A Flexible, Extensible Software Framework for Neural Net Compression

Anonymous Author(s)
Affiliation
Address
`email`

Abstract

We propose a software framework based on ideas of the Learning-Compression algorithm [1, 2, 4], that allows one to compress any neural network by different compression mechanisms (pruning, quantization, low-rank, etc.). By design, the learning of the neural net (handled by SGD) is decoupled from the compression of its parameters (handled by a signal compression function), so that the framework can be easily extended to handle different combinations of neural net and compression type. In addition, it has other advantages, such as easy integration with deep learning frameworks, efficient training time, competitive practical performance in the loss-compression tradeoff, and reasonable convergence guarantees. Our toolkit is written in Python and Pytorch and we plan to make it available by the workshop time, and eventually open it for contributions from the community.

With the great success of neural network in solving practical problems in various fields (vision, NLP, etc.) there has been an emergence of research in neural network compression techniques that allows to compress these large models in terms of memory, computation and/or power requirements. At present many ad-hoc solutions have been proposed that typically solve *one specific type of compression* (binarization and quantization [2, 5, 8, 11, 18, 23, 24], pruning [9, 10, 15, 16, 21], low-rank or tensor factorization [6, 7, 12–14, 17, 19, 20, 22], etc.), as well as several submissions to the present workshop.

Among the various research strands in neural net compression, in our view a fundamental problem is that in practice one does not know what the best type of compression (or combination of compression types) may be the best for a given network. In principle, it would be possible to try different existing algorithms, assuming one can find an implementation for them. We seek a solution that directly addresses this problem and can potentially allow non-expert end users to compress models easily and effectively[1]. It is based on a recently proposed compression framework, the LC algorithm [1, 2, 4], that by design separates the "learning" part of the problem (which involves the dataset, neural net model and loss function) from the "compression" part (how the network parameters will be compressed). This has the advantage of modularity (a pillar of structured programming and software development): we can change the compression type by simply calling a different compression routine (e.g. k-means instead of the SVD) within the overall algorithm *in a principled way*, with no other changes to the algorithm. We briefly describe the LC algorithm below and mention additional advantages it provides—such as easy integration with deep learning frameworks, efficient training time, competitive practical performance in the loss-compression tradeoff, and reasonable convergence guarantees.

[1] There have been some attempts to include compression as a black-box routines into deep learning frameworks (e.g. `https://www.tensorflow.org/performance/post_training_quantization`), but they are limited to a subset of simple methods.

Submitted to 32nd Conference on Neural Information Processing Systems (NIPS 2018). Do not distribute.

In this paper, we describe our ongoing efforts in building a software implementation that can capitalize on the modularity of the LC algorithm. At present this handles 1) (C step) various forms of quantization, pruning and low-rank compression, and we will soon add combinations of those and further compression types; and 2) (L step) various types of deep net models. Our framework is written in Python and Pytorch. We plan to make it available online as open source by the time of the workshop. We also hope that interested researchers and developers will eventually contribute their own routines for signal compression or for training of specific neural net architectures.

1 Model compression as a constrained optimization problem

Assume we have a previously trained model with weights $\overline{\mathbf{w}} = \arg\min_{\mathbf{w}} L(\mathbf{w})$. This is our *reference* model, which represents the best loss we can achieve without compression. The "Learning-Compression" paper [1] defines *compression* as finding a low-dimensional parameterization $\boldsymbol{\Delta}(\boldsymbol{\Theta})$ of $\mathbf{w}$ in terms of $Q < P$ parameters $\boldsymbol{\Theta}$. The goal is to find such $\boldsymbol{\Theta}$ that its corresponding model has (locally) optimal loss. Therefore the *model compression as a constrained optimization* problem is defined as:

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

Compression and decompression are usually seen as algorithms, but here they are regarded as mathematical mappings in parameter space. The *decompression mapping* $\boldsymbol{\Delta}$: $\boldsymbol{\Theta} \in \mathbb{R}^Q \rightarrow \mathbf{w} \in \mathbb{R}^P$ maps a low-dimensional parameterization to uncompressed model weights and the *compression mapping* $\boldsymbol{\Pi}(\mathbf{w}) = \arg\min_{\boldsymbol{\Theta}} \|\mathbf{w} - \boldsymbol{\Delta}(\boldsymbol{\Theta})\|^2$ behaves as its "inverse".

The problem in (1) is constrained, nonlinear, and potentially non-differentiable w.r.t. $\boldsymbol{\Theta}$ (e.g. binarization). The LC-algorithm is obtained by converting this problem to an equivalent formulation using penalties 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:** $\min_{\mathbf{w}} L(\mathbf{w}) + \frac{\mu}{2}\|\mathbf{w} - \boldsymbol{\Delta}(\boldsymbol{\Theta})\|^2$. This is a regular training of the uncompressed model but with a quadratic regularization term. *This step is independent of the compression type.*
- **C (compression) step:** $\min_{\boldsymbol{\Theta}} \|\mathbf{w} - \boldsymbol{\Delta}(\boldsymbol{\Theta})\|^2 \Leftrightarrow \boldsymbol{\Theta} = \boldsymbol{\Pi}(\mathbf{w})$. This means finding the best (lossy) compression of $\mathbf{w}$ (the current uncompressed model) in the ℓ_2 sense (orthogonal projection on the feasible set), and corresponds to our definition of the compression mapping $\boldsymbol{\Pi}$. *This step is independent of the loss, training set and task.*

The LC algorithm defines a continuous path $(\mathbf{w}(\mu), \boldsymbol{\Theta}(\mu))$ which, under some mild assumptions, converges to a stationary point (typically a minimizer) of the constrained problem. The beginning of this path, for $\mu \rightarrow 0^+$, corresponds to training the reference model and then compressing it disregarding the loss (*direct compression*), a simple but suboptimal approach popular in practice.

Optimizing the L and C steps The L step can be solved by stochastic gradient descent (clipping the step sizes so they never exceed $\frac{1}{\mu}$). The C step can be solved by calling a compression routine corresponding to the desired compression type. For example, for quantization with an adaptive codebook we can run k-means with a codebook of size K weights [2]; for pruning, we can prune all but the top-κ weights (where κ depends on the sparsifying norm used) [3]. Fig. 1 gives a pseudo-code for the LC algorithm.

2 Overall software approach

In the pseudo-code there is only one compression mapping $\boldsymbol{\Delta}(\boldsymbol{\Theta})$, however, practically we want to mix and match, e.g. compress first layer using quantization and all remaining ones using pruning. Therefore, there might be multiple constraints $\mathbf{w}_i = \boldsymbol{\Delta}_i(\boldsymbol{\Theta}_i)$ such that $\mathbf{w}_i \subset \mathbf{w}$ where $\mathbf{w}$ is a set of all weights of the neural network. This means, during the C-step of the LC algorithm, instead of having one optimal compression $\boldsymbol{\Pi}(\mathbf{w})$, we have multiple $\boldsymbol{\Pi}_i(\mathbf{w}_i)$ which separates from each other and can be done in parallel.

Data structures and functions defined by library. Library defines several data structures that are necessary for internal housekeeping. It abstracts away from the architecture of the neural network, and sees its weights as a vector containing P values where P is cardinality of union of all $\mathbf{w}_i$, e.g. $P = |\mathbf{w}_1 \cup \mathbf{w}_2 \cup \cdots \cup \mathbf{w}_k|$. This is achieved by a *weights view* data structure.

The compression mappings assumes a specific structure, e.g. low-rank compression expects a matrix, while pruning expects a vector. Therefore, we define a *compression view* data structure that allows to map $\mathbf{w}_i$ to the format suitable to a *compression function*,

The *compression function* ($\mathbf{\Pi}_i$) obtains $\mathbf{w}_i$ in the suitable format and computes $\mathbf{\Pi}_i(\mathbf{w}_i)$. Library comes equipped with a number of implemented compression functions: for quantization, pruning and low-rank. For example, if user wants no more than κ nonzero items, decompression mapping has form $\mathbf{\Delta}(\mathbf{\Theta}) = \mathbf{\Theta}$ s.t. $\|\mathbf{\Theta}\| \le \kappa$, C-step will be $\mathbf{\Pi}_i(\mathbf{w}_i) = \min_{\mathbf{\Theta}_i} \|\mathbf{w}_i - \mathbf{\Theta}_i\|^2$ s.t. $\|\mathbf{\Theta}_i\| \le \kappa$ which solved by zeroing all but top κ values of $\mathbf{w}_i$ in magnitude. Here we give a snippet from library that performs that:

```
pruned = np.zeros_like(weights)
indx = np.argsort(np.abs(weights), kind='mergesort')
remaining_indx = indx[-.kappa:]
pruned[remaining_indx] = weights[remaining_indx]
```

Input from the user. For compression of a neural network, the software needs following things from the user: *L-step* implementation, *list of compression tasks* and a list of μ-values $(\mu_0, \mu_1, \dots, \mu_m)$. Note that the dataset, the loss function of the neural network is abstracted away from the library.

An *implementation of L-step* is a `Python` function which will be invoked with a new value of penalty (the term $\mathbf{\Delta}(\mathbf{\Theta})$) and it must return an updated $\mathbf{w}$ that minimizes the $L(\mathbf{w}) + {}^{\mu}/_{2}\|\mathbf{w} - \mathbf{\Delta}(\mathbf{\Theta})\|^2$. This step is similar to the reference network learning and requires minimal modifications to include penalty terms. We also note that it is independent of compression, and needs to be done once for a network, and can be re-used for other compression and combinations.

A *list of compression tasks* is list of tuples where each tuple contains: *compression view* of $\mathbf{w}_i$ and *compression function* $\mathbf{\Pi}_i$ which maps subset of weights $\mathbf{w}$ to a specific compression user wants to achieve, e.g. first layer to be quantized, second layer to be pruned.

Library operations. As soon as user invokes `lc.run` function with inputs described in previous paragraph, library initializes and/or parses necessary internal data structures and enters the loop over μ-values. For each μ value it invokes user supplied *L-step implementation* function and immediately afterwards, in parallel, traverses through *list of compression tasks*, obtains updated compression-view of $\mathbf{w}_i$ and passes it to compression function $\mathbf{\Pi}_i$ to update $\mathbf{\Theta}_i$.

3 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. Both are mature fields with well-understood algorithms and efficient implementations. Both converge in the problem of model compression, where we seek the smallest model (in the sense of memory, inference time or energy, etc.). The LC algorithm achieves this by seamlessly integrating the existing algorithms to train deep nets and to compress a signal. Here, we seek to develop an extensible software framework that can easily plug in existing deep net training techniques with existing signal compression techniques and their combinations. We intend to make the software open source by the time of the workshop and eventually to seek contributions from the community.

input training data and model with parameters (weights) $\mathbf{w}$

$\mathbf{w} \leftarrow \overline{\mathbf{w}} = \arg\min_{\mathbf{w}} L(\mathbf{w})$ — reference model

$\mathbf{\Theta} \leftarrow \mathbf{\Theta}^{\text{DC}} = \mathbf{\Pi}(\overline{\mathbf{w}}) = \arg\min_{\mathbf{\Theta}} \|\overline{\mathbf{w}} - \mathbf{\Delta}(\mathbf{\Theta})\|^2$ — compress reference model

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

  $\mathbf{w} \leftarrow \arg\min_{\mathbf{w}} L(\mathbf{w}) + \frac{\mu}{2}\|\mathbf{w} - \mathbf{\Delta}(\mathbf{\Theta})\|^2$ — L step: learn model

  $\mathbf{\Theta} \leftarrow \mathbf{\Pi}(\mathbf{w}) = \arg\min_{\mathbf{\Theta}} \|\mathbf{w} - \mathbf{\Delta}(\mathbf{\Theta})\|^2$ — C step: compress model

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

return $\mathbf{w}$, $\mathbf{\Theta}$

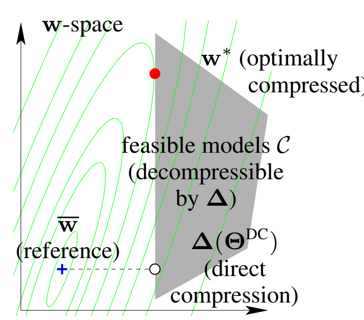

Figure 1: *Left*: A pseudo-code of LC algorithm. *Right*: an illustration of the idea of model compression by constrained optimization.

References

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

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

[3] M. Á. Carreira-Perpiñán and Y. Idelbayev. Learning-compression algorithms for neural network pruning. In *Proc. of the 2017 IEEE Computer Society Conf. Computer Vision and Pattern Recognition (CVPR'17)*, Honolulu, HI, July 21–26 2017.

[4] M. Á. Carreira-Perpiñán and Y. Idelbayev. "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, June 18–22 2018.

[5] M. Courbariaux, Y. Bengio, and J.-P. David. BinaryConnect: Training deep neural networks with binary weights during propagations. In C. Cortes, N. D. Lawrence, D. D. Lee, M. Sugiyama, and R. Garnett, editors, *Advances in Neural Information Processing Systems (NIPS)*, volume 28, pages 3105–3113. MIT Press, Cambridge, MA, 2015.

[6] M. Denil, B. Shakibi, L. Dinh, M. Ranzato, and N. de Freitas. Predicting parameters in deep learning. In C. J. C. Burges, L. Bottou, M. Welling, Z. Ghahramani, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems (NIPS)*, volume 26, pages 2148–2156. MIT Press, Cambridge, MA, 2013.

[7] E. L. Denton, W. Zaremba, J. Bruna, Y. LeCun, and R. Fergus. Exploiting linear structure within convolutional networks for efficient evaluation. In Z. Ghahramani, M. Welling, C. Cortes, N. D. Lawrence, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems (NIPS)*, volume 27, pages 1269–1277. MIT Press, Cambridge, MA, 2014.

[8] Y. Gong, L. Liu, M. Yang, and L. Bourdev. Compressing deep convolutional networks using vector quantization. In *Proc. of the 3rd Int. Conf. Learning Representations (ICLR 2015)*, San Diego, CA, May 7–9 2015.

[9] S. Han, J. Pool, J. Tran, and W. Dally. Learning both weights and connections for efficient neural network. In C. Cortes, N. D. Lawrence, D. D. Lee, M. Sugiyama, and R. Garnett, editors, *Advances in Neural Information Processing Systems (NIPS)*, volume 28, pages 1135–1143. MIT Press, Cambridge, MA, 2015.

[10] B. Hassibi and D. G. Stork. Second order derivatives for network pruning: Optimal brain surgeon. In S. J. Hanson, J. D. Cowan, and C. L. Giles, editors, *Advances in Neural Information Processing Systems (NIPS)*, volume 5, pages 164–171. Morgan Kaufmann, San Mateo, CA, 1993.

[11] K. Hwang and W. Sung. Fixed-point feedforward deep neural network design using weights $+1$, 0, and -1. In *2014 IEEE Workshop on Signal Processing Systems (SiPS)*, pages 1–6, Belfast, UK, Oct. 20–22 2014.

[12] Y. Ioannou, D. Robertson, R. Cipolla, and A. Criminisi;. Deep roots: Improving cnn efficiency with hierarchical filter groups. In *Proc. of the 2017 IEEE Computer Society Conf. Computer Vision and Pattern Recognition (CVPR'17)*, pages 1231–1240, Honolulu, HI, July 21–26 2017.

[13] M. Jaderberg, A. Vedaldi, and A. Zisserman. Speeding up convolutional neural networks with low rank expansions. In M. Valstar, A. French, and T. Pridmore, editors, *Proc. of the 25th British Machine Vision Conference (BMVC 2014)*, Nottingham, UK, Sept. 1–5 2014.

[14] Y.-D. Kim, E. Park, S. Yoo, T. Choi, L. Yang, and D. Shin. Compression of deep convolutional neural networks for fast and low power mobile applications. In *Proc. of the 4th Int. Conf. Learning Representations (ICLR 2016)*, San Juan, Puerto Rico, May 2–4 2016.

[15] Y. LeCun, J. S. Denker, and S. A. Solla. Optimal brain damage. In D. S. Touretzky, editor, *Advances in Neural Information Processing Systems (NIPS)*, volume 2, pages 598–605. Morgan Kaufmann, San Mateo, CA, 1990.

[16] B. Liu, M. Wan, H. Foroosh, M. Tappen, and M. Pensky. Sparse convolutional neural networks. In *Proc. of the 2015 IEEE Computer Society Conf. Computer Vision and Pattern Recognition (CVPR'15)*, pages 806–814, Boston, MA, June 7–12 2015.

[17] A. Novikov, D. Podoprikhin, A. Osokin, and D. P. Vetrov. Tensorizing neural networks. In C. Cortes, N. D. Lawrence, D. D. Lee, M. Sugiyama, and R. Garnett, editors, *Advances in Neural Information Processing Systems (NIPS)*, volume 28, pages 442–450. MIT Press, Cambridge, MA, 2015.

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

[19] T. N. Sainath, B. Kingsbury, V. Sindhwani, E. Arısoy, and B. Ramabhadran. 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)*, pages 6655–6659, Vancouver, Canada, Mar. 26–30 2013.

[20] C. Tai, T. Xiao, Y. Zhang, X. Wang, and W. E. Convolutional neural networks with low-rank regularization. In *Proc. of the 4th Int. Conf. Learning Representations (ICLR 2016)*, San Juan, Puerto Rico, May 2–4 2016.

[21] W. Wen, C. Wu, Y. Wang, Y. Chen, and H. Li. Learning structured sparsity in deep neural networks. In D. D. Lee, M. Sugiyama, U. von Luxburg, I. Guyon, and R. Garnett, editors, *Advances in Neural Information Processing Systems (NIPS)*, volume 29, pages 2074–2082. MIT Press, Cambridge, MA, 2016.

[22] X. Zhang, J. Zou, K. He, and J. Sun. Accelerating very deep convolutional networks for classification and detection. *IEEE Trans. Pattern Analysis and Machine Intelligence*, 38(10): 1943–1955, Oct. 2016.

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

[24] C. Zhu, S. Han, H. Mao, and W. J. Dally. Trained ternary quantization. In *Proc. of the 5th Int. Conf. Learning Representations (ICLR 2017)*, Toulon, France, Apr. 24–26 2017.