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

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
Machine learning and neural networks

Neural networks have established state-of-the-art performance nearly in every machine learning task:

- Natural Language Processing (NLP): dialogs systems, translators...
- Computer vision: image and video recognition, classification...
- Speech processing: speech-to-text, audio synthesis...
- Various signal enhancement tasks (photo, audio, video)

The models trained on these tasks have significant practical importance
Some famous neural network use cases

Voice Assistants

"Hey, Siri"

Photo/Video/AR

Translators

Current mobile devices contain dozens of neural networks, some of them running non-stop!

The images are obtained from official websites or blogposts of the services.
The improvement of NN performance and model compression

The main factor behind the significant improvement in NN accuracies is **larger size and more computation!**

- While largest neural networks 5 years ago contained millions of parameters (VGG16: 138M) current networks routinely have billions (GPT3: 175B), an exponential growth!
- Inference using such models is **expensive** and requires **high-end hardware**
- What if you want to save money or simply do not have such hardware?

This motivates the problem of model compression: reduce model's size/energy consumptions/computational requirements while maintaining the accuracy
The fundamental problem of model compression: what to choose?
Challenges

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 (builtin reusability)
- extensible, modular, and fast
- impressive compression results
- based on solid optimization principles
- open source: BSD 3-clause license
The LC algorithm: MCCO formulation

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

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

Compression details are abstracted in $\Delta(\Theta)$:

- low-rank: $\Delta(\Theta) = UV^T$ where $\Theta = \{U, V\}$
- pruning: $\Delta(\Theta) = \Theta$ s.t. $\|\Theta\|_0 \leq \kappa$

feasible models $C$ (decompressible by $\Delta$)

feasible set $C = \{w \in \mathbb{R}^P : w = \Delta(\Theta) \text{ for } \Theta \in \mathbb{R}^Q \}$ when $\lambda = 0$
The LC algorithm (cont.)

Reformulate using penalty methods and optimize the following while driving $\mu \rightarrow \infty$:

$$\min_{w, \Theta} L(w) + \lambda C(\Theta) + \frac{\mu}{2} \|w - \Delta(\Theta)\|^2$$

Apply alternating optimization wrt $w$ and $\Theta$, which gives the (LC) algorithm:

- **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.
  - L step is independent of compression mechanism.
  - We will use SGD and standard NN software

- **Compression (C) step:**

  $$\min_{\Theta} \|w - \Delta(\Theta)\|^2 + \lambda C'(\Theta)$$

  - C step has a form of optimal projection of the weights and independent of the dataset.
  - Many well studied cases with fast solutions
The LC algorithm: pseudocode

**input** training data and model with parameters $w$

$w \leftarrow \overline{w} = \arg \min_w L(w)$

pretrained model

$\Theta \leftarrow \Theta^{DC} = \Pi(\overline{w})$

init compression

$\beta \leftarrow 0$

for $\mu = \mu_0 < \mu_1 < \cdots < \infty$

$w \leftarrow \arg \min_w L(w) + \frac{\mu}{2} \| w - \Delta(\Theta) - \frac{1}{\mu} \beta \|_2^2$

L step

$\Theta \leftarrow \arg \min_{\Theta} \| w - \frac{1}{\mu} \beta - \Delta(\Theta) \|_2^2 + \lambda C(\Theta)$

C step

$\beta \leftarrow \beta - \mu (w - \Delta(\Theta))$

multipliers step

if $\| w - \Delta(\Theta) \|$ is small enough then exit the loop

return $w, \Theta$

---

class LCAlgorithm():

# Housekeeping code skipped
def run(self):

self.mu = 0

self.c_step(step_number=0)

for i, mu in enumerate(self.mu_schedule):

self.mu = mu

self.l_step(i) # user defined

self.c_step(i) # library call

self.multipliers_step()
The LC software

• Written in python using NumPy and PyTorch

• L step  We hand off the L step to the user through the lambda functions.

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

• C step  Many compression are implemented, and you can chose any. If desired, you can add your own compression too by extending the CompressionTypeBase: class.

```python
class ScaledBinaryQuantization(CompressionTypeBase):
    def compress(self, data):
        a = np.mean(np.abs(data))
        quantized = 2 * a * (data > 0) - a
        return quantized
```
The LC software: currently 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><strong>Pruning</strong></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>Low-rank</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>Additive Combinations</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>
</tbody>
</table>
The LC software: the ease of exploration

- **Mix-and-match through compression tasks.** We structured the software in such way that any compression can be applied to any part of the model, and you can mix them as well!

  For example, the following semantics:

  \[
  (\text{layer 1, layer 3}) \rightarrow \text{adaptive quantization } k = 6 \\
  (\text{layer 2}) \rightarrow \text{low-rank with } r = 3
  \]

  Translates into the following code:

  ```python
  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))
  }
  ```
Example: Apples-to-apples comparison between compressions

Tradeoff on LeNet300

Tradeoff on VGG16

See more on it in [1]
Example: Tradeoff curves for CIFAR10 networks using low-rank compression

Error-compression space of test error, inference FLOPs and number of parameters (ball size for each net). Different networks have different colors. R — reference network.

Results of our algorithm over different $\lambda$ values for a given network span a curve, shown as connected circles. Other published results using low-rank are shown as isolated circles; filter pruning results shown as isolated squares.
Example: Low-rank AlexNet models with automatic rank selection

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></th>
<th>MFLOPs</th>
<th>top-1</th>
<th>top-5</th>
<th>(\rho_{\text{FLOPs}})</th>
</tr>
</thead>
<tbody>
<tr>
<td>Caffe-AlexNet [10]</td>
<td>724</td>
<td>42.70</td>
<td>19.80</td>
<td>1.00</td>
</tr>
<tr>
<td><strong>our scheme 1</strong>, (\lambda = 0.20)</td>
<td>231</td>
<td>41.56</td>
<td>18.72</td>
<td>3.13</td>
</tr>
<tr>
<td><strong>our scheme 2</strong>, (\lambda = 0.20)</td>
<td><strong>152</strong></td>
<td><strong>41.03</strong></td>
<td><strong>18.23</strong></td>
<td><strong>4.78</strong></td>
</tr>
<tr>
<td>Kim et al. [11], Tucker</td>
<td>272</td>
<td>n/a</td>
<td>21.67</td>
<td>2.66</td>
</tr>
<tr>
<td>Tai et al. [12], scheme 2</td>
<td>185</td>
<td>n/a</td>
<td>20.34</td>
<td>3.90</td>
</tr>
<tr>
<td>Wen et al. [2], scheme 1</td>
<td>269</td>
<td>n/a</td>
<td>20.14</td>
<td>2.69</td>
</tr>
<tr>
<td>Kim et al. [13], scheme 2</td>
<td>272</td>
<td>43.40</td>
<td>20.10</td>
<td>2.66</td>
</tr>
<tr>
<td>Yu et al. [7], filter prun.</td>
<td>232</td>
<td>44.13</td>
<td>n/a</td>
<td>3.12</td>
</tr>
<tr>
<td>Li et al. [14], filter prun.</td>
<td>334</td>
<td>43.17</td>
<td>n/a</td>
<td>2.16</td>
</tr>
<tr>
<td>Ding et al. [15], filter prun.</td>
<td>492</td>
<td>43.83</td>
<td>20.47</td>
<td>1.47</td>
</tr>
</tbody>
</table>

Compression with our algorithm vs published work using low-rank methods and structured pruning.

\(\rho_{\text{FLOPs}}\) — reduction in FLOPs.

See [16] for full details.
Example: Additive compressions to achieve smallest AlexNets

The frameworks and software allows easy exploration of new compressions. For example, how about additive combination of quantization and pruning?

<table>
<thead>
<tr>
<th>Model</th>
<th>top-1 size, MB MFLOPs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Caffe-AlexNet Jia et al. [10]</td>
<td>42.70 243.5 724</td>
</tr>
<tr>
<td>$L_1 \rightarrow Q$ (1-bit) + $P$ (0.25M)</td>
<td>39.67 3.7 228</td>
</tr>
<tr>
<td>$L_2 \rightarrow Q$ (1-bit) + $P$ (0.25M)</td>
<td>40.19 2.8 185</td>
</tr>
<tr>
<td>$L_3 \rightarrow Q$ (1-bit) + $P$ (0.25M)</td>
<td>41.27 2.1 152</td>
</tr>
<tr>
<td>AlexNet-QNN of Wu et al. [17]</td>
<td>44.24 13.0 175</td>
</tr>
<tr>
<td>$P \rightarrow 1 Q$ of Han et al. [18]</td>
<td>42.78 6.9 724</td>
</tr>
<tr>
<td>$P \rightarrow 2 Q$ of Choi et al. [19]</td>
<td>43.80 5.9 724</td>
</tr>
<tr>
<td>$P \rightarrow 3 Q$ of Tung and Mori [20]</td>
<td>42.10 4.8 724</td>
</tr>
<tr>
<td>$P \rightarrow 4 Q$ of Yang et al. [21]</td>
<td>42.48 4.7 724</td>
</tr>
<tr>
<td>$P \rightarrow 5 Q$ of Yang et al. [21]</td>
<td>43.40 3.1 724</td>
</tr>
<tr>
<td>filter pruning of Li et al. [14]</td>
<td>43.17 232.0 334</td>
</tr>
</tbody>
</table>

More on it in [25]
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
• requires minimal coding
• based on solid optimization principles
• single algorithm—many compressions
• time proven (since 2017), with many publications [1, 16, 26, 27, 28, 29, 30, 31]
• we welcome comments, bug reports, requests for functionality, contributions, etc.
References


[15] X. Ding, G. Ding, Y. Guo, J. Han, and C. Yan, “Approximated oracle filter pruning for destructive CNN width optimization.”


[27] M. Á. Carreira-Perpiñán and Y. Idelbayev, “Model compression as constrained optimization, with application to neural nets. Part II: Quantization.”

[28] ——, “Learning-compression” algorithms for neural net pruning.”

[29] Y. Idelbayev and M. Á. Carreira-Perpiñán, “Neural network compression via additive combination of reshaped, low-rank matrices.”


[31] ——, “Beyond FLOPs in low-rank compression of neural networks: Optimizing device-specific inference runtime.”