Leveraging Tensor Methods in Neural Architecture Search for the automatic development of lightweight Convolutional Neural Networks

Mayur Dhanaraj[a *], Huyen Do[b], Dinesh Nair[b], and Cong Xu[b]

[a]Dept. of Electrical and Microelectronic Engineering, Rochester Institute Of Technology, Rochester, NY, USA
[b]Amazon Halo Health & Wellness, Sunnyvale, CA, USA

ABSTRACT

Most state-of-the-art Convolutional Neural Networks (CNNs) are bulky and cannot be deployed on resource-constrained edge devices. In order to leverage the exceptional generalizability of CNNs on edge-devices, they need to be made efficient in terms of memory usage, model size, and power consumption, while maintaining acceptable performance. Neural architecture search (NAS) is a recent approach for developing efficient, edge-deployable CNNs. On the other hand, CNNs used for classification, albeit developed using NAS, often contain large fully-connected (FC) layers with thousands of parameters, contributing to the bulkiness of CNNs. Recent works have shown that FC layers can be compressed, with minimal loss in performance, if any, using tensor processing methods. In this work, for the first time in literature, we leverage tensor methods in the NAS framework to discover efficient CNNs. Specifically, we employ tensor contraction layers (TCLs) to compress fully connected layers in the NAS framework and control the trade-off between compressibility and classification performance by handcrafting the ranks of TCLs. Additionally, we modify the NAS procedure to incorporate automatic TCL rank search in an end-to-end fashion, without human intervention. Our numerical studies on a wide variety of datasets including CIFAR-10, CIFAR-100, and Imagenette (a subset of ImageNet) demonstrate the superior performance of the proposed method in the automatic discovery of CNNs, whose model sizes are manyfold smaller than other cutting-edge mobile CNNs, while maintaining similar classification performance.

Keywords: Neural architecture search, CNN compression, lightweight CNNs, edge-deployable networks, Efficient deep-learning models, Tensor methods.

1. INTRODUCTION

Since their advent in 1990 [1], convolutional neural networks (CNNs) have revolutionized computer vision. In the present age, CNNs are widely used for computer vision tasks such as classification, detection, segmentation, tracking, facial recognition, and autonomous driving, to name just a few. The success of CNNs is mainly attributed to their ability to perform automatic end-to-end feature extraction. Other advantages include sparsity, weight sharing, and end-to-end trainability. Although CNNs have demonstrated state-of-the-art performances in many real-world applications, they are very deep, bulky, and often over-parameterized. We need massive number of GPU hours to train these large CNNs. The resulting networks have hundreds, if not thousands of layers, making them infeasible for use in edge devices with limited storage, computational, and power resources. In the modern era, artificially intelligent edge devices are omnipresent and play a crucial role in healthcare, defense, education, and entertainment, among others. In order to maintain superior performance, smart edge-devices rely on the impressive generalization capabilities of large deep-learning models. However, due to the hardware constraints of edge devices, deploying deep models on them is infeasible.

*Work done while interning at Amazon.
Further author information: (Send correspondence to M.D.)
M.D. : E-mail: mxd6023@rit.edu
H.D. : E-mail: huyendo@amazon.com
D.N. : E-mail: naidines@lab126.com
C.X. : E-mail: congxu@amazon.com

One may circumvent this problem by hosting the bulky deep-learning models on the cloud and accessing them remotely using internet connectivity. The superior generalization capabilities of deep models on the cloud can be leveraged using data collected by the edge devices. That is, the data acquired by the resource-constrained edge devices are first transmitted to the cloud, wherein the deep-learning models reside. Next, the deep models perform inference on this data and subsequently, the inference results are streamed down to the edge device. Although this is a straightforward workaround in general, transmitting sensitive data used in some applications including healthcare and defense pose privacy and security concerns, making cloud inference infeasible. Moreover, the speed of internet connectivity becomes a bottleneck to the speed of inference and if there is a lack of (reliable) internet connectivity, the edge-device may not be able to avail certain cloud services, resulting in degraded performance and/or user experience. In addition, hosting the large CNN models on the cloud attracts additional expense. To address these problems, researchers have focused on the development of lightweight and resource-efficient CNNs that are deployable directly on edge-devices [2–7]. Sufficiently lightweight CNNs can successfully be deployed on edge-devices to obtain satisfactory predictive performance. Edge-deployed CNNs enable uninterrupted operation of edge-devices, even in the absence of internet connectivity and help save on cloud compute costs. Additionally, performing inference on edge devices can assist in reducing the carbon footprint of large cloud compute infrastructure. Neural architecture search (NAS) [8–11] is a recent line of research for CNN compression that has attracted significant research interest. On the other hand, it is known that many real-world datasets assume a natural multi-way array (tensor) structure and maintaining the tensor structure of data enables learning methods to leverage higher-order correlations in the data, yielding better generalization performance [12]. Especially in CNNs, tensor processing methods are gaining much traction, owing to their ability to maintain and leverage the multidimensional array structure of CNN feature maps [13–20].

In this work, we employ tensor processing in the NAS framework in order to discover lightweight, efficient, and edge-deployable CNNs. First, we present a NAS method that incorporates TCLs to replace bulky fully-connected layers, resulting in lightweight CNN discovery. However, we note that TCLs warrant hand-crafted hyperparameter (rank) tuning, resulting in longer training times in some cases. Subsequently, we modify the NAS strategy to incorporate automatic hyperparameter (rank) tuning for the tensor processing layer, resulting in an end-to-end automatic CNN discovery, without human intervention. We present a brief background on existing CNN compression methods in the sequel.

2. BACKGROUND

CNN compression has attracted substantial research interest over the past decade. We discuss some directions of research to this end in the following.

Quantization. Broadly, model quantization involves the replacement of higher precision (typically 64-bit) weights by lower precision weights, while making sure the performance of the original network is preserved approximately [2, 3]. These techniques are widely applied to existing state-of-the-art networks and are shown to be successful in significant deep CNN compression.

Pruning. Model pruning [4, 5] takes a trained CNN and reduces the number of connections by removing weak connections (generally weights with small magnitude). The resulting network with fewer connections is generally smaller and fine-tuned to achieve a certain target performance. Pruning techniques can be classified into structured and unstructured pruning. Structured pruning involves the joint removal of groups of weights such that the pruned network is shallower. However, unstructured pruning is able to remove any weight connection independently of all other weights removed, allowing the network to be more sparse.

Knowledge distillation. A recent and popular approach for network compression is knowledge distillation [6, 7, 21]. Knowledge distillation involves a smaller, efficient model, referred to as the "student" that learns to mimic the performance of a larger, more complex model, known as the "teacher". The student model mimics the teacher by matching its intermediate features and final softmax output with that of the teacher during training. These methods have shown to achieve significant model compression, while maintaining performance similar to the original model.

Tensor methods for CNN compression. Tensor methods are a natural fit for CNN compression because the intermediate feature maps of a CNN naturally assume a multilinear (tensor) structure. Tensor methods preserve

and leverage this inherent multilinear structure of the feature maps in order to perform model compression. These techniques are based on some of the well-known tensor decomposition methods like Tucker decomposition (TKD), Canonical Polyadic decomposition (CPD), and Tensor Train decomposition (TT) [13–20, 22–24], among others.

Next, we discuss neural architecture search and provide the mathematical preliminaries of tensor processing in more detail as our method is based on these techniques.

2.1 Neural architecture search

It is common knowledge that designing CNNs to obtain state-of-the-art performance on a given dataset is cumbersome, time-consuming, and involves extensive handcrafting. Clearly, this is not a straightforward process and warrants the use of multiple powerful GPUs. Moreover, the search space is extremely large and the possible combinations of resulting networks is practically inexhaustible. With limited time and hardware resources, it is infeasible to build all possible architectures and train them on a given dataset. In addition, an architecture that demonstrates good performance on a dataset may not preform well on a different dataset. Further fine-tuning and/or hyperparameter tuning may be needed to improve its performance on the new dataset. All these reasons make it hard for researchers to design a CNN for any dataset at hand. It is especially painstaking to design CNNs for multiple end tasks and/or datasets, which is typical in an industry setting. Therefore, the need is to develop a strategy that performs automatic CNN design, given the search space. To this end, NAS was proposed in the literature and has attracted significant research interest in both the industry and academia. The main strength of NAS is its ability to discover the best architecture among all possible architectures from a given search space. In a mobile setting where efficient CNNs are desired, NAS automatically discovers architectures that adhere to device constraints defined by the user. The models output by NAS meet the user requirements and perform well at the same time, saving researches hundreds of hours on handcrafting a "good" architecture. Importantly, NAS aids in the automatic development of task-specific and data-specific models.

The NAS procedure involves three major steps, namely

1. **Designing the search space** – e.g., the number of layers, the number of parameters, the overall latency of the network, the possible blocks for each layer, micro versus macro design, and the level of handcrafting, to name a few.
2. **Designing the search strategy** – e.g., reinforcement learning, evolutionary search, progressive search, and gradient based search.
3. **Determining the evaluation criterion** – Estimating the performance of the discovered architecture on the target dataset is time-consuming. Possible workarounds include the use of a proxy dataset (a subset of the target dataset) or using the entire target dataset, while reducing the number of training epochs.

Early techniques employed reinforcement learning (RL) to perform NAS. This approach replaced human handcrafting with the help of controller neural networks and were shown to work well on benchmark datasets. However, RL based methods were extremely slow to train and required 1000's of GPU hours. More recently, gradient based NAS techniques were proposed to make NAS efficient. Some notable works include DARTS [9] and FBNet [8]. An excellent overview of other approaches for NAS, their strengths and weaknesses are elaborated in [25]. We employ FBNet's strategy to preform NAS and for the sake of completeness, we present a brief overview of FBNet in the sequel.

2.1.1 FBNet

FBNet is a recently proposed method for neural architecture search based on gradient descent. Therefore, unlike RL based NAS techniques, FBNet is extremely efficient. At the same time, it was shown that the CNNs discovered by FBNet's strategy outperform manually designed stat-of-the-art networks, as well as some networks discovered by other NAS methods [8]. FBNet's uniqueness can be attributed to the fact that it employs target device latency in the loss function, thereby enabling the method to discover device specific networks. A simplified version of the loss function is as follows

$$\mathcal{L} = \mathtt{class}_{\mathtt{loss}} \cdot (\mathtt{lat}_{\mathtt{loss}})^{\beta} \tag{1}$$

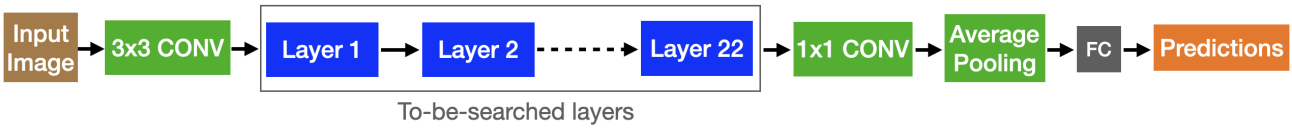

Figure 1: Graphical representation of the macro architecture of FBNet. CONV represents the convolutional layer. There are 26 layers in total, among which 22 layers are to be searched (TBS) by the NAS strategy.

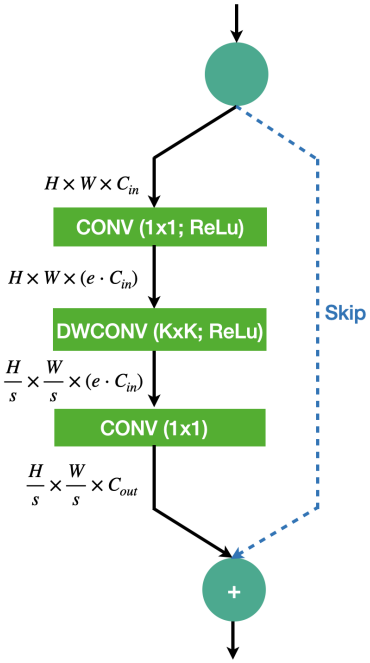

Figure 2: Micro architecture of each TBS block. CONV and DWCONV represent the convolutional layer and the depth-wise convolutional layer respectively.

where $\texttt{class}_{\texttt{loss}}$ is the cross-entropy classification loss term, $\texttt{lat}_{\texttt{loss}}$ is the overall latency of the network, and β is a modulating factor that assists the user to control the trade-off between choosing networks with better classification performance versus lower latency. The overall latency is computed as a summation of latencies of each layer in the network. The latency of each layer is first measured on the target device and recorded on a look-up table, which can later be referred to, to compute the overall latency. Since the loss function accounts for both classification performance and overall latency, FBNet is able to discover networks that jointly demonstrate cutting-edge classification performance and low latency. In terms of its architecture, FBNet employs MobileNetV2 [26] like architecture along with residual skip connections, consisting of 22 layers to be searched (TBS) by NAS strategy, with each layer able to choose from 9 unique options. A graphical representation of the macro architecture of FBNet is presented in Figure 1.

Specifically, each TBS layer in the macro architecture of FBNet has the freedom to choose a different block from the layer-wise search space, consisting of 9 options [8]. As shown in Figure 2, the block structure of each layer-wise search space consists of a 1×1 point-wise convolution with ReLu activation, followed by a $K \times K$ depth-wise convolution with ReLu activation, and finally a 1×1 convolution. In addition, a parallel residual skip connection is employed to add the activation maps obtained by the series of convolutional blocks to the input feature map of the block, if the input dimensions match the dimensions of the resulting convolutional feature maps, as shown in Figure 2. The size of the input to any block in the layer-wise search space is denoted as $H \times W \times C_{in}$, where H, W, and C_{in} denote the height, width, and depth of the input feature map respectively. A hyperparameter for expansion ratio, e, is employed to regulate how much the depth of the feature map increases after the first 1×1 convolution, compared to the input depth. The output of the $K \times K$ depth-wise convolution is H/s, W/s, $e \cdot C_{in}$, where s is the pre-determined stride of the depth-wise $K \times K$ convolution. Finally, the output of the last 1×1 convolution is $H/s \times W/s \times C_{out}$.

In order to circumvent forming and training all possible architectures from the search space, FBNet employs the concept of a stochastic supernet. The stochastic supernet has the same macro architecture as described above, with each TBS block having 9 parallel blocks, each one corresponding to a unique block from the feasibility set. During training, the supernet learns the relative importance (weights) of each of the 9 parallel blocks in a given TBS layer and also the weights of each convolutional layer of each of the 9 TBS blocks. During inference, only one candidate block is sampled from each TBS layer to obtain the overall network architecture, which is further trained and evaluated.

2.2 Mathematical preliminaries on tensor processing

In this work, we reserve lowercase and uppercase letters to denote scalars, e.g., x and D_1, uppercase boldface letters are used to denote matrices, e.g., $\mathbf{X}$, and Euler boldface letters denote tensors, e.g., $\boldsymbol{\mathcal{X}}$. In the sequel, we present some preliminaries of tensor processing.

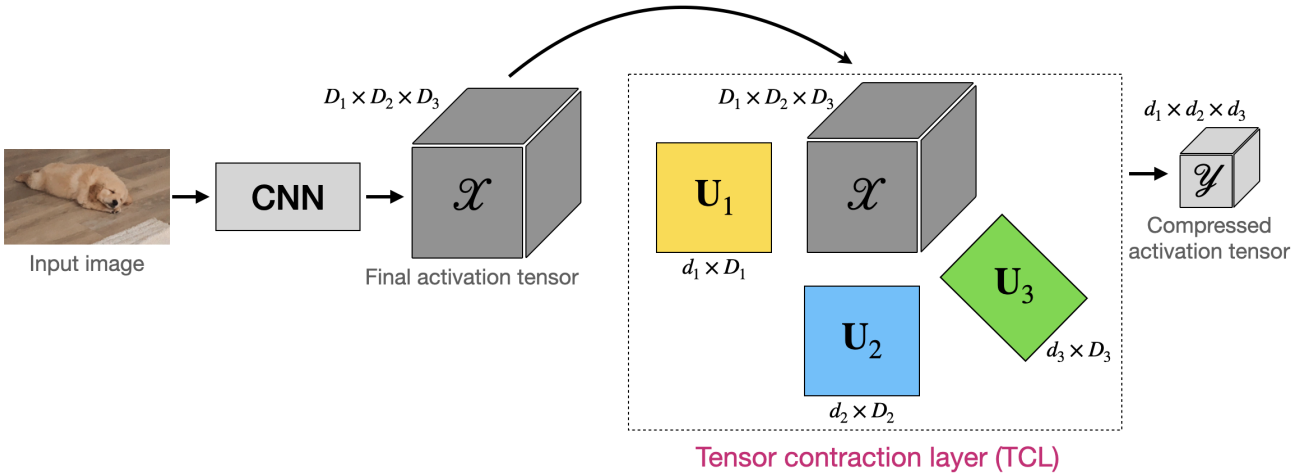

Figure 3: Tensor Contraction Layer (TCL) [28, 29]. The three learnable factor matrices $\mathbf{U}_1$, $\mathbf{U}_2$, and $\mathbf{U}_3$ for three modes of the final activation tensor $\boldsymbol{\mathcal{X}}$ are represented as colored rectangles. The values d_1, d_2, d_3 are known as TCL ranks. They are the hyperparameters of TCL and can be tuned according to the dataset at hand.

Fibers of any tensor are a higher-order equivalent of matrix columns and rows. Fibers of a tensor are obtained by fixing all but one of its dimension.

Mode-n matrix unfolding of any tensor is obtained by rearranging its mode-n fibers as the columns of a matrix. That is, the mode-n matrix unfolding of a mode-N tensor $\boldsymbol{\mathcal{X}} \in \mathbb{R}^{D_1 \times D_2 \times \ldots \times D_n \times \ldots \times D_N}$ is denoted as $\mathbf{X}_{(n)}$ and it is obtained by arranging the mode-n fibers of tensor $\boldsymbol{\mathcal{X}}$ as the columns of the matrix $\mathbf{X}_{(n)} \in \mathbb{R}^{D_n \times \prod_{k=1,k\neq n}^{N} D_k}$. Mathematically, the $(i_1, i_2, \ldots, i_N)$-th element of tensor $\boldsymbol{\mathcal{X}}$ will be mapped to (i_n, j)-th element of matrix $\mathbf{X}_{(n)}$, where $j = 1 + \sum_{k=1,k\neq n}^{N} (i_k - 1) \prod_{m=1,m\neq n}^{k-1} D_m$ [27].

n-mode product is the higher order generalization of the matrix product. The n-mode product of a tensor is obtained by first unfolding the tensor along its n-th mode into a matrix and then multiplying it with another matrix of conformable size. The n-mode product of a tensor $\boldsymbol{\mathcal{X}} \in \mathbb{R}^{D_1 \times D_2 \times \ldots \times D_N}$ with a matrix $\mathbf{W}_n \in \mathbb{R}^{d_n \times D_n}$ is denoted by $\tilde{\boldsymbol{\mathcal{X}}} = \boldsymbol{\mathcal{X}} \times_n \mathbf{W}_n$, and results in a $(N-1)$-mode tensor of size $D_1 \times D_2 \times \ldots \times D_{n-1} \times d_n \times D_{n+1} \times \ldots \times D_N$. Interestingly, the n-mode product can be expressed as matrix multiplication of $\mathbf{W}_n$ and the mode-n unfolding of $\boldsymbol{\mathcal{X}}$, where each mode-n fiber of tensor $\boldsymbol{\mathcal{X}}$ is multiplied by the matrix $\mathbf{W}_n$ i.e., $\tilde{\mathbf{X}}_{(n)} = \mathbf{W}_n \mathbf{X}_{(n)}$. n-mode product can be used to compress/expand a tensor along all modes by performing a sequence of mode-n products as follows

$$\tilde{\boldsymbol{\mathcal{X}}} = \boldsymbol{\mathcal{X}} \times_1 \mathbf{W}_1 \times_2 \mathbf{W}_2 \times_3 \cdots \times_N \mathbf{W}_N, \tag{2}$$

where, for any $\mathbf{W}_n \in \mathbb{R}^{d_n \times D_n}$, $d_n < D_n$, or $d_n \geq D_n$, and $\tilde{\boldsymbol{\mathcal{X}}} \in \mathbb{R}^{d_1 \times d_2 \times \ldots \times d_N}$. If $d_n < D_n$, then the resulting tensor $\tilde{\boldsymbol{\mathcal{X}}}$ is compressed/contracted along all modes, and the operation is called *tensor contraction*. It is important to note that for distinct modes of a sequence of mode-n products, the work in [27] showed that the order in which the mode-n products are performed is irrelevant. Mode-n product in (2) can be conveniently computed through matrix unfoldings and Kronecker products [27]

$$\tilde{\mathbf{X}}_{(n)} = \mathbf{W}_n \mathbf{X}_{(n)} (\mathbf{W}_N \otimes \ldots \otimes \mathbf{W}_{n+1} \otimes \mathbf{W}_{n-1} \otimes \ldots \otimes \mathbf{W}_1)^\top. \tag{3}$$

2.3 Tensor contraction layer

Fully connected layers in CNNs are typically bulky as depicted in Figure 4(a) and contribute to their enormous computational cost. Tensor contractions were employed in CNNs in order to reduce their computational overhead, while maintaining high classification accuracy [28, 29]. As shown in Figure 3, given the final activation tensor,

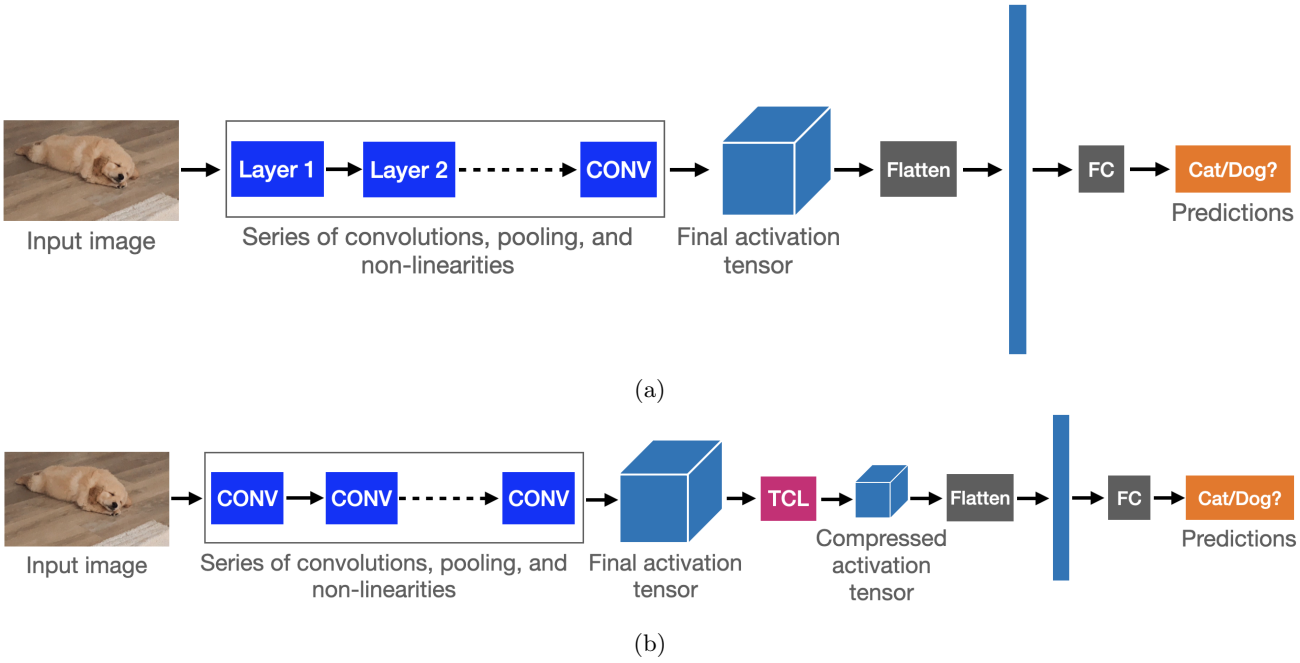

Figure 4: (a) Generic CNN. CONV represents convolutions, pooling, and non-linearity. FC represents the fully-connected layer. Note that typically the flattened final feature map is very long, creating a need for a large fully connected layer. (b) CNN with TCL. The large final feature map is compressed using TCL, subsequently resulting in a smaller flattened activation vector and fully-connected layer.

TCL compresses it along all modes to output another tensor of smaller size, using a series of n-mode products. This smaller tensor is then flattened and connected to a FC layer. We note that since the resulting tensor is of smaller size, the number of connections going into the FC layer is fewer, resulting in fewer learnable parameters. As TCL is applied on the third order activation tensor of a CNN, it is required to learn three factor matrices, one for each mode, that are updated end-to-end via backpropagation. Mathematically, given a final activation tensor $\boldsymbol{\mathcal{X}} \in \mathbb{R}^{D_1 \times D_2 \times D_3}$, TCL compresses it along all modes by using a series of mode-n products to obtain $\boldsymbol{\mathcal{Y}} \in \mathbb{R}^{d_1 \times d_2 \times d_3}$ as $\boldsymbol{\mathcal{Y}} = \boldsymbol{\mathcal{X}} \times_1 \mathbf{U}_1 \times_2 \mathbf{U}_2 \times_3 \mathbf{U}_3$, as shown in Figure 3, where every $\mathbf{U}_i \in \mathbb{R}^{d_i \times D_i}$ for $i \in \{1, 2, 3\}$ is known as the i-th factor matrix, which is initialized arbitrarily and updated jointly with the rest of the network parameters. Each d_i is a hyperparameter that needs to be handcrafted. TCLs are used to reduce the size of a fully-connected layer as shown in Figure 4(b) or replace it altogether.

3. PROPOSED METHOD

For the first time in literature, we propose to leverage tensor processing in NAS for CNN compression. Specifically, we propose to replace the parameter-dense fully connected (FC) layers within the NAS framework used to discover classification CNNs with TCL [28, 29], followed by a more compact Fully-connected layer. We introduce TCL in the macro architecture in two stages as discussed next.

3.1 Hand tuning TCL ranks

Firstly, we incorporate TCL in the framework of FBNet to replace a bulky FC layer by a TCL, followed by a smaller FC layer. In this approach, we manually tune the rank values of TCL. An illustration of this approach is provided in Figure 5(a). We perform an exploratory analysis with different TCL rank values and provide a detailed description of our observations in the experimental studies section. In general, we observe that a range of TCL rank values result in improved performance compared to the baseline FBNet model. However, hand-tuning TCL ranks may be cumbersome and time-consuming in some cases. In order to overcome this problem, we propose to extend the NAS strategy to perform a search for TCL ranks also, as explained next.

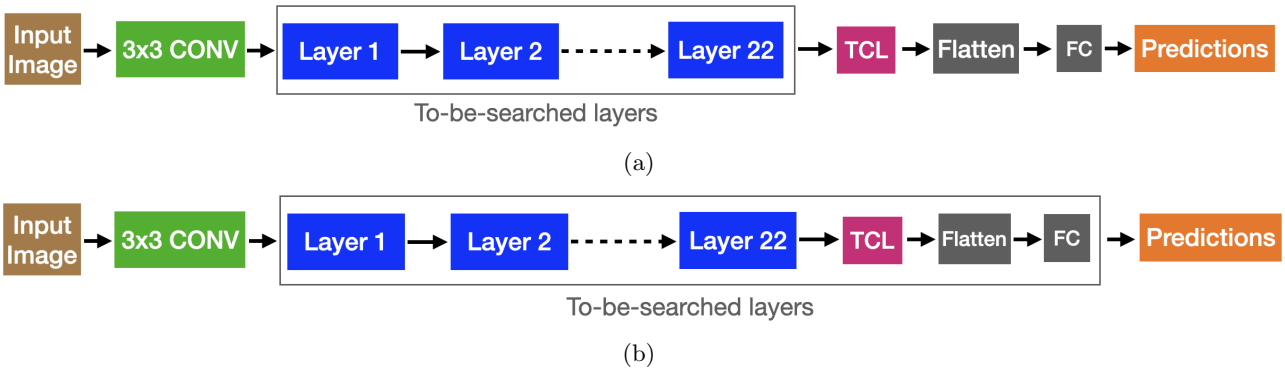

Figure 5: Incorporating TCL into the FBNet framework, where (a) TCL ranks are handcrafted and (b) TCL ranks are automatically chosen by the NAS procedure.

3.2 NAS for TCL ranks

Since hand-tuning TCL ranks is time-consuming, we propose to extend FBNet's search strategy to tune the ranks of TCL automatically, as depicted in Figure 5(b). In order to enable automatic rank tuning using NAS, we need to provide a search space for viable TCL ranks for the NAS strategy to choose from. In all our experiments, our search space for the last layer includes the original (bulky) FC layer and TCL layer with six distinct sets of rank values. Employing NAS for TCL ranks results in an overall reduction in search time and the NAS procedure is successful in discovering "meaningful" ranks values for TCL layers as detailed in our experimental studies.

4. EXPERIMENTAL STUDIES

We perform a large array of experiments on three different datasets, namely CIFAR-10 [30], CIFAR-100 [30], and Imagenette [31]. CIFAR-10 consists of 10 classes with 50,000 training images (5,000 images per class) and 10,000 testing images (1,000 images per class). Each image is of resolution 32×32. CIFAR-100 consists of 100 classes and has the same overall number of training and testing images as that of CIFAR-10. However the number of training and testing images per class is 10x fewer compared to those in CIFAR-10. Imagenette, with 10 classes, is a subset of the ImageNet dataset [32]. Imagenette consists of about 9,500 training images and about 4,000 testing images. Each image has a resolution of 224×224. All our experiments were performed on the NVIDIA V100 GPU.

4.1 Results and discussions

As mentioned earlier, we carry out our experiments in two stages, namely, stage 1, which involves hand-tuning TCL ranks, and stage 2, which involves NAS for TCL ranks. We provide a detailed description of the results obtained in the following.

4.1.1 Stage 1: Hand-tuning TCL ranks

Firstly, we employ FBNet's method to train on the CIFAR-10 dataset. The discovered model has a size of 6.8 MB and achieves a test classification accuracy of 83.15%. We note that the sampled architecture consists of seven skip layers out of the 22 layers to be searched (TBS) by the NAS procedure. This indicates that a smaller macro architecture (with fewer TBS layers) would suffice for CIFAR-10. To this end, we reduce the number of TBD layers from 22 to 8 and name the resulting model "Modified FBNet". This reduction straightforwardly results in a smaller model of size 548 KB, but the classification accuracy drops drastically to 78.76% (about 5% drop compared to the baseline performance). In order to preserve the baseline accuracy, while reducing the model size significantly, we employ TCL in the macro architecture, and hand-tune the rank values. We present the performance of FBNet models with various TCL rank values in Figure 6(a) and observe that incorporating TCL in the macro architecture improves the classification accuracy significantly. Among the different rank values, we

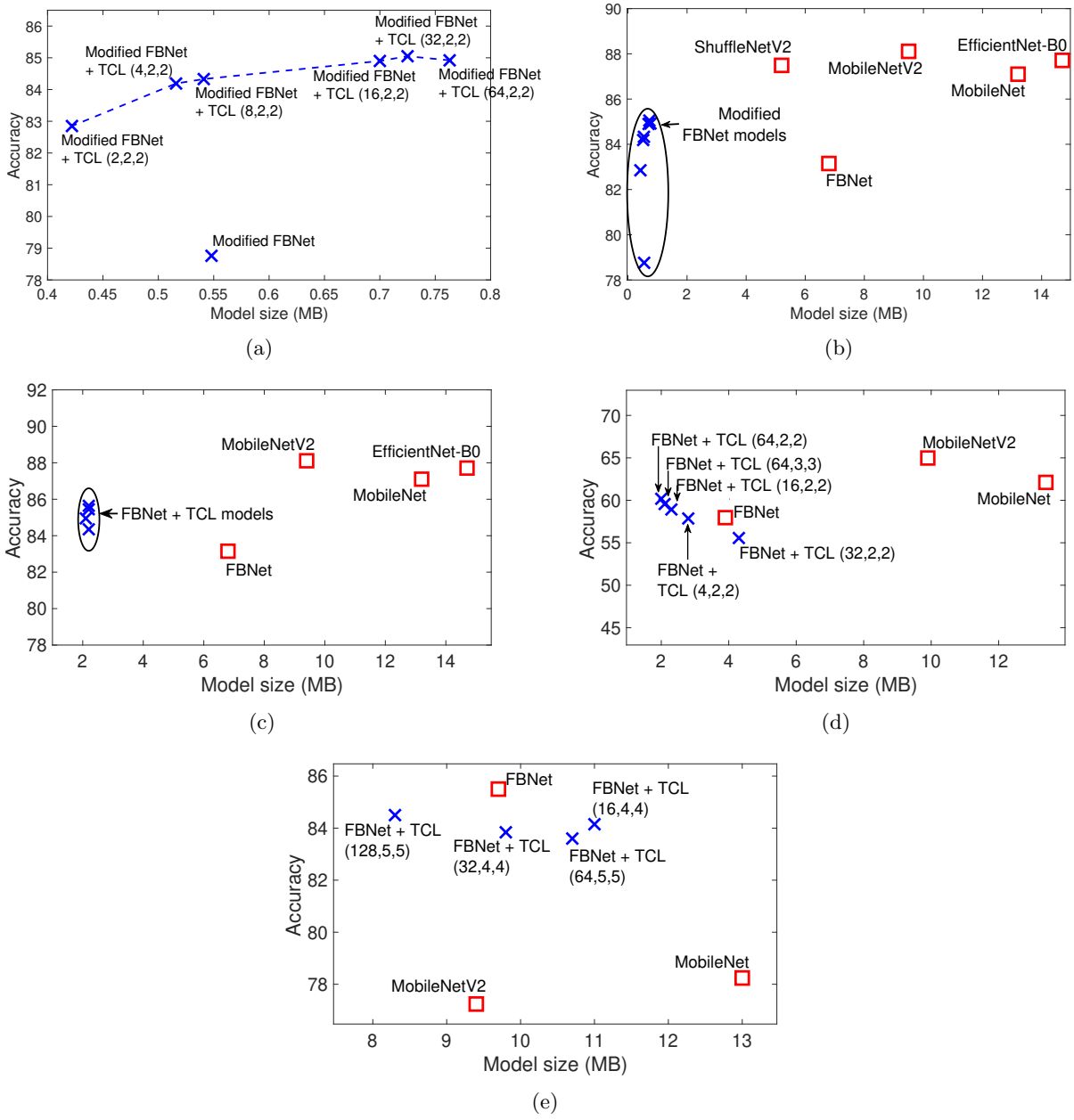

Figure 6: Classification accuracy versus model size in megabytes (MB) for (a) CIFAR-10 using modified FBNet (+ TCL) models (8 TBS layers), (b) comparison of modified FBNet + TCL models (8 TBS layers) with state-of-the-art efficient CNNs for CIFAR-10, comparison of FBNet + TCL models (22 TBS layers) with state-of-the-art efficient CNNs for (c) CIFAR-10, (d) CIFAR-100, and (e) Imagenette.

note that rank values of $(32, 2, 2)$ result in the best classification accuracy of 85.05% (about 2% better than the baseline accuracy), and a model of size 725 KB (about 9.3x smaller than the baseline model size). In addition, in Figure 6(b), we plot the relative performance of TCL + FBNet models with respect to other efficient state-of-the-art architecture such as MobileNet [33], MobileNetV2 [26], ShuffleNetV2 [34], and EfficientNet-B0 [35] and observe that the FBNet + TCL models perform similarly to the other baseline models in terms of classification accuracy. However, the size of FBNet + TCL models is significantly smaller. Clearly, employing TCL helps

in jointly reducing the model size, while improving its classification performance. The superior performance of FBNet + TCL models is attributed to the fact that TCL leverages the natural tensor structure of the final feature map and reduces its overall size, which in turn reduces the number of inputs to the FC layer, resulting in parameter savings. Moreover, since a smaller activation tensor is flattened, the loss in crucial spatial information is minimal, compared to the baseline model, leading to better classification performance.

Next, instead of first reducing the number of TBS layers in order to reduce the model size and then applying TCL to improve classification accuracy, we employ a single stage approach of incorporating TCL directly in the plain FBNet model with 22 TBS layers. Following this approach, we notice that the sampled FBNet + TCL models have a size of about 2 MB and a classification accuracy of about 85% as shown in Figure 6(c).

Subsequently, we repeat our experiments on CIFAR-100 and Imagenette datasets. In our experiments with CIFAR-100, we make similar observations compared to CIFAR-10. That is, FBNet + TCL models are jointly more efficient and demonstrate better classification performance compared to the baseline FBNet model. Additionally, these models have similar classification accuracy compared to MobileNet and MobileNetV2, but their model sizes are much smaller as shown in Figure 6(d). On Imagenette, however, we note that the baseline FBNet model performs well in terms of classification, while having a relatively small size, also as shown in the original paper [8]. When we introduced TCL with correctly chosen ranks, the model size is reduced and the classification accuracy drops marginally as seen in Figure 6(e). In general, we observe that TCL is able to automatically reduce the model size while improving the classification performance, compared to the baseline FBNet model.

However, we note that handcrafting TCL ranks is a time-consuming process. For example, it takes about seven days of FBNet training to estimate the "best" rank values. In order to reduce the training time, we employ NAS for TCL ranks as discussed next.

4.1.2 Stage 2: NAS for TCL ranks

We employ NAS for TCL rank tuning and notice that the procedure is able to choose the "optimal" rank values (exactly the ones that were determined to work the best via handcrafting) automatically. For example, on the CIFAR-10 dataset, we observe that the search returns rank values of (32, 2, 2) – the same exact values that worked the best when we tuned TCL ranks by hand. This demonstrates that the proposed method is able to discover models that perform well and have a tiny model size at the same time, without any human intervention.

5. CONCLUSIONS AND FUTURE WORK

In this work, for the first time in literature, we leverage tensor methods in NAS for efficient CNN design. Our method is able to discover efficient CNNs via end-to-end NAS. The models discovered using the proposed method are many times smaller than other state-of-the-art efficient CNN models on a variety of datasets. The proposed method has demonstrated superior compression performance on multiple datasets of interest, with negligible loss in classification performance, if any. One direction for future work involves the use of other device specific constraints including power consumption and FLOP count along with device latency in the loss function. These additional constraints may help in the discovery of device specific CNNs that are efficient in terms of memory usage, power consumption, and inference time, while delivering cutting-edge classification performance.

References

[1] Yann LeCun, Bernhard E Boser, John S Denker, Donnie Henderson, Richard E Howard, Wayne E Hubbard, and Lawrence D Jackel. Handwritten digit recognition with a back-propagation network. In *Proc. Adv. Neural Inf. Process. Syst. (NeurIPS)*, pages 396–404., Nov. 1990.

[2] Aojun Zhou, Anbang Yao, Yiwen Guo, Lin Xu, and Yurong Chen. Incremental network quantization: Towards lossless CNNs with low-precision weights. In *Proc. Int. Conf. Learn. Repr. (ICLR)*, pages 1–14., Apr. 2017.

[3] Yunchao Gong, Liu Liu, Ming Yang, and Lubomir Bourdev. Compressing deep convolutional networks using vector quantization. In *Proc. Int. Conf. Learn. Repr. (ICLR)*, pages 1–10., May 2015.

[4] Song Han, Jeff Pool, John Tran, and William Dally. Learning both weights and connections for efficient neural network. In *Proc. Adv. Neural Inf. Process. Syst. (NeurIPS)*, pages 1–9., Dec. 2015.

[5] Song Han, Huizi Mao, and William J Dally. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. In *Proc. Int. Conf. Learn. Repr. (ICLR)*, pages 1–14., May 2016.

[6] Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. Distilling the knowledge in a neural network. *ArXiv preprint.*, Mar. 2015.

[7] Jang Hyun Cho and Bharath Hariharan. On the efficacy of knowledge distillation. In *Proc. Int. Conf. Comput. Vision (ICCV)*, pages 4794–4802., Nov. 2019.

[8] Bichen Wu, Xiaoliang Dai, Peizhao Zhang, Yanghan Wang, Fei Sun, Yiming Wu, Yuandong Tian, Peter Vajda, Yangqing Jia, and Kurt Keutzer. FBNet: Hardware-aware efficient convnet design via differentiable neural architecture search. In *Proc. IEEE Conf. Comput. Vision Pattern Recogn. (CVPR)*, pages 10734–10742, Jun. 2019.

[9] Hanxiao Liu, Karen Simonyan, and Yiming Yang. Darts: Differentiable architecture search. In *Proc. Int. Conf. Learn. Repr. (ICLR)*, May 2019.

[10] Yihui He, Ji Lin, Zhijian Liu, Hanrui Wang, Li-Jia Li, and Song Han. AMC: Automl for model compression and acceleration on mobile devices. In *Proc. Euro. Conf. on Comput. Vision (ECCV)*, pages 784–800., Sep. 2018.

[11] Mingxing Tan, Bo Chen, Ruoming Pang, Vijay Vasudevan, Mark Sandler, Andrew Howard, and Quoc V Le. MnasNet: Platform-aware neural architecture search for mobile. In *Proc. IEEE Conf. Comput. Vision Pattern Recogn. (CVPR)*, pages 2820–2828., Jun. 2019.

[12] Animashree Anandkumar, Rong Ge, Daniel Hsu, Sham M Kakade, and Matus Telgarsky. Tensor decompositions for learning latent variable models. *J. Machine Learn. Res. (JMLR)*, 15:2773–2832, Aug. 2014.

[13] 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. Int. Conf. Learn. Repr. (ICLR).*, May 2016.

[14] Anh-Huy Phan, Konstantin Sobolev, Konstantin Sozykin, Dmitry Ermilov, Julia Gusak, Petr Tichavskỳ, Valeriy Glukhov, Ivan Oseledets, and Andrzej Cichocki. Stable low-rank tensor decomposition for compression of convolutional neural network. In *Proc. Euro. Conf. on Comput. Vision (ECCV)*, pages 522–539., Aug. 2020.

[15] Giuseppe G Calvi, Ahmad Moniri, Mahmoud Mahfouz, Qibin Zhao, and Danilo P Mandic. Compression and interpretability of deep neural networks via Tucker tensor layer: From first principles to tensor valued back-propagation. *ArXiv preprint.*, Jan. 2019.

[16] Zhisheng Zhong, Fangyin Wei, Zhouchen Lin, and Chao Zhang. ADA-Tucker: Compressing deep neural networks via adaptive dimension adjustment Tucker decomposition. *J. Neural Net.*, 110:104–115, Feb. 2019.

[17] Vadim Lebedev, Yaroslav Ganin, Maksim Rakhuba, Ivan Oseledets, and Victor Lempitsky. Speeding-up convolutional neural networks using fine-tuned CP-decomposition. In *Proc. Int. Conf. Learn. Repr. (ICLR).*, May 2015.

[18] Marcella Astrid and Seung-Ik Lee. CP-decomposition with tensor power method for convolutional neural networks compression. In *Proc. IEEE Int. Conf. Big Data Smart Comput. (BigComp)*, pages 115–118., Feb. 2017.

[19] Jean Kossaifi, Antoine Toisoul, Adrian Bulat, Yannis Panagakis, Timothy M Hospedales, and Maja Pantic. Factorized higher-order CNNs with an application to spatio-temporal emotion estimation. In *Proc. IEEE Conf. Comput. Vision Pattern Recogn. (CVPR)*, pages 6060–6069., Jun. 2020.

[20] Timur Garipov, Dmitry Podoprikhin, Alexander Novikov, and Dmitry Vetrov. Ultimate tensorization: compressing convolutional and FC layers alike. *ArXiv preprint.*, Nov. 2016.

[21] Lin Wang and Kuk-Jin Yoon. Knowledge distillation and student-teacher learning for visual intelligence: A review and new outlooks. *IEEE Trans. Pattern Anal. Machine Intell. (TPAMI)*, Jan. 2021.

[22] Emily L Denton, Wojciech Zaremba, Joan Bruna, Yann LeCun, and Rob Fergus. Exploiting linear structure within convolutional networks for efficient evaluation. In *Proc. Adv. Neural Inf. Process. Syst. (NeurIPS)*, pages 1269–1277., Dec. 2014.

[23] Manish Sharma, Panos P. Markopoulos, Eli Saber, M. Salman Asif, and Ashley Prater-Bennette. Convolutional auto-encoder with tensor-train factorization. In *Proc. IEEE/CVF Int.l Conf. Comput. Vis. (ICCV)*, pages 198–206, Oct. 2021.

[24] Manish Sharma, Panos P. Markopoulos, and Eli Saber. YOLOrs-lite: A lightweight cnn for real-time object detection in remote-sensing. In *IEEE Int. Geo. Remote Sens. Symp. (IGARSS)*, pages 2604–2607. IEEE, Jul. 2021.

[25] Thomas Elsken, Jan Hendrik Metzen, and Frank Hutter. Neural architecture search: A survey. *J. Machine Learn. Res. (JMLR)*, 20:1997–2017, Mar. 2019.

[26] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. Mobilenetv2: Inverted residuals and linear bottlenecks. In *Proc. IEEE Conf. Comput. Vision Pattern Recogn. (CVPR)*, pages 4510–4520, Jun. 2018.

[27] Tamara G. Kolda and Brett W. Bader. Tensor decompositions and applications. *SIAM Rev.*, 51(3):455–500, Sep. 2009.

[28] Jean Kossaifi, Aran Khanna, Zachary Lipton, Tommaso Furlanello, and Anima Anandkumar. Tensor contraction layers for parsimonious deep nets. In *Proc. IEEE Conf. Comput. Vision Pattern Recogn. (CVPR) Workshops*, pages 26–32., Jul. 2017.

[29] Jean Kossaifi, Zachary C Lipton, Arinbjörn Kolbeinsson, Aran Khanna, Tommaso Furlanello, and Anima Anandkumar. Tensor regression networks. *J. Machine Learn. Res. (JMLR)*, 21:1–21., Jul. 2020.

[30] Alex Krizhevsky, Geoffrey Hinton, et al. Learning multiple layers of features from tiny images. Apr. 2009.

[31] FastAI. Imagenette, Apr. 2019. Available: *https://github.com/fastai/imagenette*. Accessed: Jul. 2021.

[32] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. ImageNet classification with deep convolutional neural networks. In *Proc. Adv. Neural Inf. Process. Syst. (NeurIPS)*, pages 1097–1105., Dec. 2012.

[33] Andrew G Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. MobileNets: Efficient convolutional neural networks for mobile vision applications. *ArXiv preprint.*, Apr. 2017.

[34] Ningning Ma, Xiangyu Zhang, Hai-Tao Zheng, and Jian Sun. Shufflenet v2: Practical guidelines for efficient cnn architecture design. In *Proc. Euro. Conf. Comput. Vision (ECCV)*, pages 116–131, Sep. 2018.

[35] Mingxing Tan and Quoc Le. EfficientNet: Rethinking model scaling for convolutional neural networks. In *Proc. Int. Conf. Machine Learn. (ICML)*, pages 6105–6114., Jun. 2019.