A flexible, extensible software framework for model compression based on the LC algorithm

Yerlan Idelbayev and Miguel Á. Carreira-Perpiñán
Department of CSE, University of California, Merced
http://eecs.ucmerced.edu

The code is available at:
https://github.com/UCMerced-ML/LC-model-compression
The fundamental problem of model compression: what to choose?

Images are from the slides of Miguel Á. Carreira-Perpiñán
In principle, we want to explore all possible combinations, and select the best. But:

- Many compression schemes \(\Rightarrow\) many algorithms
- How to maintain a library of many compressions?
- How to make it user friendly?
  - many algorithms \(\Rightarrow\) many failure points
- How to make it extensible and easily maintainable?

We propose a software based on the Learning-Compression (LC) algorithm:

- single algorithm—many compressions
- extensible, modular, and fast
- impressive compression results
- open source: BSD 3-clause license
The LC algorithm: formulation

Given a network with weights $w$ and loss $L$:

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

The compression details are abstracted in $\Delta(\Theta)$:

- e.g., low-rank: $\Delta(\Theta) = UV^T$ where $\Theta = \{U, V\}$

![Diagram showing the w-space (uncompressed models) and the feasible models $C$ (decompressible by $\Delta$). The optimal compressed model $w^*$ is marked. The decompression mapping is illustrated by $\Delta: \Theta \rightarrow w \in \mathbb{R}^P$.](figure from the slides of Miguel Á. Carreira-Perpiñán)
The problem (1) can be solved by alternation of these two steps (while driving \( \mu \to \infty \)), which form the basis of our software:

- **Learning (L) step:**
  \[
  \min_w L(w) + \frac{\mu}{2} \| w - \Delta(\Theta) \|^2
  \]
  
  - This is a regular training of the model, but with a quadratic regularization term.
  - When you train a network, you already have the L step.

- **Compression (C) step:**
  \[
  \min_{\Theta} \| w - \Delta(\Theta) \|^2
  \]
  
  - Independent of the loss, neural network structure, and the dataset.
  - We provide a library of different C steps for many different compressions.
## The library of implemented compressions

<table>
<thead>
<tr>
<th>Type</th>
<th>Forms</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Quantization</strong></td>
<td>Adaptive Quantization into ( {c_1, c_2, \ldots c_K} )</td>
</tr>
<tr>
<td></td>
<td>Binarization into ( {-1, 1} ) and ( {-c, c} )</td>
</tr>
<tr>
<td></td>
<td>Ternarization into ( {-c, 0, c} )</td>
</tr>
<tr>
<td></td>
<td>( \ell_0 )-constraint (s.t., ( |w|_0 \leq \kappa ))</td>
</tr>
<tr>
<td></td>
<td>( \ell_1 )-constraint (s.t., ( |w|_0 \leq \kappa ))</td>
</tr>
<tr>
<td></td>
<td>( \ell_0 )-penalty ( (\alpha |w|_0) )</td>
</tr>
<tr>
<td></td>
<td>( \ell_1 )-penalty ( (\alpha |w|_1) )</td>
</tr>
<tr>
<td><strong>Pruning</strong></td>
<td>Low-rank compression to a given rank</td>
</tr>
<tr>
<td></td>
<td>Low-rank with <em>automatic</em> rank selection for FLOPs reduction</td>
</tr>
<tr>
<td></td>
<td>Low-rank with <em>automatic</em> rank selection for storage compression</td>
</tr>
<tr>
<td><strong>Low-rank</strong></td>
<td>Quantization + Pruning</td>
</tr>
<tr>
<td></td>
<td>Quantization + Low-rank</td>
</tr>
<tr>
<td></td>
<td>Pruning + Low-rank</td>
</tr>
<tr>
<td></td>
<td>Quantization + Pruning + Low-rank</td>
</tr>
<tr>
<td><strong>Additive Combinations</strong></td>
<td></td>
</tr>
</tbody>
</table>
Easy exploration of compressions

Having an L-step implementation *(you only need one)*, definition of compression is very simple:

### quantize each layer with separate codebooks

```
compression_tasks = {
    Param(l1.weight): (AsVector, AdaptiveQuantization(k=2)),
    Param(l2.weight): (AsVector, AdaptiveQuantization(k=2)),
    Param(l3.weight): (AsVector, AdaptiveQuantization(k=2))
}
```

### prune all but 5%

```
compression_tasks = {
    Param([l1.weight, l2.weight, l3.weights]):
    (AsVector, ConstraintL0Pruning(kappa=13310))  # 13310 = 5%
}
```

### 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))
}
```
Our framework achieves competitive results in many compression schemes. For example, using our code for rank-selection, we can achieve considerable speed-up on AlexNet:

<table>
<thead>
<tr>
<th>Model</th>
<th>GPU of Jetson Nano time, ms</th>
<th>speed-up</th>
</tr>
</thead>
<tbody>
<tr>
<td>AlexNet</td>
<td>23.36</td>
<td>1.00</td>
</tr>
<tr>
<td>L_1</td>
<td>11.59</td>
<td>2.01</td>
</tr>
<tr>
<td>L_2</td>
<td>8.88</td>
<td>2.63</td>
</tr>
<tr>
<td>L_3</td>
<td>7.11</td>
<td>3.29</td>
</tr>
</tbody>
</table>

\( \rho_{\text{FLOPs}} \) — reduction in FLOPs. See Idelbayev and Carreira-Perpiñán [9] for full details.
Example: Additive compressions to achieve smallest AlexNet-s

The codebase allows easy exploration of new compression mechanisms. For example, we can further compress low-rank AlexNet models to target storage:

<table>
<thead>
<tr>
<th>Model</th>
<th>top-1 size, MB</th>
<th>MFLOPs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Caffe-AlexNet Jia et al. [1]</td>
<td>42.70</td>
<td>243.5</td>
</tr>
<tr>
<td>L₁ → Q (1-bit) + P (0.25M)</td>
<td><strong>41.56</strong></td>
<td>3.7</td>
</tr>
<tr>
<td>L₂ → Q (1-bit) + P (0.25M)</td>
<td>41.91</td>
<td>2.8</td>
</tr>
<tr>
<td>L₃ → Q (1-bit) + P (0.25M)</td>
<td>42.85</td>
<td><strong>2.2</strong></td>
</tr>
<tr>
<td>AlexNet-QNN of Wu et al. [10]</td>
<td>44.24</td>
<td>13.0</td>
</tr>
<tr>
<td>P→₁Q of Han et al. [11]</td>
<td>42.78</td>
<td>6.9</td>
</tr>
<tr>
<td>P→₂Q of Choi et al. [12]</td>
<td>43.80</td>
<td>5.9</td>
</tr>
<tr>
<td>P→₃Q of Tung and Mori [13]</td>
<td>42.10</td>
<td>4.8</td>
</tr>
<tr>
<td>P→₄Q of Yang et al. [14]</td>
<td>42.48</td>
<td>4.7</td>
</tr>
<tr>
<td>P→₅Q of Yang et al. [14]</td>
<td>43.40</td>
<td>3.1</td>
</tr>
<tr>
<td>Filter pruning of Li et al. [7]</td>
<td>43.17</td>
<td>232.0</td>
</tr>
</tbody>
</table>

![Graph showing top-1 test error (%) and storage ratio ρₛ for various compression schemes](image)
Source code and library features

Our code is written in Python using PyTorch, and open source under BSD 3-clause license:

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

Using the provided code, you will be able to:

• replicate all reported experiments
• compress your own models with many available compression schemes

Our library is:

• modular and easily extensible
• only requires the L-step implementation: the regular learning of the model (using SGD)
• based on solid optimization principles
• single algorithm—many compressions
• time proven (development since 2017), with many publications [9, 15, 16, 17, 18, 19, 20, 21]
References


