theta}_m$. Then, the objective of the C step separates into M problems over each layer's weights:

$$\min_{\boldsymbol{\Theta}_m, r_m} \lambda C_m(r_m) + \frac{\mu}{2}\|\mathbf{W}_m - \boldsymbol{\Theta}_m\|^2 \quad \text{s.t.} \quad \operatorname{rank}(\boldsymbol{\Theta}_m) = r_m \le R_m.$$

The solution of this C-step problem was given in [19]: it involves an SVD and enumeration over the ranks for each layer's weights.

5 DESIGN OF THE SOFTWARE

Equipped with the Learning-Compression algorithm and the required building-block compressions, we now discuss the design of our library. Our main goal is: to have an easy to use, efficient, robust, and configurable neural network compression software. Particularly, we would like to have the flexibility of applying any available compression (Table 1) to any parts of the neural network with per-layer granularity. For example, consider the following compression tasks:

- a single compression per layer: say, low-rank compression for layer 1 with target rank of 5
- a single compression per multiple layers: e.g., prune 5% of weights in layer 1 and 3, jointly
- mixing multiple compressions: e.g., quantize layer 1 and prune jointly layers 2 and 3
- additive compressions: be able to use additive compressions in the same mix-and-match way

The mix-and-match on the level of a layer granularity is an important requirement as neural networks can have heterogeneous structures: having layers with few parameters but many FLOPs and vice-versa. As such, some layers might be better suited to the specific form of compression than others, which has been exploited in the literature with specific schemes targeting only, for example, fully-connected layers [8, 43]. To implement our desiderata, we leverage the modularity of the LC algorithm and introduce some additional building blocks next.

L step. We hand off the model training operations, the L step, to the user through the lambda functions. This gives a fine-grained control to the user on the model's actual learning: hardware utilization, data source pulling, and other essential steps required for training. Usually, the L step implementation is already available or can be extracted from the training code used for the reference (uncompressed) model. Below we give a typical way of implementing the L step in PyTorch:

```
def my_l_step(model, lc_penalty, args**):
    loss = model.loss(out_, target_) + lc_penalty()
    loss.backward()
    optimizer.step()
```

Here we skipped some code (such as the setup of the optimizer and data source configuration) for brevity. Note that the only required change is the addition of `lc_penalty` term.

C step. All provided compressions of Table 1 are implemented as subclasses of `CompressionTypeBase` class, and the actual C step is exposed through the `compress` method. This allows a straightforward extension of the library of compressions: if needed, the user simply wraps the custom C-step solution into an object of `CompressionTypeBase` class. Below we give an example implementation of the C step for binarization:

```
class ScaledBinaryQuantization(CompressionTypeBase):
    def compress(self, data):
        a = np.mean(np.abs(data))
        quantized = 2 * a * (data > 0) - a
        return quantized
```

Compression tasks. To instruct the framework on which compression types should be applied to which parts of the model, the user needs to populate a compression tasks structure. This structure is a list of simple mappings of the form: (parameters) ⟶ (compression view, compression type), which is implemented as a python dictionary. The *parameters* are the subset of model weights, which are wrapped into internal `Parameter` object. The *compression view* is another internal structure that handles reshaping of the model weights into a form suitable for compression, e.g., reshaping the weight tensor of a convolutional layer into a matrix for low-rank compression.

While our strategy of defining the compression tasks might seem unnecessarily complicated, it brings *a considerable amount of flexibility*. For instance, it erases the limitations of standard compression approaches with coarse layer-based granularity: we can compress multiple layers with a single compression, or a single layer with multiple compressions, while simultaneously mixing different compressions in a single model. This abstraction disentangles compression from the model structure and allows us to construct complicated schemes of compressions in a *mix-and-match* way. For example, consider the following compression task:

$$\begin{aligned}&\text{(layer 1, layer 3)} && \longrightarrow \text{(as a vector, adaptive quantization } k = 6\text{)},\\ &\text{(layer 2)} && \longrightarrow \text{(as is, low-rank with } r = 3\text{)}\end{aligned}$$

where we want to jointly compress a three-layer neural network so that the first and third layers are quantized with the same codebook, and the second layer is a low-rank matrix with $r = 3$, and we want these compression to be applied simultaneously. The semantics of this compression is translated almost verbatim in our framework:

```
from lc.torch import ParameterTorch as P, AsVector, AsIs
compression_tasks = {
  P([l1.weight, l3.weight]): (AsVector, AdaptiveQuantization(k=6)),
  P(l2.weight):              (AsIs,     LowRank(target_rank=3))
}
```

The fine-grained control over semantics of the compression allows to include expert knowledge about properties of a particular model (e.g., do not quantize the first layer) without much effort.

Running the software. To compress a model, the user needs to construct an `lc.Algorithm` object and provide the following: 1) a model to be compressed 2) associated compression tasks 3) implementation of the L step 4) a schedule of μ values, and 5) an evaluation function to keep track of the error during the compression.

Here is an example of running the algorithm:

```
lc_alg = lc.Algorithm(
    model,                    # a model to compress
    compression_tasks,        # specifications of compression
    l_step_optimization,      # implementation of the L step
    mu_schedule,              # schedule of the mu values
    evaluation_func           # the evaluation function
)
lc_alg.run()                  # an entry point to the LC algorithm
```

Once the `run` method is called, the LC algorithm will start execution, at which point the library will proceed in line-by-line correspondence to the pseudocode on the left of Figure 2. Currently, each of the compression tasks (and corresponding C step implementation) is called in order. Yet, due to the nature of the LC algorithm, every compression task's C steps can be executed in parallel, further improving the efficiency of the algorithm.

6 A GUIDED TOUR THROUGH THE FUNCTIONALITY OF OUR SOFTWARE

In this section, we demonstrate the flexibility of our framework by easily exploring multiple compression schemes with minimal effort. As an example, say we are tasked with compressing the storage bits of the LeNet300 neural network trained on MNIST dataset (10 classes, 28×28 gray-scale images). The LeNet300 is a three-layer neural network with 300, 100, and 10 neurons respectively on every layer; the reference network has an error of 1.66% on the test set.

In order to run the LC algorithm, we need to provide an L step implementation and compression tasks as described in sec. 5. The implementation of corresponding L step is given below:

```
def my_l_step(model, lc_penalty, step):
  params = [p for p in model.parameters() if p.requires_grad]
  lr = lr_base*(0.98**step)              # decayed learning rate
  optimizer = optim.SGD(params, lr=lr, momentum=0.9, nesterov=True)
  for epoch in range(epochs_per_step):
    for x, target in train_loader:       # loop over the dataset
      optimizer.zero_grad()
      loss = model.loss(model(x), target) + lc_penalty()
      loss.backward()
      optimizer.step()
```

Now, having the L step implementation, we can formulate the compression tasks. Say, we would like to know what would be the test error if the model is optimally quantized with a separate codebook on each layer? Test error in such case is 1.97%, which is 0.31% higher than the reference. What would be the performance of the model if one would quantize only the first and the third layers, leaving the second layer untouched? Test error in such case is 1.96%. What about if we prune all but 5% of the weights? Yes, our framework can handle all of these combinations and more; see Table 2 for other examples. We can even apply different compressions to every layer, for example, take a look at the last row of Table 2, where we apply quantization, pruning, and low-rank compression to the different parts of the LeNet300. Once the L step is given, trying a new compression scheme only requires a new compression task.

7 PRACTICAL ADVICE

We implemented the LC algorithm originally in 2017, and we have gone through multiple refinements and code reimplementations. We have applied it to compressing a wide array of relatively large neural nets, such as LeNet, AlexNet, VGG, ResNet, etc., which are themselves tricky to train well in the first place. In the process, we have gathered a considerable amount of practical knowledge on the behavior of the LC algorithm on both small and large models and datasets. We would like to share a list of common pitfalls so future users of our framework would hopefully avoid them.

- **Monitor the progression of the algorithm** Specifically, two important quantities to keep an eye on:
  - The loss of the L step: $L(\mathbf{w}) + \frac{\mu}{2}\|\mathbf{w} - \Delta(\Theta)\|^2$. The total loss at the end of the L step must be smaller than the total loss at the beginning. If some L step has not reduced the loss, optimization parameters of the step should be tuned.
  - The loss of the C step, $\|\mathbf{w} - \Delta(\Theta)\|^2$, must have a smaller value after each C step. This often fails when new compression is introduced into the pipeline, where `compress` method

Compression	Code for compression tasks	Error
no compression		Train 0.00% Test 1.66%
quantize all layers	`compression_tasks = {` `Param(l1.weight): (AsVector, AdaptiveQuantization(k=2)),` `Param(l2.weight): (AsVector, AdaptiveQuantization(k=2)),` `Param(l3.weight): (AsVector, AdaptiveQuantization(k=2))` `}`	Train 0.02% Test 1.97%
quantize first and third layers	`compression_tasks = {` `Param(l1.weight): (AsVector, AdaptiveQuantization(k=2)),` `Param(l3.weight): (AsVector, AdaptiveQuantization(k=2))` `}`	Train 0.00% Test 1.96%
prune all but 5%	`compression_tasks = {` `Param([l1.weight, l2.weight, l3.weights]):` `(AsVector, ConstraintL0Pruning(kappa=13310)) # 13310 = 5%` `}`	Train 0.00% Test 1.70%
single codebook quantization with additive pruning of all but 1%	`compression_tasks = { Param([l1.weight, l2.weight, l3.weights]): [` `(AsVector, ConstraintL0Pruning(kappa=2662)), # 2662 = 1%` `(AsVector, AdaptiveQuantization(k=2))]` `}`	Train 0.00% Test 1.85%
prune first layer, low-rank to second, quantize third	`compression_tasks = {` `Param(l1.weight): (AsVector, ConstraintL0Pruning(kappa=5000)),` `Param(l2.weight): (AsIs,     LowRank(target_rank=10))` `Param(l3.weight): (AsVector, AdaptiveQuantization(k=2))` `}`	Train 0.04% Test 1.68%

Table 2: Some of the mix-and-match compressions possible in our framework and corresponding train/test errors. Here, we use the LeNet300 neural network trained on the MNIST dataset (reference test error is 1.67%) and report final test errors after compression. Notice that trying a new combination of compressions is as simple as writing a new *compression tasks* structure.

is not fully tested. For the base compressions in the framework, we made sure they always optimize the C step.

- **On the μ schedule** Theoretically, the sequence of μ values should start at 0 and infinitesimally grow to ∞. In practice, we use an exponentially increasing schedule $\mu_k = \mu_0 \times a^k$ with small initial μ_0 and appropriately chosen $a > 1$ for the k-th step of the LC. For most of compression schemes, we have developed robust estimates of μ_0-values: for pruning see suppl.mat. of [6], for rank-selection see suppl.mat. of [19]. For the value of a, we found the range of [1.1 1.4] to be a good spot.

8 EXPERIMENTS

Due to space considerations, here we report a limited set of experiments; for an extensive empirical evaluation we refer to [18].

CIFAR10 experiments. Our toolkit allows an easy exploration of tradeoff curves. For instance, on Figure 3 we explore FLOPs-error tradeoff when compressing multiple networks on CIFAR10.

ImageNet experiments. Our library can handle the training/compression of large models on bigger datasets. As a demonstration, we train the 1140 MFLOPs version of AlexNet [30] on the ImageNet-1k task [42]: the network has 60M weights and achieves top-1/top-5 errors of 40.43/17.55%. We then explore different compression mechanisms and their nesting using our library by changing the specifics of compression tasks. We run: 1) low-rank compression with automatic rank selection targeted for FLOPs reduction, 2) additive quantization and pruning (Q+P) scheme with varying amount of pruning, and 3) we select some of our low-rank AlexNet models and further compress them with Q+P scheme (resulting in a scheme of L→Q+P). We plot our results as colored tradeoff curves on Figure 4. To give a perspective on our results, we additionally plot individual compressions of AlexNet reported in the literature with black markers of different shapes.

The resulting AlexNet models achieve considerable amount of compression: we obtain a model with the compression ratio of 87.5× (2.78 MB) with no degradation in accuracy when compared to the reference net, or a model with a compression ratio of 136× (1.79 MB) with no degradation in accuracy when compared to Caffe AlexNet (see Figure 4). These results outperform all AlexNet compression results available in literature. Most importantly, all these compressions can be done with minimal changes to the code base, yet, with a possibility to target different compression goals. For example, we can target the size or the speed of the model, or both. For instance, the chaining of the low-rank scheme with Q+P not only compresses the size of the model by 136×, but also results in a faster inference due to lower FLOPs count (4.9×) coming from low-rank structure of the weights. On Jetson Nano Developer board, such FLOPs reduction translates into 3.5× faster inference speed [22].

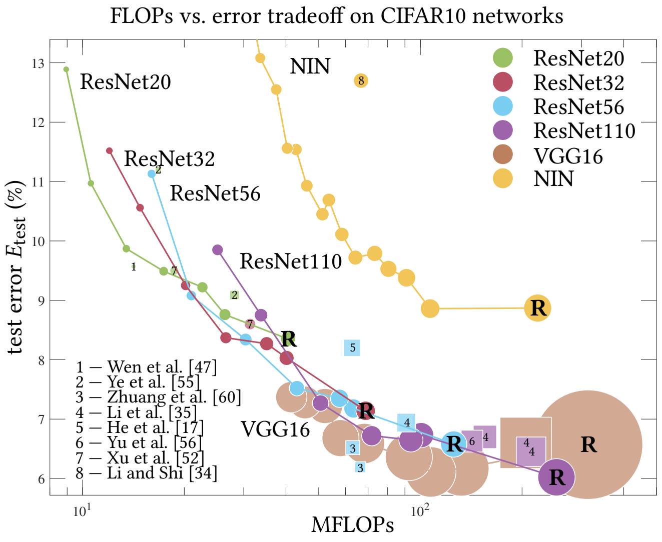


Figure 3: Error-compression space of test error (Y axis), inference FLOPs (X axis) and number of parameters (ball size for each net), for multiple networks trained on CIFAR10 and compressed with low-rank and structured pruning methods. Results of rank selection with LC algorithm over different λ values for a given network span a curve, shown as connected circles ●–●, which starts on the lower right at the reference R ($\lambda = 0$) and then moves left and up. Other published results using low-rank compression are shown as isolated circles labeled with a citation. Other published results involving structured filter pruning are shown as isolated squares labeled with a citation. The area of a circle or square is proportional to the number of parameters in the corresponding compressed model. Ideal models are small balls (having few parameters) on the left-bottom (where both error and FLOPs are the smallest). See [19] for full details of experiments.

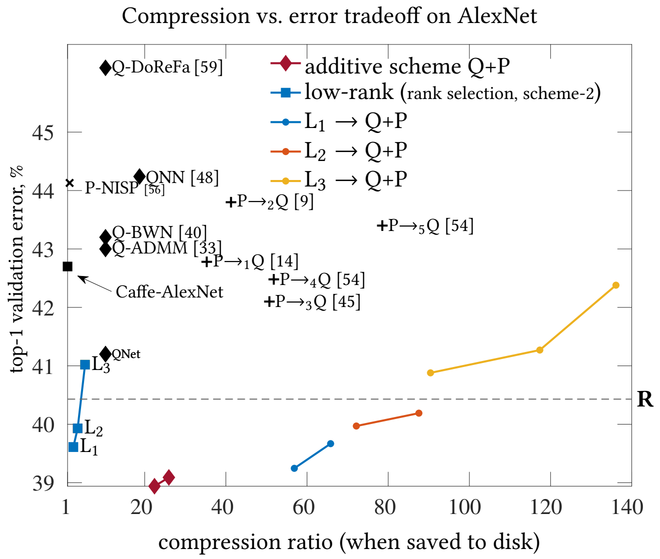


Figure 4: Compression schemes and their combinations available in our library when applied to AlexNet (trained on ImageNet-1k). We show results of: low-rank compression with rank selection, additive combination of quantization and pruning (Q+P), and nesting of low-rank scheme with Q+P. The only change required to obtain our results is in writing of a new compression task definition (about 10 lines of Python code). All other results available in the literature are given as black annotated markers and achieved using *highly specialized* algorithms. Mark descriptions: Q—quantization, P—pruning, L—low-rank. If compressions are chained, we denote it with '→', e.g., P→Q means network is quantized then pruned. Results obtained using our software are given as colored connected lines. Note: in their work, Yang et al. [54] reported 118× and 205× compression on AlexNet, yet, these numbers were reported without a proper accounting for the storage of the sparse index. When the sparse index is saved along the model, the compression ratios become 52× and 79× (results labeled as P→$_4$Q, P→$_5$Q).

9 CONCLUSION

The fields of machine learning and signal compression have developed independently for a long time: machine learning solves the problem of training a deep net to minimize a desired loss on a dataset, while signal compression solves the problem of optimally compressing a given signal. The LC algorithm allows us to seamlessly integrate the existing algorithms to train deep nets (L step) and algorithms to compress a signal (C step) by tapping on the abundant literature in the machine learning and signal compression fields. Based on this, we have described an open-source toolkit for model compression. It is primarily designed for flexibility and extensibility, in order to handle in a scalable way arbitrary choices of the model, task and loss function; other desirable cost functions such as memory, runtime or energy; and the type of compression to apply. The LC algorithm is based on solid optimization principles that guarantee that we find a local optimum of the (usually nonconvex, often nondifferentiable) compression problem, and is not much slower than training an uncompressed model. The toolkit allows reusability of code developed in machine learning for training models, and in signal processing for compressing data. This results in a highly usable and reliable solution to determine how best to compress a given model. Experimentally, in a series of papers we have observed little, if any, loss of performance compared to customized algorithms for specific choices of model and compression. We have developed the toolkit to include various forms of compressions out of the box, including quantization, pruning, and low-rank in various forms. The toolkit, available at https://github.com/UCMerced-ML/LC-model-compression under the BSD 3-clause license, is an evolving work and we expect future contributions from our own research group and the community.

ACKNOWLEDGMENTS

Work partially supported by NSF award IIS–1423515 and by several GPU donations from the NVIDIA Corporation.

REFERENCES

[1] Daniel Aloise, Amit Deshpande, Pierre Hansen, and Preyas Popat. 2009. NP-Hardness of Euclidean Sum-of-Squares Clustering. *Machine Learning* 75, 2 (May 2009), 245–248.

[2] Michaela Blott, Thomas B. Preußer, Nicholas J. Fraser, Giulio Gambardella, Kenneth O'Brien, Yaman Umuroglu, Miriam Leeser, and Kees Vissers. 2018. FINN-R: An End-to-End Deep-Learning Framework for Fast Exploration of Quantized Neural Networks. *ACM Trans. Reconfigurable Technology and Systems* 11, 3 (Dec. 2018), 16.1–16.23.

[3] James D Bruce. 1965. *Optimum quantization.* Technical Report 429. Massachussetts Institute of Technology.

[4] Miguel Á. Carreira-Perpiñán. 2017. Model Compression as Constrained Optimization, with Application to Neural Nets. Part I: General Framework. (July 5 2017). arXiv:1707.01209.

[5] Miguel Á. Carreira-Perpiñán and Yerlan Idelbayev. 2017. Model Compression as Constrained Optimization, with Application to Neural Nets. Part II: Quantization. (July 13 2017). arXiv:1707.04319.

[6] Miguel Á. Carreira-Perpiñán and Yerlan Idelbayev. 2018. "Learning-Compression" Algorithms for Neural Net Pruning. In *Proc. of the 2018 IEEE Computer Society Conf. Computer Vision and Pattern Recognition (CVPR'18).* Salt Lake City, UT, 8532–8541.

[7] Miguel Á. Carreira-Perpiñán and Arman Zharmagambetov. 2018. Fast Model Compression. In *Bay Area Machine Learning Symposium (BayLearn 2018).* Facebook, Menlo Park, CA.

[8] Patrick Chen, Si Si, Yang Li, Ciprian Chelba, and Cho-Jui Hsieh. 2018. GroupReduce: Block-Wise Low-Rank Approximation for Neural Language Model Shrinking. In *Advances in Neural Information Processing Systems (NEURIPS)*, S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett (Eds.), Vol. 31. MIT Press, Cambridge, MA, 10988–10998.

[9] Yoojin Choi, Mostafa El-Khamy, and Jungwon Lee. 2017. Towards the Limit of Network Quantization. In *Proc. of the 5th Int. Conf. Learning Representations (ICLR 2017).* Toulon, France.

[10] Sanjoy Dasgupta and Yoav Freund. 2009. Random Projection Trees for Vector Quantization. *IEEE Trans. Information Theory* 55, 7 (July 2009), 3229–3242.

[11] Misha Denil, Babak Shakibi, Laurent Dinh, Marc'Aurelio Ranzato, and Nando de Freitas. 2013. Predicting Parameters in Deep Learning. In *Advances in Neural Information Processing Systems (NIPS)*, C. J. C. Burges, L. Bottou, M. Welling, Z. Ghahramani, and K. Q. Weinberger (Eds.), Vol. 26. MIT Press, Cambridge, MA, 2148–2156.

[12] Emily L. Denton, Wojciech Zaremba, Joan Bruna, Yann LeCun, and Rob Fergus. 2014. Exploiting Linear Structure within Convolutional Networks for Efficient Evaluation. In *Advances in Neural Information Processing Systems (NIPS)*, Z. Ghahramani, M. Welling, C. Cortes, N. D. Lawrence, and K. Q. Weinberger (Eds.), Vol. 27. MIT Press, Cambridge, MA, 1269–1277.

[13] Xiaohan Ding, Guiguang Ding, Yuchen Guo, Jungong Han, and Chenggang Yan. 2019. Approximated Oracle Filter Pruning for Destructive CNN Width Optimization. In *Proc. of the 36th Int. Conf. Machine Learning (ICML 2019)*, Kamalika Chaudhuri and Ruslan Salakhutdinov (Eds.). Long Beach, CA, 1607–1616.

[14] Song Han, Huizi Mao, and William J. Dally. 2016. 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.

[15] Song Han, Jeff Pool, John Tran, and William Dally. 2015. Learning both Weights and Connections for Efficient Neural Network. In *Advances in Neural Information Processing Systems (NIPS)*, C. Cortes, N. D. Lawrence, D. D. Lee, M. Sugiyama, and R. Garnett (Eds.), Vol. 28. MIT Press, Cambridge, MA, 1135–1143.

[16] Trevor Hastie, Robert Tibshirani, and Martin Wainwright. 2015. *Statistical Learning with Sparsity: The Lasso and Generalizations.* Chapman & Hall/CRC.

[17] Yihui He, Xiangyu Zhang, and Jian Sun. 2017. Channel Pruning for Accelerating Very Deep Neural Networks. In *Proc. 16th Int. Conf. Computer Vision (ICCV'17).* Venice, Italy, 1398–1406.

[18] Yerlan Idelbayev and Miguel Á. Carreira-Perpiñán. 2020. A Flexible, Extensible Software Framework for Model Compression Based on the LC Algorithm. (May 15 2020). arXiv:2005.07786.

[19] Yerlan Idelbayev and Miguel Á. Carreira-Perpiñán. 2020. Low-Rank Compression of Neural Nets: Learning the Rank of Each Layer. In *Proc. of the 2020 IEEE Computer Society Conf. Computer Vision and Pattern Recognition (CVPR'20).* Seattle, WA, 8046–8056.

[20] Yerlan Idelbayev and Miguel Á. Carreira-Perpiñán. 2021. Beyond FLOPs in Low-Rank Compression of Neural Networks: Optimizing Device-Specific Inference Runtime. In *IEEE Int. Conf. Image Processing (ICIP 2021).* Anchorage, AK.

[21] Yerlan Idelbayev and Miguel Á. Carreira-Perpiñán. 2021. An Empirical Comparison of Quantization, Pruning and Low-Rank Neural Network Compression Using the LC Toolkit. In *Int. J. Conf. Neural Networks (IJCNN'21).* Virtual event.

[22] Yerlan Idelbayev and Miguel Á. Carreira-Perpiñán. 2021. More General and Effective Model Compression via an Additive Combination of Compressions. In *Proc. of the 32nd European Conf. Machine Learning (ECML–21).* Bilbao, Spain.

[23] Yerlan Idelbayev and Miguel Á. Carreira-Perpiñán. 2021. Neural Network Compression via Additive Combination of Reshaped, Low-rank Matrices. In *Proc. Data Compression Conference (DCC 2021)*, Ali Bilgin, Michael W. Marcellin, Joan Serra-Sagrista, and James A. Storer (Eds.). Online, 243–252.

[24] Yerlan Idelbayev and Miguel Á. Carreira-Perpiñán. 2021. Optimal Selection of Matrix Shape and Decomposition Scheme for Neural Network Compression. In *Proc. of the IEEE Int. Conf. Acoustics, Speech and Sig. Proc. (ICASSP'21).* Toronto, Canada, 3250–3254.

[25] Benoit Jacob, Skirmantas Kligys, Bo Chen, Menglong Zhu, Matthew Tang, Andrew Howard, Hartwig Adam, and Dmitry Kalenichenko. 2018. Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference. In *Proc. of the 2018 IEEE Computer Society Conf. Computer Vision and Pattern Recognition (CVPR'18).* Salt Lake City, UT, 2704–2713.

[26] Max Jaderberg, Andrea Vedaldi, and Andrew Zisserman. 2014. Speeding up Convolutional Neural Networks with Low Rank Expansions. In *Proc. of the 25th British Machine Vision Conference (BMVC 2014)*, Michel Valstar, Andrew French, and Tony Pridmore (Eds.). Nottingham, UK.

[27] Vinu Joseph, Saurav Muralidharan, Animesh Garg, Michael Garland, and Ganesh Gopalakrishnan. 2019. A Programmable Approach to Model Compression. (Nov. 6 2019). arXiv:1911.02497.

[28] Hyeji Kim, Muhammad Umar Karim Khan, and Chong-Min Kyung. 2019. Efficient Neural Network Compression. In *Proc. of the 2019 IEEE Computer Society Conf. Computer Vision and Pattern Recognition (CVPR'19).* Long Beach, CA, 12569–12577.

[29] Alexander Kozlov, Ivan Lazarevich, Vasily Shamporov, Nikolay Lyalyushkin, and Yury Gorbachev. 2020. Neural Network Compression Framework for Fast Model Inference. (Feb. 20 2020). arXiv:2002.08679.

[30] Alex Krizhevsky, Ilya Sutskever, and Geoff Hinton. 2012. ImageNet Classification with Deep Convolutional Neural Networks. In *Advances in Neural Information Processing Systems (NIPS)*, F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger (Eds.), Vol. 25. MIT Press, Cambridge, MA, 1106–1114.

[31] Chris Lattner. 2011. LLVM. In *The Architecture of Open Source Applications, Vol. I: Elegance, Evolution, and a Few Fearless Hacks*, Amy Brown and Greg Wilson (Eds.). lulu.com, Chapter 11, 151–165.

[32] Vadim Lebedev, Yaroslav Ganin, Maksim Rakhuba, Ivan Oseledets, and Victor Lempitsky. 2015. Speeding-up Convolutional Neural Networks Using Fine-Tuned CP-Decomposition. In *Proc. of the 3rd Int. Conf. Learning Representations (ICLR 2015).* San Diego, CA.

[33] Cong Leng, Hao Li, Shenghuo Zhu, and Rong Jin. 2018. Extremely Low Bit Neural Network: Squeeze the Last Bit Out with ADMM. In *Proc. of the 32nd AAAI Conference on Artificial Intelligence (AAAI 2018).* New Orleans, LA, 3466–3473.

[34] Chong Li and C. J. Richard Shi. 2018. Constrained Optimization Based Low-Rank Approximation of Deep Neural Networks. In *Proc. 15th European Conf. Computer Vision (ECCV'18)*, Vittorio Ferrari, Martial Hebert, Cristian Sminchisescu, and Yair Weiss (Eds.). Munich, Germany, 746–761.

[35] Hao Li, Asim Kadav, Igor Durdanovic, and Hans P. Graf. 2017. Pruning Filters for Efficient ConvNets. In *Proc. of the 5th Int. Conf. Learning Representations (ICLR 2017).* Toulon, France.

[36] Jiashi Li, Qi Qi, Jingyu Wang, Ce Ge, Yujian Li, Zhangzhang Yue, and Haifeng Sun. 2019. OICSR: Out-In-Channel Sparsity Regularization for Compact Deep Neural Networks. In *Proc. of the 2019 IEEE Computer Society Conf. Computer Vision and Pattern Recognition (CVPR'19).* Long Beach, CA, 7046–7055.

[37] Baoyuan Liu, Min Wan, Hassan Foroosh, Marshall Tappen, and Marianna Pensky. 2015. Sparse Convolutional Neural Networks. In *Proc. of the 2015 IEEE Computer Society Conf. Computer Vision and Pattern Recognition (CVPR'15).* Boston, MA, 806–814.

[38] Dominik Marek Loroch, Norbert Wehn, Franz-Josef Pfreundt, and Janis Keuper. 2017. TensorQuant — A Simulation Toolbox for Deep Neural Network Quantization. In *Proc. SC Workshop on Machine Learning on HPC Environments (MLHPC'17).* 1:1–1:8.

[39] Moritz B. Milde, Daniel Neil, Alessandro Aimar, Tobi Delbruck, and Giacomo Indiveri. 2017. ADaPTION: Toolbox and Benchmark for Training Convolutional Neural Networks with Reduced Numerical Precision Weights and Activation. (Nov. 13 2017). arXiv:1711.04713.

[40] Mohammad Rastegari, Vicente Ordonez, Joseph Redmon, and Ali Farhadi. 2016. XNOR-Net: ImageNet Classification Using Binary Convolutional Neural Networks. In *Proc. 14th European Conf. Computer Vision (ECCV'16)*, Bastian Leibe, Jiri Matas, Nicu Sebe, and Max Welling (Eds.). Amsterdam, The Netherlands, 525–542.

[41] Gregory C. Reinsel and Raja P. Velu. 1998. *Multivariate Reduced-Rank Regression. Theory and Applications.* Number 136 in Lecture Notes in Statistics. Springer-Verlag.

[42] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. 2015. ImageNet Large Scale Visual Recognition Challenge. *Int. J. Computer Vision* 115, 3 (Dec. 2015), 211–252.

[43] Tara N. Sainath, Brian Kingsbury, Vikas Sindhwani, Ebru Arısoy, and Bhuvana Ramabhadran. 2013. 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)*. Vancouver, Canada, 6655–6659.

[44] Cheng Tai, Tong Xiao, Yi Zhang, Xiaogang Wang, and Weinan E. 2016. Convolutional Neural Networks with Low-rank Regularization. In *Proc. of the 4th Int. Conf. Learning Representations (ICLR 2016)*. San Juan, Puerto Rico.

[45] Frederick Tung and Greg Mori. 2018. CLIP-Q: Deep Network Compression Learning by In-Parallel Pruning-Quantization. In *Proc. of the 2018 IEEE Computer Society Conf. Computer Vision and Pattern Recognition (CVPR'18)*. Salt Lake City, UT, 7873–7882.

[46] Wei Wen, Chunpeng Wu, Yandan Wang, Yiran Chen, and Hai Li. 2016. Learning Structured Sparsity in Deep Neural Networks. In *Advances in Neural Information Processing Systems (NIPS)*, D. D. Lee, M. Sugiyama, Ulrike von Luxburg, Isabelle Guyon, and R. Garnett (Eds.), Vol. 29. MIT Press, Cambridge, MA, 2074–2082.

[47] Wei Wen, Cong Xu, Chunpeng Wu, Yandan Wang, Yiran Chen, and Hai Li. 2017. Coordinating Filters for Faster Deep Neural Networks. In *Proc. 16th Int. Conf. Computer Vision (ICCV'17)*. Venice, Italy.

[48] Jiaxiang Wu, Cong Leng, Yuhang Wang, Qinghao Hu, and Jian Cheng. 2016. Quantized Convolutional Neural Networks for Mobile Devices. In *Proc. of the 2016 IEEE Computer Society Conf. Computer Vision and Pattern Recognition (CVPR'16)*. Las Vegas, NV, 4020–4028.

[49] Jiaxiang Wu, Yao Zhang, Haoli Bai, Huasong Zhong, Jinlong Hou, Wei Liu, Wenbing Huang, and Junzhou Huang. 2018. PocketFlow: An Automated Framework for Compressing and Accelerating Deep Neural Networks. In *NIPS Workshop on Compact Deep Neural Network Representation with Industrial Applications (CDNNRIA)*.

[50] Xiaolin Wu. 1991. Optimal Quantization by Matrix Searching. *J. Algorithms* 12, 4 (Dec. 1991), 663–673.

[51] Xiaolin Wu and John Rokne. 1989. An $O(KN \log N)$ Algorithm for Optimum K-level Quantization on Histograms of N Points. In *Proc. 17th ACM Annual Computer Science Conference*. Louisville, KY, 339–343.

[52] Xiaowei Xu, Yukun Ding, Sharon Xiaobo Hu, Michael Niemier, Jason Cong, Yu Hu, and Yiyu Shi. 2018. Scaling for Edge Inference of Deep Neural Networks. *Nature Electronics* 1 (April 17 2018), 216–222.

[53] Jian Xue, Jinyu Li, and Yifan Gong. 2013. Restructuring of Deep Neural Network Acoustic Models with Singular Value Decomposition. In *Proc. of Interspeech'13*, F. Bimbot, C. Cerisara, C. Fougeron, G. Gravier, L. Lamel, F. Pellegrino, and P. Perrier (Eds.). Lyon, France, 2365–2369.

[54] Haichuan Yang, Shupeng Gui, Yuhao Zhu, and Ji Liu. 2020. Automatic Neural Network Compression by Sparsity-Quantization Joint Learning: A Constrained Optimization-Based Approach. In *Proc. of the 2020 IEEE Computer Society Conf. Computer Vision and Pattern Recognition (CVPR'20)*. Seattle, WA, 2175–2185.

[55] Jianbo Ye, Xin Lu, Zhe Lin, and James Wang. 2018. Rethinking the Smaller-norm-less-informative Assumption in Channel Pruning of Convolution Layers. In *Proc. of the 6th Int. Conf. Learning Representations (ICLR 2018)*. Vancouver, Canada.

[56] Ruichi Yu, Ang Li, Chun-Fu Chen, Jui-Hsin Lai, Vlad I. Morariu, Xintong Han, Mingfei Gao, Ching-Yung Lin, and Larry S. Davis. 2018. NISP: Pruning Networks Using Neuron Importance Score Propagation. In *Proc. of the 2018 IEEE Computer Society Conf. Computer Vision and Pattern Recognition (CVPR'18)*. Salt Lake City, UT, 9194–9203.

[57] Xiangyu Zhang, Jianhua Zou, Kaiming He, and Jian Sun. 2016. Accelerating Very Deep Convolutional Networks for Classification and Detection. *IEEE Trans. Pattern Analysis and Machine Intelligence* 38, 10 (Oct. 2016), 1943–1955.

[58] Yiren Zhao, Xitong Gao, Robert Mullins, and Chengzhong Xu. 2018. Mayo: A Framework for Auto-generating Hardware Friendly Deep Neural Networks. In *Proc. 2nd Int. Workshop on Embedded and Mobile Deep Learning (EMDL'18)*. Munich, Germany, 25–30.

[59] Shuchang Zhou, Zekun Ni, Xinyu Zhou, He Wen, Yuxin Wu, and Yuheng Zou. 2016. Dorefa-Net: Training Low Bitwidth Convolutional Neural Networks with Low Bitwidth Gradients. (July 17 2016). arXiv:1606.06160.

[60] Zhuangwei Zhuang, Mingkui Tan, Bohan Zhuang, Jing Liu, Yong Guo, Qingyao Wu, Junzhou Huang, and Jinhui Zhu. 2018. Discrimination-Aware Channel Pruning for Deep Neural Networks. In *Advances in Neural Information Processing Systems (NEURIPS)*, S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett (Eds.), Vol. 31. MIT Press, Cambridge, MA, 815–886.

[61] Jacob Ziv and Abraham Lempel. 1977. A Universal Algorithm for Sequential Data Compression. *IEEE Trans. Information Theory* 23, 3 (May 1977), 337–343.

[62] Neta Zmora, Guy Jacob, Lev Zlotnik, Bar Elharar, and Gal Novik. 2019. Neural Network Distiller: a Python Package for DNN Compression Research. (Oct. 27 2019). arXiv:1910.12232.