Using Network Architecture Search for Optimizing Tensor Compression

Arunachalam Thirunavukkarasu(✉) and Domenik Helms

Deutsches Zentrum für Luft- und Raumfahrt, Linder Höhe, 51147 Köln, Germany
{arunachalam.thirunavukkarasu,domenik.helms}@dlr.de

Abstract. In this work we propose to use Network Architecture Search (NAS) for controlling the per layer parameters of a Tensor Compression (TC) algorithm using Tucker decomposition in order to optimize a given convolutional neural network for its parameter count and thus inference performance on embedded systems. TC enables a quick generation of the next instance in the NAS process, avoiding the need for a time consuming full training after each step. We show that this approach is more efficient than conventional NAS and can outperform all TC heuristics reported so far. Nevertheless it is still a very time consuming process, finding a good solution in the vast search space of layer-wise TC. We show that, it is possible to reduce the parameter size upto 85% for the cost of 0.1–1% of Top-1 accuracy on our vision processing benchmarks. Further, it is shown that the compressed model occupies just 20% of the original memory size which is required for storing the entire uncompressed model, with an increase in the inference speed of upto 2.5 times without much loss in the performance indicating potential gains for embedded systems.

Keywords: Tensor Compression · Embedded systems · Network Architecture Search · Tucker Decomposition · Convolutional Neural Network

1 Introduction

Image recognition is one of the key algorithms for advanced driver assistance systems, e.g., for autonomous vehicles which are composed of large neural networks. In order to classify a single image, the original AlexNet requires around 240 MB of memory just to store the weight parameters which are obtained after training the model. Similarly, VGG model requires around 528 MB for storing its weights with 99% of MAC operations coming from convolutional layers [1]. With increasing network size, storage space required to store their parameters also increases. This leads to a larger inference time when such AI models are employed on embedded devices. For cost, energy and reliability reasons, automotive embedded systems offer only reduced computation resources. In order to

This publication was created as part of the research project "KI Delta Learning" (project number: 19A19013K) funded by the Federal Ministry for Economic Affairs and Energy (BMWi) on the basis of a decision by the German Bundestag.

Published by Springer Nature Switzerland AG 2023
S. Henkler et al. (Eds.): IESS 2022, IFIP AICT 669, pp. 139–150, 2023.
https://doi.org/10.1007/978-3-031-34214-1_12

bridge the gap between the high demand of these AI application and the limited resources of automotive systems, various compression techniques like pruning, quantization, teacher-student reduction [2], and a promising, but rarely used Tensor Compression (TC) techniques are employed.

According to [3], pruning resulted in a nine times parameter reduction for AlexNet and sixteen times parameter reduction for VGG-16 model. Quantization technique reduces the number of bits (corresponding to filter weights) to read from the memory in a convolutional neural network. Recent methods for quantization offer quantization down to 4–5 bit accuracy for most layers (if supported by the hardware) by using different quantization for inputs and activations [4] or per layer [5] or even per kernel [6] with sacrificing less than half a percent in accuracy [7]. The work carried out by [8] shows that another compression technique 'Knowledge distillation' has relatively higher advantages and perform well on network trained on MNIST.

On the other hand, Tensor Compression splits one of the most used building blocks of image recognition networks, the (2 dimensional) convolution into a linear part (tensor multiplication) and a nonlinear post-processing (activation function and optionally pooling). For the linear part, mathematical methods are used to approximate the tensor by a series of much smaller tensors. The most prominent realization for this is the Tucker decomposition [9] - the tensor equivalent to Singular Value Decomposition (SVD). While the Tucker decomposition itself is straight-forward and easily available in mathematical libraries (in python), it needs to be controlled by two parameters (third and fourth rank of the target tensor) for which not many obvious algorithms exist.

Often a machine learning engineer is needed to design the architecture and structure of the artificial neural networks based on the problem description. Neural Architecture Search (NAS) is a better way of automating this process and Microsoft Neural Network Intelligence [10] has produced toolkits for NAS. This paper analyses, how TC can be applied on neural networks and the potential of NAS to determine the compression parameters. It speaks about the importance of Tucker decomposition, and the effect of different rank values on the compressed models. Further, how to best use the NAS functionality for controlling compression parameters (3rd and 4th ranks) and the importance of fine-tuning are investigated. Finally, performance analysis of compression rates vs. accuracy loss, size requirements and inference time are carried out to evaluate the results of TC by comparing against conventional NAS (performing search of filter-count per layer). The entire implementation is done in Keras and Tensorflow in Python and all the experiments are carried out on a high-performance AI server (NVIDIA DGX-1).

2 Related Works

This section speaks about Tensor Compression and its current state-of-the-art, followed by the deployment of NAS in finding a smaller network architecture based on per-layer filter search.

2.1 Tensor Compression

TC involves decomposing the original tensors into multiple factors and performing mathematical operations on them, especially mode-n multiplications. TC is rarely researched and no work has fully exploited it's potential. Some of the available decomposition algorithms are Singular Value Decomposition, Canonical Parafac, Tensor Train and finally Tucker Decomposition.

Singular Value Decomposition (SVD) is valid only for 2D tensors aka matrices. SVD means decomposing (factorization) the original matrix into three different matrices, thereby reducing original number of factors in real matrix. SVD simplifies matrix calculations and improves the algorithm results with less complexity [13].

$$A_{n\times m} = U_{n\times n} \cdot \ S_{n\times m} \cdot \ V^T_{m\times m} \tag{1}$$

Equation 1 gives the mathematical expression of SVD where, the matrix U represents the left singular values along its column and V^T represents the right singular values along its row respectively. The middle matrix S contains the singular values with U and V being orthogonal to each other.

2.2 Tucker Decomposition

This particular type of decomposition was introduced by Ledyard R Tucker in the year 1966 [9], which is similar to SVD but applied on tensors. Hence, is also called as Higher Order Singular Value Decomposition. TD splits the original 'n' mode (dimension) tensor into 'n' different factor matrices and a compressed version of the original tensor, called core tensor as shown in Fig. 1. TD does not follow the regular matrix multiplications instead, it works on mode multiplication method. A 3-dimensional tensor has 3 modes namely 'x', 'y', 'z' and applying TD to this tensor will yield 3 factor matrices (one factor per mode) followed by a core tensor of 3 dimension. The equation which represents TD for n dimensional tensor is given in Eq. 2.

$$X \approx G \times_1 U^{(1)} \times_2 U^{(2)} \times_3 .... \times_N U^{(N)} \tag{2}$$

X - Original tensor,
G - Core tensor,
$U^{(1)}, U^{(2)}, ..., U^{(N)}$ - Factor matrices,
$\times_n$ - denotes the n-mode tensor product

Figure 1 shows a schematic representation of TD of a tensor. The original tensor 'T' of dimensions R1 × R2 × R3 are decomposed into a core tensor 'D' of dimensions 'a × b × c' followed by three factor matrices along each direction with the mentioned dimensions. Size of the core can be decided by the users but care must be taken that minimum values of 'a', 'b', and 'c' should be at least 1. Till date, there is no standard way of selecting the optimized ranks which determine the size of the core tensor. Only trial and error methods has been adopted for appropriate rank selection along with some other search heuristics

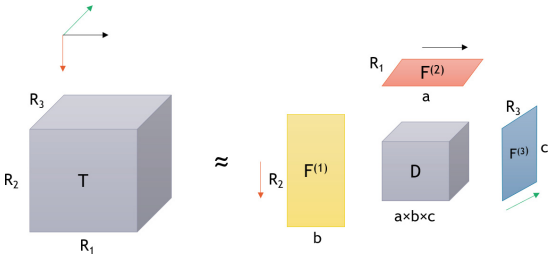

Fig. 1. Tucker Decomposition

which exist in theoretical approaches. Tensors of a CNN are 4 dimensional, so applying TD on them needs to be controlled by 4 rank factors namely [a, b, c, d] for [R1, R2, R3, R4].

The compression work by [11] conducted on various models such as AlexNet, VGG-S, VGG-16 etc. were evaluated on a smartphone and Titan X platforms led to a conclusion that the average runtime was enhanced by 2.72 times for AlexNet, by 3.68 times for VGG-S and 3.34 times for VGG-16 networks respectively. It also reported a factor of 1.4 to 3.7 reduction on inference time on embedded hardware for the cost of 0.2–1.7% accuracy. The GitHub implementation [15] which gives an overall idea of how to implement a Canonical & Parafac decomposition along with Tucker decomposition is used a reference for this paper work. Best to our knowledge, the most sophisiticated TC heuristic is discussed in [14], reducing the number of operations on a neural network by a factor of 2–4 for below 0.2% accuracy loss. This is done by a hierarchical rank search on layer clusters.

2.3 Model Optimization Using NAS

Microsoft's NNI - a tool to handle automated machine learning (AutoML) problems by choosing hyper-parameters (number of filters and layers, learning rate, activation functions, etc). NNI has a set of inbuilt tuning algorithms which searches for the most efficient architecture [10]. Based on the user defined search criteria it performs a number of trainings (trials) with different parameter values and comes up with the most optimal solution. NNI supports a number of ML frameworks and libraries such as Tensorflow, Keras, Pytorch, Scikit-learn etc.

NAS, a toolkit of NNI can be employed to find appropriate hyper-parameters (filter counts, etc.) in each layer of the neural network within the specific range of choice (search space). NAS will try to come up with an optimized architecture based on the defined tuning algorithm. NAS has a set of different in-built tuning algorithms like Gaussian, Random, annealing, etc. User can choose the most suitable tuning algorithm based on their needs. This filter search is a powerful network optimization method employed using NAS. The only disadvantage of filter-NAS is that, every time a new filter value is chosen from the search space, the entire model has to be trained from scratch requiring huge computational time. This in turn makes the system even more expensive.

3 Tensor Compression Implementation

We implemented and tested a methodology for applying TC on convolutional layers of neural networks. For that, we started by reading in the Keras graph, describing the neural network and constructed a second similar instance layer by layer. At user constrained levels, we do not copy over the layer and all its trained weights to the new model, but instead split the activation function, extract the weight tensor, apply a Tucker decomposition with user constrained ranks 'c' and 'd' for the channel (R3) and filter (R4) dimensions respectively. R1 and R2 are the kernel 'a' and 'b' sizes respectively and will not be altered by TC. From the resulting three tensors coming out of the TD we set up three sub 2D convolutional layers without an activation function and used the three tensor's values as weights for those three layers. Finally, the original activation function is added to the last of the three layers. The different steps used in the implementation of TC are as follows:

1. The first sub layer will perform a pointwise convolution on the 3rd factor matrix to reduce the number of channels to 'c' dimension.
2. The output of the previous pointwise convolution will be the input to this second layer where a normal convolution is performed on the core tensor with 'c' input channels and 'd' output channels. This becomes the input for the next sub-layer.
3. Final pointwise convolution with 'c' input channels and the original output channel (number of output filters w.r.t layer without compression). Biases and activation functions are also added as this is the last sub-layer.

While applying Tucker algorithm for each convolutional layer, the user has to specify the rank values. In our work, this process is taken care by the NAS as explained in the coming sections. Once the rank values are obtained, the above 3 steps are repeated for all convolutional layer that are subjected to compression. After applying this algorithm for the convolution layers, the new model with compressed layers is fine tuned to compensate for compression losses.

4 NAS Setup

As explained in the previous section, Tucker algorithm is applied on convolutional layers, decomposing it into a sequential 3 sub layers. The dimensions of the 3 sub-layers are determined by the 3rd and 4th rank factors. NAS is assigned the task of finding out the best possible rank values from the user defined search space. The different steps involved in implementing TC using NAS are as follows:

The first step is to define a search space which contains the key parameters (rank values) that are to be tuned using the NNI. The range of values/choices are written in JSON format from which specific rank values for each trials are chosen. Second step is to modify the existing Python codes by including NNI commands in them. The final step in implementing a trial run is to write a 'configuration' file in YAML format containing the experiment details (duration, tuners, assessors,

etc). Some of the available tuning algorithms which NAS provides are: TPE, Random Tuner, GP Tuner, etc. In our experiments, only Gaussian Process (GP) tuner is considered which is based on the Bayesian optimization techniques.

4.1 Evaluation Metric

Microsoft NNI allows the users to define their own target functions based on which NAS will tune for optimal rank values. Since we are interested in smaller model size without degradation in its performance, an evaluation metric is designed such that it combines both, size and accuracy.

$$t = (1 - a) + (p \cdot \alpha) \tag{3}$$

where, t - target function,
a - validation / top-1 accuracy
p - parameter count of the model
α - constant

The main optimization technique is to lower the target function as much as possible. Parameter count of the model is denoted by 'p'. It can be either total parameters in a model or just the parameter count of the layers which are subjected to compression. Based on the value of 'p' the constant 'α' is chosen. As the parameter count reduces with a significant increase in accuracy (lowering of loss function (1-a)), the entire target function value will be minimized. This target function is written in the python training script and reported to the NNI tuner after each training. Based on the previous value of target function, NAS tuner chooses the rank values for subsequent trials from the search space. User can choose the tuning algorithm based on which the tuners will tune for rank values.

5 Evaluation

In order to analyze the potential of TC, different deep neural networks are chosen and the Tucker algorithm is applied on them. Since CNN tensors are 4 dimensional, applying TD on them needs to be controlled by 4 rank values. Excluding the first two ranks which corresponds to kernel sizes, NAS is employed to choose the third and fourth rank parameters. The behavior of NAS and compression is studied on 3 different networks ranging from smaller generic AI up to a larger modified GoogleNet. These experiments are run on a NVIDIA DGX station inside different Docker containers. Since performing these experiments require even more powerful platforms, in order to fit the experiments within the available resources, some modifications are made on the original model architecture such that their performance is not degraded and still they yield a good result even after modification. To better understand the working of Tucker compression, the results are compared against the normal filter-NAS optimization technique as discussed in Sect. 2.3. The inference time and storage size of the best performing

compressed models are also analyzed and compared against the uncompressed model demonstrating the potential of TC.

Evaluation of Generic AI on MNIST

As a first test with reasonable execution times, we set up an ad hoc network consisting of only four convolutional layers (32 filters each) and trained it on the MNIST digits dataset. The output layer with 10 different classes constitutes a total parameter count of around 28000. After complete training (10 epochs) the model produces a validation accuracy of 98.6%. Decomposition is performed on all convolutional layers except the input layer and the third & fourth rank factors are controlled by NAS. The search space for these rank values is limited from [1, 32] where 32 being the highest rank value (because maximum number of filters per layer is only 32) which denotes the full rank decomposition. NAS chooses the rank values for each trial from this search space based on the tuner's suggestion after each trial. Maximum of 4000 trials are performed.

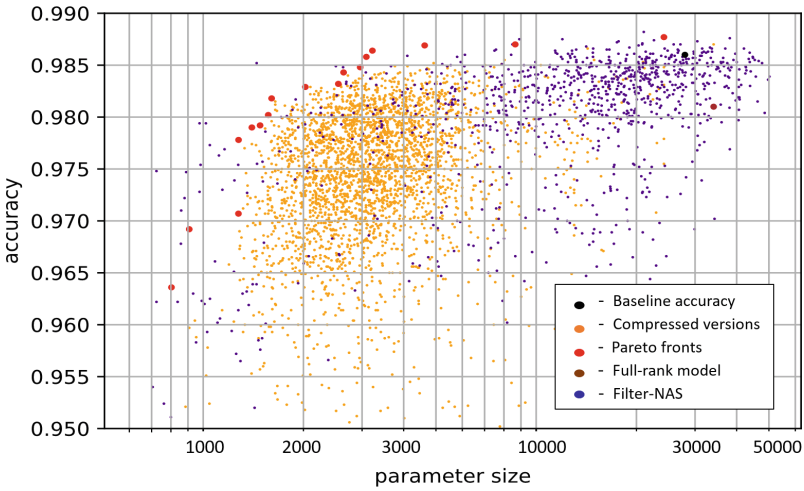

Fig. 2. Performance of TC-NAS vs. Filter-NAS for MNIST benchmark (Color figure online)

In the Fig. 2, black point represents the baseline accuracy (uncompressed model), orange points represent the performance of the compressed models at the end of fine-tuning for 3 epochs for different ranks and brown point represents decomposed version for full ranks. Red ones are the pareto points showing a good performance in the tensor compressed models with relatively lower parameter count. Parameter count refers only to convolutional parameters. There were very few points which reported below 95% of accuracy hence they are ignored and plot is clipped from 0.95 to 0.99 along the y-axis. Violet points represents the results of filter-NAS. Pareto points are drawn only for TC-NAS (orange points) and not for filter-NAS (violet points) for this and further analysis discussed in this paper.

As it can be seen from the Fig. 2, there are few models which reports top-1 accuracy of more than 98.5% with significant drop in the parameter count. The most promising compressed model which performs even better than the baseline model accounts for around 12.5% of the parameters being compressed with an increase in 0.17% of accuracy. The next good performing model reports a top-1 accuracy of 98.7%, which is 0.1% more than the baseline model with nearly 83% of the parameters being compressed. Typically, when full ranks (brown point) are applied, it is no longer compressed. In order to compress, the rank values have to be chosen accordingly such that the parameter counts are reduced.

The violet points show the results of 1000 trials (trained for 10 epochs each) of NAS choosing optimal filter counts per convolutional layer. It is observed that NAS on filter tuning seems to perform slightly better for medium to high compression rates (30%–100%). For medium compressions (10%–30%), TC-NAS performs slightly ahead than filter-NAS. For very low compression rates (below 10%), it is clear that NAS on filter tuning is better, hence we have some violet points outside of the pareto fronts (red points). Each TC versions need far less computational time since it is only fine-tuned and hence more results are obtained within a short span of time. Even though Filter-NAS produces better solutions with low parameters, they are very rare and are way further distributed with only few points being actually useful. Filter-NAS on an average takes upto 480 s training time per trial. In contrast, TC-NAS took only around 90 s of fine-tuning per trial on average. Hence, filter-NAS is extremely expensive implying that for complex models it consumes significantly huge time to find optimal solutions.

Analyzing the storage size, the uncompressed model occupied 636 KB of memory to store its weights and its inference time evaluated on Intel UHD Graphics 630 was observed to be 0.407 ms. Similarly, the compressed model which had an accuracy of 98.7% with 83% parameters being compressed exhibited an inference time of around 0.381 ms occupying a storage space of 558 KB. Since this small ad hoc network has more number of dense layer parameters than the actual convolutional parameters that are subjected to compression, the inference time and storage size is not impacted to a larger extent.

AlexNet on CIFAR-10

AlexNet was first proposed by Alex Krizhevsky, mainly for image classification problems [16] and is usually designed for ($224 \times 224 \times 3$) images. Considering our limited resources and timing constraints, our model is made to train on a relatively smaller dataset: CIFAR-10 of dimensions (32, 32, 3). AlexNet has 5 very deep convolutional layers which are subjected to compression except the first input layer. It has a total of around 3.7 million parameters. After training for 100 epochs a top-1 accuracy of 69.01% is achieved (baseline accuracy). Since 89% of the parameters in this model are composed of convolutional layers, we can have a good visualization of compression. A maximum of 4000 trials are made to run on NNI using NAS for tensor compression. Each trial is fine-tuned for 25 epochs consuming 10–12 min on average per trial. In contrast, it took 60 min on average to train the model completely for 100 epochs. The search space is

set between [1, 512], depending on the filter counts in each convolutional layer. A scatter plot is drawn between the convolutional parameters count and top-1 accuracy as shown in Fig. 3.

Just like the generic AI analysis, this study is carried out to find the potential of TC against traditional filter-NAS in finding a good optimal solution with low parameter counts. In the Fig. 3 violet points represents the filter-NAS results for 300 trials doing filter search on convolutional layers. Each filter-NAS trial are made to train for 100 full epochs. Due to time constraints, only small number of filter-NAS trials were run (each trial took around 50–60 min even on the very powerful DGX-1 AI accelerator).

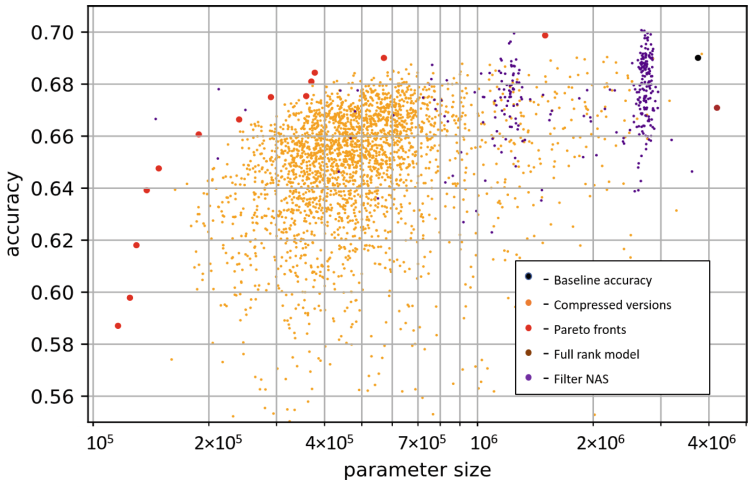

Fig. 3. TC-NAS vs. filter-NAS on AlexNet

This analysis can be concluded with the following learnings: for a compression rate of below 10%, clearly NAS on TC is the best choice as it produces good number of pareto points in that range. Also, the filter search approach seems not to produce much versions below 10% of the compression rate. For a range of compression between 10%–25%, again TC-NAS seems to outperform the filter-NAS. It can be seen from the plot as there is a cloud of points in this region produced by compressed versions, leading to more pareto fronts. On the other hand, filter-NAS produces very few points in this region.

The two blue clouds in plot Fig. 3 is due to the NAS trying to learn the optimal rank values. Exploring a particular region of search space, before the tuner moves to other values results in such clouds of points. These clouds will disappear with larger number of trials with significantly high number of performance points. For compression rates from 25% to 50%, filter-NAS seems to dominate slightly producing many optimal solutions and for compression rates of more than 50%, undoubtedly filter-NAS dominates the region. It seems to produce more solutions for greater than 50% of target compression rates.

The uncompressed model occupied a storage space of 16.6 MB with an inference time of 0.847 ms on Intel UHD graphics 630. In order to study the impact of TC on inference time and memory space requirements, one of the best performing tensor compressed models produced by NAS is picked up. The chosen model exhibits a negligible accuracy loss of 0.01% with almost 85% of the convolutional parameters being compressed. The storage space of the compressed model was found to be 3.4 MB, which is just 20% of the size required by the uncompressed model. Its inference time was evaluated to be 0.349 ms indicating that the compressed models are 2.5 times faster than uncompressed versions.

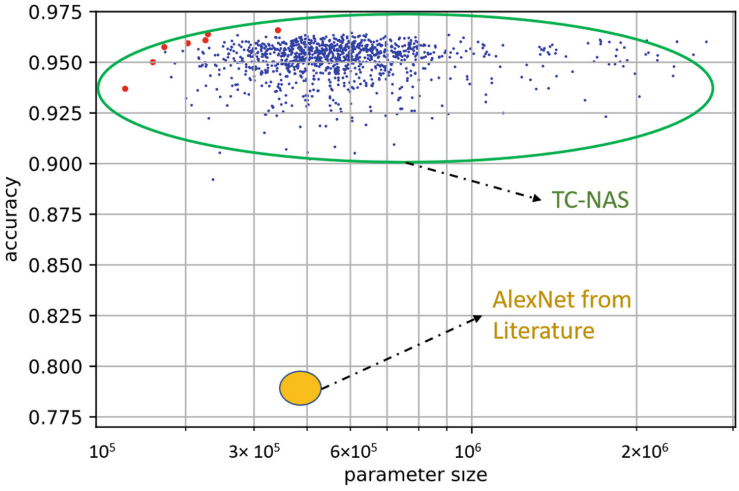

Fig. 4. TC-NAS vs. SoTA on AlexNet (Color figure online)

Figure 4 shows the comparison of our work with the current State-of-The Art (SoTA) in Tensor Compression for top-5 accuracy. Yellow circle shows the performance of AlexNet obtained from the work [11] where they have compressed all 5 convolutional layers using TD and VBMF algorithm for rank selection. For a fair analysis, we have drawn a comparison considering the parameters of 2–4 convolutional layers. It can be seen clearly that the top-5 accuracy of TC-NAS exceeds the SoTA. Even though TC-NAS is slower in finiding an optimized model compared the to SoTA, TC-NAS results has the power to provide a cluster of solutions instead of a single one.

GoogLeNet on MNIST

The original GoogLeNet architecture consists of 9 Inception modules and 2 auxiliary networks and hence it takes significantly larger training time. In order to fit it based on our available resources, the architecture is scaled down such that it is designed to have only 3 inception modules and one auxiliary network and is made to train on the MNIST dataset (tricolor images of dimensions (32, 32, 3)). There are multiple convolutional layers in these inception blocks and almost all of them are subjected to compression in NAS. Hence, the search space relatively

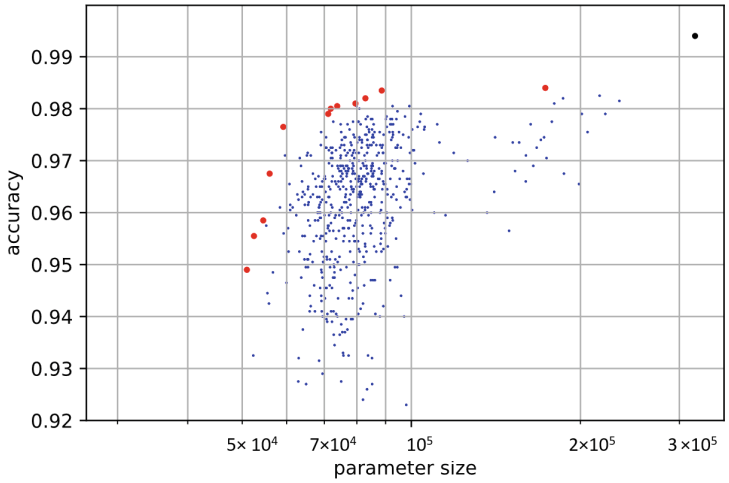

Fig. 5. TC-NAS on GoogLeNet

huge taking larger duration to converge. The search space is set between [1, 512], depending on the filter counts in each convolutional layer. After training for 20 epochs, the top-1 accuracy is around 99.4%, which is used as the baseline accuracy for comparison against different compressed versions. The uncompressed version has around 319,960 parameters in total.

Since this model has more than 10 large convolutional layers, despite running around 4000 trials for weeks together, only handful results (∼600 trials) could be obtained as shown in Fig. 5. Remaining trials were early stopped by the NAS tuner which it anticipated to have poor performance. Each successfully compressed version took around 20 min on an average for a fine-tune of 3 epochs. Filter-NAS was not performed for this model as it requires complete training for 20 epochs from scratch demanding huge computational resources.

As it can be seen from the Fig. 5, there is a big gap between the black point (baseline) and compressed versions. This gap can be filled if sufficiently more NAS trials are performed. The best performing compressed model had a compression rate of upto 40% with 1% accuracy loss, occupying a memory space of 0.8 MB. On the other hand, the size of the uncompressed model was 5.9 MB. From the analysis, it is evident that TC is capable of producing smaller models with a minimal trade-off for the performance.

6 Conclusion

Although, NAS is a very resource demanding optimization technique, combination with TC can speed up its performance by a factor of 4, reducing the need for full training. Because of high computational load, it is hard to fully evaluate the potential of TC-NAS on reasonable benchmarks - even after a month of GPU

time on very recent 100k€ machines. Another drawback is that the number of layers are increased when Tucker algorithm is applied, which may be an issue for already very deep networks. Nevertheless, we could clearly show, that we can outperform the state of the art in TC, which, to be fair executes in hours rather than month for small benchmarks resulting in optimized models with increased inference speed and reduced storage size. Instead of compressing both input and output channels (R3 and R4), trying to compress only one channel with one rank is a possible future work. This paper speaks about compression on sequential networks only. Extending this approach to non-sequential models are challenging due to the non-linearities and they are in the scope of future works.

References

1. Choudhary, T., Mishra, V., Goswami, A., et al.: A comprehensive survey on model compression and acceleration. Artif. Intell. Rev. **53**, 5113–5155 (2020). https://doi.org/10.1007/s10462-020-09816-7
2. Helms, D., Amende, K., Bukhari, S., et al.: Optimizing neural networks for embedded hardware. In: SMACD/PRIME 2021; International Conference on SMACD and 16th Conference on PRIME, pp. 1–6 (2021)
3. Han, S., Mao, H., Dally, W:J.: Deep compression: compressing deep neural networks with pruning, trained quantization and Huffman coding. Published as a Conference Paper at ICLR (oral) (2016). https://doi.org/10.48550/ARXIV.1510.00149
4. Zhou, S., et al.: DoReFa-Net: training low bitwidth convolutional neural networks with low bitwidth gradients, CoRR; abs/1606.06160
5. Uhlich, S., et al.: Mixed precision DNNs: all you need is a good parametrization. arXiv:1905.11452 (2019)
6. Yang, L., Jin, Q.: FracBits: mixed precision quantization via fractional bit-widths. arXiv:2007.02017 (2020)
7. Esser, SK., et al.: Learned step size quantization, CoRR. arXiv:1902.08153 (2019)
8. Hinton, G., Vinyals, O., Dean, J.: Distilling the knowledge in a neural network (2015). arXiv:1503.02531
9. Tucker, LR.: Some mathematical notes on three-mode factor analysis. Psychometrika **3**1, 279–311 (1966). https://doi.org/10.1007/bf02289464
10. Microsoft NNI (2021). https://nni.readthedocs.io/en/stable/index.html
11. Kim, Y.-D., Park, E., Yoo, S., et al.: Compression of deep convolutional neural networks for fast and low power mobile applications (2016)
12. Cao, X., Rabusseau, G.: Tensor regression networks with various low-rank tensor approximations (2018). arXiv:1712.09520v2
13. Alter, O., Brown, P.O., Botstein, D.: Singular value decomposition for genome-wide expression data processing and modeling. Proc. Nat. Acad. Sci. (2000). https://www.pnas.org/content/97/18/10101
14. Kim, H., Khan, M.U.K., et al.: Efficient neural network compression. arXiv:1811.12781
15. Accelerating Deep Neural Networks with Tensor Decompositions. https://jacobgil.github.io/deeplearning/tensor-decompositions-deep-learning
16. Krizhevsky, A., Sutskever, I., Hinton, G.E.: Advances in Neural Information Processing Systems. 2nd edn. Curran Associates, Inc. (2012). https://dl.acm.org/doi/10.5555/2999134.2999257