A Programming System for Model Compression

Vinu Joseph
University of Utah
vinu@cs.utah.edu

Saurav Muralidharan
NVIDIA
sauravm@nvidia.com

Animesh Garg
University of Toronto, NVIDIA
garg@cs.toronto.edu

Michael Garland
NVIDIA
mgarland@nvidia.com

Ganesh Gopalakrishnan
University of Utah
ganesh@cs.utah.edu

Abstract

Deep neural networks frequently contain far more weights, represented at a higher precision, than is required for the specific task which they are trained to perform. Consequently, they can often be compressed using techniques such as weight pruning and quantization that reduce both model size and inference time without appreciable loss in accuracy. Compressing models before they are deployed can therefore result in significantly more efficient systems. However, while these benefits are desirable, finding the best compression strategy for a given neural network, target platform, and optimization objective often requires extensive experimentation. Moreover, finding optimal hyperparameters for a given compression strategy typically results in even more expensive, frequently manual, trial-and-error exploration. In this paper, we introduce a programmable system for model compression, called CONDENSA. Users programmatically compose simple operators, in Python, to build complex compression strategies. Given a strategy and a user-provided objective, such as minimization of running time, CONDENSA uses a novel sample-efficient constrained Bayesian optimization-based algorithm to automatically infer optimal sparsity ratios. Our experiments on three real-world image classification and language modeling tasks demonstrate memory footprint reductions of up to $65\times$ and runtime throughput improvements of up to 2.22x using at most 10 samples per search.

1 Introduction

Modern deep neural networks (DNNs) are complex, and often contain millions of parameters spanning dozens or even hundreds of layers [24, 29]. This complexity engenders substantial memory and runtime costs on hardware platforms at all scales. Recent work has demonstrated that DNNs are often over-provisioned and can be compressed without appreciable loss of accuracy. Model compression can be used to reduce both model memory footprint and inference latency using techniques such as weight pruning [23, 39], quantization [19], and low-rank factorization [30, 10]. Unfortunately, the requirements of different *compression contexts*—DNN structure, target hardware platform, and the user's optimization objective—are often in conflict. The recommended compression strategy for reducing inference latency may be different from that required to reduce total memory footprint. For example, for a Convolutional Neural Network (CNN), the former strategy may prune convolutional filters [36], while the latter may prune individual non-zero weights. Similarly, even for the *same optimization objective*, say reducing inference latency, one may employ filter pruning for a CNN, while prune 2D blocks of non-zero weights [18] for a language modeling network such as Transformer [52], since the latter has no convolutional layers. Thus, it is crucial to enable convenient expression of alternative compression schemes, yet none of today's model compression approaches help the designer tailor compression schemes to their needs.

Preprint. Under review.

Current approaches to model compression also require manual specification of compression hyperparameters, such as the target sparsity ratio, which is the proportion of zero-valued parameters in the compressed model vs. the original. Finding the best sparsity ratio often becomes a trial-and-error search in practice, since compression hyperparameter values vary unpredictably with changes in the compression context. This makes it difficult to provide users with a rule of thumb, much less a single number, to apply when faced with the need to select a hyperparameter value. Each trial in this approach has a huge cost (hours or days for larger models), as it requires training the compressed model to convergence, with most of these manually orchestrated trials ending up in unmet compression objectives. Thus, automation is a crucial requirement to support the needs of designers who must adapt a variety of neural networks to a broad spectrum of platforms targeting a wide range of tasks.

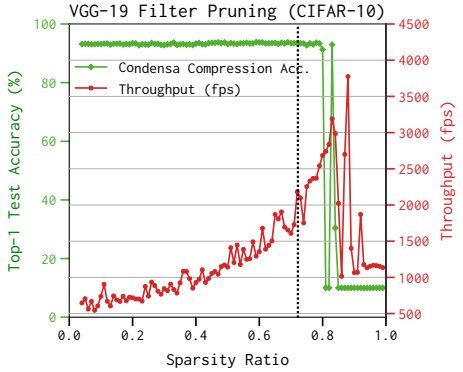


Figure 1: Top-1 accuracy (green) and Throughput (red) vs. sparsity ratio for VGG-19 on CIFAR-10. CONDENSA framework is designed to solve for constrained optimization of the form "maximize throughput, with a lower bound on accuracy". In this case, CONDENSA automatically discovers a sparsity ratio (vertical dashed line) and compresses the model to this ratio, improving throughput by $2.17\times$ and accuracy by 0.5%.

As an illustration of the level of automation provided by CONDENSA, consider the problem of improving the inference throughput of VGG-19 [49] on the CIFAR-10 image classification task [33]. Since VGG-19 is a convolutional neural network, one way to improve its inference performance on modern hardware such as GPUs is by pruning away individual convolutional filters [25]. Figure 1 shows the accuracy and throughput obtained by Condensa on this task. Here, we plot the compressed model's top-1 test accuracy and throughput as a function of the sparsity ratio (green and red lines, respectively).[1] Condensa's solution corresponds to a sparsity ratio of 0.73 and is depicted as the vertical dashed line. This result is significant for two reasons: (1) using the Condensa library, the filter pruning strategy employed for this experiment was expressed in less than 10 lines of Python code, and (2) the optimal sparsity ratio of 0.73 (shown as the vertical dashed line in the Figure) that achieves a state-of-the-art throughput of 2130 images/sec ($2.17\times$ improvement) and a top-1 accuracy *improvement* of 0.5% was obtained automatically by Condensa using a sample-efficient constrained Bayesian optimization algorithm. For this to work, the user didn't have to specify any sparsity ratios manually, and instead only had to define a domain-specific objective function to maximize (inference throughput, in this case).

As captured by this illustration, Condensa supports the expression of the overall *compression scheme* in Python using operators provided by the Condensa library. Since each scheme is a Python function, users are able to programmatically compose elementary schemes to build much more complex and practically interesting schemes. Condensa accepts a black-box objective function (also expressed in Python) on the target compressed model that is maximized or minimized to automatically find corresponding compression hyperparameters such as sparsity ratios. This programmable approach to model compression enables users to experiment and rapidly converge to an ideal scheme for a given compression context, avoiding manual trial and error search. Given Condensa's ability to support the expression of meaningful high level objective functions—for example, the throughput (images/sec) of a convolutional neural network—users are freed from the burden of having to specify compression hyperparameters manually.

2 CONDENSA Framework

Figure 2 shows a high-level overview of the Condensa framework. As shown on the left side of the figure, a user compresses a pre-trained model $\overline{\mathbf{w}}$ by specifying a compression scheme and an objective function f. Both the scheme and objective are specified in Python using operators from the Condensa library; alternatively, users may choose from a selection of commonly used built-in schemes and objectives. The Condensa library is described in more detail in Section 2.1. Apart from the operator

[1] Note that these curves are not known a priori and are often extremely expensive to sample; they are only plotted here to better place the obtained solution in context.

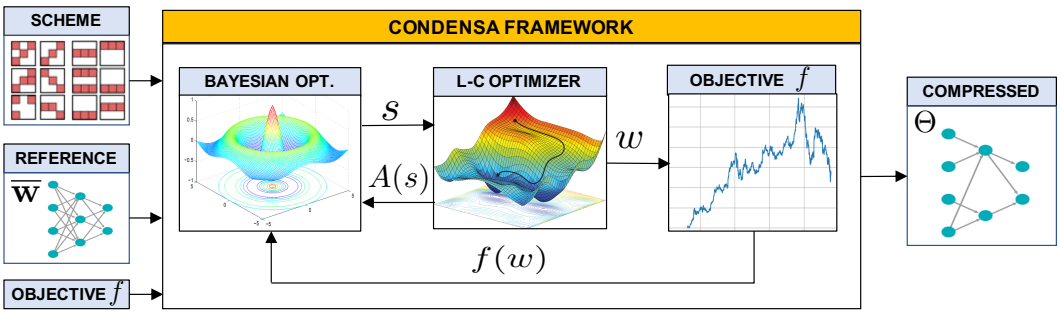


Figure 2: CONDENSA framework overview. The user provides the pre-trained model ($\overline{\mathbf{w}}$), a compression scheme, and an objective function f. CONDENSA uses the Bayesian and L-C optimizers to infer an optimal sparsity ratio s^* and corresponding compressed model Θ.

library, the core framework, shown in the middle of the figure, consists primarily of two components: (1) the constrained Bayesian optimizer for inferring optimal sparsity ratios, and (2) the L-C optimizer for accuracy recovery. These components interact with each other as follows: at each iteration, the Bayesian optimizer samples a sparsity ratio s, which is fed into the L-C optimizer. The L-C optimizer distributes this global sparsity across all the layers of the network and performs accuracy recovery (this process is described in more detail in Section 2.2), passing the final obtained accuracy $A(s)$ back to the Bayesian optimizer. The compressed model w obtained by the L-C optimizer is also used to evaluate the user-provided objective function f, the result of which is fed into the Bayesian optimizer. Based on these inputs ($A(s)$ and $f(w)$), the Bayesian optimizer decides the next point to sample. The sparsity ratio that satisfies both the accuracy and objective constraints (s^*) is used to obtain the final compressed model (denoted as Θ in the figure). The L-C and Bayesian optimizers are described in more detail in Sections 2.2 and 2.3, respectively, and the sparsity inference algorithm is presented in Algorithm 1.

Algorithm 1 Bayesian Hyperparameter Inference

Input: $\overline{\mathbf{w}}$,ϵ
Output: s^*
AcqFn $\leftarrow$ `ILS-UCB`($L = \overline{\mathbf{w}}_{acc} - \epsilon$, $s = (0, 1)$)
$s_{acc} \leftarrow$ `BayesOpt`($\mathcal{B}_f$ = L-C, AcqFn)
AcqFn $\leftarrow$ `GP-UCB`($s = (0, s_{acc})$)
$s^* \leftarrow$ `BayesOpt`($\mathcal{B}_f = f$, AcqFn)

function `BayesOpt`
Input: $\mathcal{B}_f$, AcqFn
Output: s
`GP` $\leftarrow$ `GP-Regressor.initialize()`
for t $\leftarrow$ 0, 1, 2, ... **do**
  $s_t \leftarrow$ `argmax`$_s$AcqFn($s|D_{1:t-1}$)
  $y_t \leftarrow f(s_t)$
  $D_{1:t} \leftarrow \{D_{1:t-1}, (s_t, y_t)\}$
  `GP.Update`($D_{1:t}$)
  if $t > 0$ **and** $s_t == s_{t-1}$ **then**
    return s_t
  end if
end for

2.1 Condensa Library

The Condensa Library provides a set of operators for constructing complex compression schemes programmatically in Python. Three sets of operators are currently supported: (1) the `quantize` and `dequantize` operators for converting network parameters from a 32-bit floating-point representation to a lower-precision one such as 16-bit floating-point, and in the opposite direction, respectively; (2) the

`prune` operator for unstructured magnitude-based pruning, and (3) the `filter_prune`, `neuron_prune`, and `blockprune` operators for pruning blocks of nonzeros (structure pruning). Each operator can be applied on a per-layer basis. A decompression scheme needs to be specified only when at least one of the operators in the corresponding compression scheme performs quantization, as described in Section 2.2.

Pre-built Schemes In addition to the layer-wise operators described above, the Condensa Library also includes a set of pre-built compression *schemes* that operate on the full model. Condensa includes schemes for unstructured and structured pruning, quantization, and composition of individual schemes. These schemes handle a number of low-level details such as magnitude threshold computation from a sparsity ratio, filter/neuron/block aggregation, etc., enabling non-expert users to quickly get started with Condensa without knowledge of its low-level implementation details. In the supplementary material, we have among other details (1) The current set of pre-built schemes is listed, along with their source code (2) Listing 1 shows example usage of the Condensa library.

2.2 Accuracy Recovery using L-C

As described earlier in this section, given a reference model, compression scheme, and compression hyperparameter values (obtained automatically by the Bayesian hyperparameter optimization subsystem described in Section 2.3), Condensa tries to recover any accuracy lost due to compression. While the compressed model, denoted as Θ, can be obtained by directly zeroing out lower-magnitude parameters from the reference model $\overline{\mathbf{w}}$ (a technique referred to as *direct compression*), the resulting model Θ is generally sub-optimal w.r.t. the loss since the latter is ignored in learning Θ. Instead, we desire an *accuracy recovery algorithm* that obtains an *optimally compressed model* with locally optimal loss. An effective accuracy recovery mechanism for Condensa must ideally have three important attributes: (1) able to handle all the compression operators supported by Condensa, (2) be efficient with relatively low overheads, and (3) provide optimality guarantees whenever possible. In this paper, we use the recently proposed L-C algorithm [6], since it satisfies all three of the above requirements. In L-C, model compression is formulated as a constrained optimization problem: $\min_{\mathbf{w},\boldsymbol{\Theta}} L(\mathbf{w}) \quad \text{s.t.} \quad \mathbf{w} = \mathcal{D}(\Theta)$ Here, the *decompression mapping* $\mathcal{D} : \Theta \in \mathbb{R}^Q \rightarrow \mathbf{w} \in \mathbb{R}^P$ maps a low-dimensional parameterization to uncompressed model weights, and the *compression mapping* $\mathcal{C}(\mathbf{w}) = \operatorname{argmin}_{\Theta} \|\mathbf{w} - \mathcal{D}(\Theta)\|^2$ behaves similar to the inverse of $\mathcal{D}$. This formulation naturally supports a number of well-known compression techniques. In particular, pruning is defined as $\mathbf{w} = \mathcal{D}(\Theta) = \Theta$ where $\mathbf{w}$ is real and Θ is constrained to have fewer nonzero values by removing (zeroing out) lower magnitude weights; low-precision approximation defines a constraint $w_i = \theta_i$ per parameter where w_i is in a higher-precision representation and θ_i is in a lower-precision one. While a number of non-convex algorithms may be used to solve the optimization, we focus on the augmented Lagrangian (AL) method [53] implemented in the L-C algorithm [6]. Due to space restrictions, we refer the reader to [6] for a more detailed description of the L-C algorithm.

2.3 Bayesian Hyperparameter Optimization

It is intuitive to split the problem of finding optimal sparsity ratios into two stages: (I) find the highest sparsity value that loses at most ϵ accuracy w.r.t the original uncompressed model, and (II) in a constrained sparsity regime obtained from stage I, optimize a user-provided objective function f (for eg., throughput, memory or footprint) and return the solution as the final sparsity ratio. It is worth noting that optimizing performance characteristics (accuracy, throughput, and so on) against sparsity ratios requires access to function f, and often assumes cheap function evaluation. For compression, each function evaluation can amount to optimizing the full model, is computationally prohibitive.

CONDENSA leverages black-box sample efficient Bayesian optimization to optimize objective f with accuracy constraints. Bayesian optimization solves for the minimum of a black-box function $f(\mathbf{x})$ on some bounded set $\mathcal{X}$, which we take to be a subset of $\mathbb{R}^D$ [42, 31]. BO methods construct a probabilistic model of f with sequential evaluation, and then exploits this model for sequential selection of information gathering actions – the choice of $x \in \mathcal{X}$. This procedure leverages all function evaluations instead of only local gradient approximations, and hence is sample efficient even for non-convex black-box functions [4].

A bayesian optimization algorithm requires two design choices: a prior and an acquisition function. The prior captures assumptions about smoothness and continuity of function f. While the acquisition function expresses an utility function over the model posterior for sequential decisions.

Gaussian Process Prior. The Gaussian Process (GP) is a computationally convenient prior distribution on functions that allows for closed form marginal and conditional computations [47]. The

GP is defined by the property that any finite set of N points $\{\mathbf{x}_n \in \mathcal{X}\}_{n=1}^{N}$ induces a multivariate Gaussian distribution on $\mathbb{R}^N$. We assume that the function $f(x)$ is drawn from a GP prior and that our observations are of the form $\{\mathbf{x}_n, y_n\}_{n=1}^{N}\}$, where $y_n \sim \mathcal{N}(f(\mathbf{x}_n), \nu)$ and ν is the variance of noise introduced into the funciton observations. The support and properties of the resulting distribution on functions are determined by a mean function $m : \mathcal{X} \to \mathbb{R}$ and a positive definite covariance function $K : \mathcal{X} \times \mathcal{X} \to \mathbb{R}$.

Design of Acquisition Function. The GP prior and sequential function evaluations induce a posterior over function f of interest (eg. throughput, memory or footprint). An acquisition function is the utility model which guides next best point for function evaluation. Under the Gaussian process prior, acquisition function depends on the model solely through its predictive mean function $\mu(\mathbf{x}; \mathbf{x}_n, y_n, \theta)$ and predictive variance function $\sigma^2(\mathbf{x}; \mathbf{x}_n, y_n, \theta)$. For this discussion, we denote the best current value as $\mathbf{x}_{\text{next}} = \text{argmin}_{\mathbf{x}_n} f(\mathbf{x}_n)$ and the cumulative distribution function of the standard normal as $\Phi(\cdot)$. The choice of acquisition function depends on the overall problem objective, as illustrated following. *Level-Set Optimization*. In addition to unconstrained optimization, to enable CONDENSA to achieve constraint satification we build on top of level-set black-box optimization [3, 14, 56]. We leverage a Gaussian Process Adaptive Sampling criterion called Implicit Level Set Upper Confidence Bound (ILS-UCB) [14], that prioritizes sampling near a level set of the estimate. This algorithm prioritizes searching the expected LC curve intersection with user accuracy constraints, conditional on estimated uncertainty, and does not seek to precisely learn the shape of the entire LC curve. Intuitively, by reducing the estimation space to specifically localize the sparsity that meets user accuracy constraints, we can reduce the total number of measurements-and consequently the time required to achieve an optimal value for the sparsity. Hence, rather than prioritizing both high variance and high mean like UCB, ILS-UCB prioritizes sampling in areas near a level set of the mean represented by the Gaussian Process Implicit Surface, i.e. to minimize the implicit potential defined by $\mu(\mathbf{x}) - L$, and where the confidence interval is large:$\mathbf{x}_t = \underset{\mathbf{x} \in X}{\text{argmax}} \; (1-\gamma)\sigma(\mathbf{x}) - \gamma * |\mu(\mathbf{x}) - L|$

3 Evaluation

We conduct extensive experiments and fully analyze Condensa on three tasks: (1) image classification on CIFAR-10 [33], (2) image classification on ILSVRC (ImageNet) [9], and (3) language modeling on WikiText-2 [40]. We optimize the networks in each task for two distinct objectives: (1) minimize their memory footprint, and (2) maximize their inference throughput.

Image Classification on ImageNet and CIFAR-10 We use the VGG-16 neural network [49] trained on the challenging ImageNet task [9], specifically the ILSVRC2012 version. We use PyTorch [45] and default pretrained models as a starting point. The CIFAR-10 dataset [33] consists of $50k$ training and $10k$ testing 32×32 images in 10 classes. We train the VGG-19 [49] and ResNet56 [24] neural networks on this dataset for 160 epochs with batch normalization, weight decay (10^{-4}), decreasing learning rate schedules (starting from 0.1) and augmented training data.

Language Modeling on WikiText-2 We trained a 2-layer LSTM model to perform a language modeling task on the WikiText-2 dataset [40]. We used a hidden state size of 650 and included a dropout layer between the two RNN layers with a dropout probability of 0.5. The LSTM received word embeddings of size 650. For training, we used truncated Backpropagation Through Time (truncated BPTT) with a sequence length of 50. The training batch size was set to 30, and models were optimized using SGD with a learning rate of 20. This setup is similar to Yu et al. [55].

Bayesian Optimizer Settings We use a Gaussian Processes prior with the Matern kernel ($\nu = 2.5$), length scale of 1.0 and α value of 0.1 with normalization of the predictions. For the GP regressor, the noise level in the covariance matrix is governed by another parameter, which we set to a very low value $10e^{-6}$. For the ILS-UCB acquisition function, we use a κ value of 0.95 for all our experiments with a bias towards sampling more in the area of level set, with the intention that the Bayesian optimizer results in a favorable sparsity level in as few samples as possible. We stop the Bayesian optimization loop according to the termination condition specified in Algorithm 1.

L-C Optimizer Settings The L-C optimizer was configured as follows: for all experiments, we use $\mu_j = \mu_o a^j$, with $\mu_0 = 10^{-3}$ and $a = 1.1$ where j is the L-C iteration. For CIFAR-10 and ImageNet, we use the SGD optimizer in the learning (L) step with a momentum value of 0.9, with the learning rate decayed from 0.1 to 10^{-5} over each mini-batch iteration. We use the Adam

Table 1: CONDENSA performance results on CIFAR-10, ImageNet, and WikiText-2. s^* is the sparsity ratio obtained by CONDENSA, r_c is the memory footprint reduction, and r_T/s_F is the throughput improvement/FLOP reduction.

METHOD	DATASET	NETWORK	ACCURACY/LOG PERPLEXITY	s^*	BO-SAMPLES	r_c	r_T/s_F
BASELINE	CIFAR-10	VGG19-BN	92.98%				
CONDENSA P+Q ($\epsilon = 2\%$)	CIFAR-10	VGG19-BN	93.04%	0.97	8,7	65.25×	N/A
CONDENSA FILTER ($\epsilon = 2\%$)	CIFAR-10	VGG19-BN	93.51%	0.72	9,8	N/A	$r_T = 2.22\times$
BASELINE	CIFAR-10	RESNET56	92.75%				
AMC [26]	CIFAR-10	RESNET56	90.1%	N/A	N/A	N/A	$s_F = 2\times$
CONDENSA P+Q ($\epsilon = 2\%$)	CIFAR-10	RESNET56	91.2%	0.94	7,7	27×	N/A
CONDENSA FILTER ($\epsilon = 2\%$)	CIFAR-10	RESNET56	91.29%	0.72	7,7	N/A	$r_T = 1.07\times$
BASELINE	IMAGENET	VGG16-BN	91.5%				
FILTER PRUNING [27]	IMAGENET	VGG16-BN	89.80%	N/A	N/A	≈ 4×	N/A
AUTOSLIM [37]	IMAGENET	VGG16-BN	90.90%	N/A	N/A	6.4×	N/A
AMC [26]	IMAGENET	VGG16-BN	90.10%	N/A	N/A	N/A	$s_F = 1.25\times$
CONDENSA P+Q ($\epsilon = 2\%$)	IMAGENET	VGG16-BN	89.89%	0.92	8,7	25.59×	N/A
CONDENSA FILTER ($\epsilon = 2\%$)	IMAGENET	VGG16-BN	90.25%	0.12	9,7	N/A	$r_T = 1.16\times$
BASELINE	WIKITEXT-2	LSTM	4.70				
[55]	WIKITEXT-2	LSTM	6.5	N/A	N/A	≈ 10×	N/A
CONDENSA P+Q ($\epsilon = 2\%$)	WIKITEXT-2	LSTM	4.75	0.92	9,7	4.2×	N/A
CONDENSA BLOCK ($\epsilon = 2\%$)	WIKITEXT-2	LSTM	4.77	0.61	8,7	N/A	$s_F = 2.2\times$

optimizer in the L-step of WikiText-2 with a fixed learning rate of 10^{-4}. We ran between 4000-5000 mini-batch iterations in each L-step, with a higher number of iterations in the first L-step ($30k$ for CIFAR-10 and ImageNet, and $7k$ for WikiText-2) as recommended by [6]. We ran 5, 30, and 50 L-C iterations for WikiText-2, ImageNet, and CIFAR-10, respectively; compared to CIFAR-10, we ran relatively fewer iterations for ImageNet due to its significantly higher computational cost, and ran an extra 5 fine-tuning iterations instead. We use the same mini-batch sizes as during training for all experiments, and use validation datasets to select the best model during compression (we perform a 9:1 training:validation split for CIFAR-10 since it doesn't include a validation dataset).

Objective 1: Minimize Memory Footprint The memory footprint of a model is defined as the number of bytes consumed by the model's *non-zero* parameters. Reducing the footprint below a threshold value is desirable, especially for memory-constrained devices such as mobile phones, and can be accomplished through either pruning or quantization, or both.

Objective 2: Maximize Throughput Inference throughput is defined as the number of input samples processed by a model per second, and is commonly used for measuring real-world performance. For CIFAR-10 and ImageNet, we measure hardware inference throughput of the compressed model in the objective function. We use an NVIDIA Titan V GPU with the TensorRT 5 framework to obtain throughput data.

We present the memory footprint reductions and inference throughput improvements obtained by Condensa for each of the three tasks we evaluate in Table 1. For each task, we list the sparsity ratio obtained by the Condensa Bayesian optimizer, its corresponding accuracy, memory footprint reductions using pruning and quantization (column labeled r_c), and inference throughput/FLOP improvements using filter/block pruning (column labeled r_T/s_F). We also show the number of samples required by the Bayesian optimizer for each phase of the sparsity ratio inference algorithm (shown in Algorithm 1) to arrive at the final solution. We also compare our approach with recent work on automated model compression. For CIFAR-10 and ImageNet, we compare our results with AMC [26] and AutoSlim [37], and for WikiText-2, we compare with [55]. We notice that Condensa significantly outperforms current state-of-the-art approaches in terms of accuracy, throughput, and model footprint reduction.

4 Conclusions

This paper has presented Condensa, which is a programming system for model compression. Condensa enables users to programmatically compose elementary schemes to build much more complex and practically interesting schemes, and includes a novel sample-efficient constrained Bayesian optimization-based algorithm for automatically inferring desirable sparsity ratios based on a user-provided objective function (also expressed in Python). On three real-world image classification and language modeling tasks, Condensa achieves memory footprint reductions of up to $65\times$ and runtime throughput improvements of up to $2.17\times$ using at most 10 samples per search.

References

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: a system for large-scale machine learning. In *OSDI*, volume 16, pages 265–283, 2016.

[2] Sajid Anwar and Wonyong Sung. Compact deep convolutional neural networks with coarse pruning. *arXiv preprint arXiv:1610.09639*, 2016.

[3] Ilija Bogunovic, Jonathan Scarlett, Andreas Krause, and Volkan Cevher. Truncated variance reduction: A unified approach to bayesian optimization and level-set estimation. In *Advances in neural information processing systems*, pages 1507–1515, 2016.

[4] Eric Brochu, Vlad M Cora, and Nando De Freitas. A tutorial on bayesian optimization of expensive cost functions, with application to active user modeling and hierarchical reinforcement learning. *arXiv preprint arXiv:1012.2599*, 2010.

[5] Miguel A Carreira-Perpinán. Model compression as constrained optimization, with application to neural nets. part I: General framework. *arXiv preprint arXiv:1707.01209*, 2017.

[6] Miguel A Carreira-Perpinán and Yerlan Idelbayev. "learning-compression" algorithms for neural net pruning. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 8532–8541, 2018.

[7] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. *arXiv preprint arXiv:1512.01274*, 2015.

[8] Xiaoliang Dai, Peizhao Zhang, Bichen Wu, Hongxu Yin, Fei Sun, Yanghan Wang, Marat Dukhan, Yunqing Hu, Yiming Wu, Yangqing Jia, et al. Chamnet: Towards efficient network design through platform-aware model adaptation. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 11398–11407, 2019.

[9] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*, pages 248–255. Ieee, 2009.

[10] Emily L Denton, Wojciech Zaremba, Joan Bruna, Yann LeCun, and Rob Fergus. Exploiting linear structure within convolutional networks for efficient evaluation. In *Advances in neural information processing systems*, pages 1269–1277, 2014.

[11] Xuanyi Dong, Junshi Huang, Yi Yang, and Shuicheng Yan. More is less: A more complicated network with less inference complexity. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 5840–5848, 2017.

[12] fmfn. A Python implementation of global optimization with Gaussian processes. `https://github.com/fmfn/BayesianOptimization`, 2019. [Online; accessed 1-September-2019].

[13] Jonathan Frankle and Michael Carbin. The lottery ticket hypothesis: Training pruned neural networks. *arXiv preprint arXiv:1803.03635*, 2018.

[14] Animesh Garg, Siddarth Sen, Rishi Kapadia, Yiming Jen, Stephen McKinley, Lauren Miller, and Ken Goldberg. Tumor localization using automated palpation with gaussian process adaptive sampling. In *2016 IEEE International Conference on Automation Science and Engineering (CASE)*, pages 194–200. IEEE, 2016.

[15] Ross Girshick. Fast r-cnn. In *Proceedings of the IEEE international conference on computer vision*, pages 1440–1448, 2015.

[16] Yunchao Gong, Liu Liu, Ming Yang, and Lubomir Bourdev. Compressing deep convolutional networks using vector quantization. *arXiv preprint arXiv:1412.6115*, 2014.

[17] Google. TensorFlow model optimization toolkit. `https://github.com/tensorflow/model-optimization`, 2019. [Online; accessed 1-September-2019].

[18] Scott Gray, Alec Radford, and Diederik P Kingma. GPU kernels for block-sparse weights. *arXiv preprint arXiv:1711.09224*, 2017.

[19] Suyog Gupta, Ankur Agrawal, Kailash Gopalakrishnan, and Pritish Narayanan. Deep learning with limited numerical precision. In *International Conference on Machine Learning*, pages 1737–1746, 2015.

[20] Song Han, Junlong Kang, Huizi Mao, Yiming Hu, Xin Li, Yubin Li, Dongliang Xie, Hong Luo, Song Yao, Yu Wang, et al. Ese: Efficient speech recognition engine with sparse LSTM on FPGA. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 75–84. ACM, 2017.

[21] Song Han, Xingyu Liu, Huizi Mao, Jing Pu, Ardavan Pedram, Mark A Horowitz, and William J Dally. Eie: efficient inference engine on compressed deep neural network. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, pages 243–254. IEEE, 2016.

[22] Song Han, Huizi Mao, and William J Dally. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv preprint arXiv:1510.00149*, 2015.

[23] Song Han, Jeff Pool, John Tran, and William Dally. Learning both weights and connections for efficient neural network. In *Advances in neural information processing systems*, pages 1135–1143, 2015.

[24] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[25] Yang He, Xuanyi Dong, Guoliang Kang, Yanwei Fu, and Yi Yang. Progressive deep neural networks acceleration via soft filter pruning. *arXiv preprint arXiv:1808.07471*, 2018.

[26] Yihui He, Ji Lin, Zhijian Liu, Hanrui Wang, Li-Jia Li, and Song Han. Amc: Automl for model compression and acceleration on mobile devices. In *Proceedings of the European Conference on Computer Vision (ECCV)*, pages 784–800, 2018.

[27] Yihui He, Xiangyu Zhang, and Jian Sun. Channel pruning for accelerating very deep neural networks. In *International Conference on Computer Vision (ICCV)*, volume 2, 2017.

[28] Hengyuan Hu, Rui Peng, Yu-Wing Tai, and Chi-Keung Tang. Network trimming: A data-driven neuron pruning approach towards efficient deep architectures. *arXiv preprint arXiv:1607.03250*, 2016.

[29] Gao Huang, Zhuang Liu, Laurens Van Der Maaten, and Kilian Q Weinberger. Densely connected convolutional networks. In *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 2261–2269. IEEE, 2017.

[30] Max Jaderberg, Andrea Vedaldi, and Andrew Zisserman. Speeding up convolutional neural networks with low rank expansions. *arXiv preprint arXiv:1405.3866*, 2014.

[31] Donald R Jones. A taxonomy of global optimization methods based on response surfaces. *Journal of global optimization*, 21(4):345–383, 2001.

[32] Donald R Jones, Cary D Perttunen, and Bruce E Stuckman. Lipschitzian optimization without the lipschitz constant. *Journal of optimization Theory and Applications*, 79(1):157–181, 1993.

[33] Alex Krizhevsky, Vinod Nair, and Geoffrey Hinton. The cifar-10 dataset. *online: http://www. cs. toronto. edu/kriz/cifar. html*, 55, 2014.

[34] Harold J Kushner. A new method of locating the maximum point of an arbitrary multipeak curve in the presence of noise. *Journal of Basic Engineering*, 86(1):97–106, 1964.

[35] Vadim Lebedev, Yaroslav Ganin, Maksim Rakhuba, Ivan Oseledets, and Victor Lempitsky. Speeding-up convolutional neural networks using fine-tuned cp-decomposition. *arXiv preprint arXiv:1412.6553*, 2014.

[36] Hao Li, Asim Kadav, Igor Durdanovic, Hanan Samet, and Hans Peter Graf. Pruning filters for efficient convnets. *arXiv preprint arXiv:1608.08710*, 2016.

[37] Ning Liu, Xiaolong Ma, Zhiyuan Xu, Yanzhi Wang, Jian Tang, and Jieping Ye. Autoslim: An automatic dnn structured pruning framework for ultra-high compression rates. *arXiv preprint arXiv:1907.03141*, 2019.

[38] Daniel James Lizotte. *Practical bayesian optimization*. University of Alberta, 2008.

[39] Jian-Hao Luo, Jianxin Wu, and Weiyao Lin. Thinet: A filter level pruning method for deep neural network compression. *arXiv preprint arXiv:1707.06342*, 2017.

[40] Stephen Merity, Caiming Xiong, James Bradbury, and Richard Socher. Pointer sentinel mixture models. *arXiv preprint arXiv:1609.07843*, 2016.

[41] Jonas Mockus. Application of bayesian approach to numerical methods of global and stochastic optimization. *Journal of Global Optimization*, 4(4):347–365, 1994.

[42] Jonas Mockus, Vytautas Tiesis, and Antanas Zilinskas. The application of bayesian methods for seeking the extremum. *Towards global optimization*, 2(117-129):2, 1978.

[43] Pavlo Molchanov, Stephen Tyree, Tero Karras, Timo Aila, and Jan Kautz. Pruning convolutional neural networks for resource efficient inference. *arXiv preprint arXiv:1611.06440*, 2016.

[44] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I Jordan, et al. Ray: A distributed framework for emerging {AI} applications. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*, pages 561–577, 2018.

[45] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch. In *NIPS-W*, 2017.

[46] Adam Polyak and Lior Wolf. Channel-level acceleration of deep face representations. *IEEE Access*, 3:2163–2175, 2015.

[47] Carl Edward Rasmussen and C Williams. Gaussian processes for machine learning the mit press, 2006.

[48] Mohammad Rastegari, Vicente Ordonez, Joseph Redmon, and Ali Farhadi. Xnor-net: Imagenet classification using binary convolutional neural networks. In *European Conference on Computer Vision*, pages 525–542. Springer, 2016.

[49] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[50] Jasper Snoek, Hugo Larochelle, and Ryan P Adams. Practical bayesian optimization of machine learning algorithms. In *Advances in neural information processing systems*, pages 2951–2959, 2012.

[51] Niranjan Srinivas, Andreas Krause, Sham M Kakade, and Matthias Seeger. Gaussian process optimization in the bandit setting: No regret and experimental design. *arXiv preprint arXiv:0912.3995*, 2009.

[52] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in neural information processing systems*, pages 5998–6008, 2017.

[53] Stephen Wright and Jorge Nocedal. Numerical optimization. *Springer Science*, 35(67-68):7, 1999.

[54] Jian Xue, Jinyu Li, and Yifan Gong. Restructuring of deep neural network acoustic models with singular value decomposition. In *Interspeech*, pages 2365–2369, 2013.

[55] Haonan Yu, Sergey Edunov, Yuandong Tian, and Ari S Morcos. Playing the lottery with rewards and multiple languages: lottery tickets in rl and nlp. *arXiv preprint arXiv:1906.02768*, 2019.

[56] Andrea Zanette, Junzi Zhang, and Mykel J Kochenderfer. Robust super-level set estimation using gaussian processes. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, pages 276–291. Springer, 2018.

[57] Tianyun Zhang, Kaiqi Zhang, Shaokai Ye, Jiayu Li, Jian Tang, Wujie Wen, Xue Lin, Makan Fardad, and Yanzhi Wang. Adam-ADMM: A unified, systematic framework of structured weight pruning for DNNs. *arXiv preprint arXiv:1807.11091*, 2018.

[58] Chenzhuo Zhu, Song Han, Huizi Mao, and William J Dally. Trained ternary quantization. *arXiv preprint arXiv:1612.01064*, 2016.

[59] Zhuangwei Zhuang, Mingkui Tan, Bohan Zhuang, Jing Liu, Yong Guo, Qingyao Wu, Junzhou Huang, and Jinhui Zhu. Discrimination-aware channel pruning for deep neural networks. In *Advances in Neural Information Processing Systems*, pages 883–894, 2018.

[60] Neta Zmora, Guy Jacob, and Gal Novik. Neural network distiller, June 2018.

5 Supplementary Material

This paper makes the following contributions: (1) it introduces Condensa, a novel programming system for model compression and demonstrates its ease-of-use for expressing complex compression schemes, (2) it presents the first sample-efficient constrained Bayesian optimization-based method for automatically inferring optimal sparsity ratios based on a user-provided objective function, and (3) it demonstrates the effectiveness of Condensa on three image classification and language modeling tasks, resulting in memory footprint reductions of up to $65\times$ and runtime throughput improvements of up to $2.17\times$ using at most 10 samples per search.

6 Background

An effective accuracy recovery mechanism for Condensa must ideally have three important attributes: (1) able to handle all the compression operators supported by Condensa, (2) be efficient with relatively low overheads, and (3) provide optimality guarantees whenever possible. In this paper, we use the recently proposed L-C algorithm [6], since it satisfies all three of the above requirements. In L-C, model compression is formulated as a constrained optimization problem:

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

The optimization is non-convex due to two reasons: (1) the original problem of training the reference model is already non-convex for models such as DNNs, making the objective function non-convex, and (2) the decompression mapping $\mathcal{D}(\Theta)$ typically adds another layer of non-convexity caused by an underlying combinatorial problem.

For a given task such as image classification, assume we have trained a large *reference* model $\overline{\mathbf{w}} = \text{argmin}_{\mathbf{w}} L(\mathbf{w})$, where $L()$ denotes a *loss function* (e.g., cross-entropy on a given training set), and $\mathbf{w} \in \mathbb{R}^P$. *Model compression* refers to finding a smaller model Θ that can be applied to the same task and ideally achieves the same accuracy as $\mathbf{w}$. Model compression can be performed in various ways, and Condensa currently supports two commonly used techniques: pruning and quantization. In pruning, non-zero values from $\mathbf{w}$ are eliminated or "pruned" to obtain Θ. Pruning is usually performed using some kind of thresholding (for eg., magnitude-based) and can be unstructured (prune any non-zero value) or structured (prune only *blocks* of non-zeros). On the other hand, quantization retains the number of parameters in Θ but assigns parameters in $\mathbf{w}$ one of K codebook values, where the codebook may be fixed or adaptive. Condensa supports low-precision approximation, which refers to assigning each parameter in $\mathbf{w}$ a corresponding lower-precision representation (for example, converting from 32-bit to 16-bit floating-point) and is equivalent to quantization using a fixed codebook.

General Compression Algorithms and Tools General accuracy recovery algorithms capable of handling a wide variety of compression techniques provide the foundation for systems like Condensa. Apart from the L-C algorithm [5] which Condensa uses, other recent accuracy recovery algorithms have been proposed. ADAM-ADMM [57] proposes a unified framework for structured weight pruning based on ADMM that performs dynamic regularization in which the regularization target is updated in each iteration. DCP [59] introduces additional losses into the network to increase the discriminative power of intermediate layers and select the most discriminative channels for each layer by considering the additional loss and the reconstruction error. Condensa can readily support such algorithms as additional optimizers as described in Section 2. Neural network distiller [60] and TensorFlow model optimization toolkit [17] are two recent open-source model compression frameworks that support multiple compression schemes. While these projects share a number of common goals with Condensa, they differ in two important ways: first, they do not support the expression of schemes as imperative programs containing control-flow, iteration, recursion, etc. (Distiller requires a declarative compression specification in YAML, while the TensorFlow model optimization toolkit operates by modifying the DNN computation graph directly); second, these frameworks do not support automatic compression hyperparameter optimization for black-box objective functions.

B.O. and Automated Model Compression Bayesian optimization has previously been demonstrated to work well for general hyperparameter optimization in machine learning and neural architecture search [50, 8]. To the best of our knowledge, we are the first to use sample-efficient

search via Bayesian optimization for obtaining compression hyperparameters. Automation in model compression is currently achieved either through reinforcement learning (RL) algorithms [26] or simulated annealing [37]. In particular, the automation procedure for AMC [26] uses four arbitrary stages of pruning and re-training for RL training; additionally, the reward function is difficult to design, and even given a good reward, local optima can be hard to escape. It is also difficult to determine when such methods may just be overfitting to irrelevant patterns in the environment. Even disregarding generalization issues, AMC's agent (DDPG) uses trial and error, which is charecterized to have an underlying incompatibility with the target pruning problem [37]. AutoSlim [37] proposes an automated approach based on simulated annealing, and use the ADMM algorithm for accuracy recovery, which is an AL-based method very similar to the L-C algorithm; AutoSlim, however, only supports weight pruning and does not support general compression schemes as Condensa does.

DNN Compression Techniques There is considerable prior work on accelerating neural networks using structured weight pruning [13, 23, 22, 39, 20, 11, 21, 46, 28, 2, 43], quantization [58, 16, 48] and low-rank tensor factorization [35, 54, 10, 15]. Most of these individual compression schemes for pruning and quantization and their combinations can be expressed in Condensa. Two common problems with these methods are: (1) determining optimal sparsity ratios at a global (network) level, and (2) distributing global sparsity into a particular sparsity ratio for each layer. We tackle these problems efficiently and systematically using our Bayesian and L-C optimizers, respectively.

6.1 Sparsity Profile Analysis

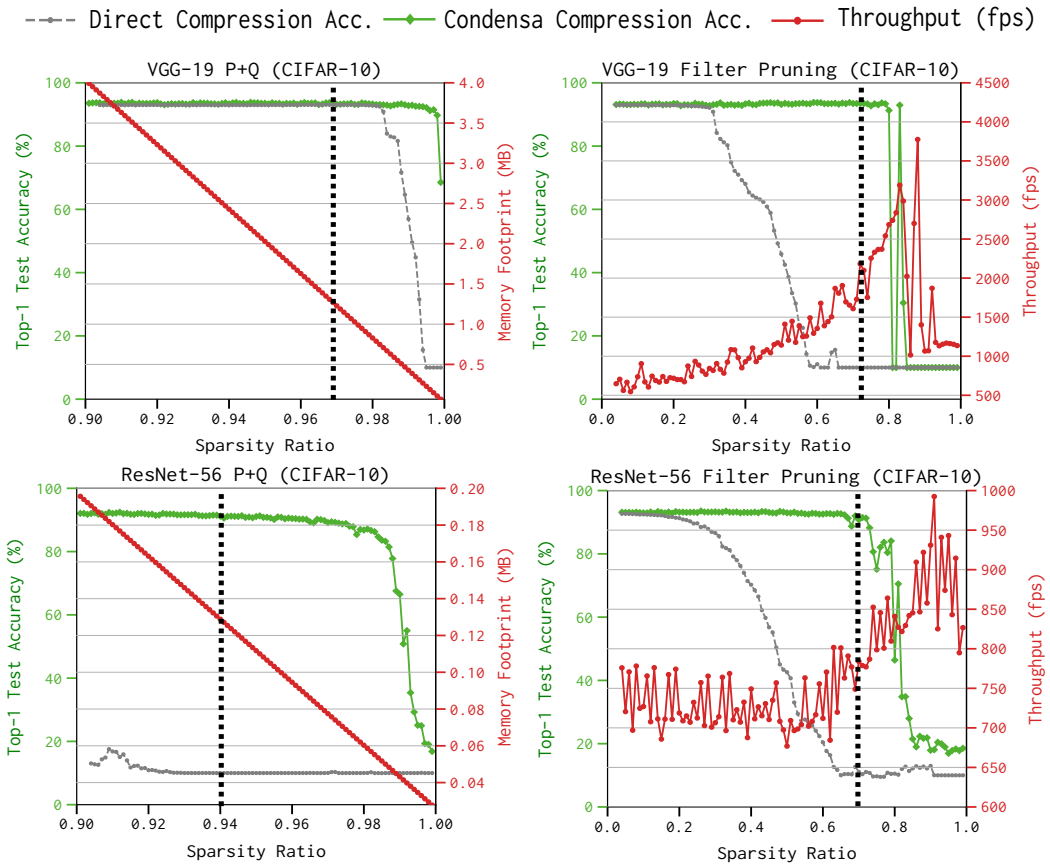


Figure 3: Examples of CONDENSA operating on VGG19BN and ResNet56 for CIFAR-10. Column 1 shows the problem of the form "minimize *Memory* with a lower bound on accuracy", while Column 2 illustrates "maximize *Throughput* with a lower bound on accuracy". The DC line (grey) shows accuracy values if no fine tuning with LC is performed.

Figures 3 and 4 illustrate how a compressed model's accuracy, inference performance, and memory footprint vary w.r.t. sparsity ratios for the CIFAR-10 and WikiText-2 tasks. All three of these functions

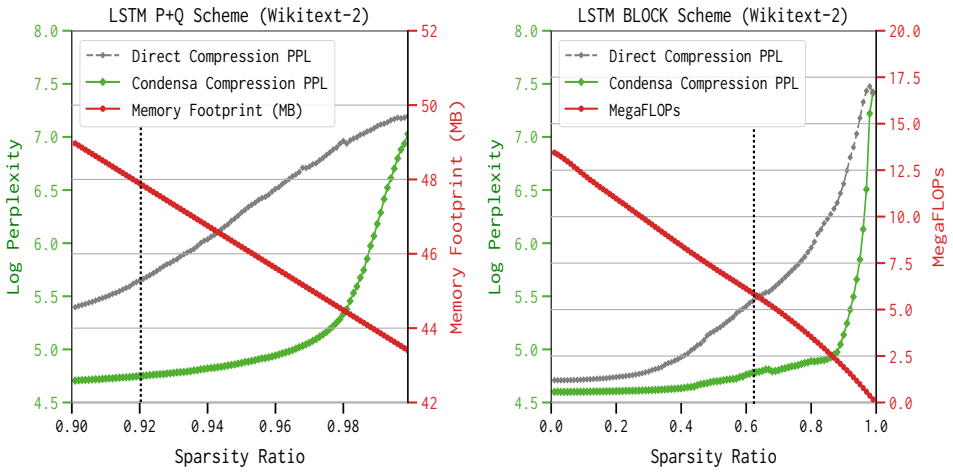


Figure 4: 2-layer LSTM WikiText-2 task results for pruning + quantization (left) and block pruning with block size of 5.

are assumed to be unknown in our problem formulation, but we compute them explicitly here to better understand the quality of solutions produced by Condensa. For each figure, compression accuracies (shown in green) are obtained by running the L-C algorithm to convergence for 100 sparsity ratios ranging from 0.9 to 1.0 (for pruning + quantization), and from 0 to 1 for the filter and block pruning schemes; collecting each such point requires between 30 minutes to 8 hours of time on a single NVIDIA Tesla V100 GPU.

We notice three important trends in Figures 3 and 4: (1) Condensa consistently finds solutions near the 'knee' of the L-C accuracy curves, signifying the effectiveness of the ILS-UCB acquisition function; (2) local minima/maxima is avoided while optimizing the objective function, demonstrating that the UCB acquisition function for objective function optimization is working as expected, and (3) the knee of the D-C accuracy curves occur at significantly lower sparsity ratios; the L-C optimizer, on the other hand is able to recover accuracies up to much higher sparsity ratios.

Algorithm Summary: We describe CONDENSA's two-stage optimization pipeline in Algorithm 1. Here, we first find a sparsity value s_{acc} that constrains the accuracy function A to the provided ϵ. We then constrain the search space to $(0, s_{acc})$ while optimizing the user-provided objective function f. The `BAYESOPT` function runs a Bayesian optimization loop given a target objective function $\mathcal{B}_f$ and an acquisition function. Note that we assume that A decreases monotonically w.r.t. sparsity in the region $(0, s_{acc})$.

Scheme	Description
`Quantize(dtype)`	Quantizes network weights to given datatype `dtype`.
`Prune()`	Performs unstructured pruning of network weights.
`NeuronPrune(criteria)`	Aggregates and prunes neurons (1D blocks) according to `criteria`.
`FilterPrune(criteria)`	Aggregates and prunes filters (3D blocks) according to `criteria`.
`StructurePrune(criteria)`	Combines neuron and filter pruning.
`BlockPrune(criteria, bs)`	Aggregates and prunes n-D blocks of size `bs` according to `criteria`.
`Compose(slist)`	Composes together all schemes in `slist`.

Table 2: List of pre-built compression schemes in Condensa.

Listing 1 provides a concrete example of invoking Condensa to compress a model. Here, we first train the reference models (lines 2-3) and instantiate the pre-built `Prune` scheme for unstructured pruning (line 6; see Table 2 for a full list of pre-built schemes). We also define our objective function to be throughput (line 8) and specify that it must be maximized (line 10); note that while users may define their own objective functions, Condensa also comes bundled with some common objective functions such as model memory footprint and throughput. Next, we instantiate the L-C optimizer (line 12) and the model compressor (lines 14-24). The model compressor (`Compressor` class in

```
# Construct pre-trained model
criterion = torch.nn.CrossEntropyLoss()
train(model, num_epochs, trainloader, criterion)

# Instantiate compression scheme
prune = condensa.schemes.Prune()
# Define objective function
tput = condensa.objectives.throughput
# Specify optimization operator
obj = condensa.searchops.Maximize(tput)
# Instantiate L-C optimizer
lc = condensa.optimizers.LC(steps=30, lr=0.01)
# Build model compressor instance
compressor = condensa.Compressor(
    model=model, # Trained model
    objective=obj, # Objective
    eps=0.02, # Accuracy threshold
    optimizer=lc, # Accuracy recovery
    scheme=prune, # Compression scheme
    trainloader=trainloader, # Train dataloader
    testloader=testloader, # Test dataloader
    valloader=valloader, # Val dataloader
    criterion=criterion # Loss criterion
)
# Obtain compressed model
wc = compressor.run()
```

Listing 1: Example usage of the Condensa library.

Listing) automatically samples and evaluates global sparsity ratios as described in Section 2.3 and returns the final compressed model.

The objective function f for memory footprint case is defined as follows:

```
from torch.nn.utils import parameters_to_vector
def footprint(w):
    return parameters_to_vector(w.parameters())
           .view(-1).nonzero().numel() * 2.0

from schemes import Compose, Prune, Quantize
scheme = Compose([Prune(), Quantize(float16)])
```

6.2 Bayesian Optimizer ILS-UCB Trace

In the following plots we show an example run of the Bayesian optimizer trace running on the black-box function set as Condensa Compression accuracy with an acquicision function setup as `ILS-UCB`. The `BayesOpt` is attempting to find the level set on the Top-1 test accuracy of the compressed model. The X-axis on the top plots is Sparsity and Y-Axis is Top-1 test accuracy (top-plot) and Utility function on the bottom plot. The red dots on the top plots are samples of the expensive function and the star on the bottom plots are the maxima of the utility function. The blue dark line represents the Target function, recall that this function can be programmed by the user as throughput, FLOPs, memory footprint or Top-1 Accuracy. The dotted line is the GP's belief of the shape of the function, notice how it evolves over each sample. These figures show a Gaussian process (GP) approximation of the objective function over four iterations of sampled values of the objective function. The figure also shows the acquisition function in the lower plots. The acquisition is high where the GP predicts a high objective (exploitation) and where the prediction uncertainty is high (exploration)—areas with both attributes are sampled first. Note that the area on the far left remains unsampled, as while it has high uncertainty, it is (correctly) predicted to offer little improvement over the highest observation.

The expectation of the improvement function with respect to the predictive distribution of the Gaussian process enables us to balance the trade-off of exploiting and exploring. When exploring, we should choose points where the surrogate variance is large. When exploiting, we should choose points where the surrogate mean is high. To sample efficiently, Bayesian optimization uses these acquisition

Figure 5: The Bayes Opt is Initialized with 2-points to begin with, denoted by the two red-dots.

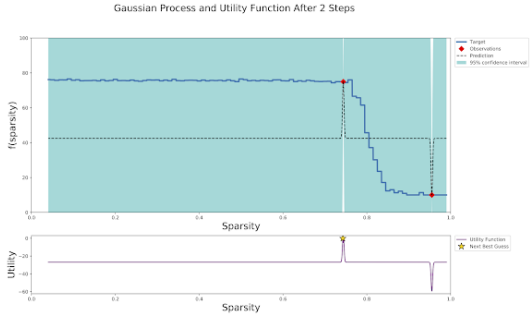


Figure 6: Illustrating the state of the GP-regressor, notice the change in Variance of the GP

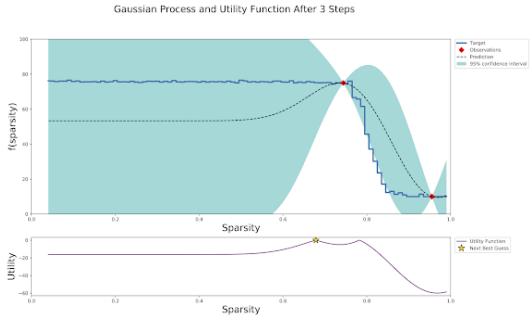


Figure 7: This is the state of the BayesOpt after three samples, its is important to note that the BayesOpt decided not to take any more samples on the right hand side of the curve.

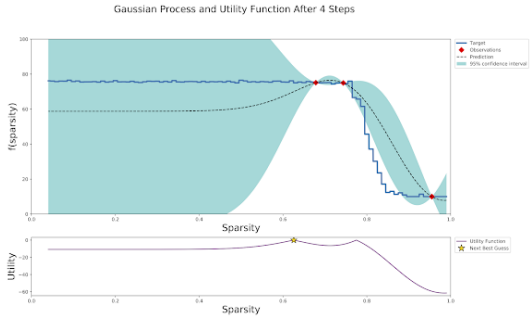
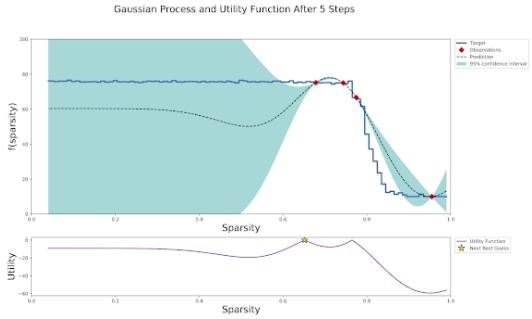


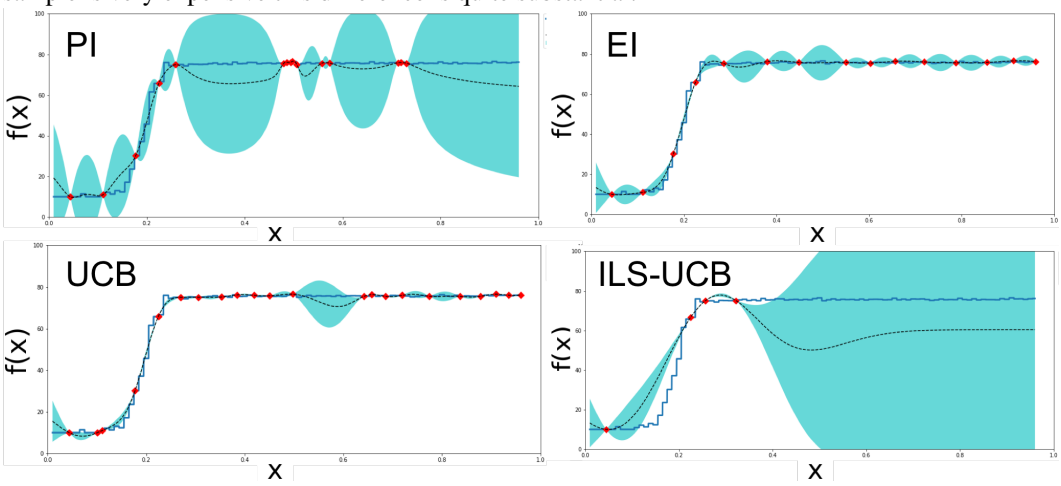
Figure 8: This plot illustrates the sample efficiency of BayesOpt, as you notice the fourth sample is drawn on the knee of the curve, attemping to minimize the difference between the Implicit Level Set and the function representation.



function to determine the next location $x_{t+1} \in A$ to sample. The decision represents an automatic

trade-off between exploration (where the objective function is very uncertain) and exploitation (trying values of x where the objective function is expected to be high). This optimization technique has the nice property that it aims to minimize the number of objective function evaluations. Moreover, it is likely to do well even in settings where the objective function has multiple local maxima. Our acquisition is currently both myopic and permits only a single sample per iteration. Looking forward to some horizon would be extremely valuable, as well as in trying to optimize within a known budget of future observations.

Figure 9: In this figure below, we have density in the x-axis and perform ablation studies by running our Bayesian Optimizer with different acquisition functions that are available in CONDENSA framework: PI, EI, GP-UCB as the acquisition model after 15 steps and ILS-UCB after 5 samples. GP-ILS-UCB gets a good estimate around the requested level set quickly with *3x* fewer samples, while UCB adn EI also perform reasonably but with many more samples. In this domain, where each sample is very expensive this difference is quite substantial.



Maximizing the acquisition function CONDENSA uses a function to find the maximum of the acquisition function and use a combination of random sampling and the L-BFGS-B optimization method. First by sampling a few warmup $(1e5)$ points at random, and then running L-BFGS-B from (250) random starting points. To find the point at which to sample, we still need to maximize the constrained objective $u(\mathbf{x})$. *Unlike the original objective function,* $u(\cdot)$ *can be cheaply sampled*. Existing works optimize the acquisition function using DIRECT [32], a deterministic, derivative-free optimizer. It uses the existing samples of the objective function to decide how to proceed to divide the feasible space into finer rectangles. Other methods such as Monte Carlo and multi-start have also been used, and seem to perform reasonably well [41, 38]. Note that the second term in the equation is negative, as we are trying to sample in locations where the distance to the level set is minimized. To find the point at which to sample, we still need to maximize the constrained objective $u(\mathbf{x})$. Unlike the original objective function f, $u(\cdot)$ can be cheaply sampled. In CONDENSA we use GP-UCB (GP-LCB) for function maximization (minimization) and ILS-UCB for solving constraints, as shown in Algorithm 1.

Bayesian Optimizer Settings *1. Probability of Improvement* This intuitive stategy maximizes the probability of improving over the best current value [34]. Under the GP this can be computed analytically as: $a_{\text{PI}}(\mathbf{x}; \{\mathbf{x}_n, y_n\}, \theta) = \Phi(\gamma(\mathbf{x}))$, where $\gamma(\mathbf{x}) = \frac{f(\mathbf{x}_{\text{best}}) - \mu(\mathbf{x}; \{\mathbf{x}_n, y_n\}, \theta)}{\sigma(\mathbf{x}; \{\mathbf{x}_n, y_n\}, \theta)}$.

2. Expected Improvement. Alternatively, one could choose to maximize the expected improvement (EI) over the current best. This also has closed form under the Gaussian process: $a_{\text{EI}}(\mathbf{x}; \{\mathbf{x}_n, y_n\}, \theta) = \sigma(x; \mathbf{x}_n, y_n, \theta) - \kappa\sigma(\mathbf{x}; \{\mathbf{x}_n, y_n\}, \theta)$, with a tunable κ to balance exploitation against exploration.

3. Upper/Lower Confidence Bound. Herein, the functional approximation uncertainty is leveraged for acquisition through lower (upper) confidence bounds for functional min (max) [51]. These

acquisition functions have the form $a_{\text{UCB}}(\mathbf{x}; \{\mathbf{x}_n, y_n\}; \theta) = \mu(\mathbf{x}; \{\mathbf{x}_n, y_n\}, \theta) - \kappa\sigma(\mathbf{x}; \{\mathbf{x}_n, y_n\}, \theta)$, with a tunable κ to balance exploitation against exploration.

4. Level-Set Optimization. In addition to unconstrained optimization, to enable CONDENSA to achieve constraint satification we build on top of level-set black-box optimization [3, 14, 56]. We leverage a Gaussian Process Adaptive Sampling criterion called Implicit Level Set Upper Confidence Bound (ILS-UCB) [14], that prioritizes sampling near a level set of the estimate.

This algorithm prioritizes searching the expected LC curve intersection with user accuracy constraints, conditional on estimated uncertainty, and does not seek to precisely learn the shape of the entire LC curve. Intuitively, by reducing the estimation space to specifically localize the sparsity that meets user accuracy constraints, we can reduce the total number of measurements-and consequently the time required to achieve an optimal value for the sparsity. Hence, rather than prioritizing both high variance and high mean like UCB, ILS-UCB prioritizes sampling in areas near a level set of the mean represented by the Gaussian Process Implicit Surface, i.e. to minimize the implicit potential defined by $\mu(\mathbf{x}) - L$, and where the confidence interval is large:$\mathbf{x}_t = \underset{\mathbf{x} \in X}{\text{argmax}}\ (1-\gamma)\sigma(\mathbf{x}) - \gamma * |\mu(\mathbf{x}) - L|$

6.3 Layer-Wise Runtime Performance

In this section, we analyze how improving throughput using compression translates to execution time improvements for each layer on actual hardware. For this experiment, we focus on VGG-19 on CIFAR-10, since it has a relatively simple structure and is easy to analyze on a layer-by-layer basis. We use filter pruning with a sparsity ratio of 0.7 for this experiment. We report the mean runtimes over 100 executions as obtained using TensorRT.

Table 3 shows layer-by-layer compression ratios and mean runtimes collected over 100 runs for filter pruning. Here, the columns labeled *R and C* represent results for the reference, and filter-pruned models, respectively. We only show data for convolutional layers as they dominate computation time for this network. We observe large inference runtime speedups in later layers of the network and observe a geometric mean speedup of $3.21x$ over the original model. This result helps us gain more insight into how the L-C algorithm distributes global sparsity ratios to each layer, resulting in actual hardware speedups.

LAYER	SHAPE		TIME(MS)		SPEEDUP
	R	C	R	C	
CONV1	3 x 3 x 3 x 64	3 x 3 x 3 x 23	0.07	0.05	1.4X
CONV2	3 x 3 x 64 x 64	3 x 3 x 23 x 58	0.23	0.11	2.09X
CONV3	3 x 3 x 64 x 128	3 x 3 x 58 x 126	0.12	0.13	0.92X
CONV4	3 x 3 x 128 x 128	3 x 3 x 126 x 127	0.22	0.24	0.92X
CONV5	3 x 3 x 128 x 256	3 x 3 x 127 x 256	0.22	0.22	1
CONV6	3 x 3 x 256 x 256	3 x 3 x 256 x 255	0.41	0.41	1
CONV7	3 x 3 x 256 x 256	3 x 3 x 255 x 251	0.41	0.41	1
CONV8	3 x 3 x 256 x 256	3 x 3 x 251 x 241	0.41	0.41	1
CONV9	3 x 3 x 256 x 512	3 x 3 x 241 x 214	0.28	0.22	1.27X
CONV10	3 x 3 x 512 x 512	3 x 3 x 214 x 71	0.54	0.16	3.38X
CONV11	3 x 3 x 512 x 512	3 x 3 x 71 x 30	0.53	0.03	17.67X
CONV12	3 x 3 x 512 x 512	3 x 3 x 30 x 38	0.53	0.03	17.67X
CONV13	3 x 3 x 512 x 512	3 x 3 x 38 x 48	0.56	0.04	14X
CONV14	3 x 3 x 512 x 512	3 x 3 x 48 x 38	0.56	0.04	14X
CONV15	3 x 3 x 512 x 512	3 x 3 x 38 x 48	0.56	0.04	14X
CONV16	3 x 3 x 512 x 512	3 x 3 x 28 x 102	0.56	0.04	14X

Table 3: Layer-wise TensorRT run-times and speedups for filter pruning of VGG-19. R and C denote reference and compressed models, respectively.

6.4 Implementation Notes

The Condensa library and L-C optimizer are implemented in Python and are designed to inter-operate seamlessly with the PyTorch framework [45]. While we chose PyTorch for its widespread use in the machine learning community, it is worth noting that Condensa's design is general and that its

```
import torch

import condensa
import condensa.tensor as T
import condensa.functional as F
```

Listing 2: Preamble code for all scheme implementations.

features can be implemented in other similar frameworks such as TensorFlow [1] and MXNET [7]. We currently use a publicly available Python library for Bayesian global optimization with Gaussian Processes [12]. In a large-scale production setting, the configuration spaces may be larger and the trade-offs more complex; we plan to use a more scalable Bayesian optimization library such as Ray [44] in the future to address these issues.

Network Thinning Condensa comes pre-built with three *structure pruning* schemes: filter, neuron, and block pruning, as shown in Table 2. The application of these schemes may yield *zero structures*, which refer to blocks of zeros within a DNN's parameters. *Network thinning* refers to the process of identifying and removing such zero structures and consequently reducing the number of floating-point operations executed by the target hardware platform. Condensa employs a three-phase network thinning algorithm for structured pruning: in the first phase, we construct an in-memory graph representation of the target DNN. PyTorch makes this non-trivial, as its eager execution semantics preclude it from ever building a full graph-based representation of the DNN. To overcome this, we trace a forward execution path of the DNN and use it to construct an in-memory representation based on the ONNX format. In the next phase, we create a *thinning strategy* by analyzing the dependencies between the nodes of the graph constructed in the first phase. This step primarily involves keeping track of tensor dimension changes in a node due to thinning and ensuring that the corresponding tensor dimensions of the node's successors are appropriately adjusted. Due to the possibility of complex dependence patterns such as skip nodes in real-world DNNs (for example, deep residual networks [24]), this step is the most challenging to implement. In the final phase, we apply the thinning strategy obtained in phase 2 and physically alter tensor shapes to obtain the final thinned network. The Condensa Library provides a `thin` method which can be used to thin a given compressed model.

6.5 Pre-Built Schemes: Source Code

Condensa's tight integration with Python makes the expression of common compression patterns more natural. For example, operators can be combined with conditional statements to selectively compress layers based on properties of the input DNN and/or target hardware platform, as shown below:

```
# Prune only non-projection layers in ResNets
if not layer.is_projection: prune(layer)
# Quantize only if FP16 hardware is available
if platform_has_fast_fp16(): quantize(layer)
```

We list the full source code for the pre-built compression schemes shown in Table 2 in Listings 2 to 9.

```python
class Prune(object):
    """Performs unstructured pruning."""
    def __init__(self, layer_types):
        self._density = None
        self.layer_types = layer_types

    @property
    def density(self):
        return self._density

    @density.setter
    def density(self, d):
        self._density = d

    def threshold(self, module):
        vec = []
        for m in module.modules():
            if type(m) in self.layer_types and not hasattr(
                    m, 'condensa_nocompress'):
                all_weights = [n for n, _ in m.named_parameters()]
                weights = [x for x in all_weights if x.startswith('weight')]
                for w in weights:
                    vec.append(getattr(m, w).data.view(-1))
        return T.threshold(torch.cat(vec), self._density)

    def pi(self, module):
        threshold = self.threshold(module)
        for m in module.modules():
            if type(m) in self.layer_types and not hasattr(
                    m, 'condensa_nocompress'):
                all_weights = [n for n, _ in m.named_parameters()]
                weights = [x for x in all_weights if x.startswith('weight')]
                for w in weights:
                    condensa.prune(m, threshold, parameter=w)

    def delta(self, module):
        pass
```

Listing 3: Unstructured pruning scheme.

```python
class Quantize(object):
    """Quantizes network to given data-type."""
    def __init__(self, layer_types, dtype):
        self.dtype = dtype
        self.layer_types = layer_types

    def pi(self, module):
        for m in module.modules():
            if type(m) in self.layer_types and not hasattr(
                    m, 'condensa_nocompress'):
                condensa.quantize(m, self.dtype)

    def delta(self, module):
        for m in module.modules():
            if type(m) in self.layer_types and not hasattr(
                    m, 'condensa_nocompress'):
                condensa.dequantize(m, condensa.float32)
```

Listing 4: Quantization scheme.

```python
class NeuronPrune(object):
    """Prunes neurons from fully-connected layers."""
    def __init__(self, criteria=F.l2norm, prune_bias=True):
        self._density = None
        self.criteria = criteria
        self.prune_bias = prune_bias

    @property
    def density(self): return self._density

    @density.setter
    def density(self, d): self._density = d

    def threshold(self, module):
        vec = []
        for m in module.modules():
            if isinstance(m, torch.nn.Linear) and not hasattr(m, 'condensa_nocompress'):
                agg = T.aggregate_neurons(m.weight.data, self.criteria)
                vec.append(agg.view(-1))
        return T.threshold(torch.cat(vec), self._density)

    def pi(self, module):
        threshold = self.threshold(module)
        for m in module.modules():
            if isinstance(m, torch.nn.Linear) and not hasattr(m, 'condensa_nocompress'):
                condensa.neuron_prune(m, threshold, criteria=self.criteria, prune_bias=self.prune_bias)

    def delta(self, module): pass
```

Listing 5: Neuron pruning scheme.

```python
class FilterPrune(object):
    """Prunes filters from convolutional layers."""
    def __init__(self, criteria=F.l2norm, prune_bias=True):
        self._density = None
        self.criteria = criteria
        self.prune_bias = prune_bias

    @property
    def density(self): return self._density

    @density.setter
    def density(self, d): self._density = d

    def threshold(self, module):
        vec = []
        for m in module.modules():
            if isinstance(m, torch.nn.Conv2d) and not hasattr(m, 'condensa_nocompress'):
                agg = T.aggregate_filters(m.weight.data, self.criteria)
                vec.append(agg.view(-1))
        return T.threshold(torch.cat(vec), self._density)

    def pi(self, module):
        threshold = self.threshold(module)
        for m in module.modules():
            if isinstance(m, torch.nn.Conv2d) and not hasattr(m, 'condensa_nocompress'):
                condensa.filter_prune(m, threshold, criteria=self.criteria, prune_bias=self.prune_bias)

    def delta(self, module): pass
```

Listing 6: Filter pruning scheme.

```python
class StructurePrune(object):
    """Combines neuron and filter pruning."""
    def __init__(self, criteria=F.l2norm, prune_bias=True):
        self.density = None
        self.criteria = criteria
        self.prune_bias = prune_bias

    @property
    def density(self):
        return self._density

    @density.setter
    def density(self, d):
        self._density = d

    def threshold(self, module):
        vec = []
        for m in module.modules():
            if isinstance(m, torch.nn.Linear) and not hasattr(
                    m, 'condensa_nocompress'):
                agg = T.aggregate_neurons(m.weight.data, self.criteria)
                vec.append(agg.view(-1))
            if isinstance(m, torch.nn.Conv2d) and not hasattr(
                    m, 'condensa_nocompress'):
                agg = T.aggregate_filters(m.weight.data, self.criteria)
                vec.append(agg.view(-1))
        return T.threshold(torch.cat(vec), self._density)

    def pi(self, module):
        threshold = self.threshold(module)
        for m in module.modules():
            if isinstance(m, torch.nn.Linear) and not hasattr(
                    m, 'condensa_nocompress'):
                condensa.neuron_prune(m,
                                      threshold,
                                      criteria=self.criteria,
                                      prune_bias=self.prune_bias)
            if isinstance(m, torch.nn.Conv2d) and not hasattr(
                    m, 'condensa_nocompress'):
                condensa.filter_prune(m,
                                      threshold,
                                      align=self.align,
                                      criteria=self.criteria,
                                      prune_bias=self.prune_bias)

    def delta(self, module):
        pass
```

Listing 7: Structure pruning scheme.

```
class BlockPrune(object):
    """Prunes blocks in Linear layers."""
    def __init__(self, block_size, layer_types, criteria=F.l2norm):
        self._density = None
        self.block_size = block_size
        self.criteria = criteria
        self.layer_types = layer_types

    @property
    def density(self):
        return self._density

    @density.setter
    def density(self, d):
        self._density = d

    def threshold(self, module):
        vec = []
        for m in module.modules():
            if type(m) in self.layer_types and not hasattr(m, 'condensa_nocompress'):
                all_weights = [n for n, _ in m.named_parameters()]
                weights = [x for x in all_weights if x.startswith('weight')]
                for w in weights:
                    agg = T.aggregate(getattr(m, w).data, self.block_size, self.
                        criteria)
                    vec.append(agg.view(-1))
        return T.threshold(torch.cat(vec), self._density)

    def pi(self, module):
        threshold = self.threshold(module)
        for m in module.modules():
            if type(m) in self.layer_types and not hasattr(m, 'condensa_nocompress'):
                all_weights = [n for n, _ in m.named_parameters()]
                weights = [x for x in all_weights if x.startswith('weight')]
                for w in weights:
                    condensa.blockprune(m,
                                        threshold,
                                        block_size=self.block_size,
                                        criteria=self.criteria,
                                        parameter=w)

    def delta(self, module):
        pass
```

Listing 8: Block pruning scheme.

```
class Compose(object):
    """Composes two or more schemes together."""
    def __init__(self, schemes):
        if not isinstance(schemes, list):
            raise TypeError('Please␣specify␣schemes␣to␣compose␣as␣a␣list')
        self.schemes = schemes

    def pi(self, module):
        for s in self.schemes:
            s.pi(module)

    def delta(self, module):
        for s in reversed(self.schemes):
            s.delta(module)
```

Listing 9: Scheme composition.