

A PROGRAMMABLE APPROACH TO MODEL COMPRESSION

Vinu Joseph¹ Saurav Muralidharan² Animesh Garg³ Michael Garland² Ganesh Gopalakrishnan¹

ABSTRACT

Deep neural networks frequently contain far more weights, represented at a higher precision, than are 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 the results 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 CONDENSE. 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, CONDENSE uses a novel sample-efficient constrained Bayesian optimization algorithm to automatically infer desirable 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 $2.22\times$ using at most 10 samples per search. We have released a reference implementation of CONDENSE at <https://github.com/NVlabs/condense>.

1 INTRODUCTION

Modern deep neural networks (DNNs) are complex, and often contain millions of parameters spanning dozens or even hundreds of layers (He et al., 2016; Huang et al., 2017). 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 (Han et al., 2015b; Luo et al., 2017), quantization (Gupta et al., 2015), and low-rank factorization (Jaderberg et al., 2014; Denton et al., 2014). 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 Net-

work (CNN), the former strategy may prune convolutional filters (Li et al., 2016), 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 pruning 2-D blocks of non-zero weights (Gray et al., 2017) for a language modeling network such as Transformer (Vaswani et al., 2017), 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.

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.

¹University of Utah ²NVIDIA ³University of Toronto. Correspondence to: Vinu Joseph <vinu@cs.utah.edu>.

This research was developed with funding from the Defense Advanced Research Projects Agency (DARPA). The views, opinions and/or findings expressed are those of the author and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government. Distribution Statement "A" (Approved for Public Release, Distribution Unlimited).

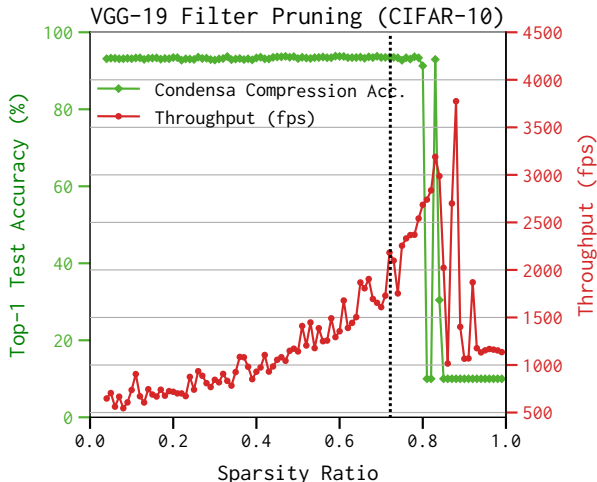


Figure 1. Top-1 test accuracy (green) and throughput (red) vs. sparsity ratio for VGG-19 on CIFAR-10. CONDENSEA is designed to solve constrained optimization problems of the form “maximize throughput, with a lower bound on accuracy”. In this case, CONDENSEA automatically discovers a sparsity ratio (vertical dashed line) and compresses the model to this ratio, improving throughput by $2.22\times$ and accuracy by 0.5%.

As an illustration of the level of automation provided by CONDENSEA, consider the problem of improving the inference throughput of VGG-19 (Simonyan & Zisserman, 2014) on the CIFAR-10 image classification task (Krizhevsky et al., 2014). 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 (He et al., 2018a). Figure 1 shows the accuracy and throughput obtained by CONDENSEA 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 CONDENSEA’s solution corresponds to a sparsity ratio of 0.72 and is depicted as the vertical dashed line. This result is significant for two reasons: (1) using the CONDENSEA 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 0.72 (shown as the vertical dashed line in the Figure) that achieves a state-of-the-art throughput of 2130 images/sec ($2.22\times$ improvement) and a top-1 accuracy improvement of 0.5% was obtained automatically by CONDENSEA 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).

¹Note 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.

CONDENSEA supports the expression of the overall *compression scheme* in Python using operators provided by the CONDENSEA 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. CONDENSEA 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 CONDENSEA’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.

This paper makes the following contributions: (1) it introduces CONDENSEA, 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 CONDENSEA 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 $2.22\times$ using at most 10 samples per search.

2 BACKGROUND

For a given task such as image classification, assume we have trained a large *reference* model $\bar{\mathbf{w}} = \operatorname{argmin}_{\mathbf{w}} L(\mathbf{w})$, 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 $\bar{\mathbf{w}}$. Model compression can be performed in various ways, and CONDENSEA currently supports two commonly used techniques: pruning and quantization. In pruning, non-zero values from $\bar{\mathbf{w}}$ are eliminated or “pruned” to obtain Θ . Pruning is usually performed using some kind of thresholding (for e.g., 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 $\bar{\mathbf{w}}$ one of K codebook values, where the codebook may be fixed or adaptive. CONDENSEA supports low-precision approximation, which refers to assigning each parameter in $\bar{\mathbf{w}}$ 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.

DNN Compression Techniques There is considerable prior work on accelerating neural networks using structured weight pruning (Frankle & Carbin, 2018; Han et al., 2015b;a; Luo et al., 2017; Han et al., 2017; Dong et al., 2017; Han et al., 2016; Polyak & Wolf, 2015; Hu et al., 2016; Anwar & Sung, 2016; Molchanov et al., 2016), quantization (Zhu et al., 2016; Gong et al., 2014) and low-rank tensor factorization (Lebedev et al., 2014; Xue et al., 2013; Denton et al., 2014; Girshick, 2015). Most of these individual compression schemes for pruning and quantization and their combinations can be expressed in CONDENSEA. Two common problems with these existing 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, as described in Section 3.

Automated Model Compression Bayesian optimization has previously been demonstrated to work well for general hyperparameter optimization in machine learning and neural architecture search (Snoek et al., 2012; Dai et al., 2019). 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 (He et al., 2018b) or simulated annealing (Liu et al., 2019). In particular, the automation procedure for AMC (He et al., 2018b) 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 characterized to have an underlying incompatibility with the target pruning problem (Liu et al., 2019). AutoSlim (Liu et al., 2019) proposes an automated approach based on simulated annealing, and uses 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 CONDENSEA does.

General Compression Algorithms and Tools General accuracy recovery algorithms capable of handling a wide variety of compression techniques provide the foundation for systems like CONDENSEA. Apart from the L-C algorithm (Carreira-Perpinán, 2017) which CONDENSEA uses, other recent accuracy recovery algorithms have been proposed. ADAM-ADMM (Zhang et al., 2018) pro-

poses 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 (Zhuang et al., 2018) 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. CONDENSEA can readily support such algorithms as additional optimizers as described in Section 3. Neural network distiller (Zmora et al., 2018) and TensorFlow model optimization toolkit (Google, 2019) are two recent open-source model compression frameworks that support multiple compression schemes. While these projects share a number of common goals with CONDENSEA, 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.

3 CONDENSEA FRAMEWORK

Figure 2 provides a high-level overview of the CONDENSEA framework. As shown on the left side of the figure, a user compresses a pre-trained model \bar{w} by specifying a compression scheme and an objective function f . Both the scheme and objective are specified in Python using operators from the CONDENSEA library; alternatively, users may choose from a selection of commonly used built-in schemes and objectives. The CONDENSEA library is described in more detail in Section 3.1. Apart from the operator 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 3.2), 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 3.2 and 3.3, respectively, and the sparsity

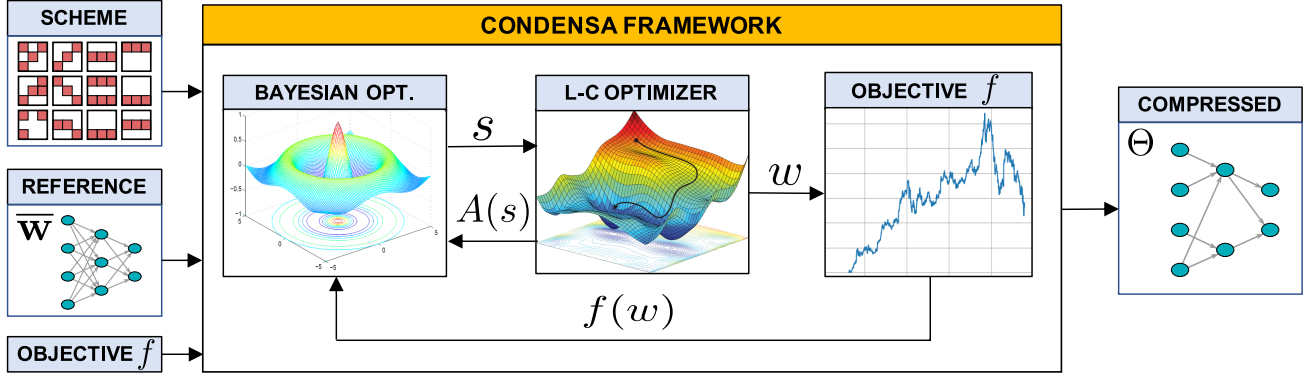


Figure 2. CONDENSED framework overview. The user provides the pre-trained model (\bar{w}), a compression scheme, and an objective function f . CONDENSED uses the Bayesian and L-C optimizers to infer an optimal sparsity ratio s^* and corresponding compressed model Θ .

inference algorithm is presented in Algorithm 1.

Listing 1 provides a concrete example of invoking CONDENSED 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 1 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, CONDENSED 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 Listing) automatically samples and evaluates global sparsity ratios as described in Section 3.3 and returns the final compressed model.

3.1 Condense Library

The CONDENSED 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 3.2.

CONDENSED’s tight integration with the Python ecosystem 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

```

1  # Construct pre-trained model
2  criterion = torch.nn.CrossEntropyLoss()
3  train(model, num_epochs, trainloader, criterion)
4
5  # Instantiate compression scheme
6  prune = condensa.schemes.FilterPrune()
7  # Define objective function
8  tput = condensa.objectives.throughput
9  # Specify optimization operator
10 obj = condensa.searchops.Maximize(tput)
11 # Instantiate L-C optimizer
12 lc = condensa.optimizers.LC(steps=30, lr=0.01)
13 # Build model compressor instance
14 compressor = condensa.Compressor(
15     model=model, # Trained model
16     objective=obj, # Objective
17     eps=0.02, # Accuracy threshold
18     optimizer=lc, # Accuracy recovery
19     scheme=prune, # Compression scheme
20     trainloader=trainloader, # Train dataloader
21     testloader=testloader, # Test dataloader
22     valloader=valloader, # Val dataloader
23     criterion=criterion # Loss criterion
24 )
25 # Obtain compressed model
26 wc = compressor.run()

```

Listing 1. Example usage of the CONDENSED library.

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)

```

Similarly, the use of iteration statements obviates the need for applying compression operators individually for each layer, resulting in more concise and readable schemes. This is in contrast to frameworks such as Distiller (Zmora et al., 2018) which require a per-layer declarative compression specification.

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 1. List of pre-built compression schemes in CONDENSEA.

Pre-built Schemes In addition to the layer-wise operators described above, the CONDENSEA Library also includes a set of pre-built compression *schemes* that operate on the full model. CONDENSEA 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 CONDENSEA without knowledge of low-level implementation details. The current set of pre-built schemes is listed in Table 1. Line 6 in Listing 1 shows an example instantiation of the Prune scheme.

3.2 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 3.3), CONDENSEA 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 $\bar{\mathbf{w}}$ (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 CONDENSEA must ideally have three important attributes: (1) able to handle all the compression operators supported by CON-

DENSEA, (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 (Carreira-Perpinán & Idelbayev, 2018), since it satisfies all three of the above requirements. In L-C, model compression is formulated as a constrained optimization problem:

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

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.

Eq. 1 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 of Eq 1 non-convex, and (2) the decompression mapping $\mathcal{D}(\Theta)$ typically adds another layer of non-convexity caused by an underlying combinatorial problem. While a number of non-convex algorithms may be used to solve Eq 1, we focus on the augmented Lagrangian (AL) method (Wright & Nocedal, 1999) implemented in the L-C algorithm (Carreira-Perpinán & Idelbayev, 2018) in this paper, since it is relatively efficient and easy to implement. As its name indicates, the L-C algorithm alternates between two steps: a learning (L) step which trains the uncompressed model but with a quadratic regularization term, and a compression (C) step, which finds the best compression of \mathbf{w} (the current uncompressed model) and corresponds to the definition of compression mapping \mathcal{C} . Due to space restrictions, we refer the reader to (Carreira-Perpinán & Idelbayev, 2018) for a more detailed description of the L-C algorithm. Other recent AL-based algorithms that could potentially be used include ADMM (Zhang et al., 2018) and DCP (Zhuang et al., 2018).

3.3 Bayesian Hyperparameter Optimization

It is intuitive to split the problem of finding optimal sparsity ratios into two stages: (1) find the highest sparsity value that loses at most ϵ accuracy w.r.t the original uncompressed model, and (2) in a constrained sparsity regime obtained from stage I, optimize a user-provided objective function f (e.g., throughput, or memory 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 func-

tion evaluation. However, for compression, each function evaluation may amount to optimizing the full model, which 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 (Mockus et al., 1978; Jones, 2001). These methods construct a probabilistic model of f with sequential evaluation, and then exploit 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 (Brochu et al., 2010).

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 a 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 (Rasmussen & Williams, 2006). 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 function observations. The support and properties of the resulting distribution on functions are determined by a mean function $m : \mathcal{X} \rightarrow \mathbb{R}$ and a positive definite covariance function $K : \mathcal{X} \times \mathcal{X} \rightarrow \mathbb{R}$.

Design of Acquisition Function. The GP prior and sequential function evaluations induce a posterior over the function of interest f ; the acquisition function, which we denote by $a : \mathcal{X} \rightarrow \mathbb{R}^+$ is the utility model that guides the next best point for function evaluation. These acquisition functions depend on the previous observations $\{\mathbf{x}_n, y_n\}$ and the GP hyperparameters ρ ; we denote this dependence as $a(\mathbf{x}; \{\mathbf{x}_n, y_n\}, \rho)$. Under the Gaussian process prior, the acquisition function depends on the model solely through its predictive mean function $\mu(\mathbf{x}; \{\mathbf{x}_n, y_n\}, \rho)$ and predictive variance function $\sigma^2(\mathbf{x}; \{\mathbf{x}_n, y_n\}, \rho)$. 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.

1. Probability of Improvement. This intuitive strategy maximizes the probability of improving over the best current value (Kushner, 1964). Under the GP this can be computed

Algorithm 1 Bayesian Hyperparameter Inference

```

1: Input:  $\bar{\mathbf{w}}, \epsilon$ 
2: Output:  $s^*$ 
3:  $\text{AcqFn} \leftarrow \text{ILS-UCB}(L = \bar{\mathbf{w}}_{acc} - \epsilon, s = (0, 1))$ 
4:  $s_{acc} \leftarrow \text{BayesOpt}(\mathcal{B}_f = \text{L-C}, \text{AcqFn})$ 
5:  $\text{AcqFn} \leftarrow \text{GP-UCB}(s = (0, s_{acc}))$ 
6:  $s^* \leftarrow \text{BayesOpt}(\mathcal{B}_f = f, \text{AcqFn})$ 
7:
8: BayesOpt
9: Input:  $\mathcal{B}_f, \text{AcqFn}$ 
10: Output:  $s$ 
11:  $\text{GP} \leftarrow \text{GP-Regressor.initialize}()$ 
12: for  $t \leftarrow 0, 1, 2, \dots$  do
13:    $s_t \leftarrow \text{argmax}_s \text{AcqFn}(s | D_{1:t-1})$ 
14:    $y_t \leftarrow f(s_t)$ 
15:    $D_{1:t} \leftarrow \{D_{1:t-1}, (s_t, y_t)\}$ 
16:    $\text{GP.Update}(D_{1:t})$ 
17:   if  $t > 0$  and  $s_t == s_{t-1}$  then
18:     return  $s_t$ 
19:   end if
20: end for

```

analytically as: $a_{\text{PI}}(\mathbf{x}; \{\mathbf{x}_n, y_n\}, \rho) = \Phi(\gamma(\mathbf{x}))$, where $\gamma(\mathbf{x}) = \frac{f(\mathbf{x}_{\text{best}}) - \mu(\mathbf{x}; \{\mathbf{x}_n, y_n\}, \rho)}{\sigma(\mathbf{x}; \{\mathbf{x}_n, y_n\}, \rho)}$.

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: $a_{\text{EI}}(\mathbf{x}; \{\mathbf{x}_n, y_n\}, \rho) = \sigma(x; \{\mathbf{x}_n, y_n\}, \rho) - \kappa \sigma(\mathbf{x}; \{\mathbf{x}_n, y_n\}, \rho)$, with a tunable κ to balance exploitation against exploration.

3. Upper/Lower Confidence Bound. Here, the functional approximation uncertainty is leveraged for acquisition through lower (upper) confidence bounds for functional min (max) (Srinivas et al., 2009). These acquisition functions have the form $a_{\text{UCB}}(\mathbf{x}; \{\mathbf{x}_n, y_n\}, \rho) = \mu(\mathbf{x}; \{\mathbf{x}_n, y_n\}, \rho) - \kappa \sigma(\mathbf{x}; \{\mathbf{x}_n, y_n\}, \rho)$, with a tunable κ to balance exploitation against exploration.

4. Level-Set Optimization. In addition to unconstrained optimization, to enable CONDENSA to achieve constraint satisfaction we build on top of level-set black-box optimization (Bogunovic et al., 2016; Garg et al., 2016; Zanette et al., 2018). We leverage a Gaussian Process Adaptive Sampling criterion called Implicit Level Set Upper Confidence Bound (ILS-UCB) (Garg et al., 2016), that prioritizes sampling near a level set of the estimate. This algorithm prioritizes searching the expected L-C curve intersection with user accuracy constraints, conditional on estimated uncertainty, and does not seek to precisely learn the shape of the entire L-C 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 conse-

quently 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:

$$\mathbf{x}_t = \operatorname{argmax}_{\mathbf{x} \in \mathcal{X}} (1 - \gamma)\sigma(\mathbf{x}) - \gamma|\mu(\mathbf{x}) - L| \quad (2)$$

Maximizing the acquisition function. CONDENSEA uses a combination of random sampling and the L-BFGS-B optimization method to find the maximum of the acquisition function. We first sample a few ($1e5$) warmup points at random, and then run 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 (Jones et al., 1993), 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 (Mockus, 1994; Lizotte, 2008). Note that the second term in Equation 2 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 CONDENSEA we use GP-UCB (GP-LCB) for function maximization (minimization) and ILS-UCB for solving constraints, as shown in Algorithm 1.

Algorithm Summary. We describe CONDENSEA’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 $(0, s_{acc})$.

3.4 Implementation

The CONDENSEA library and L-C optimizer are implemented in Python and are designed to inter-operate seamlessly with the PyTorch framework (Paszke et al., 2017). While we chose PyTorch for its widespread use in the machine learning community, it is worth noting that CONDENSEA’s design is general and that its features can be implemented in other similar frameworks such as TensorFlow (Abadi et al., 2016) and MXNET (Chen et al., 2015). We currently use a publicly available Python library for Bayesian global optimization

with Gaussian Processes (fmfn, 2019).

Network Thinning Condensa comes pre-built with three *structure pruning* schemes: filter, neuron, and block pruning, as shown in Table 1. 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 structure 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 (He et al., 2016)), 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 the `thin` method which can be used to thin a given compressed model.

4 EVALUATION

We conduct extensive experiments and fully analyze CONDENSEA on three tasks: (1) image classification on CIFAR-10 (Krizhevsky et al., 2014), (2) image classification on ILSVRC (ImageNet) (Deng et al., 2009), and (3) language modeling on WikiText-2 (Merity et al., 2016). We optimize the networks in each task for two distinct objectives: (1) minimize their memory footprint, and (2) maximize their inference throughput. We now describe the individual tasks and optimization objectives in more detail. We also describe how the Bayesian and L-C optimizers are set up.

Image Classification on CIFAR-10 The CIFAR-10 dataset (Krizhevsky et al., 2014) consists of $50k$ training and $10k$ testing 32×32 images in 10 classes. We train the VGG-19 (Simonyan & Zisserman, 2014) and ResNet56 (He et al., 2016) 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.

Image Classification on ImageNet Here, we use the VGG-

16 neural network (Simonyan & Zisserman, 2014) trained on the challenging ImageNet task (Deng et al., 2009), specifically the ILSVRC2012 version. We use PyTorch (Paszke et al., 2017) and default pretrained models as a starting point.

Language Modeling on WikiText-2 We trained a 2-layer LSTM model to perform a language modeling task on the WikiText-2 dataset (Merity et al., 2016). 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 the one used by Yu et al. (Yu et al., 2019).

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. While some real-world sparse matrix formats such as CSR and COO incur additional overheads in terms of storage, we believe that the footprint of a model is an effective proxy. The objective function f in this 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
```

For reducing footprint, we define a compression scheme that performs unstructured pruning of each learnable layer (except batch normalization layers), and then quantizes it to half-precision floating-point, yielding an additional 2x reduction. In CONDENSEA, this scheme can be constructed using the Compose operator as shown below (see Table 1 for the full list of schemes):

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

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. For WikiText-2, due to the lack of optimized block-sparse kernels for PyTorch, we measure the floating-point operations (FLOPs) of the compressed model instead as a proxy for inference performance. To improve throughput,

we focus on removing entire blocks of non-zeros, such as convolutional filters, since they have been proven to improve performance on real-world hardware (He et al., 2018a; Gray et al., 2017). For CIFAR-10 and ImageNet, we use filter pruning, since all the networks we consider are CNNs. In WikiText-2, we employ block pruning with a block size of 5. CONDENSEA makes it convenient to customize schemes in this manner based on the input DNN structure.

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 of 10^{-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 $\mu_j = \mu_0 a^j$, 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 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 (Carreira-Perpinán & Idelbayev, 2018). 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).

4.1 Results

We present the memory footprint reductions and inference throughput improvements obtained by CONDENSEA for each of the three tasks we evaluate in Table 2. For each task, we list the sparsity ratio obtained by the CONDENSEA Bayesian optimizer (s^* in the table), its corresponding accuracy/perplexity (top-1 accuracy, top-5 accuracy, and log perplexity for CIFAR-10, ImageNet, and WikiText-2, respectively), memory footprint reductions using pruning and quantization (column labeled r_c), and inference throughput/FLOP

Table 2. CONDENSEA performance results on CIFAR-10, ImageNet, and WikiText-2. s^* is the sparsity ratio obtained by CONDENSEA, 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%				
CONDENSEA P+Q ($\epsilon = 2\%$)	CIFAR-10	VGG19-BN	93.04%	0.97	8,7	65.25×	N/A
CONDENSEA 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 (HE ET AL., 2018B)	CIFAR-10	RESNET56	90.1%	N/A	N/A	N/A	$s_F = 2\times$
CONDENSEA P+Q ($\epsilon = 2\%$)	CIFAR-10	RESNET56	91.2%	0.94	7,7	27×	N/A
CONDENSEA 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 (HE ET AL., 2017)	IMAGENET	VGG16-BN	89.80%	N/A	N/A	$\approx 4\times$	N/A
AUTO SLIM (LIU ET AL., 2019)	IMAGENET	VGG16-BN	90.90%	N/A	N/A	6.4×	N/A
AMC (HE ET AL., 2018B)	IMAGENET	VGG16-BN	90.10%	N/A	N/A	N/A	$s_F = 1.25\times$
CONDENSEA P+Q ($\epsilon = 2\%$)	IMAGENET	VGG16-BN	89.89%	0.92	8,7	25.59×	N/A
CONDENSEA 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				
(YU ET AL., 2019)	WIKITEXT-2	LSTM	4.70	N/A	N/A	$\approx 10\times$	N/A
CONDENSEA P+Q ($\epsilon = 2\%$)	WIKITEXT-2	LSTM	4.75	0.92	9,7	4.2×	N/A
CONDENSEA BLOCK ($\epsilon = 2\%$)	WIKITEXT-2	LSTM	4.77	0.61	8,7	N/A	$s_F = 2.2\times$

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 (He et al., 2018b) and AutoSlim (Liu et al., 2019), and for WikiText-2, we compare with (Yu et al., 2019). Since AMC (He et al., 2018b) and (Yu et al., 2019) do not report actual runtime numbers on hardware, we report the corresponding FLOP improvements instead (values marked s_F). We also use FLOP reduction as a metric for LSTM block pruning, as described above. Overall, we obtain memory footprint reductions of up to 65.25× and inference throughput improvements of up to 2.22×. On CIFAR-10, we notice relatively smaller throughput improvements on ResNet56 due to inter-layer filter dependencies that prevented us from performing aggressive network thinning (we describe network thinning in more detail in Section 3.4).

4.2 Sparsity Profile Analysis

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 are assumed to be unknown in our problem formulation, but we compute them explicitly here to better understand the quality of solutions produced by CONDENSEA. 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 are unable to show the full profile for ImageNet due to its significantly higher computation cost: collecting each data point for compression accuracy requires over 12 hours of compute time on a node with 8 Tesla V100 GPUs. Inference throughput, FLOPs, and memory footprint data is collected for each compressed model and depicted by red lines in the figures (right-hand-side y-axis). We also show direct compression (DC) accuracies in gray for comparison (DC is described in more detail in Section 3.2). In each figure, the sparsity ratio found by CONDENSEA is shown as a black vertical dashed line.

We notice three important trends in Figures 3 and 4: (1) CONDENSEA 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.

4.3 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 0.72 (found by the Bayesian optimizer, as shown in Table 2) for this experiment. We report the mean runtimes over 100 executions as obtained using TensorRT.

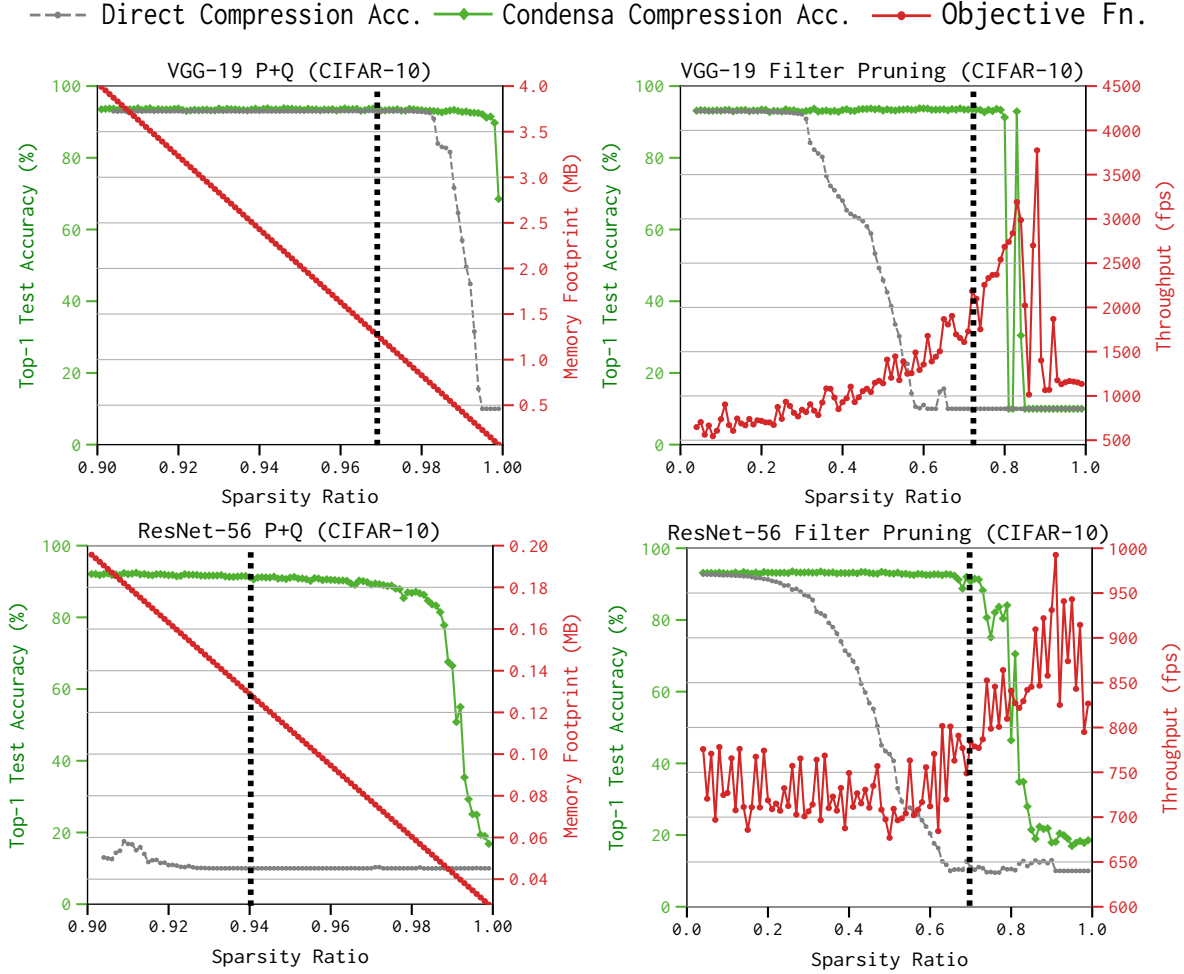


Figure 3. CONDENSEA sparsity profiles for VGG19-BN and ResNet56 for CIFAR-10. Column 1 shows the problem of the form “minimize *memory footprint* with a lower bound on accuracy”, while Column 2 illustrates “maximize *throughput* with a lower bound on accuracy”. The DC line (gray) shows accuracy values if no accuracy recovery with L-C is performed. Note that the x-axis ranges are different: the plots on the left have sparsity ratio values ranging from 0.9 to 1.0 while those on the right have values ranging from 0 to 1.

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. 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.

5 CONCLUSIONS

This paper has presented CONDENSEA, which is a programming system for model compression. CONDENSEA 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. On three real-world image classification and language modeling tasks, CONDENSEA achieves memory footprint reductions of up to $65\times$ and runtime throughput improvements of up to $2.22\times$ using at most 10 samples per search. With the initial framework in place, we envision a number of directions to expand on CONDENSEA’s capability. For example, we plan to augment automatic sparsity ratio inference with support for additional compression hyperparameters such as block sizes in block-sparsification (Gray et al., 2017), and data types for quantization. Our long-term goal is a framework that makes model compression easier, more flexible, and accessible to a wide range of users.

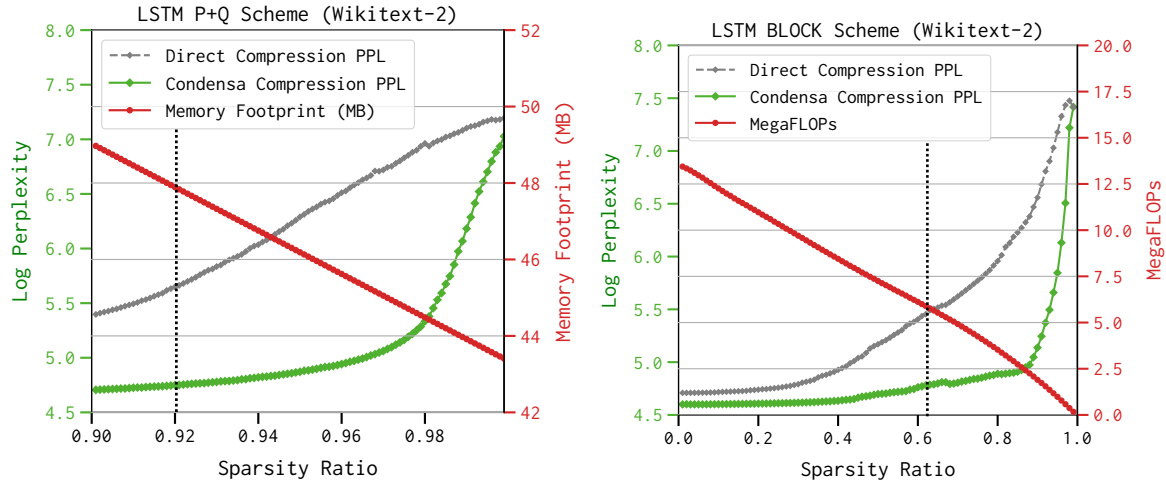


Figure 4. WikiText-2 two-layer LSTM results for pruning + quantization (left) and block pruning with block size of 5 (right). Note that the x-axis ranges are different: the plot on the left has sparsity ratio values ranging from 0.9 to 1.0 while the one on the right has values ranging from 0 to 1.

Table 3. Layer-wise TensorRT runtimes and speedups for filter pruning of VGG-19 on CIFAR-10. R and C denote reference and compressed models, respectively.

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	1.4×
CONV2	3 x 3 x 64 x 64	3 x 3 x 23 x 58	0.23	0.11	2.09×
CONV3	3 x 3 x 64 x 128	3 x 3 x 58 x 126	0.12	0.17	0.71×
CONV4	3 x 3 x 128 x 128	3 x 3 x 126 x 127	0.22	0.23	0.95×
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	0.35	1.17×
CONV9	3 x 3 x 256 x 512	3 x 3 x 241 x 214	0.28	0.16	1.75×
CONV10	3 x 3 x 512 x 512	3 x 3 x 214 x 71	0.54	0.03	18×
CONV11	3 x 3 x 512 x 512	3 x 3 x 71 x 30	0.53	0.02	26.5×
CONV12	3 x 3 x 512 x 512	3 x 3 x 30 x 38	0.53	0.01	53×
CONV13	3 x 3 x 512 x 512	3 x 3 x 38 x 48	0.56	0.03	18.66×
CONV14	3 x 3 x 512 x 512	3 x 3 x 48 x 38	0.56	0.02	28×
CONV15	3 x 3 x 512 x 512	3 x 3 x 38 x 48	0.56	0.02	28×
CONV16	3 x 3 x 512 x 512	3 x 3 x 28 x 102	0.56	0.03	18.66×

REFERENCES

- Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., et al. Tensorflow: a system for large-scale machine learning. In *OSDI*, volume 16, pp. 265–283, 2016.
- Anwar, S. and Sung, W. Compact deep convolutional neural networks with coarse pruning. *arXiv preprint arXiv:1610.09639*, 2016.
- Bogunovic, I., Scarlett, J., Krause, A., and Cevher, V. Truncated variance reduction: A unified approach to bayesian optimization and level-set estimation. In *Advances in neural information processing systems*, pp. 1507–1515, 2016.
- Brochu, E., Cora, V. M., and De Freitas, N. 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.
- Carreira-Perpinán, M. A. Model compression as constrained optimization, with application to neural nets. part I: General framework. *arXiv preprint arXiv:1707.01209*, 2017.
- Carreira-Perpinán, M. A. and Idelbayev, Y. “learning-compression” algorithms for neural net pruning. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 8532–8541, 2018.
- Chen, T., Li, M., Li, Y., Lin, M., Wang, N., Wang, M., Xiao,

- T., Xu, B., Zhang, C., and Zhang, Z. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. *arXiv preprint arXiv:1512.01274*, 2015.
- Dai, X., Zhang, P., Wu, B., Yin, H., Sun, F., Wang, Y., Dukhan, M., Hu, Y., Wu, Y., Jia, Y., et al. Chamnet: Towards efficient network design through platform-aware model adaptation. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 11398–11407, 2019.
- Deng, J., Dong, W., Socher, R., Li, L.-J., Li, K., and Fei-Fei, L. Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*, pp. 248–255. Ieee, 2009.
- Denton, E. L., Zaremba, W., Bruna, J., LeCun, Y., and Fergus, R. Exploiting linear structure within convolutional networks for efficient evaluation. In *Advances in neural information processing systems*, pp. 1269–1277, 2014.
- Dong, X., Huang, J., Yang, Y., and Yan, S. More is less: A more complicated network with less inference complexity. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 5840–5848, 2017.
- fmfn. A Python implementation of global optimization with Gaussian processes. <https://github.com/fmfn/BayesianOptimization>, 2019. [Online; accessed 1-September-2019].
- Frankle, J. and Carbin, M. The lottery ticket hypothesis: Training pruned neural networks. *arXiv preprint arXiv:1803.03635*, 2018.
- Garg, A., Sen, S., Kapadia, R., Jen, Y., McKinley, S., Miller, L., and Goldberg, K. Tumor localization using automated palpation with gaussian process adaptive sampling. In *2016 IEEE International Conference on Automation Science and Engineering (CASE)*, pp. 194–200. IEEE, 2016.
- Girshick, R. Fast r-cnn. In *Proceedings of the IEEE international conference on computer vision*, pp. 1440–1448, 2015.
- Gong, Y., Liu, L., Yang, M., and Bourdev, L. Compressing deep convolutional networks using vector quantization. *arXiv preprint arXiv:1412.6115*, 2014.
- Google. TensorFlow model optimization toolkit. <https://github.com/tensorflow/model-optimization>, 2019. [Online; accessed 1-September-2019].
- Gray, S., Radford, A., and Kingma, D. P. GPU kernels for block-sparse weights. *arXiv preprint arXiv:1711.09224*, 2017.
- Gupta, S., Agrawal, A., Gopalakrishnan, K., and Narayanan, P. Deep learning with limited numerical precision. In *International Conference on Machine Learning*, pp. 1737–1746, 2015.
- Han, S., Mao, H., and Dally, W. J. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv preprint arXiv:1510.00149*, 2015a.
- Han, S., Pool, J., Tran, J., and Dally, W. Learning both weights and connections for efficient neural network. In *Advances in neural information processing systems*, pp. 1135–1143, 2015b.
- Han, S., Liu, X., Mao, H., Pu, J., Pedram, A., Horowitz, M. A., and Dally, W. J. Eie: efficient inference engine on compressed deep neural network. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, pp. 243–254. IEEE, 2016.
- Han, S., Kang, J., Mao, H., Hu, Y., Li, X., Li, Y., Xie, D., Luo, H., Yao, S., Wang, Y., 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*, pp. 75–84. ACM, 2017.
- He, K., Zhang, X., Ren, S., and Sun, J. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 770–778, 2016.
- He, Y., Zhang, X., and Sun, J. Channel pruning for accelerating very deep neural networks. In *International Conference on Computer Vision (ICCV)*, volume 2, 2017.
- He, Y., Dong, X., Kang, G., Fu, Y., and Yang, Y. Progressive deep neural networks acceleration via soft filter pruning. *arXiv preprint arXiv:1808.07471*, 2018a.
- He, Y., Lin, J., Liu, Z., Wang, H., Li, L.-J., and Han, S. Amc: Automl for model compression and acceleration on mobile devices. In *Proceedings of the European Conference on Computer Vision (ECCV)*, pp. 784–800, 2018b.
- Hu, H., Peng, R., Tai, Y.-W., and Tang, C.-K. Network trimming: A data-driven neuron pruning approach towards efficient deep architectures. *arXiv preprint arXiv:1607.03250*, 2016.
- Huang, G., Liu, Z., Van Der Maaten, L., and Weinberger, K. Q. Densely connected convolutional networks. In *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 2261–2269. IEEE, 2017.
- Jaderberg, M., Vedaldi, A., and Zisserman, A. Speeding up convolutional neural networks with low rank expansions. *arXiv preprint arXiv:1405.3866*, 2014.

- Jones, D. R. A taxonomy of global optimization methods based on response surfaces. *Journal of global optimization*, 21(4):345–383, 2001.
- Jones, D. R., Perttunen, C. D., and Stuckman, B. E. Lipschitzian optimization without the lipschitz constant. *Journal of optimization Theory and Applications*, 79(1):157–181, 1993.
- Krizhevsky, A., Nair, V., and Hinton, G. The cifar-10 dataset. online: <http://www.cs.toronto.edu/kriz/cifar.html>, 55, 2014.
- Kushner, H. J. 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.
- Lebedev, V., Ganin, Y., Rakhuba, M., Oseledets, I., and Lempitsky, V. Speeding-up convolutional neural networks using fine-tuned cp-decomposition. *arXiv preprint arXiv:1412.6553*, 2014.
- Li, H., Kadav, A., Durdanovic, I., Samet, H., and Graf, H. P. Pruning filters for efficient convnets. *arXiv preprint arXiv:1608.08710*, 2016.
- Liu, N., Ma, X., Xu, Z., Wang, Y., Tang, J., and Ye, J. Autoslim: An automatic dnn structured pruning framework for ultra-high compression rates. *arXiv preprint arXiv:1907.03141*, 2019.
- Lizotte, D. J. *Practical bayesian optimization*. University of Alberta, 2008.
- Luo, J.-H., Wu, J., and Lin, W. Thinet: A filter level pruning method for deep neural network compression. *arXiv preprint arXiv:1707.06342*, 2017.
- Merity, S., Xiong, C., Bradbury, J., and Socher, R. Pointer sentinel mixture models. *arXiv preprint arXiv:1609.07843*, 2016.
- Mockus, J. Application of bayesian approach to numerical methods of global and stochastic optimization. *Journal of Global Optimization*, 4(4):347–365, 1994.
- Mockus, J., Tiesis, V., and Zilinskas, A. The application of bayesian methods for seeking the extremum. *Towards global optimization*, 2(117-129):2, 1978.
- Molchanov, P., Tyree, S., Karras, T., Aila, T., and Kautz, J. Pruning convolutional neural networks for resource efficient inference. *arXiv preprint arXiv:1611.06440*, 2016.
- Paszke, A., Gross, S., Chintala, S., Chanan, G., Yang, E., DeVito, Z., Lin, Z., Desmaison, A., Antiga, L., and Lerer, A. Automatic differentiation in pytorch. In *NIPS-W*, 2017.
- Polyak, A. and Wolf, L. Channel-level acceleration of deep face representations. *IEEE Access*, 3:2163–2175, 2015.
- Rasmussen, C. E. and Williams, C. Gaussian processes for machine learning the mit press, 2006.
- Simonyan, K. and Zisserman, A. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.
- Snoek, J., Larochelle, H., and Adams, R. P. Practical bayesian optimization of machine learning algorithms. In *Advances in neural information processing systems*, pp. 2951–2959, 2012.
- Srinivas, N., Krause, A., Kakade, S. M., and Seeger, M. Gaussian process optimization in the bandit setting: No regret and experimental design. *arXiv preprint arXiv:0912.3995*, 2009.
- Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., and Polosukhin, I. Attention is all you need. In *Advances in neural information processing systems*, pp. 5998–6008, 2017.
- Wright, S. and Nocedal, J. Numerical optimization. *Springer Science*, 35(67-68):7, 1999.
- Xue, J., Li, J., and Gong, Y. Restructuring of deep neural network acoustic models with singular value decomposition. In *Interspeech*, pp. 2365–2369, 2013.
- Yu, H., Edunov, S., Tian, Y., and Morcos, A. S. Playing the lottery with rewards and multiple languages: lottery tickets in rl and nlp. *arXiv preprint arXiv:1906.02768*, 2019.
- Zanette, A., Zhang, J., and Kochenderfer, M. J. Robust super-level set estimation using gaussian processes. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, pp. 276–291. Springer, 2018.
- Zhang, T., Zhang, K., Ye, S., Li, J., Tang, J., Wen, W., Lin, X., Fardad, M., and Wang, Y. Adam-ADMM: A unified, systematic framework of structured weight pruning for DNNs. *arXiv preprint arXiv:1807.11091*, 2018.
- Zhu, C., Han, S., Mao, H., and Dally, W. J. Trained ternary quantization. *arXiv preprint arXiv:1612.01064*, 2016.
- Zhuang, Z., Tan, M., Zhuang, B., Liu, J., Guo, Y., Wu, Q., Huang, J., and Zhu, J. Discrimination-aware channel pruning for deep neural networks. In *Advances in Neural Information Processing Systems*, pp. 883–894, 2018.
- Zmora, N., Jacob, G., and Novik, G. Neural network distiller, June 2018. URL <https://doi.org/10.5281/zenodo.1297430>.