ALF: Autoencoder-based Low-rank Filter-sharing for Efficient Convolutional Neural Networks

Alexander Frickenstein[1*], Manoj-Rohit Vemparala[1*], Nael Fasfous[2*], Laura Hauenschild[2*],
Naveen-Shankar Nagaraja[1], Christian Unger[1], Walter Stechele[2]
[1]*Autonomous Driving, BMW Group, Munich, Germany*
[2]*Department of Electrical and Computer Engineering, Technical University of Munich, Munich, Germany*
[1]{< firstname >.< lastname >}@bmw.de, [2]{< firstname >.< lastname >}@tum.de

***Abstract*—Closing the gap between the hardware requirements of state-of-the-art convolutional neural networks and the limited resources constraining embedded applications is the next big challenge in deep learning research. The computational complexity and memory footprint of such neural networks are typically daunting for deployment in resource constrained environments. Model compression techniques, such as pruning, are emphasized among other optimization methods for solving this problem. Most existing techniques require domain expertise or result in irregular sparse representations, which increase the burden of deploying deep learning applications on embedded hardware accelerators. In this paper, we propose the autoencoder-based low-rank filter-sharing technique (ALF). When applied to various networks, ALF is compared to state-of-the-art pruning methods, demonstrating its efficient compression capabilities on theoretical metrics as well as on an accurate, deterministic hardware-model. In our experiments, ALF showed a reduction of 70% in network parameters, 61% in operations and 41% in execution time, with minimal loss in accuracy.**

I. Introduction

In recent years, deep learning solutions have gained popularity in several embedded applications ranging from robotics to autonomous driving [1]. This holds especially for computer vision based algorithms [2]. These algorithms are typically demanding in terms of their computational complexity and their memory footprint. Combining these two aspects emphasizes the importance of constraining neural networks in terms of model size and computations for efficient deployment on embedded hardware. The main goal of model compression techniques lies in reducing redundancy, while having the desired optimization target in mind.

Hand-crafted heuristics are applied in the field of model compression, however, they severely reduce the search space and can result in a poor choice of design parameters [3]. Particularly for convolutional neural networks (CNNs) with numerous layers, such as ResNet1K [4], hand-crafted optimization is prone to result in a sub-optimal solution. In contrast, a learning-based policy would, for example, leverage reinforcement learning (RL) to automatically explore the CNN's design space [5]. By evaluating a cost function, the RL-agent learns to distinguish between good and bad decisions, thereby converging to an effective compression strategy for a given CNN. However, the effort of designing a robust cost function and the time spent for model exploration render learning-based compression policies as a complex method, requiring domain expertise.

In this work, we propose the autoencoder-based low-rank filter-sharing technique (ALF) which generates a dense, compressed CNN during task-specific training. ALF uses the inherent properties of sparse autoencoders to compress data in an unsupervised manner. By introducing an information bottleneck, ALF-blocks are used to extract the most salient features of a convolutional layer during training, in order to dynamically reduce the dimensionality of the layer's tensors. To the best of our knowledge, ALF is the first method where autoencoders are used to prune CNNs. After optimization, the resulting model consists of fewer filters in each layer, and can be efficiently deployed on any embedded hardware due to its structural consistency. The contributions of this work are summarized as follows:

- Approximation of weight filters of convolutional layers using ALF-blocks, consisting of sparse autoencoders.
- A two player training scheme allows the model to learn the desired task while slimming the neural architecture.
- Comparative analysis of ALF against learning and rule-based compression methods w.r.t. known metrics, as well as layer-wise analysis on hardware model estimates.

II. Related Work

Efforts from both industry and academia have focused on reducing the redundancy, emerging from training deeper and wider network architectures, with the aim of mitigating the challenges of their deployment on edge devices [6]. Compression techniques such as quantization, low-rank decomposition and pruning can potentially make CNNs more efficient for the deployment on embedded hardware. Quantization aims to reduce the representation redundancy of model parameters and arithmetic [7]–[9]. Quantization and binarization are orthogonal to this work and can be applied in conjunction with the proposed ALF method. In the following sections, we classify works which use low-rank decomposition and pruning techniques into rule-based and learning-based compression.

A. Rule-based Compression

Rule-based compression techniques are classified as having static or pseudo-static rules, which are followed when

*Corresponding Authors

978-1-7281-1085-1/20/$31.00 ©2020 IEEE

compressing a given CNN. Low-rank decomposition or representation techniques have been used to reduce the number of parameters (`Params`) by creating separable filters across the spatial dimension or reducing cross-channel redundancy in CNNs [10], [11]. Hand-crafted pruning can be implemented based on heuristics to compute the saliency of a neuron. Han et al. [3] determined the saliency of weights based on their magnitude, exposing the superfluous nature of state-of-the-art neural networks. Pruning individual weights, referred to as irregular pruning, leads to inefficient memory accesses, making it impractical for general purpose computing platforms. Regularity in pruning becomes an important criterion towards accelerator-aware optimization. Frickenstein et al. [12] propose structured, kernel-wise magnitude pruning along with a scalable, sparse algorithm. He et al. [13] prunes redundant filters using a geometric mean heuristic. Although the filter pruning scheme is useful w.r.t. hardware implementations, it is challenging to remove filters as they directly impact the input channels of the subsequent layer. Rule-based compression techniques overly generalize the problem at hand. Different CNNs vary in complexity, structure and target task, making it hard to set such *one-size-fits-all* rules when considering the different compression criteria.

B. Learning-based Compression

Recent works such as [5] and [14] have demonstrated that it is difficult to formalize a rule to prune networks. Instead, they expose the pruning process as an optimization problem to be solved through an RL-agent. Through this process, the RL-agent learns the criteria for pruning, based on a given cost function.

Huang et al. [5] represent a CNN as an environment for an RL-agent. An accuracy term and an efficiency term are combined to formulate a non-differentiable policy to train the agent. The target is to maximize the two contrary objectives. Balancing the terms can eventually vary for different models and tasks. An agent needs to be trained individually for each layer. Moreover, layers with multiple channels may slow down the convergence of the agent, rendering the model exploration as a slow and greedy process. In the work proposed by He et al. [14], an RL-agent prunes channels without fine-tuning at intermediate stages. This cuts down the time needed for exploration. Layer characteristics such as size, stride and operations (`OPs`), serve the agent as input. These techniques require carefully crafted cost functions for the optimization agent. The formulation of the cost functions is non-trivial, requiring expert knowledge and some iterations of trial-and-error. Furthermore, deciding what the agent considers as the environment can present another variable to the process, making it challenging to test many configurations of the problem. As mentioned earlier, each neural network and target task combination present a new and unique problem, for which this process needs to be repeated.

More recently, Neural Architectural Search (NAS) techniques have been successful in optimizing CNN models at design-time. Combined with Hardware-in-the-Loop (HIL) testing, a much needed synergy between efficient CNN design and the target hardware platform is achieved. Tan et al. [15] propose MNAS-Net, which performs NAS using an RL-agent for mobile devices. Cai et al. [16] propose ProxylessNAS, which derives specialized, hardware-specific CNN architectures from an over-parameterized model.

Differently, Guo et al. [17] dynamically prune the CNN by learnable parameters, which have the ability to recover when required. Their scheme incorporates pruning in the training flow, resulting in irregular sparsity. Zhang et al. [18] incorporate a cardinality constraint into the training objective to obtain different pruning regularity. Bagherinezhad et al. [19] propose Lookup-CNN (LCNN), where the model learns a dictionary of shared filters at training time. During inference, the input is then convolved with the complete dictionary profiting from lower computational complexity. As the same accounts for filter-sharing, this weight-sharing approach is arguably closest to the technique developed in this work. However, the methodology itself and the training procedure are still fundamentally different from one another.

TABLE I: Classification of model compression methods.

Method\Advantage	No Pre-trained Model	Learning Policy	No Extensive Exploration
Rule-based Compression:			
Low-Rank Dec. [10], [11]	✗	✗	✗
Prune (Handcrafted) [3], [12], [13]	✗	✗	✗
Learning-based Compression:			
Prune (RL-Agent) [5], [14]	✗	✓	✗
NAS [15], [16]	✓	✓	✗
Prune (Automatic) [17]–[19]	✓	✓	✓
ALF [Ours]	✓	✓	✓

III. AUTOENCODER-BASED LOW-RANK FILTER-SHARING

The goal of the proposed filter-sharing technique is to replace the standard convolution with a more efficient alternative, namely the autoencoder-based low-rank filter-sharing (ALF)-block. An example of the ALF-block is shown in Fig. 1.

Without loss of generality, $A^{l-1} \in \mathbb{R}^{H_i \times W_i \times C_i}$ is considered as an input feature map to a convolutional layer $l \in [1, L]$ of an L-layer CNN, where H_i and W_i indicate the height and width of the input, and C_i is the number of input channels. The weights $W^l \in \mathbb{R}^{K \times K \times C_i \times C_o}$ are the trainable parameters of the layer l, where K and C_o are the kernel dimensions and the number of output channels respectively.

In detail, the task is to approximate the filter bank W in a convolutional layer during training by a low-rank version $W_{\text{code}} \in \mathbb{R}^{K \times K \times C_i \times C_{\text{code}}}$, where $C_{\text{code}} < C_o$. The low-rank version of the weights W_{code} is utilized later in the deployment stage for an embedded-friendly application.

In contrast to previous structured pruning approaches [12], [13], [19], this method does not intend to alter the structure of the model in a way which results in a changed dimensionality of the output feature maps $A^l \in \mathbb{R}^{H_o \times W_o \times C_o}$, where H_o and W_o indicate the height and width of the output. This is done by introducing an additional expansion layer [20]. The advantages are twofold. First, each layer can be trained individually without affecting the other layers. Second, it

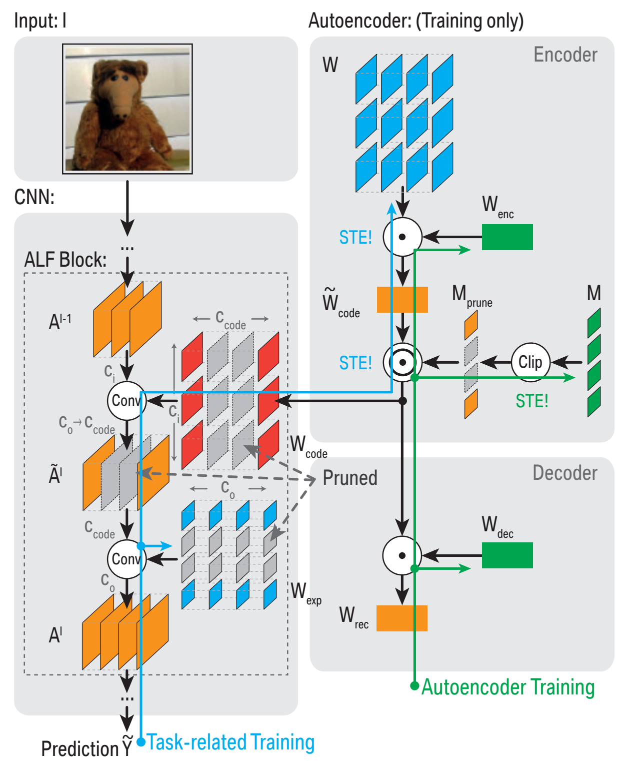


Fig. 1: ALF block in training mode showing the forward and backward path for the task-related and autoencoder training.

simplifies the end-to-end training and allows comparison of the learned features.

The expansion layer is comprised of point-wise convolutions with weights $W_{\text{exp}} \in \mathbb{R}^{1\times1\times C_{\text{code}}\times C_o}$, for mapping the intermediate feature maps after an ALF-block $\tilde{A}^l \in \mathbb{R}^{H_o\times W_o\times C_{\text{code}}}$, to the output feature map A^l, as expressed in Eq. 1.

$$A^l = \sigma(\tilde{A}^l * W_{\text{exp}}) = \sigma(\sigma_{\text{inter}}(A^{l-1} * W_{\text{code}}) * W_{\text{exp}}) \quad (1)$$

As the point-wise convolution introduces a certain overhead with regard to operations and weights, it is necessary to analyze the resource demands of the ALF-block compared to the standard convolution and ensure $C_{\text{code}} < C_{\text{code,max}}$, where $C_{\text{code,max}}$ denotes the number of filters which have to be removed to attain an efficiency improvement, see Eq. 2.

$$\frac{C_i C_o K^2}{C_{\text{code}}(C_i K^2 + C_o)} \rightarrow C_{\text{code,max}} = \lfloor \frac{C_i C_o K^2}{C_i K^2 + C_o} \rfloor \quad (2)$$

A. Autoencoder-based Low-rank Filter-sharing Block

As stated before, the autoencoder is required to identify correlations in the original weights W and to derive a compressed representation W_{code} from them. The autoencoder is only required in the training stage and is discarded in the deployment stage.

Referring back to Fig. 1, the autoencoder setup including the pruning mask $M_{\text{prune}} \in \mathbb{R}^{1\times1\times1\times C_o}$ is illustrated. According to the design of an autoencoder, Eq. 3 gives the complete expression for calculating the compressed weights W_{code}. The encoder performs a matrix multiplication between the input W and the encoder filters $W_{\text{enc}} \in \mathbb{R}^{K\times K\times C_o\times C_{\text{code}}}$. M_{prune} zeroizes elements of $\tilde{W}_{\text{code}}$ and σ_{ae} refers to a non-linear activation function, i.e. $\texttt{tanh}()$. Different configurations of σ_{ae}, σ_{inter} and initialization schemes are studied in Sec. IV-A.

$$W_{\text{code}} = \sigma_{\text{ae}}(\tilde{W}_{\text{code}} \odot M_{\text{prune}}) = \sigma_{\text{ae}}((W \cdot W_{\text{enc}}) \odot M_{\text{prune}}) \quad (3)$$

Eq. 4 provides the corresponding formula for the reconstructed filters W_{rec} of the decoding stage. The symbol $\cdot$ stands for a matrix multiplication and $\odot$ for a Hadamard product respectively. The pruning mask M_{prune} acts as a gate, allowing only the most salient filters to appear as non-zero values in W_{code}, in the same manner as sparse autoencoders. The decoder must, therefore, learn to compensate for the zeroized filters to recover a close approximate of the input filter bank.

$$W_{\text{rec}} = \sigma_{\text{ae}}(W_{\text{code}} \cdot W_{\text{dec}}) \quad (4)$$

In order to dynamically select the most salient filters, an additional trainable parameter, denoted mask $M \in \mathbb{R}^{1\times1\times1\times C_o}$, is introduced with its individual elements $m_i \in M$. By exploiting the sparsity-inducing property of L1 regularization, individual values in the mask M are driven towards zero during training. Since the optimizer usually reaches values close to zero, but not exactly zero, clipping is performed to zero out values that are below a certain threshold t. Further, the clipping function $M_{\text{prune}} = \texttt{Clip}(M, t) = \mathbb{I}_{\{|m_i| > t\}} m_i$ allows the model to recover a channel when required.

B. Training Procedure

For understanding the training procedure of ALF, it is important to fully-comprehend the training setup, including the two player game of the CNN and the ALF-blocks.

Task-related Training: The weights W are automatically compressed by the ALF-blocks. Allowing the weights W to remain trainable, instead of using fixed filters from a pre-trained model, is inspired by binary neural networks (BNNs) [8]. The motivation is that weights from a full-precision CNN might not be equally applicable when used in a BNN and, thus, require further training. Analogously, the weights from a pre-trained model might not be the best fit for the instances of W in the filter-sharing use-case. Therefore, training these variables is also part of the task optimizer's job. It's objective is the minimization of the loss function $\mathcal{L}_{\text{task}}$, which is the accumulation of the cross-entropy loss $\mathcal{L}_{\text{CE}}$, of the model's prediction $\tilde{Y}$ and the label Y of an input image I, and the weight decay scaling factor ν_{wd} multiplied with the L2 regularization loss $\mathcal{L}_{\text{reg}}$.

It is important to point out that no regularization is applied to the instances of either W or W_{code}. Even though each W_{code} contributes to a particular convolution operation with a considerable impact on the task loss $\mathcal{L}_{\text{task}}$, the task optimizer can influence this variable only indirectly by updating W. As neither of the variables W_{enc}, W_{dec}, or M of the appended autoencoder are trained based on the task loss, they introduce a sufficiently high amount of noise, which arguably makes any further form of regularization more harmful than helpful.

In fact, this noise introduced by autoencoder variables does affect the gradient computation for updating variable W. As this might hamper the training progress, Straight-Through-Estimator (STE) [8] is used as a substitute for the gradients of Hadamard product with the pruning mask, as well as for multiplication with the encoder filters. This ensures that the gradients for updating the input filters can propagate through the autoencoder without extraneous influence.

This measure is especially important in case of the Hadamard product, as a significant amount of weights in M_{prune} might be zero. When including this operation in the gradient computation, a correspondingly large proportion would be zeroized as a result, impeding the information flow in the backward pass. Nevertheless, this problem can be resolved by using the STE. In Eq. 5, the gradients for the variables g_W for a particular ALF-block are derived.

$$\begin{aligned} g_W = \frac{\partial \mathcal{L}_{\text{task}}}{\partial W} &= \frac{\partial \mathcal{L}_{\text{task}}}{\partial \tilde{A}} \cdot \frac{\partial \tilde{A}}{\partial W_{\text{code}}} \cdot \frac{\partial W_{\text{code}}}{\partial \tilde{W}_{\text{code}}} \cdot \frac{\partial \tilde{W}_{\text{code}}}{\partial W_{\text{orig}}} \\ &\overset{\text{STE}}{=} \frac{\partial \mathcal{L}_{\text{task}}}{\partial \tilde{A}} \cdot \frac{\partial \tilde{A}}{\partial W_{\text{code}}} \end{aligned} \tag{5}$$

Autoencoder Training: Each autoencoder is trained individually by a dedicated SGD optimizer, referred to as an autoencoder optimizer. The optimization objective for such an optimizer lies in minimizing the loss function $\mathcal{L}_{\text{ae}} = \mathcal{L}_{\text{rec}} + \nu_{\text{prune}} \cdot \mathcal{L}_{\text{prune}}$. The reconstruction loss is computed using the MSE metric and can be expressed by $\mathcal{L}_{\text{rec}} = \texttt{MSE}(W, W_{\text{rec}})$. In the field of knowledge distillation [21], similar terms are studied. The decoder must learn to compensate for the zeroized filters to recover a close approximate of the input filter. If a lot of values in M_{prune} are zero, a large percentage of filters are pruned. To mitigate this problem, the mask regularization function $\mathcal{L}_{\text{prune}} = 1/C_o \sum |m|$ is multiplied with a scaling factor $\nu_{\text{prune}} = \texttt{max}(0, 1 - e^{(m*(\theta - \text{pr}_{\text{max}}))})$, which decays with increasing zero fraction in the mask and slows down the pruning rate towards the end of the training. In detail, ν_{prune} adopts the pruning sensitivity [3] of convolutional layers, where $m \in [1, 10]$ is the slope of the sensitivity, $pr_{\text{max}} \in [0, 1]$ is the maximum pruning rate and $\theta = C_{\text{code,zero}}/C_{\text{code}}$ is the zero fraction, with $C_{\text{code,zero}}$ referring to the number of zero filters in W_{code}, As a consequence, the regularization effect decreases and fewer filters are zeroized in the code eventually. In other words, the task of $\mathcal{L}_{\text{rec}}$ is to imitate W by W_{rec} by learning W_{dec}, W_{enc} and M while $\mathcal{L}_{\text{prune}}$ steadily tries to prune further channels.

The autoencoder optimizer updates the variables W_{enc}, W_{dec} and M, based on the loss derived from a forward pass through the autoencoder network. Since M_{prune} and W_{code} are intermediate feature maps of the autoencoder, they are not updated by the optimizer. While the gradient calculation for the encoder and decoder weights is straight forward, the gradients for updating the mask M require special handling. The main difficulty lies in the clipping function which is non-differentiable at the data points t and $-t$. As previously mentioned, for such cases the STE can be used to approximate the gradients. The mathematical derivation for the gradients, corresponding to the variables updated by the autoencoder optimizer, are given in Eq. 6.

$$\begin{aligned} g_M = \frac{\partial \mathcal{L}_{\text{ae}}}{\partial M} &= \frac{\partial \mathcal{L}_{\text{ae}}}{\partial W_{\text{rec}}} \cdot \frac{\partial W_{\text{rec}}}{\partial W_{\text{code}}} \cdot \frac{\partial W_{\text{code}}}{\partial M_{\text{prune}}} \cdot \frac{\partial M_{\text{prune}}}{\partial M} \\ &\overset{\text{STE}}{=} \frac{\partial \mathcal{L}_{ae}}{\partial W_{\text{rec}}} \cdot \frac{\partial W_{\text{rec}}}{\partial W_{\text{code}}} \cdot \frac{\partial W_{\text{code}}}{\partial M_{\text{prune}}} \end{aligned} \tag{6}$$

C. Deployment

The utility of the autoencoders is limited to the training process, since the weights are fixed in the deployment stage. The actual number of filters is still the same as at the beginning of the training ($C_{code} = C_o$). However, the code comprises of a certain amount of filters containing only zero-valued weights which are removed. For the succeeding expansion layer, fewer input channels imply that the associated channels in W_{exp} are not used anymore and can be removed as well (Fig.1 pruned channels (gray)). After post-processing, the model is densely compressed and is ready for efficient deployment.

IV. Experimental Results

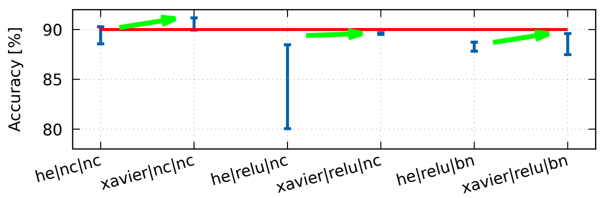
We evaluate the proposed ALF technique on CIFAR-10 [22] and ImageNet [23] datasets. The 50k train and 10k test images of CIFAR-10 are used to respectively train and evaluate ALF. The images have a resolution of 32×32 pixel. ImageNet consists of ~ 1.28 Mio. train and 50K validation images with a resolution of (256×256) px. If not otherwise mentioned, all hyper-parameters specifying the task-related training were adopted from the CNN's base implementation. For ALF, the hyperparameters $m = 8$ and $pr = 0.85$ are set.

A. Configuration Space Exploration

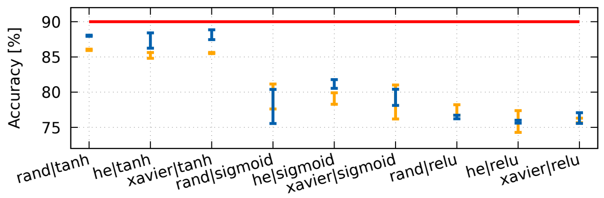
Crucial design decisions are investigated in this section. The aforementioned novel training setup includes a number of new parameters, namely the weight initialization scheme for $W_{\text{exp}}, W_{\text{enc}}$ and W_{dec}, consecutive activation functions and batch normalization layers. For that purpose, Plain-20 [4] is trained on CIFAR-10. Experiments are repeated at least twice to provide results with decent validity (bar-stretching).

Setup 1: The effect of additional expansion layers is studied in Fig. 2a taking the initialization (He [24] and Xavier [25]), extra non-linear activations σ_{inter} (i.e. ReLU) and batch normalization (BN) into account. The results suggest that expansion layers can lead to a tangible increase in accuracy. In general, the Xavier initialization yields slightly better results than the He initialization and is chosen for the expansion layer of the ALF blocks. In addition, the incorporation of the BN_{inter} layer seems to not have perceivable advantages. The influence of σ_{inter} is confirmed in the next experiment.

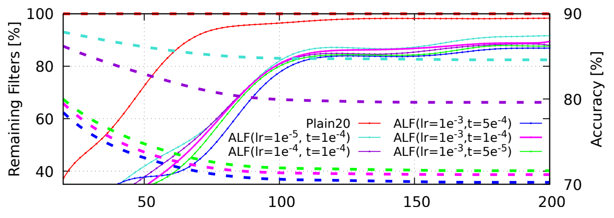
Setup 2: In this setup, the ALF block's pruning mask M is not applied, thus, no filters are pruned. This is applicable to select a weight initialization scheme for W_{enc} and W_{dec} (referred as $W_{\text{ae,init}}$) and an activation function σ_{ae}. In case no activation function σ_{inter} (blue) is applied to $\tilde{A}^l$, the accuracy is higher than with ReLU layers. Based on the results the Xavier initialization is selected for $W_{ae,init}$. Moreover, $\texttt{tanh}()$ outperforms other non-linear activation functions σ_{ae}. As the



(a) Configuration $[W_{\text{exp,init}}|\sigma_{\text{inter}}|\text{BN}_{\text{inter}}]$.



(b) Config. $[W_{\text{ae,init}}|\sigma_{\text{ae}}]$, σ_{inter}=none), σ_{inter}=ReLU.



(c) Training epochs. ALF trained on different lr and t.

Fig. 2: Design space exploration for Plain-20 on CIFAR-10.

pruning mask M is not active, the regularization property of the autoencoder is absent causing a noticeable accuracy drop.

Setup 3: Five variants of ALF, resulting in different sparsity rates, are explored in Fig 2c and compared to the uncompressed Plain-20 (90.5% accuracy). The first three variants differ in terms of the threshold $t \in \{5\cdot10^{-5}, 1\cdot10^{-4}, 5\cdot10^{-4}\}$ while the learning rate $lr_{\text{ae}} = 1\cdot10^{-3}$. We observe that the pruning gets more aggressive when the threshold t is increased. The number of non-zero filters remaining are 40.17%, 38.6% and 35.71% respectively (see green, pink and blue curves). We select the threshold $t = 1e^{-4}$ as a trade-off choice between sparsity and accuracy. The behaviour of ALF is also accessed by changing the learning rate of the autoencoder. The learning rate $lr_{\text{ae}} \in \{1\cdot10^{-5}, 1\cdot10^{-4}\}$ is explored in the consecutive variants (see turquoise and purple curves). The number of remaining non-zero filters increases as there are less updates to the sparsity mask M. In case of $lr_{\text{ae}} = 1\cdot10^{-4}$ (purple), the resulting network has less number of non-zero filters with a significant accuracy drop. However, considering the trade-off between accuracy and pruning rate, we choose the learning rate $lr_{\text{ae}} = 1\cdot10^{-3}$ for the autoencoder.

B. Comparison with State-of-the-Art

CIFAR-10: Tab. II compares ALF against its full precision counterpart (ResNet-20) and state-of-the-art pruned models [13], [14]. The ALF model consists of the least number of operations and training parameters compared to other pruning works, with an accuracy drop of 1.9% compared to its full precision model.

Hardware-Model Estimates: Improvements in the number of `OPs` and `Params` of a CNN do not always indicate tangible

TABLE II: Pruned CNNs on CIFAR-10, for Conv layers only.

Method	**Policy**	`Params`	`OPs`$[10^6]$	**Acc[%]**
Plain-20 [4]	—	0.27M	81.1	90.5
ResNet-20 [4]	—	0.27M	81.1	91.3
AMC [14]	RL-Agent	0.12M (-55%)	39.4 (-51%)	90.2
FPGM [13]	Handcrafted	—	36.2 (-54%)	**90.6**
ALF (ours) ($t = 10^{-4}$)	Automatic	**0.07M (-70%)**	**31.5 (-61%)**	89.4

advantages on real hardware execution. Here, we analyze the real advantage of ALF by investigating its implications on hardware. The deterministic execution of CNNs on target hardware platforms eases the creation of hardware models with highly accurate estimates of normalized latency and energy. We use an accurate modeling framework, Timeloop [26], to replicate the execution and scheduling scheme of the Eyeriss [27] accelerator. The Eyeriss model used in the experiments consists of a 16×16 array of processing elements (PEs). Each PE contains three separate register files (RFs), one for each datatype (inputs, weights and outputs). The word-width of all datatypes is fixed to 16-bits. The combined RFs in a single PE add up to 220 words. The global buffer has a total size of 128KB and can hold output and input datatypes. Weights bypass the global buffer and are directly fed to the weight RFs of the PEs. The mapper's search algorithm follows an exhaustive method with a timeout of 100K iterations and victory condition of 1K iterations per thread. We simulate the compressed configurations achieved by the ALF-block on the Plain-20 and ResNet-20 models. The batch size for all experiments is set to 16 and the energy values are normalized against the energy cost of a single register file read [27]. The latency is normalized to the bandwidth of a register (2 bytes/cycle).

Fig. 3 shows the advantages of the ALF technique. The results show a trend of high contributions from the RF memory, particularly in the deeper layers of each CNN. This is due to the efficient dataflow of the accelerator, which increases the data reuse at the lowest-level memory, reducing the number of redundant data accesses between the higher memory levels.

By analyzing the energy breakdown results of both ALF models, an increase in DRAM energy is observed. This is due to additional off-chip data movement introduced by the expansion layer. This is highlighted in the initial layers, as their inputs are larger. Practically, such codependent layers can be fused with some advanced scheduling techniques [28], eliminating this overhead. Nevertheless, the strong improvements in the deeper layers offset this degradation, leading to an overall 29% lower energy consumption and 41% latency reduction over the vanilla model.

The latency line-plots reveal an anomaly for layer `conv312` on the ALF-Plain-20 compressed model. This compressed layer requires more energy and execution cycles than the vanilla Plain-20 execution. Taking a closer look, ALF-Plain-20 utilizes only 8% of the PEs for processing this layer. Although ALF is a structured pruning technique, the resulting non-sparse model could have reduced parallelism opportunities

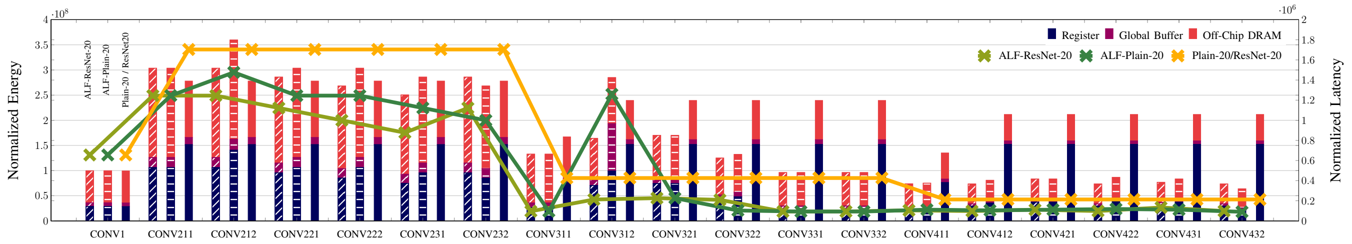


Fig. 3: Energy consumption breakdown and latency of vanilla and ALF-compressed ResNet/Plain-20 execution.

under the restrictions enforced by the row-stationary dataflow on the hardware. These results cannot be intuitively concluded by simply considering `OPs` and `Params` as metrics for a compression technique. This emphasizes the importance of hardware-aware validation of pruning advantages, as performed in this subsection.

TABLE III: Benchmarking results on ImageNet, comparing models and methods with regard to `MACs` and `Params`.

Method	**Policy**	`Params`	`OPs` $[10^6]$	**Acc [%]**
SqueezeNet [20]	—	**1.23M**	1722	57.2 %
GoogleNet [2]	—	6.80M	3004	66.8%
ResNet-18 [4]	—	11.83M	3743	**69.8 %**
Pruned ResNet-18:				
LCNN [19]	Automatic	–	**749**	62.2 %
FPGM [13]	Handcrafted	–	2178	67.8 %
AMC [14]	RL-Agent	8.9M	1874	67.7 %
ALF (ours)	Automatic ($t = 10^{-4}$)	4.24M	1239	64.3 %

ImageNet: Tab. III compares ALF against other pruned models, i.e. LCNN [19] and FPGM [13]. For comparison, GoogleNet [2], SqueezeNet [20] and ResNet-18 [4] are also detailed in the table. Compared to SqueezeNet, GoogleNet and ResNet, ALF requires $\times 1.4$, $\times 2.4$, $\times 3.0$ less `OPs`, respectively. Moreover, ALF outperforms GoogleNet and ResNet w.r.t. the parameters. Compared to other pruned models, ALF lies on the pareto-front for the number of parameters, the operations and the accuracy. On one hand, LCNN obtained a very competitive computational complexity (-40%), at the cost of an accuracy degradation of -2.1% compared to ALF. On the other hand, AMC and FPGM gain $+3.4\%$, $+3.5\%$ accuracy improvement, but has $+51\%$, $+76\%$, more `OPs` than ALF respectively.

V. Conclusion

Pruning is a promising compression technique, ranging from rule-based to handcrafted and learning-based methods. Most approaches require either a pre-trained model or an extensive model exploration, making the compression a time consuming process. In this paper, we propose the autoencoder-based low-rank filter-sharing technique (ALF) to dynamically prune a given CNN during task-specific training. ALF employs a sparse autoencoder to approximate the weight filters of the CNN for an embedded-friendly application. The novel method is applied to computer vision tasks, CIFAR-10 and ImageNet. Comparison to state-of-the-art methods is performed on well known theoretical metrics (`OPs` and `Params`), as well as an analysis on hardware-model estimates, i.e. normalized latency and energy.

References

[1] A. Frickenstein, M.-R. Vemparala, J. Mayr, *et al.*, "Binary DAD-Net: Binarized driveable area detection network for autonomous driving," in *ICRA*, 2020.

[2] C. Szegedy, Wei Liu, Yangqing Jia, *et al.*, "Going deeper with convolutions," in *CVPR*, June 2015.

[3] S. Han, J. Pool, J. Tran, *et al.*, "Learning both weights and connections for efficient neural networks," in *NeurIPS*, 2015.

[4] K. He, X. Zhang, S. Ren, *et al.*, "Deep residual learning for image recognition," in *CVPR*, 2016.

[5] Q. Huang, K. Zhou, S. You, *et al.*, "Learning to prune filters in convolutional neural networks," in *WACV*, 2018.

[6] A. Frickenstein, C. Unger, and W. Stechele, "Resource-Aware Optimization of DNNs for Embedded Applications," in *CRV*, 2019.

[7] S. Vogel, M. Liang, A. Guntoro, *et al.*, "Efficient hardware acceleration of CNNs using logarithmic data representation with arbitrary log-base," in *ICCAD*, 2018.

[8] I. Hubara, M. Courbariaux, D. Soudry, *et al.*, "Binarized Neural Networks," in *NeurIPS*, 2016.

[9] N. Fasfous, M.-R. Vemparala, A. Frickenstein, *et al.*, "Orthruspe: Runtime reconfigurable processing elements for binary neural networks," in *DATE*, 2020.

[10] X. Zhang, J. Zou, K. He, *et al.*, "Accelerating very deep convolutional networks for classification and detection," *TPAMI*, 2016.

[11] O. A. Malik and S. Becker, "Low-rank tucker decomposition of large tensors using tensorsketch," in *NeurIPS*, 2018.

[12] A. Frickenstein, M. R. Vemparala, C. Unger, *et al.*, "DSC: Dense-sparse convolution for vectorized inference of cnns," in *CVPR-W*, 2019.

[13] Y. He, P. Liu, Z. Wang, *et al.*, "Filter pruning via geometric median for deep convolutional neural networks acceleration," in *CVPR*, 2019.

[14] Y. He, J. Lin, Z. Liu, *et al.*, "AMC: Automl for model compression and acceleration on mobile devices," in *ECCV*, 2018.

[15] M. Tan, B. Chen, R. Pang, *et al.*, "MnasNet: Platform-aware neural architecture search for mobile," *arXiv:1807.11626*.

[16] H. Cai, L. Zhu, and S. Han, "ProxylessNAS: Direct neural architecture search on target task and hardware," in *ICLR*, 2019.

[17] Y. Guo, A. Yao, and Y. Chen, "Dynamic network surgery for efficient DNNs," in *NeurIPS*, 2016.

[18] T. Zhang, S. Ye, K. Zhang, *et al.*, "Structadmm: A systematic, high-efficiency framework of structured weight pruning for dnns," 2018.

[19] H. Bagherinezhad, M. Rastegari, and A. Farhadi, "LCNN: Lookup-based convolutional neural network," in *CVPR*, 2017.

[20] F. N. Iandola, S. Han, M. W. Moskewicz, *et al.*, "SqueezeNet: AlexNet-level accuracy with 50$\times$ fewer parameters and $<$0.5MB model size," *arXiv:1602.07360*, 2016.

[21] Y. Tian, D. Krishnan, and P. Isola, "Contrastive Representation Distillation," in *ICLR*, 2020.

[22] A. Krizhevsky, V. Nair, and G. Hinton, "Cifar-10 (canadian institute for advanced research),"

[23] J. Deng, W. Dong, R. Socher, *et al.*, "ImageNet: A Large-Scale Hierarchical Image Database," in *CVPR*, 2009.

[24] K. He, X. Zhang, S. Ren, *et al.*, "Delving deep into rectifiers: Surpassing human-level performance on imagenet classification," in *ICCV*, 2015.

[25] X. Glorot and Y. Bengio, "Understanding the difficulty of training deep feedforward neural networks," in *PMLR*, 2010.

[26] A. Parashar, P. Raina, Y. S. Shao, *et al.*, "Timeloop: A systematic approach to dnn accelerator evaluation," in *ISPASS*, 2019.

[27] Y. Chen, J. Emer, and V. Sze, "Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks," in *ISCA*, 2016.

[28] M. Alwani, H. Chen, M. Ferdman, *et al.*, "Fused-layer CNN accelerators," in *MICRO*, 2016.