Advancing Weight and Channel Sparsification with Enhanced Saliency Appendix

1. Difference from DST Methods (e.g. RigL)

Dynamic sparse training(DST) methods [2, 6, 13, 14, 19, 27, 28] like RigL perform growing with ***single*** mini-batch of data B by ranking *sparse* gradients over Θ_P:

$$\frac{\partial}{\partial \Theta_P} \sum_{i=1}^{|B|} \ell(f(\Theta_K \cup \Theta_P; x^i), y^i)|_{\Theta_P=\mathbf{0}}$$

this greedy technique only cares effectiveness of growing for immediate next gradient descent step. In contrast, since our importance criterion is consistent in both *Prune* and *Grow*, combining our *Reactivate & Explore* and *Grow* stages, our growing criterion could be considered as leveraging both ***"prior"*** importance information and performing ***"posterior"*** correction and adjustment based on newly selected Θ_K in the current IEE update step. For example with magnitude criterion and $Q = 1$, criterion of IEE can be reformulated as:

$$Prior(|\Theta_P|) + \frac{\partial}{\partial \Theta_P} \sum_{i=1}^{|B|} \ell(f(\Theta_K \cup \Theta_P; x^i), y^i)$$

, which considers importance from prior weights of Θ_P and also its adjusted weights in the *Reactivate & Explore* stage based on a new Θ_K selected in the current IEE step.

This offers another perspective why our method could effectively reduce greediness in exploring new sparse architectures than others. Quantitative comparisons between our growing and the previous are also provided in **Sec.4.4** of the main paper.

2. Ampere Pruning Results

With the introduction of the NVIDIA Ampere GPU, researchers in the community began to consider leveraging ampere sparsity for acceleration and compression. With $N : M$ sparsity, we sparsify N neurons out of M contiguous neurons. With a $2 : 4$ ampere sparsity, the total number of parameters in the network will be halved, but this will be slightly more structured than a non-uniformly sparsified 50% sparsity network and thus enjoy acceleration and computation saving. The proposed scheme of IEE can also be instantiated in ampere pruning scenario. In Table 1, we compare IEE with three strong latest *specialized(i.e. designed only for)* ampere pruning methods [11, 18, 30] and found that IEE beats them in Top-1 by a margin with the same $2 : 4$ sparsity and training FLOPs needed. For example, compared with SR-STE [30], IEE improves the Top-1 by 0.5 with same $2 : 4$ sparsity and $\times 0.83$ training cost. Compared with the latest SRigL [11], we surpass its performance by almost 1 point in Top-1 (**77.5** v.s. 76.6). These results again validate the generalizability of our proposed IEE scheme in ampere sparsity.

METHOD	TOP-1 ACC(%)↑	N:M	TRAIN FLOPS($\times e^{18}$)↓
ASP [18]	76.8	$2:4$	$\times 1.61$
STE [30]	76.4	$2:4$	$\times 0.83$
SR-STE [30]	77.0	$2:4$	$\times 0.83$
SRIGL [11]	76.6	$2:4$	$\times 0.83$
OURS	**77.5**	$2:4$	$\times 0.83$

Table 1. ImageNet1K N:M sparsity results with ResNet50. IEE surpasses strong latest specialized ampere pruning methods in Top-1 with the same training FLOPs needed.

3. Training FLOPs Computation

In tables presented in the paper, we demonstrate the training cost of IEE as well as other methods. FLOPs needed for a single forward pass inference of sparse model is computed by counting the total number of multiplications and additions. However, during training, the FLOPs computation would be slightly different due to different usage of the back-propagation gradients. In summary, training a neural network consists of 2 main steps which are *forward pass* and the *backward pass*. During the *forward pass*, we calculate the loss of the given batch of data using the current set of model parameters. Activations of each layer are stored in memory for the following backward pass. During the *backward pass*, we use the loss value as the initial error signal and back-propagate the error signal to calculate the gradients of parameters. We calculate respectively the gradient of the activations of the previous layer and the gradient of its parameters. Roughly, the FLOPs needed for backward pass will be ***twice*** the FLOPs needed for forward pass. Suppose a given dense architecture has forward pass FLOPs represented as ζ_D and

its pruned or sparsified model has FLOPs ζ_P. Training a sample with dense model can be expressed as $3 \cdot \zeta_D$.

IEE Each IEE step consists of three training stages, namely: *Importance Estimation*, *Accuracy Improvement*, and *Reactivate & Explore*. For each *Importance Estimation* and *Accuracy Improvement*, we need $3 \times \zeta_P$ FLOPs for both sparse forward and backward pass. For *Reactivate & Explore*, since we are training with temporarily reactivated Θ_P, we need $2 \times \zeta_P + \zeta_D$ FLOPs to take care of the dense forward pass. We still use sparse gradients for updating due to the frozen Θ_K. After the entire IEE update period, the FLOPs needed would simply be $3 \times \zeta_P$. Since the update period ends at $3/4$ of the entire training epochs, the average training cost can be calculated as:

$$\frac{3}{4} \cdot \frac{(H+J) \cdot 3 \cdot \zeta_P + Q \cdot (2 \cdot \zeta_P + \zeta_D)}{H+J+Q} + \frac{1}{4} \cdot 3 \cdot \zeta_P$$

With $H = J = Q$, the cost would be:

$$\frac{11 \cdot \zeta_P + \zeta_D}{4}$$

This would be slightly higher than completely training a sparse model from scratch which is $3 \cdot \zeta_P$ but still substantially lower than dense model training cost ($3 \cdot \zeta_D$). Also notice that, according to our above description of IEE with structured sparsity, we follow the exponential scheduler of HALP [25], and the update period ends much earlier than $3/4$ of the total training epochs. The update with IEE for latency-constrained structured sparsity will instead end at the 5th epoch. The average training cost of IEE will also be much lower. With 130 training epochs in total, according to the calculation we provide above, it will instead be:

$$\frac{5}{130} \cdot \frac{(H+J) \cdot 3 \cdot \zeta_P + Q \cdot (2 \cdot \zeta_P + \zeta_D)}{H+J+Q} + \frac{125}{130} \cdot 3 \cdot \zeta_P$$

With $H = J = Q$, the cost would approximately be:

$$\frac{388.3 \cdot \zeta_P + 1.7 \cdot \zeta_D}{130}$$

SOFT MASKING Now for the family of soft masking methods like SNFS [4], DPF [12], and DCIL [9], training cost vary based on different methods. Since these methods typically maintain dense gradients during backpropagation, training cost would usually be noticeably higher than typical sparse training approaches. For SNFS [4], the total number of training FLOPs scales with $2 \cdot \zeta_P + \zeta_D$. For DCIL [9], the work requires two forward and backward passes each time to measure two sets of gradients(one with dense weight and one with sparse weight) for weights update, and the total number of training FLOPs scales with $5 \cdot \zeta_D + \zeta_P$, which is nearly doubled dense model training cost ($6 \cdot \zeta_D$).

ZERO-SHOT PRUNING For the family of static sparse training or zero-shot pruning, the cost can be expressed as $3 \cdot \zeta_P$ for both sparse forward and backward pass.

PRUNING FROM PRETRAINED Most of the pruning from pretrained methods nowadays employed iterative pruning. For simplicity here, we estimate a *very loose theoretical lowerbound* assuming one-shot pruning and no further gradients calculation on the pruned parameters during finetuning. The training cost of pretrained dense model scales with $3 \cdot \zeta_D$ as discussed. In the later finetuning stage, the cost would scale with $3 \cdot \zeta_P$ since the model deals with a sparse model now.

RIGL [6] For the representative state-of-the-art dynamic sparse training work RigL, iterations with no connections updates need $3 \cdot \zeta_P$ FLOPs. At every ΔT iteration, RigL calculates the dense gradients. The averaged FLOPs for RigL is given by $\frac{3 \cdot \zeta_P + 2 \cdot \zeta_P + \zeta_D}{\Delta T + 1}$.

GRANET [13] The difference between GraNet and RigL is at the starting sparsity of the method. RigL, same as our IEE, starts from a sparse model of the target sparsity; whereas for GraNet, they start from a denser model of smaller sparsity. The best result from their paper, also shown in our main paper, starts at 50% ERK sparsity(5.8 FLOPs). However, the reported training FLOPs does not take the denser model pretraining into account. We explain here how we correct the training FLOPs calculation. In the first 30 epochs, as described by GraNet [13], they gradually prune to the target sparsity, and the final model has 3.0 FLOPs. We compute the average FLOPs in the first 30 epochs simply as $(3.0 + 5.8)/2 = 4.4$ and use it as ζ_P for the first 30 epochs and 3.0 as ζ_P for the remaining training epochs.

MEST, SPFDE [27, 28] We just use the reported training FLOPs in the paper. However, notice that these two methods, besides sparse training, also leverage orthogonal augmentation techniques like data sieving and layer freezing to additionally reduce training costs. In IEE, we only perform sparse training as in RigL and others for a fair comparison.

INTERSPACE PRUNING For the very latest interspace pruning work [26], authors use FB convolution layers which introduce additional forward and backward overhead. Given the information provided in the paper, for a particular convolution layer with size $c_{out} \times c_{in} \times K \times K$, the relative increase of forward pass would be K^2/c_{out} times the dense forward pass. Notice that this is a constant overhead independent of the pruning rate and sparsity of the model. Similarly, the authors provide that the backward pass would introduce an additional constant overhead of K^2/c_{in} times the dense computation of gradients. Since authors provide no exact FLOPs of the model, we also estimate a lower bound of K^2/c_{out} and K^2/c_{in} as $3^2/128 \approx 0.07$ for

ResNet-50. This is a lower bound since as identified in many works before the spatial size is the largest in the early layers with a large K and small c_{out} and c_{in} processing large-sized feature maps and dominating the overall FLOPs of the model. Now we could calculate the FLOPs needed to train a single example as $\zeta_P + 0.07 \cdot \zeta_D + 2 \cdot (\zeta_P + 0.07 \cdot \zeta_D)$ which is approximately $3 \cdot \zeta_P + 0.21 \cdot \zeta_D$.

NAS-BASED METHODS We also demonstrate the results of some NAS-based methods [5, 7, 16] in the main paper table for comparison. Since the searching involved is very hard to quantify the training cost estimation, we only report the estimated training cost of the discovered pruned model calculated as $(3 \cdot \zeta_P)$. Now notice that this is a very loose lower bound, and the actual cost could be much higher with the architecture search.

4. Integration of IEE with Latency-Constrained Structured Sparsity

We now present IEE with latency-constrained structured sparsity setting. Specifically, we will highlight how we integrate with the latest latency pruning method HALP [24].

4.1. Recap of HALP and Latency-Constrained Pruning

For our latency-constrained structured sparsification, we follow the latest resource-constrained pruning method HALP [25] but impose the dynamic regime of IEE to enhance the quality of the pruned model structure. Same as HALP, we formulate the pruning step as a global cost-constraint importance maximization problem, where we take into account the latency benefits incurred every time we remove or grow a channel from one of the layers of the network. Similarly, we also formulate our unique growing part as a cost-constraint importance maximization problem. In this section, we will provide a brief recap of HALP and how it's used our IEE iterative prune-and-grow setup. Given a global resource constraint Ψ defining the maximum amount of resource we could use, HALP aims to find a set of channels defining a sub-network achieving the best performance under the constraint Ψ. In this case, Ψ represents the inference latency for a target hardware platform, for example the Nvidia TitanV GPU.

HALP then prepare a latency lookup table $\mathcal{T}$, where $\mathcal{T}^l(p^{l-1}, p^l)$ records the layer latency at layer l with p^{l-1} active input channels and p^l active output channels. With this latency look-up table, HALP associates a potential latency reduction value R_j^l to each jth channel of layer l, computed as follows:

$$R_j^l = T^l(p^{l-1}, j) - T^l(p^{l-1}, j-1), 1 \leq j \leq p^l \quad (1)$$

R_j^l estimates the potential latency saving if we prune the corresponding channel. Now, in order to estimate the performance of the selected sub-newtwork, HALP measures the importance score $\mathcal{I}_j^l$ for each jth channel of layer l. The importance score metric adopted here is Taylor importance [20], which is evaluated as follows:

$$\mathcal{I}_j^l = |g_{\gamma_j^l} \gamma_j^l + g_{\beta_j^l} \beta_j^l|, \quad (2)$$

where γ and β are the BatchNorm layer's weight and bias. With R and $\mathcal{I}$ calculated, HALP formulates the latency-constrained channel pruning as a Knapsack problem where we try to maximize the total importance but under the latency constraint Ψ. The pruned channels can then be selected by an augmented Knapsack solver $Knapsack(\mathcal{I}, R, \Psi)$, which returns the items achieving maximum importance while the accumulated latency cost is below the global constraint Ψ.

4.2. IEE Prune Step

As described in the main paper, in the *Prune* phase of the t-th IEE step, we are going to prune a number of parameters that satisfy the "update budget" Ω^t. For integration with the HALP framework, we first collect the importance $\mathcal{I}$ and channel latency cost R from the **active sparse structure** Θ_K. Since the initialized compute resource is Ψ, after the Prune step, our desired target is $\Psi - \Omega^t$. We then leverage the Knapsack solver $Knapsack(\mathcal{I}, R, \Psi - \Omega^t)$ to choose which channels to transfer from Θ_K to Θ_P.

4.3. IEE Grow Step

During growing, we also want to take the model latency into account to prevent some latency-costly channels from getting added back. We perform a similar latency-constrained selection as above. Here, we collect the importance $\mathcal{I}$ and channel latency cost R from the **exploration space** Θ_P after our *Reactive & Explore* step. Then, we grow a number of parameters that satisfy the "update budget" Ω^t. Channels selected by the Knapsack solver $Knapsack(\mathcal{I}, R, \Omega^t)$ are transferred from Θ_P to Θ_K.

5. Detailed Experiment Hyperparameter and Optimization Settings

The large-scale image classification dataset ImageNet [3] is of version ILSVRC2012 [23], which consists of $1.3M$ images of 1000 classes. We run all experiments on ImageNet and PASCAL VOC with eight NVIDIA Tesla V100 GPUs. Experiments on CIFAR10 [10] are conducted with a single NVIDIA Tesla V100 GPU. All experiments are conducted with PyTorch [22] V1.4.0.

5.1. Layerwise Sparsity Distribution

In the main paper, in our unstructured sparsity setting, we demonstrated results of IEE and comparison with three

types of sparsity distribution, namely: Uniform, ERK, and Non-Uniform. Here, to clear possible confusion, we provide more detailed explanations here. Given a predefined sparsity S or a number of available neurons, we have different ways to allocate them across layers, which also results in different FLOPs. With Uniform sparsity, the sparsity of each layer S^l is equal to the total sparsity S throughout training, i.e., $S = S^l$. With ERK sparsity, we use the *Erdős-Rényi-Kernel* (ERK) formulation [6, 19] to set sparsity for each layer, which means higher sparsity is assigned to those layers with more parameters i.e., $S^{l_i} > S^{l_j}$ if $m^{l_i} > m^{l_j}$, where m^{l_i} represents the number of parameters for layer l_i. With Non-Uniform sparsity, we do not pose any constraints for layerwise distribution. Concretely in pruning, with Non-Uniform sparsity, we simply rank all neurons in the model globally. Though all Uniform, ERK, and Non-Uniform are *unstructured* sparsity, Uniform is slightly more *structured* than ERK which is also naturally more *structured* than Non-Uniform.

5.2. Unstructured Weight Sparsity on ResNet50-ImageNet

We use an individual batch size of 128 per GPU and follow NVIDIA's recipe [21] with mixed precision and Distributed Data Parallel training. The learning rate is warmed up linearly in the first 8 epochs reaching its highest learning rate then follows a cosine decay [17] over the remaining epochs. The pretrained model weight is kept consistent with RigL [6] to ensure a fair comparison.

5.3. Unstructured Weight Sparsity on WideResNet22-2-CIFAR10

In our experiments section, we also include results of WideResNet22-2, which is Wide Residual Network [29] with 22 layers using a width multiplier of 2. We use an individual batch size of 128, an initial learning rate of 0.1 decaying by a factor of 5 every 30000 iterations, an L2 regularization coefficient of $5e-4$, and a SGD momentum of 0.9. Similarly, results of RigL are reproduced using the same hyperparameters and optimization settings as ours, ensuring a fair comparison.

5.4. Latency Constrained Structured Sparsity

ImageNet We follow HALP [25] for setting the hyperparameters and optimization settings of experiments on latency constrained structured sparsity with ResNet50 and MobileNet-V1. They are also similar to the recipe described in 5.2. We also follow HALP [25] for constructing the latency lookup table, which is pre-generated targetting the NVIDIA TITAN V GPU inference by iteratively reducing the number of channels in a layer and characterize the corresponding latency with NVIDIA cuDNN [1] V7.6.5. The latency measurement is conducted 100 times to avoid randomness. We also refer to HALP for some special implementation detail such as how to deal with the group convolution in MobileNet-V1, negative latency contribution, pruning of the first model layer, which are all described in detail in HALP.

PASCAL VOC We follow the "07 + 12" setting as in [15] and use the union of VOC2007 and VOC2012 trainval as our training set and VOC2007 test as test set. Our SSD model, similar to HALP [25], is based on [15]. Following [8], for efficiency, we remove the last stage of convolution layers, last avgpool, and fc layers from the original ResNet50 classification structure. Also, all strides in the third stage of ResNet50 layer are set to 1×1. We train our models for 900 epochs with SGD optimizer and learning rate schedule same as [25] with an initial learning rate of $8e-3$ which warms up in the first 50 epochs then decays by $3/8, 1/3, 2/5, 1/10$ at the $700, 800, 840, 870$th epoch

References

[1] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. cudnn: Efficient primitives for deep learning. *arXiv preprint arXiv:1410.0759*, 2014. 4

[2] Xiaoliang Dai, Hongxu Yin, and Niraj K Jha. Nest: A neural network synthesis tool based on a grow-and-prune paradigm. *IEEE Transactions on Computers*, 68(10):1487–1497, 2019. 1

[3] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *CVPR*, pages 248–255. Ieee, 2009. 3

[4] Tim Dettmers and Luke Zettlemoyer. Sparse networks from scratch: Faster training without losing performance. *arXiv preprint arXiv:1907.04840*, 2019. 2

[5] Xuanyi Dong and Yi Yang. Network pruning via transformable architecture search. *NeurIPS*, 32, 2019. 3

[6] Utku Evci, Trevor Gale, Jacob Menick, Pablo Samuel Castro, and Erich Elsen. Rigging the lottery: Making all tickets winners. In *ICML*, pages 2943–2952. PMLR, 2020. 1, 2, 4

[7] Shaopeng Guo, Yujie Wang, Quanquan Li, and Junjie Yan. Dmcp: Differentiable markov channel pruning for neural networks. In *CVPR*, pages 1539–1547, 2020. 3

[8] Jonathan Huang, Vivek Rathod, Chen Sun, Menglong Zhu, Anoop Korattikara, Alireza Fathi, Ian Fischer, Zbigniew Wojna, Yang Song, Sergio Guadarrama, et al. Speed/accuracy trade-offs for modern

convolutional object detectors. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 7310–7311, 2017. 4

[9] Jangho Kim, Jayeon Yoo, Yeji Song, KiYoon Yoo, and Nojun Kwak. Dynamic collective intelligence learning: Finding efficient sparse model via refined gradients for pruned weights. *arXiv preprint arXiv:2109.04660*, 2021. 2

[10] Alex Krizhevsky, Geoffrey Hinton, et al. Learning multiple layers of features from tiny images. 2009. 3

[11] Mike Lasby, Anna Golubeva, Utku Evci, Mihai Nica, and Yani Ioannou. Dynamic sparse training with structured sparsity. *arXiv preprint arXiv:2305.02299*, 2023. 1

[12] Tao Lin, Sebastian U Stich, Luis Barba, Daniil Dmitriev, and Martin Jaggi. Dynamic model pruning with feedback. *ICLR*, 2020. 2

[13] Shiwei Liu, Tianlong Chen, Xiaohan Chen, Zahra Atashgahi, Lu Yin, Huanyu Kou, Li Shen, Mykola Pechenizkiy, Zhangyang Wang, and Decebal Constantin Mocanu. Sparse training via boosting pruning plasticity with neuroregeneration. *NIPS*, 2021. 1, 2

[14] Shiwei Liu, Lu Yin, Decebal Constantin Mocanu, and Mykola Pechenizkiy. Do we actually need dense over-parameterization? in-time over-parameterization in sparse training. In *ICML*, 2021. 1

[15] Wei Liu, Dragomir Anguelov, Dumitru Erhan, Christian Szegedy, Scott Reed, Cheng-Yang Fu, and Alexander C Berg. Ssd: Single shot multibox detector. In *ECCV*, pages 21–37. Springer, 2016. 4

[16] Zechun Liu, Haoyuan Mu, Xiangyu Zhang, Zichao Guo, Xin Yang, Kwang-Ting Cheng, and Jian Sun. Metapruning: Meta learning for automatic neural network channel pruning. In *ICCV*, pages 3296–3305, 2019. 3

[17] Ilya Loshchilov and Frank Hutter. Sgdr: Stochastic gradient descent with warm restarts. *arXiv preprint arXiv:1608.03983*, 2016. 4

[18] Asit Mishra, Jorge Albericio Latorre, Jeff Pool, Darko Stosic, Dusan Stosic, Ganesh Venkatesh, Chong Yu, and Paulius Micikevicius. Accelerating sparse deep neural networks. *arXiv preprint arXiv:2104.08378*, 2021. 1

[19] Decebal Constantin Mocanu, Elena Mocanu, Peter Stone, Phuong H Nguyen, Madeleine Gibescu, and Antonio Liotta. Scalable training of artificial neural networks with adaptive sparse connectivity inspired by network science. *Nature communications*, 2018. 1, 4

[20] Pavlo Molchanov, Arun Mallya, Stephen Tyree, Iuri Frosio, and Jan Kautz. Importance estimation for neural network pruning. In *CVPR*, pages 11264–11272, 2019. 3

[21] NVIDIA. Nvidia. convolutional networks for image classification in pytorch. 4

[22] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch. *NeurIPS Workshop*, 2017. 3

[23] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, et al. Imagenet large scale visual recognition challenge. *IJCV*, 115(3):211–252, 2015. 3

[24] Maying Shen, Lei Mao, Joshua Chen, Justin Hsu, Xinglong Sun, Oliver Knieps, Carmen Maxim, and Jose M Alvarez. Hardware-aware latency pruning for real-time 3d object detection. In *2023 IEEE Intelligent Vehicles Symposium (IV)*, pages 1–6. IEEE, 2023. 3

[25] Maying Shen, Hongxu Yin, Pavlo Molchanov, Lei Mao, Jianna Liu, and Jose Alvarez. Structural pruning via latency-saliency knapsack. In *Advances in Neural Information Processing Systems*, 2022. 2, 3, 4

[26] Paul Wimmer, Jens Mehnert, and Alexandru Condurache. Interspace pruning: Using adaptive filter representations to improve training of sparse cnns. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 12527–12537, 2022. 2

[27] Geng Yuan, Yanyu Li, Sheng Li, Zhenglun Kong, Sergey Tulyakov, Xulong Tang, Yanzhi Wang, and Jian Ren. Layer freezing & data sieving: Missing pieces of a generic framework for sparse training. In *NIPS*, 2022. 1, 2

[28] Geng Yuan, Xiaolong Ma, Wei Niu, Zhengang Li, Zhenglun Kong, Ning Liu, Yifan Gong, Zheng Zhan, Chaoyang He, Qing Jin, et al. Mest: Accurate and fast memory-economic sparse training framework on the edge. *NIPS*, 2021. 1, 2

[29] Sergey Zagoruyko and Nikos Komodakis. Wide residual networks. In *BMVC*. British Machine Vision Association, 2016. 4

[30] Aojun Zhou, Yukun Ma, Junnan Zhu, Jianbo Liu, Zhijie Zhang, Kun Yuan, Wenxiu Sun, and Hongsheng Li. Learning n: m fine-grained structured sparse neural networks from scratch. *ICLR*, 2021. 1