# A FLEXIBLE, EXTENSIBLE SOFTWARE FRAMEWORK FOR MODEL COMPRESSION BASED ON THE LC ALGORITHM

**Yerlan Idelbayev** [1] **Miguel Á. Carreira-Perpiñán** [1]

## ABSTRACT

Compression of the neural network models has become an important systems problem for practical machine learning workflows. While various compression mechanisms and algorithms have been proposed to address the issue, many solutions rely on highly specialized procedures and require substantial domain knowledge to use efficiently. To streamline the compression to a large body of users, we propose an extensible open-source library based on the ideas of learning-compression (LC) algorithm—the LC toolkit. The software is written in Python using Pytorch and currently supports multiple forms of pruning, quantization, and low-rank compressions that can be applied to the model's parts individually or in combination to reduce model's size, computational requirements, or the on-device inference time. The toolkit's versatility comes from the separation of the model learning from the model compression in the LC algorithm: once the learning (L) step is given, any compression (C) steps can be used for the model.

## 1 INTRODUCTION

The widespread application of deep neural networks in practical tasks has been fueled by their state-of-the-art performances in the fields of computer vision, natural language processing, imaging, and other machine learning domains. To achieve the best performance, these very deep neural networks require many million weights and enormous computational resources. However, in terms of actual deployment, models are often required to be running on smaller and less powerful devices like phones, cameras, and watches. The mismatch between neural network's resource demands and the constraints of the systems give rise to the problem of model compression: *how to reduce the model's requirements in terms of disk storage, computation speed, and/or power demands?*

The need for a model compression resulted in plethora of works addressing the problem using different means of compression like quantization, pruning, low-rank decomposition or tensor factorizations. *However, among these strands in neural net compression approaches, in our view, the fundamental problem is that in practice, one does not know what type of compression (or combination of compression types) may be the best for a given network.* In principle, it may be possible to try different existing compression schemes with corresponding algorithms, assuming one can find the implementations of those. Though, practically it is impossible due to several factors:

- **Code availability and generality** Compression algorithms published in research literature have ad-hoc codebase: usually, the provided scripts are highly specialized for a small set of architectures discussed in the paper itself.
- **Compression combinations** If a user wants to combine several combinations in some way (say, additively, or by mixing and matching) it is unclear how to achieve it in a systematic way. Very few papers (if at all) address the question of efficiently combining diverse compressions that would be applicable for a large class of techniques.
- **Maintenance** If one manages to collect a set of diverse, state-of-the-art compression techniques in a single place, maintaining such a collection becomes a very complex task full of technical-debt: every model's compression is as if you have a separate software altogether with minimal reuse of the code.

*Aforementioned issues highlight a need for a generic software that will allow to compress machine learning models with a minimal effort from the end user while promoting good software engineering principles like code reuse and testing.* Such a software package would be beneficial for the entire community: for the researchers to establish solid comparisons and baselines, for the practitioners to quickly reduce model's size or computational demands, and for the

[1]Department of CSE, University of California, Merced, California. Correspondence to: Yerlan Idelbayev <yidelbayev@ucmerced.edu>.

*Proceedings of the $4^{th}$ MLSys Conference*, San Jose, CA, USA, 2021. 

designers of ML hardware, software, and systems—to have a single entry point to the field of model compression.

Our ongoing research in model compression is focused on solving the generic form of the problem given by a constrained optimization formulation. As we have shown in a series of publications (Carreira-Perpiñán, 2017; Carreira-Perpiñán & Idelbayev, 2017; 2018; Idelbayev & Carreira-Perpiñán, 2020a;b; 2021a;b;c;d) such a formulation allows to efficiently handle multiple compressions of interest like pruning (Carreira-Perpiñán & Idelbayev, 2018), quantization (Carreira-Perpiñán & Idelbayev, 2017), low-rank compression (Idelbayev & Carreira-Perpiñán, 2020a; 2021a;b), resource-targeted (Idelbayev & Carreira-Perpiñán, 2020a) and device-targeted (Idelbayev & Carreira-Perpiñán, 2021c) compressions in a unified algorithmic footing while achieving state-of-the-art results. Our efforts have culminated in an open-source[1], flexible, and extensible software framework that allows to compress a neural network with any of the supported compressions (Table 1).

| Type | Forms |
|---|---|
| Quantization | Adaptive Quantization into $\{c_1, c_2, \dots c_K\}$<br>Binarization into $\{-1, 1\}$ and $\{-c, c\}$<br>Ternarization into $\{-c, 0, c\}$ |
| Pruning | $\ell_0$-constraint (s.t., $\|\mathbf{w}\|_0 \leq \kappa$)<br>$\ell_1$-constraint (s.t., $\|\mathbf{w}\|_0 \leq \kappa$)<br>$\ell_0$-penalty ($\alpha\|\mathbf{w}\|_0$)<br>$\ell_1$-penalty ($\alpha\|\mathbf{w}\|_1$) |
| Low-rank | Low-rank compression to a given rank<br>Low-rank with *automatic* rank selection |
| Additive Combinations | Quantization + Pruning<br>Quantization + Low-rank<br>Pruning + Low-rank<br>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.

The theoretical foundation that allows us to combine diverse compression schemes under the same algorithmic umbrella is in the *learning-compression* algorithm (Carreira-Perpiñán, 2017) that decouples the model training (L step) from the model compression (C step). The optimization happens by an iterative alternation of L and C steps. As we explain in sec 2, the form of the L step is given as training of a regular machine learning problem *regardless of the associated compression scheme*, and the C step often comes as a standard signal compression problem with a well-studied solution *independent of the model structure and weights*. This separation makes the framework and the software *modular*: we can change the compression type by simply calling a different compression routine (e.g., $k$-means instead of the SVD), with no other changes to the algorithm.

**What makes our approach special?** The LC algorithm is efficient in runtime; it does not take much longer than training the uncompressed model in the first place. The compressed models perform competitively and allow the user to easily explore the space of prediction accuracy of the model vs compression ratio (which can be defined in terms of memory, inference time, energy or other criteria). Our code has been extensively tested since 2017 through usage in internal research projects, and has resulted in multiple publications that improve the state of the art in several compression schemes (Carreira-Perpiñán & Idelbayev, 2017; 2018; Idelbayev & Carreira-Perpiñán, 2020a;b; 2021a;b;c;d).

But what truly makes the approach practical is its flexibility and extensibility. If one wants to compress a specific type of model with a specific compression scheme, 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. Furthermore, one is not restricted to a single compression scheme; multiple compression schemes (say, low-rank plus pruning plus quantization) can be combined automatically, so they best cooperate to compress the model. The compression schemes that our code already supports make it possible for a user to mix and match them as desired with minimal effort. We expect to include further schemes in the future.

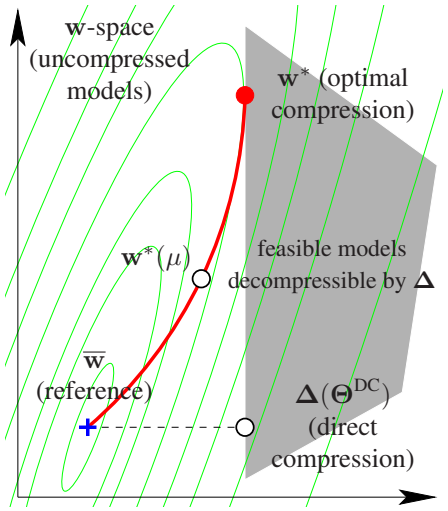
## 2 Model compression as a constrained optimization

The goal of model compression is to find a low-dimensional parametrization $\boldsymbol{\Delta}(\boldsymbol{\Theta})$ of the weights $\mathbf{w}$ so that corresponding decompressed model has (locally) optimal loss. Therefore the *model compression as a constrained optimization* problem is defined as:

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

Here the $\lambda C(\boldsymbol{\Theta})$ term with positive scalar $\lambda$ controls the amount of compression as measured by the deployment needs: it might be a storage size, number of FLOPs, or energy consumption requirements. Notice that compression cost is defined wrt parameters of the compressed model. The problem in (1) is constrained, nonlinear, and usually non-differentiable with respect to the compression parameters $\boldsymbol{\Theta}$ (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

[1] https://github.com/UCMerced-ML/LC-model-compression




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

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

employing an alternating optimization. This results in an algorithm that alternates two generic steps while slowly driving the penalty parameter $\mu \to \infty$:

- **L (learning) step:** $\min_{\mathbf{w}} L(\mathbf{w}) + \frac{\mu}{2}\|\mathbf{w} - \boldsymbol{\Delta}(\boldsymbol{\Theta})\|^2$. This is regular training of the uncompressed model but with a quadratic regularization term. *This step is independent from the form of chosen compression.*
- **C (compression) step:** $\min_{\boldsymbol{\Theta}} \|\mathbf{w} - \boldsymbol{\Delta}(\boldsymbol{\Theta})\|^2 + \lambda C(\boldsymbol{\Theta})$. This means finding the best lossy compression of the current uncompressed model weights $\mathbf{w}$ in the $\ell_2$ sense, and the solution is given by orthogonal projection on the feasible set. The solution of this step depends on the actual form of of the compression scheme ($\boldsymbol{\Delta}(\boldsymbol{\Theta})$). *However, this step is independent of the model loss and does not require training set.*

We will be using the quadratic penalty (QP) formulation throughout this paper to make exposition simpler. Yet, in practice, we implement the augmented Lagrangian (AL) version which has an additional vector of Lagrange multipliers. Fig. 1 illustrates the idea of model compression as constrained optimization, and depicts the traced solution $\mathbf{w}^*(\mu)$.

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 $\boldsymbol{\Theta}$ 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 $\ell_2$ minimization, and as 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.

## 3 The Design of the LC Toolkit

There are three important concepts to run a network compression using the LC algorithm: implementation of the L step, implementation of the C step, and the definition of the correspondence between compression types and parameters of the neural network. We discuss these building blocks next.

**L step** We give to the user the full control over the L step through the functional interface of the Python. This gives a fine-grained control to the user on the model's training which involves many systems considerations like hardware utilization, data source pulling, and other essential hardware/data steps. With such a control, user can best utilize the available hardware like TPU/GPU or custom learning environments. A typical implementation of the L step in PyTorch for our toolkit is given below:

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

**C step** The C steps (including the ones in Table 1) are implemented as subclasses of `CompressionTypeBase`. This allows to extend the library with new compression schemes via a simple class inheritance. 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** The user instructs the toolkit on intended compression and its usage parameters through *compression tasks structure* which is a simple dictionary of the form: (network parameters) $\rightarrow$ (compression view, compression type). Here, the *parameters* are the subset of model weights $\mathbf{w}$ that are going to be compressed. The *compression view* is an internal structure that handles reshaping of the weights into a suitable form, e.g., reshaping the weights of a convolutional layer (typically a tensor) into a matrix to apply low-rank compression.

Our strategy of handling compression mappings allows to combine the compressions in a mix-and-match way. For example, consider the following mapping of:

(layer 1, layer 3) $\rightarrow$ (as a vector, quantization $k = 6$),
(layer 2) $\rightarrow$ (as is, low-rank with $r = 3$)

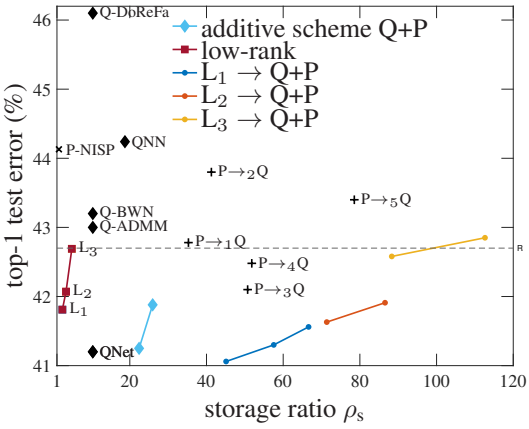
Here we want to quantize the first and third layer of the network using quantization, but apply low-rank compression with a target rank of 3 to the weights of the second layer. Such desired compression structure translates word-by-word into a Python code in our toolkit:

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

**Running the software** To compress a model, the user needs to construct an `lc.Algorithm` object and provide the structure of compression mappings (compression tasks), the implementations of the L and C steps, and the schedule of the $\mu$-values.

## 4 Experimental Evaluation

Our library allows easy exploration of various compressions and their combinations. To demonstrate its effectiveness we compress AlexNet trained on ILSVRC2012 dataset (Russakovsky et al., 2015) with some of the supported compressions. We first compress the AlexNet using automatic low-rank compression (see points $L_1, L_2, L_3$ on Fig 2). While these low-rank models achieve a considerable amount of compression wrt other low-rank models, we further compress it with additive combination of quantization and pruning: which results in state-of-the-art compression results on AlexNet. Our L$\rightarrow$Q+P models achieve get $112\times$ compression (2.16MB) without degradation in accuracy and $66\times$ compression with more than 1% improvement in the top-1 accuracy when compared to the Caffe-AlexNet. These results do not come at the cost of higher inference speed, in fact, our $66\times$ compressed model ($L_1 \rightarrow$ Q+ P) runs $2\times$ faster on Jetson Nano edge device, and the $112\times$ compressed AlexNet runs $3\times$ faster.




Inference time and speed-up for a single image

| Model | GPU of Jetson Nano time, ms | GPU of Jetson Nano speed-up |
|---|---|---|
| Caffe-AlexNet | 23.36 | 1.00 |
| $L_1 \rightarrow$ Q (1-bit ) + P (0.25M) | 11.59 | 2.01 |
| $L_2 \rightarrow$ Q (1-bit ) + P (0.25M) | 8.88 | 2.63 |
| $L_3 \rightarrow$ Q (1-bit ) + P (0.25M) | 7.11 | 3.29 |

*Figure 2.* Compression schemes and their combinations available in our library when applied to AlexNet. The only change required to obtain our results is in writing of a new compression task definition (about 10 lines of Python code). Mark descriptions: Q (quantization), P (pruning), L (low-rank). If compressions are chained, we denote it with '$\rightarrow$', e.g., P$\rightarrow$Q means network is quantized then pruned. Other results are as follows: P$\rightarrow_1$Q (Han et al., 2016), P$\rightarrow_2$Q (Choi et al., 2017), P$\rightarrow_3$Q (Tung & Mori, 2018), P$\rightarrow_4$Q (Yang et al., 2020)P$\rightarrow_5$Q (Yang et al., 2020), QNN (Wu et al., 2016), Q-DoReFa (Zhou et al., 2016), Q-BWN (Rastegari et al., 2016),Q-ADMM (Leng et al., 2018), P-NISP (Yu et al., 2018).

## 5 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. We invite practitioners to try out our library and contribute new forms of compressions.

## REFERENCES


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

Carreira-Perpiñán, M. Á. and Idelbayev, Y. Model compression as constrained optimization, with application to neural nets. Part II: Quantization. arXiv:1707.04319, July 13 2017.

Carreira-Perpiñán, M. Á. and Idelbayev, Y. "Learning-compression" algorithms for neural net pruning. In *Proc. of the 2018 IEEE Computer Society Conf. Computer Vision and Pattern Recognition (CVPR'18)*, pp. 8532–8541, Salt Lake City, UT, June 18–22 2018.

Choi, Y., El-Khamy, M., and Lee, J. Towards the limit of network quantization. In *Proc. of the 5th Int. Conf. Learning Representations (ICLR 2017)*, Toulon, France, April 24–26 2017.

Han, S., Mao, H., and Dally, W. J. Deep compression: Compressing deep neural networks with pruning, trained quantization and Huffman coding. In *Proc. of the 4th Int. Conf. Learning Representations (ICLR 2016)*, San Juan, Puerto Rico, May 2–4 2016.

Idelbayev, Y. and Carreira-Perpiñán, M. Á. 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)*, pp. 8046–8056, Seattle, WA, June 14–19 2020a.

Idelbayev, Y. and Carreira-Perpiñán, M. Á. A flexible, extensible software framework for model compression based on the LC algorithm. arXiv:2005.07786, May 15 2020b.

Idelbayev, Y. and Carreira-Perpiñán, M. Á. Neural network compression via additive combination of reshaped, low-rank matrices. In *Proc. of the 2021 Data Compression Conference (DCC'21)*, Snowbird, UT, March 23–26 2021a.

Idelbayev, Y. and Carreira-Perpiñán, M. Á. 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, June 6–11 2021b.

Idelbayev, Y. and Carreira-Perpiñán, M. Á. Beyond FLOPs in low-rank compression of neural networks: Optimizing device-specific inference runtime. Submitted, 2021c.

Idelbayev, Y. and Carreira-Perpiñán, M. Á. Comprehensive empirical comparison of quantization, pruning and low-rank compression using the lc toolkit. Submitted, 2021d.

Leng, C., Li, H., Zhu, S., and Jin, R. Extremely low bit neural network: Squeeze the last bit out with ADMM. In *Proc. of the 32nd AAAI Conference on Artificial Intelligence (AAAI 2018)*, pp. 3466–3473, New Orleans, LA, February 2–7 2018.

Rastegari, M., Ordonez, V., Redmon, J., and Farhadi, A. XNOR-net: ImageNet classification using binary convolutional neural networks. In Leibe, B., Matas, J., Sebe, N., and Welling, M. (eds.), *Proc. 14th European Conf. Computer Vision (ECCV'16)*, pp. 525–542, Amsterdam, The Netherlands, October 11–14 2016.

Russakovsky, O., Deng, J., Su, H., Krause, J., Satheesh, S., Ma, S., Huang, Z., Karpathy, A., Khosla, A., Bernstein, M., Berg, A. C., and Fei-Fei, L. ImageNet large scale visual recognition challenge. *Int. J. Computer Vision*, 115 (3):211–252, December 2015.

Tung, F. and Mori, G. 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)*, pp. 7873–7882, Salt Lake City, UT, June 18–22 2018.

Wu, J., Leng, C., Wang, Y., Hu, Q., and Cheng, J. Quantized convolutional neural networks for mobile devices. In *Proc. of the 2016 IEEE Computer Society Conf. Computer Vision and Pattern Recognition (CVPR'16)*, pp. 4020–4028, Las Vegas, NV, June 26 – July 1 2016.

Yang, H., Gui, S., Zhu, Y., and Liu, J. 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)*, pp. 2175–2185, Seattle, WA, June 14–19 2020.

Yu, R., Li, A., Chen, C.-F., Lai, J.-H., Morariu, V. I., Han, X., Gao, M., Lin, C.-Y., and Davis, L. S. NISP: Pruning networks using neuron importance score propagation. In *Proc. of the 2018 IEEE Computer Society Conf. Computer Vision and Pattern Recognition (CVPR'18)*, pp. 9194–9203, Salt Lake City, UT, June 18–22 2018.

Zhou, S., Ni, Z., Zhou, X., Wen, H., Wu, Y., and Zou, Y. Dorefa-net: Training low bitwidth convolutional neural networks with low bitwidth gradients. arXiv:1606.06160, July 17 2016.