# BEYOND FLOPS IN LOW-RANK COMPRESSION OF NEURAL NETWORKS: OPTIMIZING DEVICE-SPECIFIC INFERENCE RUNTIME

*Yerlan Idelbayev* *Miguel Á. Carreira-Perpiñán*

Department of Computer Science and Engineering, University of California, Merced

{yidelbayev, mcarreira-perpinan}@ucmerced.edu

http://eecs.ucmerced.edu

## ABSTRACT

Neural network compression has become an important practical step when deploying trained models. We consider the problem of low-rank compression of the neural networks with the goal of optimizing the measured inference time. Given a neural network and a target device to run it, we want to find the matrix ranks and the weight values of the compressed model so that network runs as fast as possible on the device while having best task performance (e.g., classification accuracy). This is a hard optimization problem involving weights, ranks, and device constraints. To tackle this problem, we first implement a simple yet accurate model of the on-device runtime that requires only a few measurements. Then we give a suitable formulation of the optimization problem involving the proposed runtime model and solve it using alternating optimization. We validate our approach on various neural networks and show that by using our estimated runtime model we achieve better task performance compared to FLOPs based methods for the same runtime budget on the actual device.

***Index Terms*—** inference-targeted compression, device-targeted compression, low-rank compression, rank selection, neural network compression

## 1. INTRODUCTION

The state-of-the-art neural network solutions to many computer vision and machine learning problems have prompted a widespread deployment of these models on small, close-to-user devices like phones, smartwatches, and other IoT devices. With modern sizes and computational demands of the neural networks (having millions of weights and requiring GFLOPs of compute), the deployment onto an IoT device poses *a question of model compression*: how to efficiently compress a large network to a small device so that it fits into the underlying hardware/quality-of-service constraints and yet performs as good as possible.

Often, an important target of compression is on-device inference time: for many tasks (e.g., real-time audio/video enhancement), a too high inference time is unacceptable. Different compression schemes have been proposed to address the inference time speed-up; however, most of the works handle it indirectly through a proxy optimization target: the total number of floating-point operations (FLOPs). While a smaller FLOPs count is indicative of a faster inference time, there is no one-to-one correspondence between these quantities. For example, on our testbed, the 727 MFLOPs version of the AlexNet (trained on ImageNet) runs a single image inference in 328 ms. In comparison, the CIFAR10 version of VGG16 has 314 MFLOPs, which is $2.32\times$ fewer than AlexNet; yet, it runs $6.07\times$ faster (54 ms) illustrating that on-device runtime does not only depend on the total FLOPs. Indeed, the inference runtime is the function of the neural network's overall structure and the hardware characteristics (e.g., frequency of CPU/GPU, size of the cache, memory speed), and it cannot be extrapolated from a single FLOPs-count number.

We consider the problem of inference-targeted compression of a neural network for a given device and adopt the low-rank compression as our method of choice. While such a scheme has a history of usage for network compression problems to reduce FLOPs and size of the networks (see sec. 1.1), we show that it can be effectively used to directly target the on-device inference time of compressed models due to the following. First, as we discuss in section 2, the low-rank scheme gives rise to a simple yet accurate device-runtime model that can be used to a precise estimation of the actual inference time of the compressed model. Second, the computational reductions of low-rank compression are realizable without specific hardware support (unlike, for instance, elementwise pruning [1]): if the layer with weights $\mathbf{W}$ is compressed with $r$-rank matrix $\mathbf{U}\mathbf{V}^T$, then forward pass of $\mathbf{W}\mathbf{x}$ through that layer can be implemented as a forward pass through a sequence of layers with weights $\mathbf{V}^T$ and $\mathbf{U}$.

The problem we are solving is challenging: we need to find the best configuration of ranks (one rank per layer, integer values) and corresponding low-rank weights (floating-point values) so that network has the fastest on-device inference time while maintaining its original task performance. Assuming we have $K$ layers with $M$ possible ranks per layer, the problem involves selection over the set of $M^K$ distinct rank configurations. However, as we show in section 3, a suitable formulation of this problem using the proposed device-runtime model admits an efficient algorithm involving alternation of simple steps: a step over weights of the neural networks (solved by stochastic gradient descent, SGD) and a step over the rank configurations (solved by enumeration). In section 4 we experimentally validate our approach's effectiveness by compressing the AlexNet and VGG16 to have fast inference time on the ARM Cortex-A57 CPU of the NVIDIA's Jetson Nano embedded computing platform.

In the remainder of this section, we give a brief overview of related work (sec. 1.1), and discuss the application of low-rank to the convolutional layers (sec. 1.2).

### 1.1. Related work

The low-rank methods have been long used for the purposes of compression and speeding-up the inference of neural networks. The prior methods [2–7] did not include the rank selection into the optimization problem and rather relied on setting the ranks upfront by various heuristics (e.g., select a certain portion of rank per layer, or thresh-

We thank NVIDIA Corporation for several GPU donations.

| | |
|---|---|
| CPU | Quad-core ARM Cortex-A57, 1.4 GHz |
| GPU | 128 CUDA cores at 0.9 GHz |
| RAM | 4 GB 64-bit LPDDR4, 1.6 GHz |
| OS | Ubuntu 18.04.5 LTS |
| Kernel | GNU/Linux 4.9.140-tegra |
| Storage | 128 GB microSDXC memory card |
| Software | PyTorch v1.6.0, ONNXRuntime v1.4.0 |

**Table 1**. Specifications of NVIDIA's Jetson Nano Developer kit used as our target testbed. While it has a built-in GPU, we used the CPU inference time (parallelized on two threads) as our compression goal.

old using a cumulative sum of singular values) and then optimize the low-rank weights using SGD. The results of low-rank compression can be drastically improved if the rank selection is included in the optimization by means of regularization [8, 9] or pruning constraints on the ranks [10, 11]. Along with the low-rank methods, the usage of tensor decompositions has been studied as well [12, 13].

Several works use the resulting number of FLOPs as an optimization criterion when optimizing over the ranks [9, 10, 14, 15]. However, we are not aware of any methods that directly optimize the on-device inference speed.

### 1.2. Low-rank compression of convolutional layers

When an $m \times n$ matrix $\mathbf{W}$ is compressed via an $r$-rank matrix, it can be written as $\mathbf{W} = \mathbf{U}\mathbf{V}^T$ where matrices $\mathbf{U}$ and $\mathbf{V}$ are of dimensions $m \times r$ and $n \times r$. Such transformation is easily applicable to the fully connected layers, however, the weights of convolutional layers come as 4D tensors: for example, with NCWH format (as in PyTorch) the weights are stored as a tensor of dimension $n \times c \times d \times d$. Here, $n$ is the number of convolutional filters, $c$ is the number of input image channels, and $d \times d$ is the spatial resolution of the filter. To apply the low-rank compression, this tensor must be reshaped into a matrix. Several reshapes have been studied in the literature: the scheme 1 [5, 16] reshapes the tensor into a matrix of $nd^2 \times c$, the scheme 2 [4, 9, 11] reshapes the tensor into a matrix of $nd \times cd$. The advantage of these particular reshaping schemes lies in the implementation of low-rank convolution: both schemes can be implemented as a sequence of smaller convolutional layers (with appropriate shapes).

## 2. DEVICE RUNTIME MODEL

Assume we are given a neural network with $K$ layers and the weights $\mathbf{W} = \{\mathbf{W}_1, \mathbf{W}_2, \ldots, \mathbf{W}_K\}$ where $\mathbf{W}_k$ is a weight matrix (or tensor) of the $k$th layer. The weights $\mathbf{W}$ implicitly define a computational graph for an inference pass through the network. When we execute this graph on the given hardware, we can measure the inference time. Throughout this paper, we define the inference time as the total time required to complete a forward pass of a single image through the computational graph, and call it $\mathcal{R}(\mathbf{W})$.

In our model, we assume that the total inference time $\mathcal{R}(\mathbf{W})$ is the sum of the inference times through each of the $K$ layers since layers have to be processed sequentially:

$$\mathcal{R}(\mathbf{W}) = \mathcal{R}_1(\mathbf{W}_1) + \mathcal{R}_2(\mathbf{W}_2) + \cdots + \mathcal{R}_K(\mathbf{W}_K). \quad (1)$$

Here, each of the $\mathcal{R}_k(\mathbf{W}_k)$ measures the total inference time through a layer $k$: this involves the time to load the weights and the inputs, actual computation time, and time to unload the output.

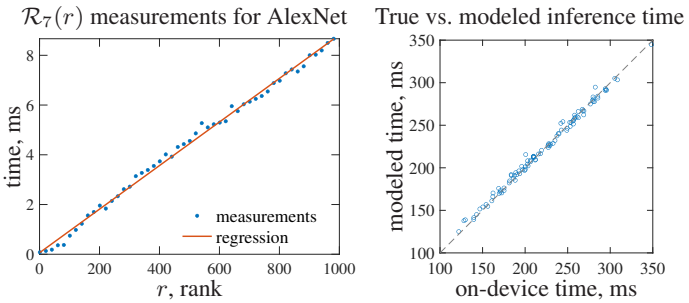


**Fig. 1**. *Left:* Measurements and regression fit to model the inference time as a function of rank for the 7th layer of AlexNet. *Right:* Plot of the actual, on device inference time for 100 randomly sampled low-rank configurations of AlexNet vs. the predictions of our model $\mathcal{R}(\mathbf{r})$. On these samples, the mean average error was 3.03 ms.

In reality, the right hand side of eq. (1) is an upper bound to the total runtime $\mathcal{R}(\mathbf{W})$: when the computational graph is executed optimally, some weights and inputs can be prefetched and layer-to-layer computations can be pipelined, thus, finishing earlier than the sum of separate inferences through each layer.

When we compress the network using the low-rank decomposition, the $k$th layer is compressed by an $r_k$-rank matrix, and the forward pass through the layer can be implemented as a sequence of fully-connected or convolutional layers (sec. 1.2). Since the computational graph is defined by the shape of the weight matrices, and not by the weight values, we conclude that the inference time through a layer $k$ is a function of the rank, and our model simplifies as:

$$\mathcal{R}(\mathbf{W}) = \mathcal{R}(\mathbf{r}) = \mathcal{R}_1(r_1) + \mathcal{R}_2(r_2) + \cdots + \mathcal{R}_K(r_K). \quad (2)$$

We make several observations. First, due to a small number of possible ranks per layer, we can directly measure the value of $\mathcal{R}_k(r)$ on the device. Essentially, $\mathcal{R}_k(r)$ is a lookup table with a single measurement for each $r$. Second, the proposed model is computationally efficient and avoids a combinatorial number of measurements. Assuming there are $M$ possible ranks per layer ($r_k = 1, \ldots, M$), rather than making $M^K$ measurements for all possible rank configuration we only need $MK$ on-device measurement in total.

Even though we need to consider $M$ ranks per layer, collecting the inference times might be time consuming and impractical. Particularly, the measurements need to be repeated many times to reduce the noise, however, too many measurements at a time might induce the thermal throttling[1] of the target device which adds inconsistencies to the model, and measurements need to be taken at intermittent intervals.

Due to the aforementioned considerations, in the actual implementation of the proposed model we collected the low-rank inference measurements at certain rank intervals and then fit a regression curve. To make the measurements, we use highly-optimized implementation of forward pass through ONNX runtime. When we used the CPU of Jetson Nano Developer board as our target device (see Table 1), we noticed that measurements within each layer follow a line trend except for some outliers (which are presumably caused by noise). Therefore, we computed an $\ell_1$-fit and used the fitted lines as our $\mathcal{R}_k$ functions, see left of Fig. 1. In our experiments, we found that $\ell_1$-fitted regression can model rank-dependent device runtime pretty accurately across all layers.

[1] https://en.wikipedia.org/wiki/CPU_throttling

How good is our model? To answer this question, we sampled random rank configurations for the AlexNet architecture by choosing each layer's rank uniformly (out of possible ones) and measured the true inference speeds of the sampled architectures. On the right of Fig. 1 we compare true inference times to the modeled inference times. As we can see, the difference between our model and the true inference time is minuscule: the average difference on the sampled low-rank architectures was 3.03 milliseconds.

## 3. RUNTIME-TARGETED LOW-RANK COMPRESSION

Having developed the device runtime model $\mathcal{R}(\mathbf{r})$ for a given $K$-layer network with weights $\mathbf{W} = \{\mathbf{W}_1, \dots, \mathbf{W}_K\}$, now we give a low-rank compression formulation that targets the inference time on the given device. We denote the network's task loss (e.g., cross-entropy) as $\mathcal{L}$ and define the following optimization problem of

$$\min_{\mathbf{W},\mathbf{r}} \mathcal{L}(\mathbf{W}) + \lambda\mathcal{R}(\mathbf{r}) \text{ s.t. } \operatorname{rank}(\mathbf{W}_k) = r_k,\ k = 1 \dots K, \quad (3)$$

where the term $\lambda\mathcal{R}(\mathbf{r})$ with user-chosen $\lambda > 0$ controls the amount of desired reduction of the inference time.

The problem given by eq. (3) is a mixed-integer optimization involving the floating-point weights of the neural network and the integer rank values. Typically, even the neural network part on its own (without the rank constraints) requires many iterations over the training dataset to be properly optimized (with SGD), and combination with rank constraints makes it truly challenging. Fortunately, this formulation falls into the category of *model compression as constrained optimization* problems [17] and admits an efficient solution based on the *Learning-Compression* (LC) algorithm [9, 14, 15, 18].

To derive the LC algorithm corresponding to our formulation, let us equivalently rewrite the constraints by introducing the auxiliary variables $\mathbf{\Theta}_k$ for each $k = 1, \dots, K$ as

$$\operatorname{rank}(\mathbf{W}_k) = r_k \iff \mathbf{W}_k = \mathbf{\Theta}_k,\ \operatorname{rank}(\mathbf{\Theta}_k) = r_k,$$

and then apply penalty method [19, ch.17] to the matrix terms (i.e., $\mathbf{W}_k = \mathbf{\Theta}_k$) while driving $\mu \to \infty$ (norms are Frobenius):

$$\begin{aligned} \min_{\mathbf{W},\mathbf{\Theta},\mathbf{r}} \mathcal{L}(\mathbf{W}) + \frac{\mu}{2}\sum_{k=1}^{K}\|\mathbf{W}_k - \mathbf{\Theta}_k\|^2 + \lambda\mathcal{R}(\mathbf{r}) \\ \text{s.t. } \operatorname{rank}(\mathbf{\Theta}_k) = r_k,\ k = 1, \dots, K. \end{aligned} \quad (4)$$

We use the quadratic penalty to simplify the derivations; however, in practice, we use the augmented Lagrangian method, which has an additional step over the vector of Lagrange multipliers. If we apply alternating optimization over variables $\mathbf{W}$ and $\{\mathbf{\Theta}, \mathbf{r}\}$ we obtain the substeps that can be efficiently handled:

- Learning (L) step. The step over $\mathbf{W}$ has the form of:

$$\min_{\mathbf{W}} \mathcal{L}(\mathbf{W}) + \frac{\mu}{2}\sum_{k=1}^{K}\|\mathbf{W}_k - \mathbf{\Theta}_k\|^2.$$

- Compression (C) step: The step over $\mathbf{\Theta}$ and $\mathbf{r}$ separates into $K$ smaller substeps due to the form of $\mathcal{R}$ (eq. 2):

$$\min_{\mathbf{\Theta}_k, r_k} \frac{\mu}{2}\|\mathbf{W}_k - \mathbf{\Theta}_k\|^2 + \lambda\mathcal{R}_k(r_k) \text{ s.t. } \operatorname{rank}(\mathbf{\Theta}_k) = r_k.$$

**Algorithm 1** LC algorithm to jointly learn weights and ranks when applying the low-rank compression to on-device inference speed.

**input** $K$-layer neural net with weights $\mathbf{W} = \{\mathbf{W}_1, \dots, \mathbf{W}_K\}$, hyperparameter $\lambda$, device runtime model $\mathcal{R}$.

$\mathbf{W} = (\mathbf{W}_1, \dots, \mathbf{W}_K) \leftarrow \arg\min_{\mathbf{W}} \mathcal{L}(\mathbf{W})$ reference net

$\mathbf{r} = (r_1, \dots, r_K) \leftarrow \mathbf{0}$ ranks

$\mathbf{\Theta} = (\mathbf{\Theta}_1, \dots, \mathbf{\Theta}_K) \leftarrow \mathbf{0}$ auxillary variables

**for** $\mu = \mu_1 < \mu_2 < \cdots < \mu_T$

  $\mathbf{W} \leftarrow \arg\min_{\mathbf{W}} \mathcal{L}(\mathbf{W}) + \frac{\mu}{2}\sum_{k=1}^{K}\|\mathbf{W}_k - \mathbf{\Theta}_k\|^2$ L step

  **for** $k = 1, \dots, K$ C step

    $\mathbf{\Theta}_k, r_k \leftarrow \arg\min_{\mathbf{\Theta}_k, r_k} \frac{\mu}{2}\|\mathbf{\Theta}_k - \mathbf{W}_k\|^2 + \lambda\mathcal{R}_k(r_k)$

  **if** $\|\mathbf{W} - \mathbf{\Theta}\|$ is small enough **then** exit the loop

**return** $\mathbf{W}, \mathbf{\Theta}, \mathbf{r}$

### 3.1. Solutions of L and C steps

The L-step problem is a standard neural network training (learning) with added $\ell_2$ regularization. We solve it using SGD. The C-step problem can be interpreted as finding best low-rank approximation (compression) to the matrix $\mathbf{W}_k$ in the presence of a cost function over the ranks. The solution of this problem was given in [9] and requires computing a singular value decomposition of $\mathbf{W}_k$ followed by enumeration. For completeness, below we give its full solution.

Assuming $\mathbf{W}_k$ is $a_k \times b_k$ matrix (w.l.o.g. $a_k \geq b_k$) let $\mathbf{W}_k = \mathbf{U}_k\mathbf{S}_k\mathbf{V}_k^T$ be the SVD of $\mathbf{W}_k$, where $\mathbf{U}_k$ of $a_k \times b_k$ and $\mathbf{V}_k$ of $b_k \times b_k$ are orthogonal matrices, and $\mathbf{S}_k = \operatorname{diag}(s_1, \dots, s_{b_k})$ with $s_1 \geq \cdots \geq s_{b_k} \geq 0$ is a matrix of sorted singular values. Then C-step problem is equivalent to:

$$\min_{r_k} \lambda\mathcal{R}_k(r_k) + \frac{\mu}{2}\sum_{i=r_k+1}^{R_k} s_{ki}^2 \quad \text{s.t.} \quad r_k \in \{0, 1, \dots, R_k\} \quad (5)$$

which can be solved by trying all $R_k + 1$ values of $r_k$. Therefore, the C step is solved exactly by 1) computing the full SVD of $\mathbf{W}_k$ 2) finding the optimal $r_k$ minimizing eq. (5) using enumeration, and 3) forming $\mathbf{\Theta}_k = \mathbf{U}_k(:, 1:r_k)\mathbf{S}_k(1:r_k, 1:r_k)\mathbf{V}(:, 1:r_k)^T$ based on the top $r_k$ singular values and corresponding singular vectors.

Overall, the LC algorithm alternates between L and C steps while driving $\mu \to \infty$. The L step finds (locally) optimal weights $\mathbf{W}$ that are close to the current selection of the low-rank matrices ($\mathbf{\Theta}$) with the rank configuration $\mathbf{r}$. The C step finds the best configuration of the ranks and the optimal numeric values of the low-rank matrices that approximate the current weights $\mathbf{W}$. Once $\mu$ is sufficiently large, neural network weights $\mathbf{W}$ and its compressed form $\mathbf{\Theta}$ will reach equality by satisfying $\mathbf{W}_k = \mathbf{\Theta}_k$.

## 4. EXPERIMENTS

We demonstrate the effectiveness of our approach by compressing batch normalized versions of AlexNet (trained on ImageNet) and VGG16 (trained on CIFAR10) networks. We initialize the algorithm from the reasonably well-trained reference models. Our reference AlexNet has 62.3M parameters, 1140 MFLOPs, and the top-1/top-5 validation error of 40.43%/17.55%. We did not use group convolutions in our reference version of AlexNet, therefore it has a slightly larger FLOPs count of 1140 MFLOPs, whereas standard (Caffe-version) has 727 MFLOPs [20, 21]. The reference CIFAR10

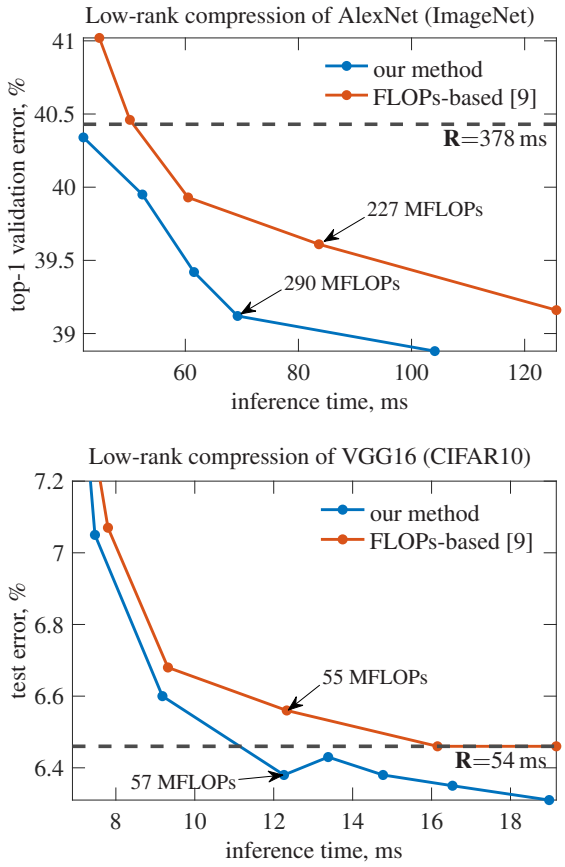


**Fig. 2**. Inference speed vs. error plot for our (blue) compressed AlexNet (top) and VGG16 models (bottom); for both networks, we additionally compare to the FLOPs based low-rank compression of [9] (given with red). The test errors and inference times of the reference models are indicated by horizontal dashed line labeled as **R**.

VGG16 model has 15.3M parameters, 313.73 MFLOPs, and a test error of 6.46%.

As our target device we use the ARM Cortex-A57 CPU of the NVIDIA's Jetson Nano; full specifications are available in Table 1. Single image inference times on this CPU (using two threads) are 378.45 ms for AlexNet and 53.99 ms for VGG16. For each network we build the runtime model as specified in sec. 2. The weights of the convolutional layers are reshaped using the scheme 2 (sec. 1.2).

We run our LC algorithm for $T$ steps with an exponential schedule on $\mu$ with $\mu_t = a \times b^t$ at the $t$th step: for AlexNet we set $T = 30, a = 10^{-4}$ and $b = 1.2$; for VGG16: $T = 60, a = 10^{-5}$ and $b = 1.2$. Each L step was trained with stochastic gradient descent using the following settings: for AlexNet we used the learning rate of 0.001 (decayed by 0.9 after each epoch) with the momentum of 0.9 on minibatches of 256 images; for VGG16 we used the learning rate of $7 \times 10^{-4}$ (decayed by 0.99 after each epoch) with the momentum of 0.9 on minibatches of size 128 images. Each L step used a predetermined number of epochs (i.e., full passes over the dataset): 5 epochs for AlexNet and 20 epochs for VGG16. Once the algorithm finished, we finetuned the decomposed weights for a small number of epochs (AlexNet: 30 epochs, VGG16: 100 epochs). *Overall, the entire compression pipelines requires not more than* $2.5\times$ *the time required to train the reference networks in the first place*.

| | Model | MFLOPs | Infr. time | top-1 err | top-5 err |
|---|---|---|---|---|---|
| | reference (**R**) | 1140 | 378.5 ms | 40.43% | 17.55% |
| | Caffe-AlexNet [20, 21] | 727 | 328.7 ms | 42.90% | 19.80% |
| ours | $\lambda = 5.0 \times 10^{-3}$ | 421 | 104.1 ms | **38.88%** | **16.83%** |
| ours | $\lambda = 1.0 \times 10^{-2}$ | 290 | 69.2 ms | 39.12% | 17.03% |
| ours | $\lambda = 2.0 \times 10^{-2}$ | 186 | **42.0 ms** | 40.34% | 17.64% |
| | low-rank AlexNet [9] | 227 | 83.6 ms | 39.61% | 17.40% |
| | low-rank AlexNet [9] | **166** | 50.2 ms | 40.46% | 17.71% |
| | ENC-AlexNet [22] | 272 | 93.3 ms | 43.40% | 19.93% |
| | SqueezeNet 1.1 [23] | 352 | 63.8 ms | 42.90% | 19.70% |

**Table 2**. Details of selected low-rank AlexNets obtained with our algorithm, and comparison to some of the available low-rank AlexNets in the literature. We additionally include a comparison to the SqueezeNet [23] that has similar accuracy to the AlexNet but was manually designed to be small and fast. All reported runtime measurements are performed on our testbed: CPU of Jetson Nano.

To explore the error-compression tradeoff, we run our compression with various values of $\lambda$. We report our results in Fig. 2 as inference time vs. validation error over the range of the obtained networks. To put our result in perspective, we additionally plot the results of FLOPs guided low-rank compression of [9].

For both networks, we achieve lower test error for the same inference speed when compared to the results of FLOPs guided low-rank compression. Notably, with $\lambda = 1 \times 10^{-2}$ we obtain a low-rank AlexNet model that has the validation error of 39.12% and requires only 69.2 ms to complete its inference pass on our target device. This results in a speed-up of $5.47\times$ wrt reference model and $4.74\times$ wrt Caffe-AlexNet while having 1.5% improvement in the test error wrt reference. The FLOPs count of this particular network is not that small: it requires 290 MFLOPs of compute; and compressed AlexNets with fewer FLOPs are available in the literature (see Table 2). However, the architecture of our compressed network was directly optimized to run as fast as possible on the target device, therefore, even with 290 MFLOPs it runs faster than the 227 MFLOPs low-rank AlexNet of [9] and the 272 MFLOPs low-rank AlexNet of [22], while additionally having a better accuracy.

We see a similar pattern for VGG16 results. For instance, with $\lambda = 1.4 \times 10^{-2}$ our algorithm achieves a network that requires only 12.26 ms of CPU time ($4.40\times$ faster) while having a test error of 6.38%. This network has a total of 57.3 MFLOPs, yet, it runs faster than 55.3 MFLOPs low-rank VGG16 of [9]: 12.26 ms vs. 12.33 ms.

## 5. CONCLUSION

We have presented a method that allows targeting on-device inference time when compressing the neural networks with the low-rank decomposition. The technique consists of two parts. First, it relies on a simple yet accurate device runtime model that can be automatically obtained with a few measurements using the target device. Second, it formulates a well-defined optimization problem and optimizes it using the Learning-Compression algorithm. We experimentally validate that targeting the on-device inference time yields faster networks than the methods that optimize the total count of floating-point operations (FLOPs). We will release all necessary code and scripts to replicate our experiments as part of the Learning-Compression toolbox [24] at `https://github.com/UCMerced-ML/LC-model-compression`.

## 6. REFERENCES


[1] 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 *Proc. 43rd Int. Symposium on Computer Architecture (ISCA 2016)*, Seoul, Korea, June 18–22 2016, pp. 243–254.

[2] Max Jaderberg, Andrea Vedaldi, and Andrew Zisserman, "Speeding up convolutional neural networks with low rank expansions," in *Proc. of the 25th British Machine Vision Conference (BMVC 2014)*, Nottingham, UK, Sept. 1–5 2014.

[3] 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 (NIPS)*, 2014, vol. 27, pp. 1269–1277, MIT Press, Cambridge, MA.

[4] Cheng Tai, Tong Xiao, Yi Zhang, Xiaogang Wang, and Weinan E, "Convolutional neural networks with low-rank regularization," in *Proc. of the 4th Int. Conf. Learning Representations (ICLR 2016)*, San Juan, Puerto Rico, May 2–4 2016.

[5] Wei Wen, Cong Xu, Chunpeng Wu, Yandan Wang, Yiran Chen, and Hai Li, "Coordinating filters for faster deep neural networks," in *Proc. 16th Int. Conf. Computer Vision (ICCV'17)*, Venice, Italy, Dec. 11–18 2017.

[6] Xiangyu Zhang, Jianhua Zou, Kaiming He, and Jian Sun, "Accelerating very deep convolutional networks for classification and detection," *IEEE Trans. Pattern Analysis and Machine Intelligence*, vol. 38, no. 10, pp. 1943–1955, Oct. 2016.

[7] Yong-Deok Kim, Eunhyeok Park, Sungjoo Yoo, Taelim Choi, Lu Yang, and Dongjun Shin, "Compression of deep convolutional neural networks for fast and low power mobile applications," in *Proc. of the 4th Int. Conf. Learning Representations (ICLR 2016)*, San Juan, Puerto Rico, May 2–4 2016.

[8] Jose M. Alvarez and Mathieu Salzmann, "Compression-aware training of deep networks," in *Advances in Neural Information Processing Systems (NIPS)*, 2017, vol. 30, pp. 856–867, MIT Press, Cambridge, MA.

[9] Yerlan Idelbayev and Miguel Á. Carreira-Perpiñán, "Low-rank compression of neural nets: Learning the rank of each layer," in *Proc. of the 2020 IEEE Computer Society Conf. Computer Vision and Pattern Recognition (CVPR'20)*, Seattle, WA, June 14–19 2020, pp. 8046–8056.

[10] Chong Li and C. J. Richard Shi, "Constrained optimization based low-rank approximation of deep neural networks," in *Proc. 15th European Conf. Computer Vision (ECCV'18)*, Munich, Germany, Sept. 8–14 2018, pp. 746–761.

[11] Yuhui Xu, Yuxi Li, Shuai Zhang, Wei Wen, Botao Wang, Yingyong Qi, Yiran Chen, Weiyao Lin, and Hongkai Xiong, "TRP: Trained rank pruning for efficient deep neural networks," in *Proc. of the 29th Int. Joint Conf. Artificial Intelligence (IJCAI'20)*, Yokohama, Japan, Jan. 21–15 2020, pp. 977–983.

[12] Vadim Lebedev, Yaroslav Ganin, Maksim Rakhuba, Ivan Oseledets, and Victor Lempitsky, "Speeding-up convolutional neural networks using fine-tuned CP-decomposition," in *Proc. of the 3rd Int. Conf. Learning Representations (ICLR 2015)*, San Diego, CA, May 7–9 2015.

[13] Alexander Novikov, Dmitrii Podoprikhin, Anton Osokin, and Dmitry P. Vetrov, "Tensorizing neural networks," in *Advances in Neural Information Processing Systems (NIPS)*, 2015, vol. 28, pp. 442–450, MIT Press, Cambridge, MA.

[14] Yerlan Idelbayev and Miguel Á. Carreira-Perpiñán, "Neural network compression via additive combination of reshaped, low-rank matrices," in *Proc. Data Compression Conference (DCC 2021)*, Mar. 23–26 2021, pp. 243–252.

[15] Yerlan Idelbayev and Miguel Á. Carreira-Perpiñán, "Optimal selection of matrix shape and decomposition scheme for neural network compression," in *Proc. of the IEEE Int. Conf. Acoustics, Speech and Sig. Proc. (ICASSP'21)*, Toronto, Canada, June 6–11 2021.

[16] Misha Denil, Babak Shakibi, Laurent Dinh, Marc'Aurelio Ranzato, and Nando de Freitas, "Predicting parameters in deep learning," in *Advances in Neural Information Processing Systems (NIPS)*, 2013, vol. 26, pp. 2148–2156, MIT Press, Cambridge, MA.

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

[18] Miguel Á. Carreira-Perpiñán and Yerlan Idelbayev, ""Learning-compression" algorithms for neural net pruning," in *Proc. of the 2018 IEEE Computer Society Conf. Computer Vision and Pattern Recognition (CVPR'18)*, Salt Lake City, UT, June 18–22 2018, pp. 8532–8541.

[19] Jorge Nocedal and Stephen J. Wright, *Numerical Optimization*, Springer Series in Operations Research and Financial Engineering. Springer-Verlag, New York, second edition, 2006.

[20] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell, "Caffe: Convolutional architecture for fast feature embedding," arXiv:1408.5093, June 20 2014.

[21] Alex Krizhevsky, Ilya Sutskever, and Geoff Hinton, "ImageNet classification with deep convolutional neural networks," in *Advances in Neural Information Processing Systems (NIPS)*, 2012, vol. 25, pp. 1106–1114, MIT Press, Cambridge, MA.

[22] Hyeji Kim, Muhammad Umar Karim Khan, and Chong-Min Kyung, "Efficient neural network compression," in *Proc. of the 2019 IEEE Computer Society Conf. Computer Vision and Pattern Recognition (CVPR'19)*, Long Beach, CA, June 16–20 2019, pp. 12569–12577.

[23] Forrest N. Iandola, Song Han, Matthew W. Moskewicz, Khalid Ashraf, William J. Dally, and Kurt Keutzer, "SqueezeNet: AlexNet-level accuracy with 50$times$ fewer parameters and <0.5MB model size," arXiv:1602.07360, Nov. 4 2016.

[24] Yerlan Idelbayev and Miguel Á. Carreira-Perpiñán, "A flexible, extensible software framework for model compression based on the LC algorithm," arXiv:2005.07786, May 15 2020.