HADS: HARDWARE-AWARE DEEP SUBNETWORKS

Francesco Corti
Graz University of Technology, Austria

Balz Maag
ABB Research, Switzerland

Joachim Schauer
University of Applied Sciences, FH Joanneum, Austria

Ulrich Pferschy
University of Graz, Austria

Olga Saukh
Graz University of Technology and CSH Vienna, Austria

ABSTRACT

We propose Hardware-Aware Deep Subnetworks (HADS) to tackle model adaptation to dynamic resource contraints. In contrast to the state-of-the-art, HADS use structured sparsity constructively by exploiting permutation invariance of neurons, which allows for hardware-specific optimizations. HADS achieve computational efficiency by skipping sequential computational blocks identified by a novel iterative knapsack optimizer. HADS support conventional deep networks frequently deployed on low-resource edge devices and provide computational benefits even for small and simple networks. We evaluate HADS on six benchmark architectures trained on the GOOGLE SPEECH COMMANDS, FMNIST and CIFAR10 datasets, and test on four off-the-shelf mobile and embedded hardware platforms. We provide a theoretical result and empirical evidence for HADS outstanding performance in terms of submodels' test set accuracy, and demonstrate an adaptation time in response to dynamic resource constraints of under 40μs, utilizing a 2-layer fully-connected network on Arduino Nano 33 BLE Sense.

1 INTRODUCTION

Data processing pipelines in edge devices increasingly rely on deep learning models to identify patterns and extract insights from multimodal sensor data. However, deep models are deployed and run along with other tasks, under constraints and priorities dictated by the current context and available resources, including storage, CPU time, energy and bandwidth. Network pruning and quantization (Han et al., 2016) have become part of standard deep learning deployment pipelines, *e.g.*, TFLMicro (David et al., 2021) and TensorRT (Vanholder, 2016), to enable deep learning on severely constrained embedded hardware operated by low-power microcontrollers with only a few kB or RAM. One drawback of compile-time optimizations is that the resulting models are *resource-agnostic*: they yield suboptimal performance in many interesting applications where resource availability depends on different factors such as available energy, task priority and timing constraints. Another drawback of one-shot model compression techniques is that these are applied to the *whole model*, making exploration of different options for a resource-aware on-device model reconfiguration challenging.

Dynamic resource constraints. Many interesting applications can make use of resource-aware deep models, *i.e.*, models that can adapt their execution to available computational resources and time constraints. For example, camera image processing by a drone or a car may depend on the respective speed (Qu et al., 2022). A naive solution to address dynamic resource constraints is to store several independent deep models and switch between them as resource availability and task priorities change. The drawback of this approach is both the increased memory consumption to store these independent models which does not scale, and the overhead of switching between models at runtime. Several recent research studies recognize and address the problem in specific domains. F. Bambusi et al. (2022) propose the concept of approximate intermittent computing by designing support vector machines that can adapt to available energy while sacrificing accuracy. Subspace-configurable

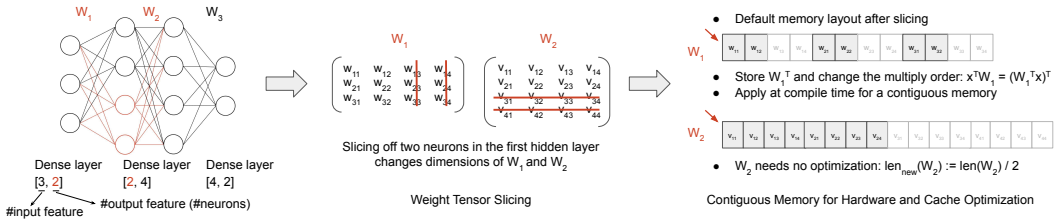

Figure 1: **Layers slicing in a fully-connected model.** The weight tensor dimensions are shown in brackets. We slice off two neurons, *i.e.*, computational units, in the first hidden layer of the network with 4 neurons, 3 inputs and 2 outputs. The matrices W_1 and W_2 store the weights along all connections, and the respective columns and rows in W_1 and W_2 get eliminated by layer slicing. This breaks the contiguous memory layout and the memory arrangement of W_1, yet not W_2. A transpose of W_1 and a change of the multiply order preserve contiguous memory of W_1 after slicing.

networks (Saukh et al., 2023) offer dynamic model reconfiguration as a resource-efficient alternative to training a static model with data augmentations. DRESS (Qu et al., 2022) and NestDNN (Fang et al., 2018) pioneered in designing nested subnetworks. However, DRESS trains nested models using a 2:4 sparsity pattern and, thus, requires specific hardware, such as NVIDIA Ampere accelerators. NestDNN converts a pre-trained model into a nested structure by applying an iterative filter pruning, growing and fine-tuning, leading to high construction overhead and complicated learning. In contrast to DRESS and NestDNN, HADS utilizes insights from Entezari et al. (2021) to preserve the model's structural density across all lower-capacity subnetworks, enabling dense matrix operations on edge devices (Howard et al., 2017). Furthermore, the structured sparsity pattern selected by HADS to build nested subnetworks is modeled through an iterative knapsack problem.

Contributions. This paper presents the design of Hardware-Aware Deep Subnetworks (HADS) featuring a nested submodel structure to address dynamic resource constraints on mobile and IoT devices. HADS introduces a novel hardware-aware method to design the nested subnetwork structure by formulating and solving an iterative knapsack problem, provide theoretical guarantees and empirical evidence of outstanding solution performance. Furthemore, we leverage permutation invariance of neurons (Entezari et al., 2021) to keep the subnetwork weight tensors in contiguous memory regions, *i.e.*, dense layers remain dense in all lower-capacity subnetworks. This makes the description of the subnetwork structure elegant, reduces HADS adaptation time, and allows for further hardware-specific optimizations. HADS is evaluated on six benchmark architectures from (Zhang et al., 2017) trained on GOOGLE SPEECH COMMANDS (Warden, 2018b), FMNIST (Xiao et al., 2017) and CIFAR10 (Krizhevsky et al., 2009) and tested on four mobile and IoT devices (Section 4). Our code is publicly available.[1]

2 HADS SUBNETWORKS STRUCTURE

HADS organizes a model into a set of *nested submodels*, *i.e.*, the active weights of a child subnetwork are fully contained in its parent subnetwork. We slice each parent subnetwork into an *active* and an *inactive* part, where the active part shapes the child subnetwork. HADS makes use of structured sparsity, *i.e.*, model slicing occurs at the level of individual neurons and convolutional filters. A tensor slice is defined as a temporary tensor object that points to the original weight tensor.

Firstly, even though any neuron can be removed from a layer, we re-order neurons to have a group of active neurons followed by inactive neurons due to the *permutation invariance* phenomenon of neural networks (Entezari et al., 2021). A permutation does not change the function of the network, but allows optimizing the memory layout to keep subnetwork weights in contiguous memory. The above observations make the subnetwork layout efficient to be stored on-device, in fact it is sufficient to store one integer value denoting the *slicing point* for each layer in each subnetwork, which corresponds to the number of active computational units. This is possible since active and inactive units build continuous groups in memory. Activating a particular subnetwork means changing the length of the dimension of the weight tensor in each layer.

[1] https://github.com/FraCorti/hads

Secondly, HADS achieves *minimal adaptation overhead*: only the widths of the layers have to be updated to switch to a different model. Switching from one HADS submodel to another requires adjusting only the sizes of the layers. Due to a local scope of a slice, the modification affects only the incoming and outgoing connections. An example of slicing dense and convolutional networks are shown in Figure 1. At inference time, the computational cost of running inference using a subnetwork is not affected by the presence of other HADS networks, in sharp contrast to the approaches that use binary masks to select active neurons or channels.

3 ITERATIVE KNAPSACK PROBLEM

Given a pre-trained model, HADS identifies how to best slice the weight tensors by formulating the problem as an iterative knapsack problem with k stages: the items included in a knapsack with capacity c have to be included in all later stages, *i.e.*, knapsacks with larger capacities (Della Croce et al., 2019). Each item, *i.e.*, computational unit of the model encoder architecture, is characterized by an importance score, denoted as I_c, and a computational cost defined in terms of model inference latency. The former is computed as $I_i = |g_i \gamma_i|$, where g_i is the sum of the accumulated gradients which approximates each unit contribution to the final prediction loss (Molchanov et al., 2019). The latter is defined as the number of multiply–accumulate operations (MACs) as the inference latency predictor for each unit on an MCUs (Liberis et al., 2021).

Given a list of MAC values obtained by multiplying the computational cost of the full model by predefined percentages, HADS solves an iterative knapsack problem with as many stages as the predefined number of subnetworks. We give two heuristic algorithms that solve the corresponding iterative knapsack problem and hence find subnetwork architectures, *i.e.*, slicing points for each layer and each subnetwork, that satisfy these capacity constraints while maximizing I to keep the most important units in each subnetwork. Both heuristics are based on solving several knapsack problems formulated as integer programs (Perron & Furnon) which we then solve by using a mixed integer programming solver (Gurobi Optimization, LLC, 2023). Due to permutation invariance, the items are stored and passed to the solver in descending order of their importance scores. This property makes the solver select items in descending order, thereby creating contiguous active and inactive units for the weight tensors in each subnetwork. Given C_{MACs} as the maximum number of MACs in a subnetwork, a single stage knapsack problem is formulated as follows:

$$
\begin{aligned}
\max \quad & \sum_{l=1}^{L} \sum_{i=1}^{u_l} x_{il} \cdot I_{il} \\
\text{s.t.} \quad & \sum_{l=1}^{L} \sum_{i=1}^{u_l} x_{il} \cdot MACs_{il} \leq C_{MACs}\,, \\
& x_{il} \in \{0,1\}, \quad \forall l \in \{1,\ldots,L\}, \;\; i \in \{1,\ldots,u_l\}, \\
& I_{i1} \geq I_{i2} \geq \cdots \geq I_{il},
\end{aligned}
\tag{1}
$$

where x_{il} is a binary decision variable taking value 1 if an item i from layer l is selected and 0 otherwise; I_{il} is the importance score of item i from layer l; $MACs_{il}$ is the number of MACs of item i from layer l; L is the number of layers in the model and u_l is the number of items in layer l. The above knapsack problem formulation is limited to the fully-connected and standard convolution architectures, its generalization for depthwise-convolution architectures (Howard et al., 2017) is moved to the Appendix A.1 for clarity of presentation.

3.0.1 BOTTOM-UP (BU) AND TOP-DOWN (TD) HEURISTICS

HADS examines two heuristics for solving the iterative knapsack problem named bottom-up (BU) and top-down (TD). The former iteratively calculates the subnetwork architectures by considering the tightest MACs constraints for the smallest subnetwork first. Once a solution is found by the knapsack solver, these units are frozen, *i.e.*, they are now part of all nested subnetworks, and the second smallest subnetwork is now being computed by the solver. The latter top-down method determines the solutions by considering the weakest MACs constraints first and then iteratively searching the architectures for increasingly smaller subnetworks. In Appendix A.2 we prove that the solution found by the two-stage bottom-up iterative knapsack heuristic is not worse than $\frac{2}{3} \cdot Opt$, where Opt is the optimal solution of the knapsack with larger capacity, and that this bound is tight. We then show that the tight bound for the top-down iterative knapsack is $\frac{1}{2} \cdot Opt$, where Opt in this

case is the optimal solution of the knapsack with smaller capacity. Since our generalized problem suited for DS-CNN architectures has the classical 0-1 knapsack as its core problem, we believe that a similar result is valid for this case as well. In other words, the bottom-up knapsack offers a better solution in the worst case than the top-down knapsack.

4 EVALUATION

4.1 HADS FINE-TUNING

HADS subnetworks found by the heuristic algorithms are fine-tuned simultaneously to recover their accuracy. For each subnetwork we slice each layer to construct a subnetwork architecture by storing only the slicing point corresponding to the number of active computational units. During fine-tuning, the weights of each subnetwork i are reused by all lower-capacity subnetworks $\{i+k\}_{k=1}^{N}$. We balance the contribution of the loss of each individual model with parameters $\{\pi_i\}_{i=1}^{N}$ equal to the percentage of weights used by the subnetwork i (Qu et al., 2022).

4.2 EVALUATION SETUP

HADS is evaluated on three pre-trained GOOGLE SPEECH COMMANDS architectures (DNN, CNN and DS-CNN) of two sizes (S and L) each introduced in Zhang et al. (2017). DNN is a 2-layer fully-connected architecture with layer width 144 (S) and 436 (L), CNN is a convolutional architecture with two convolutional layers followed by 3 fully-connected layers. The widths of convolutional layers are 28 and 30 for the S architecture and 60 and 76 for the L architecture, respectively. DS-CNN is composed of a standard convolutional layer followed by several blocks of depth-wise and point-wise convolutional layers. There are 4 blocks for the network of size S, and 5 blocks for the network of size L with a layer width of 64 and 276 respectively. The main paper presents the results for DS-CNN architectures (S and L) on GOOGLE SPEECH COMMANDS. Additional results for other architectures and datasets are available in our arXiv preprint (Corti et al., 2024).

We analyzed HADS in the low-data regime, where only few samples per class are available for fine-tuning, the performance differences between BU and TD heuristics are remarkable. Figure 2 (left) shows the performance when few-shot learning is applied to fine-tune HADS structures. Our empirical findings confirm a consistently superior performance of the BU heuristic compared to the TD alternative. The differences diminish as more samples are used for fine-tuning (Entezari et al., 2023). We observe in Figure 2 (right) consistent differences in each layer slicing points as we move from parent to child submodels. The distribution of computational units across subnetworks in each layer depends on the model architecture and dataset.

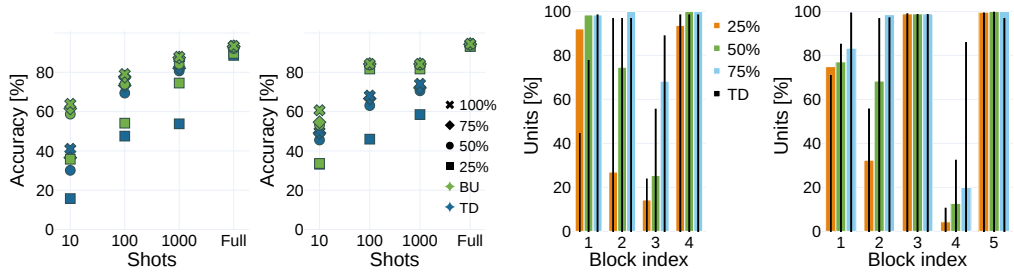

Figure 2: **Analysis of the BU and TD DS-CNN S and L subnetworks.** Few-shots finetuning accuracies (left) and structured sparsity (right) on GOOGLE SPEECH COMMANDS (Warden, 2018a).

4.3 HADS ON MOBILE AND IOT

We evaluate HADS models on two mobile phones, specifically Xiaomi Redmi Note 9 Pro and Google Pixel 6, and on two IoT devices, namely Arduino Nano 33 BLE Sense and Infineon CY8CKIT-062S2. For the mobile phone evaluation, we employed the Google Tensorflow Lite benchmarking tool and for IoT devices we evaluated the models using Edge Impulse (Hymel et al., 2022). We report the number of parameters, accuracy, and latency on the mobiles and on the IoT devices for the full architectures (100% MACs) and the subnetworks architecture (75%, 50% and 25% MACs) found by the BU knapsack solvers. The obtained results are reported in Figure 3 for the DNN, CNN and DS-CNN architectures S (top row) and L (bottom row) on GOOGLE SPEECH COMMANDS. Tensorflow Lite and TFLMicro lack support for runtime adaptation of model weight

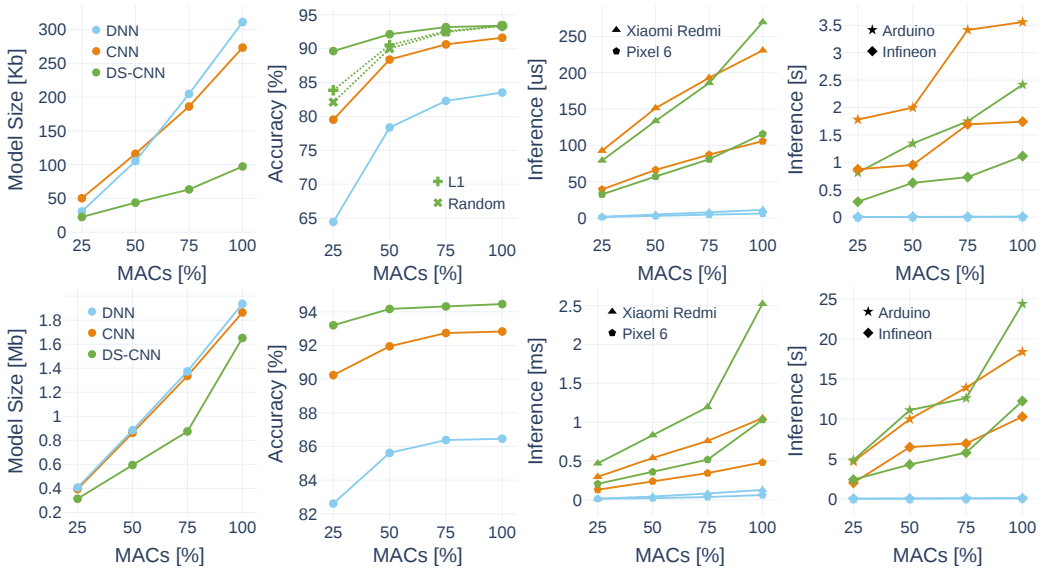

Figure 3: **HADS size S (top row) and L (bottom row) architecture analysis.** The two plots on the left show the number of model parameters and model accuracy as a function of MAC percentage in each HADS submodel. The two plots on the right evaluate model inference time on two classes of devices: Xiaomi Redmi Note 9 Pro, Pixel 6; and low-power IoT platforms including Arduino Nano 33 BLE Sense and Infineon CY8CKIT-062S2. All results are averages over three runs.

tensors. We extended the TFLMicro framework to support HADS out of the box and measure on-device inference and submodel switching times.

The first left-most column of plots in Figure 3 shows the relationship between the subnetworks' MACs and the number of model parameters. All curves are close to linear, even though the parameters of these linear relationships are architecture-specific. The DS-CNN models have the lowest number of parameters, thanks to their more sophisticated architecture. DS-CNN models also yield better accuracy, even for only 25% of MACs, while DNN models perform worst, as can be observed in the second column of plots. This result can be attributed to the successful generalization of the iterative knapsack problem to support depth-wise convolutions (Appendix A.1). We compare HADS to pruning an equal share of computational blocks in each layer (Tan & Le, 2019). We use the block selection strategy based on the L1 norm (Cai et al., 2020) and a random selection. HADS knapsack solution outperforms both baselines achieving higher accuracy for the same percentage of MACs. The last two plots on the right show HADS performance on mobile and IoT devices. We observe a difference of three orders of magnitude in the inference times between the two categories. DNN models perform best thanks to the more optimized algorithm and libraries for matrix-matrix multiplication (Goto & Geijn, 2008). All models show a linear relationship between the percentage of MACs and inference time. This empirical evidence validates MACs as a robust predictor of model latency, extending its applicability to mobile devices. On Arduino Nano 33 BLE Sense, HADS yield $38\pm1\mu$s model adaptation time for a 2-layer fully-connected network, while the model inference times for the same network with 25% and 50% of MACs are $2'131\pm27\mu$s and $4'548\pm13\mu$s respectively. This highlights the efficiency of the dense structured sparsity approach of HADS to adapt the model to dynamic constraints.

5 Conclusion, Discussion, Outlook

This paper introduces Hardware-Aware Deep Subnetworks (HADS), an approach for adapting deep neural networks to the dynamic resource constraints typically encountered on mobile and IoT devices. We formulate the design of HADS' subnetworks' structured sparsity pattern as an iterative knapsack problem, leveraging permutation invariance of neurons, and we perform a theoretical analysis of the solution space. Experimental evaluations on six benchmark architectures demonstrate HADS' effectiveness on mobile and IoT devices, showing superior performance compared to baselines. Currently, HADS does not support skip connections used in advanced architectures like MobileNet v2. This limitation, however, is expected to be addressed in future ongoing research.

ACKNOWLEDGEMENTS

The authors are grateful to Markus Gallacher for his support with the energy efficiency analysis of HADS. Christopher Hinterer and Julian Rudolf contributed to extending the TFL Micro framework to support HADS on Arduino Nano 33 BLE Sense. This research was funded in part by the Austrian Science Fund (FWF) within the DENISE doctoral school (grant number DFH 5). The results presented in this paper were computed using computational resources of HLR resources of the Zentralen Informatikdienstes of Graz University of Technology.

REFERENCES

Han Cai, Chuang Gan, Tianzhe Wang, Zhekai Zhang, and Song Han. Once-for-all: Train one network and specialize it for efficient deployment. 2020. URL https://arxiv.org/abs/1908.09791.

Francesco Corti, Balz Maag, Joachim Schauer, Ulrich Pferschy, and Olga Saukh. Reds: Resource-efficient deep subnetworks for dynamic resource constraints, 2024.

Robert David, Jared Duke, Advait Jain, Vijay Janapa Reddi, Nat Jeffries, Jian Li, Nick Kreeger, Ian Nappier, Meghna Natraj, Shlomi Regev, Rocky Rhodes, Tiezhen Wang, and Pete Warden. TensorFlow Lite micro: Embedded machine learning for TinyML systems. *Proceedings of Machine Learning and Systems*, 3:800–811, 2021.

Federico Della Croce, Ulrich Pferschy, and Rosario Scatamacchia. On approximating the incremental knapsack problem. *Discrete Applied Mathematics*, 264:26–42, 2019.

Rahim Entezari, Hanie Sedghi, Olga Saukh, and Behnam Neyshabur. The role of permutation invariance in linear mode connectivity of neural networks. *arXiv preprint*, 2021. URL https://arxiv.org/abs/2110.06296.

Rahim Entezari, Mitchell Wortsman, Olga Saukh, M. Moein Shariatnia, Hanie Sedghi, and Ludwig Schmidt. The role of pre-training data in transfer learning. 2023. URL https://arxiv.org/abs/2302.13602.

F. Bambusi et al. The case for approximate intermittent computing. In *IPSN*, pp. 463–476, 2022. doi: 10.1109/IPSN54338.2022.00044. URL https://doi.org/10.1109/IPSN54338.2022.00044.

Yuri Faenza, Danny Segev, and Lingyi Zhang. Approximation algorithms for the generalized incremental knapsack problem. *Mathematical Programming*, 198:27–83, 2023.

Biyi Fang, Xiao Zeng, and Mi Zhang. NestDNN-device deep learning for continuous mobile vision. In *Proceedings of the 24th Annual International Conference on Mobile Computing and Networking*, pp. 115–127, 2018.

Kazushige Goto and Robert A van de Geijn. Anatomy of high-performance matrix multiplication. *ACM Transactions on Mathematical Software (TOMS)*, 34(3):1–25, 2008.

Gurobi Optimization, LLC. Gurobi Optimizer Reference Manual, 2023. URL https://www.gurobi.com.

Song Han, Huizi Mao, and William J. Dally. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. 2016. URL http://arxiv.org/abs/1510.00149.

Andrew G Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. Mobilenets: Efficient cnns for mobile vision applications. 2017. URL http://arxiv.org/abs/1704.04861.

Shawn Hymel, Colby Banbury, Daniel Situnayake, Alex Elium, Carl Ward, Mat Kelcey, Mathijs Baaijens, Mateusz Majchrzycki, Jenny Plunkett, David Tischler, et al. Edge impulse: An mlops platform for tiny machine learning. 2022. URL http://arxiv.org/abs/2212.03332.

Hans Kellerer, Ulrich Pferschy, and David Pisinger. *Knapsack Problems*. Springer, 2004.

Alex Krizhevsky, Vinod Nair, and Geoffrey Hinton. Cifar-100 and cifar-10 (canadian institute for advanced research), 2009. URL http://www.cs.toronto.edu/~kriz/cifar.html. MIT License.

Edgar Liberis, Łukasz Dudziak, and Nicholas D Lane. μNAS: Constrained neural architecture search for microcontrollers. *Proceedings of the 1st Workshop on Machine Learning and Systems*, pp. 70–79, 2021.

Pavlo Molchanov, Arun Mallya, Stephen Tyree, Iuri Frosio, and Jan Kautz. Importance estimation for neural network pruning. pp. 11256–11264, 2019.

Laurent Perron and Vincent Furnon. Or-tools. URL https://developers.google.com/optimization/.

Zhongnan Qu, Syed Shakib Sarwar, Xin Dong, Yuecheng Li, Ekin Sumbul, and Barbara De Salvo. DRESS: Dynamic real-time sparse subnets. 2022. URL https://arxiv.org/abs/2207.00670.

Olga Saukh, Dong Wang, Xiaoxi He, and Lothar Thiele. Subspace-configurable networks, 2023.

Mingxing Tan and Quoc Le. Efficientnet: Rethinking model scaling for convolutional neural networks. In *ICML*, pp. 6105–6114. PMLR, 2019.

Han Vanholder. Efficient inference with tensorRT. In *GPU Technology Conference*, volume 1, pp. 2, 2016.

Pete Warden. Speech commands: A dataset for limited-vocabulary speech recognition. *arXiv preprint arXiv:1804.03209*, 2018a.

Pete Warden. Speech commands: A dataset for limited-vocabulary speech recognition. *CoRR*, 2018b. URL http://arxiv.org/abs/1804.03209.

Han Xiao, Kashif Rasul, and Roland Vollgraf. Fashion-MNIST: a novel image dataset for benchmarking machine learning algorithms. 2017. URL http://arxiv.org/abs/1708.07747.

Yundong Zhang, Naveen Suda, Liangzhen Lai, and Vikas Chandra. Hello edge: Keyword spotting on microcontrollers. 2017. URL http://arxiv.org/abs/1711.07128.

A Appendix

A.1 Knapsack for depth-wise convolutions

In this section we define a generalized knapsack framework for depth-wise convolutions. Let the input be a 2D matrix A with dimensions $m \times n$. The first layer L_0 is a standard convolutional layer applied to A that consists of N_0 filters and hence produces N_0 output channels. It is followed by depth-wise convolutional blocks $B_1, \ldots, B_d$, where each block i is built by a depth-wise layer B_i^D and a point-wise layer B_i^P for $i = 1, \ldots, d$. A depth-wise layer applies one filter on each of the N_{i-1} input channels. The point-wise layer B_i^P then applies N_i filters, where the kernel number for each filter has to be equal to the number of filters of the previous layer. From a knapsack perspective, if we want to reduce the size of the network, we can decide on how many filters we use at L_0 and at each B_i^P. For example, if we choose k filters at L_0 the layer B_1^D will have exactly k filters and one filter of B_1^P will have k as kernels number. We give an integer programming formulation for this network structure. The resulting optimization method can be used for structural pruning of a neural network, *i.e.*, for choosing an optimal number ($\leq N_0$) of filters of L_0 and (at most N_i) filters of B_i^P such that we maximize performance of the network obeying a constraint on time or space complexity, which in HADS is the number of MACs.

Formally, we introduce an integer decision variable x_0 to decide on the number of filters we use at L_0, and x_i on the number of filters to use at each B_i^P. Since every computational unit consumes

computational resources and contributes to the overall accuracy, we also introduce decision variables that control whether a unit is used by a subnetwork or not. For L_0 we introduce N_0 binary variables $y_1, \ldots, y_{N_0}$, and for every block B_i, $i = 1, \ldots, d$, we introduce binary variables f_k^i and g_{kt}^i, where $k \in \{1, \ldots, N_i\}$ and $t \in \{1, \ldots, N_{i-1}\}$, f_k^i to indicate whether a filter k is used in B_i^P, and g_{kt}^i if filter k with kernel t of B_i^P is used. For B_i^D we introduce the decision variables d_t^i to decide if a depth-wise filter $t \in \{1, \ldots, N_{i-1}\}$ is used.

In the model P^1 is the importance score of a standard convolution filter i in the first layer and W_1 is its number of MACs. For each depth-wise point-wise block i, P_t^i is the importance score for the depth-wise filter t, P_{kt}^i is the importance score of the corresponding kernel k of the point-wise filter t in the subsequent layer N_i. The problem is formulated as:

$$\max \quad \sum_{i=1}^{N_0} y_i \cdot P^1 + \sum_{i=1}^{d} \sum_{t=1}^{N_{i-1}} \left(d_t^i \cdot P_t^i + \sum_{k=1}^{N_i} g_{kt}^i \cdot P_{kt}^i \right) \quad (1)$$

$$s.t. \quad \sum_{i=1}^{N_0} y_i \cdot W_1 + \sum_{i=1}^{d} \sum_{t=1}^{N_{i-1}} \left(d_t^i \cdot W_2 + \sum_{k=1}^{N_i} g_{kt}^i \cdot W_3 \right) \leq C \quad (2)$$

$$\sum_{i=1}^{N_0} y_i = x_0 \quad (3) \quad \text{and} \quad \sum_{t=1}^{N_{i-1}} d_t^i = x_{i-1} \ \forall i \quad (4)$$

$$\sum_{k=1}^{N_i} f_k^i = x_i \ \forall i \quad (5) \quad \text{and} \quad f_k^i \geq f_{k+1}^i \ \forall i \quad (6)$$

$$g_{kt}^i \leq f_k^i \ \forall i, k, t \quad (7) \quad \text{and} \quad f_k^i \leq \sum_{t=1}^{N_{i-1}} g_{kt}^i \ \forall i, k \quad (8)$$

$$\sum_{t=1}^{N_{i-1}} g_{kt}^i \leq x_{i-1} \ \forall i, k \quad (9)$$

$$\sum_{t=1}^{N_{i-1}} g_{kt}^i \geq x_{i-1} - (1 - f_k^i) \cdot N_{i-1} \ \forall i, k \quad (10)$$

The knapsack for depth-wise convolutions is described as:

(1) The objective function maximizes the total importance of the chosen architecture.

(2) Solution MACs must comply with the constraint C.

(3) The number of filters chosen in the first convolution layer.

(4) The number of filters chosen in the depth-wise layer i must match the number of filters picked in the previous layer $i - 1$.

(5) The number of the point-wise filters chosen in layer i.

(6) Point-wise filters are chosen in ascending order to impose a contiguous solution.

(7) If kernel t of filter k is chosen then the whole filter k is chosen.

(8) A point-wise filter k is chosen only if one of its kernels is chosen.

(9) The number of kernels in the filter k of point-wise layer i must $\leq$ the number of filters taken in the previous depth-wise layer.

(10) If filter k in layer i is chosen then the constraints (9) and (10) together ensure that the number of kernels t of filter k in layer i equals the number of point-wise filters in the previous block (*i.e.*, the number of filters in the depth-wise layer i). If filter k in layer i is not chosen, constraints (9) and (10) together imply that all kernels t of filter k at layer i are zero (the right-hand side of (9) is ≤ 0, since N_{i-1} is an upper bound on x_{i-1}).

A.2 ITERATIVE KNAPSACK PROBLEM

In this section we prove that the order matters if we want to pack a knapsack iteratively. We are looking at the 2 stage iterative knapsack problem, where the items of a solution for capacity $c/2$ have to be a subset of the items of a solution for capacity c. We give our analysis and theoretical findings under the natural assumption that all items of the knapsack have weight $\leq c/2$. We first consider the bottom-up iterative knapsack heuristic and show that the quality of a worst-case solution is bounded by $\frac{2}{3} \cdot Opt$ and the bound is tight. We then analyze the top-down iterative knapsack heuristic and show that in this case a worst-case solution has a tight lower worst-case bound of $\frac{1}{2} \cdot Opt$. Approximation results for the related incremental knapsack problem were given in Della Croce et al. (2019) and Faenza et al. (2023). For a set of items I, let $P(I)$ ($W(I)$) denote the total profit (weight) of all items in I. For a binary knapsack problem (see Kellerer et al. (2004) for a general overview) we say that a *split item* I_s according to some ordering O of the items and capacity c exists, if there is an item I_s with the following property: all the items I_b^O that appear in O before I_s fulfill $W(I_b^O) < c$ and $W(I_b^O \cup I_s) > c$.

Bottom-up knapsack. Let $I(Opt_c)$ denote the optimal solution set for a knapsack of capacity c. Consider the following iterative heuristic A_c: we first find an optimal solution of the knapsack with capacity $c/2$, then we fix the selected items $I(Opt_{c/2})$ and solve the knapsack defined on the remaining items and capacity $c - W(I(Opt_{c/2}))$. We denote this second set of items as $I(A_{c/2})$. Hence, the overall item set of this heuristic is given by $I(A_c) = I(Opt_{c/2}) \cup I(A_{c/2})$. Note that there may be $P(I(A_{c/2})) > P(I(Opt_{c/2}))$, since the corresponding knapsack capacity in the second step can be larger than $c/2$.

Theorem 1. *A_c yields a worst case ratio of $\frac{2}{3}$,* i.e.*, $P(I(A_c)) \geq \frac{2}{3} \cdot P(I(Opt_c))$, if all items of the knapsack have a weight $\leq c/2$. This bound is tight.*

Proof. Let us arrange the items of $I(Opt_c)$ in the following order O: we first take all the items from $I(Opt_c) \cap I(Opt_{c/2})$ in an arbitrary order. Note that these are the items that are in both optimal solutions, *i.e.*, for both capacity c and $c/2$. Then we take all the items that are not included in $I(A_{c/2})$ followed by the items of $I(Opt_c) \cap I(A_{c/2})$ (again in arbitrary order). Now we have two cases:

Case 1: There does not exist a split item in $I(Opt_c)$ with respect to O and capacity $c/2$. Hence $W(I(Opt_{c/2})) = c/2$. It is easy to see that in this case $A_c = Opt_c$.

Case 2: Let I_s be the split item in $I(Opt_c)$ with respect to O and capacity $c/2$. In this case we get that the weight of all the items I_b^O before I_s as well as the weight of all the items I_f^O that follow I_s is smaller than $c/2$. It follows that $P(I(Opt_{c/2})) \geq P(I_b^O)$ and that $P(I(A_{c/2})) \geq P(I_f^O)$. Since all items have a weight $\leq c/2$ and by the fact that I_s is not contained in $I(Opt_{c/2})$ we know that its profit is less or equal than the minimum of $P(I(A_{c/2}))$ and $P(I(Opt_{c/2}))$. Therefore, it holds that $P(I_s) \leq \frac{1}{2} P(I(A_c))$. Hence we get:

$$\begin{aligned} P(I(Opt_c)) &= P(I_b^O) + P(I_s) + P(I_f^O) \leq P(I(A_c)) + \frac{1}{2} P(I(A_c)) \\ &= \frac{3}{2} P(I(A_c)) \end{aligned}$$

It remains to show the bound is tight. We introduce the following knapsack instance with four items and a large positive constant P.

item:	1	2	3	4
weight:	$c/3 + \epsilon$	$c/3$	$c/3$	$c/3$
profit:	$P + \epsilon$	P	P	P

Here $I(Opt_{c/2}) = \{1\}$ which only leaves space for one additional item for the larger capacity. Hence we get that $P(I(A_c)) = 2P + \epsilon$, whereas $P(I(Opt_c)) = 3P$. □

Top-down knapsack. We now consider a heuristic $D_{c/2}$ consisting of an iterative top-down knapsack packing. We first solve the knapsack with capacity c to optimality, and then solve the knapsack problem defined only on the items $I(Opt_c)$ with capacity $c/2$ to optimality. $I(D_{c/2})$ corresponds to the items in this second and smaller knapsack.

Theorem 2. *$D_{c/2}$ yields a worst case ratio of $\frac{1}{2}$,* i.e.*, $P(I(D_{c/2})) \geq \frac{1}{2} \cdot P(I(Opt_{c/2}))$ if all items of the knapsack have a weight $\leq c/2$. This bound is tight.*

Proof. Consider a knapsack of size c with optimal solution set $I(Opt_c)$ and the knapsack problem with capacity $c/2$ defined on the restricted item set $I(Opt_c)$ with solution set $I(D_{c/2})$. We will show that: $P(I(D_{c/2})) \geq \frac{1}{2} \cdot P(I(Opt_{c/2}))$.

We first arrange the items of $I(Opt_c)$ in an ordering O' such that they start with those items contained also in $I(Opt_{c/2})$. Then we identify the split item I_s according to O' for capacity $c/2$ and partition $I(Opt_c)$ into three parts. $D_1 = I_b^{O'}$, $D_2 = I_s$ and D_3 contains all the remaining items. If no split item exists, we simply set $I_s = \emptyset$. We now show that:

$$\max(P(D_1), P(D_2), P(D_3)) \geq \frac{P(I(Opt_{c/2}))}{2} \tag{2}$$

Assuming that this is not the case, we would get that:

$$\max(P(D_1), P(D_2), P(D_3)) < \frac{P(I(Opt_{c/2}))}{2}$$

This would imply

$$P(I(Opt_c)) = P(D_1) + P(D_2) + P(D_3) < P(I(Opt_{c/2})) + P(D_3).$$

However, since $I(Opt_{c/2}) \cap D_3 = \emptyset$ and $W(I(Opt_{c/2})) \leq c/2$, $I(Opt_{c/2}) \cup D_3$ would constitute a feasible solution better than $I(Opt_c)$, which is a contradiction. Thus, we have shown (2).

For $i = 1, \ldots, 3$, there is $W(D_i) \leq c/2$ and all items in D_i are available for $D_{c/2}$. Therefore, (2) implies

$$P(I(D_{c/2})) \geq \max(P(D_1), P(D_2), P(D_3)) \geq \frac{1}{2} P(I(Opt_{c/2})).$$

It remains to show the bound is tight. We introduce the following knapsack instance with four items and a large positive constant P.

Item:	1	2	3	4
Weight:	$c/3$	$c/3$	$c/3$	$c/2$
Profit:	$P + \epsilon$	$P + \epsilon$	$P + \epsilon$	2P

Here $I(Opt_c) = \{1, 2, 3\}$. $D_{c/2}$ then selects one of these items and no more items fit into the knapsack. $Opt_{c/2}$ selects item 4, which shows that the ratio of $\frac{1}{2}$ is tight. □

Note that in case that we have instances, where the weight of certain items is greater that $c/2$, it is easy to construct instances with arbitrary bad ratios for both cases.