Pruning and Accelerating Fine-Structured Sparsity

Charlie Tan*
Department of Computer Science
University of Cambridge
{ct632}@cam.ac.uk

Sepand Dyanatkar*
Department of Computer Science
University of Cambridge
{sd974}@cam.ac.uk

Abstract

Sparsity combined with efficient arithmetic over matrices is a long sought-after ideal in the machine learning world. However, due to structured, rigid design, most hardware accelerators struggle to optimize for unstructured sparsity. Fine-structured sparsity is an alternative that has provided potential for GPU speedup while also presenting the benefits of sparsity. In particular, NVIDIA recently released Sparse Tensor Cores supporting mid-range sparsity in the range of 50%, with up to a 2x speedup. We investigate the speedup provided by Sparse Tensor Cores and procedures for pruning networks of suitable structure. Our code is available https://github.com/char-tan/sparsity.

1 Introduction

Deep learning is a branch of machine learning, characterised by deep neural networks. Deep learning has been applied to great success in numerous application areas, including image classification [10], machine translation [28], and audio classification [15] Pruning is a method wherein parameters are removed (set to zero) to sparsify a neural network [5]. Pruning reduces the number of parameters required to represent the neural network, and reduces the number of operations required to process inputs. However, with most deep learning being conducted on GPUs - hardware specialised for parallel computation - pruning does not necessarily reduce inference latency until high sparsity levels are achieved.

The recent introduction of Sparse Tensor Cores in the NVIDIA Ampere architecture offers another solution for hardware acceleration of sparse neural networks [21]. Notably, this hardware is designed specifically for 50% sparsity in a row-structured format, allowing efficient memory access and paralellisation. This level of sparsity is much lower than required to get a performance benefit using classical sparse algorithms, making this an exciting area of research. This is the motivation of our project, in which we investigate the speed-up achieved by these Sparse Tensor Cores over dense matrix multiplication, and different methods for training networks with the desired fine-structured sparsity.

The key contributions of our work are listed as follows:

1. We benchmark the Sparse Tensor Cores versus existing matrix multiplication algorithms.
2. We evaluate the lottery ticket hypothesis as a method for producing fine-structured sparse neural networks, finding it to be successful but marginally inferior to the method of [21].
3. We reproduce results for channel permutations, finding them to not offer benefit on our task or architectures.

Motivated by the Tensor Core sparsity support we investigate different methods for training 2:4 sparse neural networks. In particular, we investigate the performance of 2:4 sparse lottery tickets compared

*These authors contributed equally to this work

Preprint. Under review.

to the continued training methodology employed by [21]. We further compare with re-initialising the network before masking, with the lottery ticket hypothesis indicating such models should be inferior to lottery tickets [16]. We additionally evaluated the efficacy of channel permutations in mitigating the accuracy reduction of pruning [22].

2 Background

2.1 Hardware Acceleration

Speeding up math operations have been an important component of the machine learning boom in the past 2 decades, since models have required increasing computational resources to train and validate. Much of the computations have been structured and parallelizable, and thus a focus for acceleration. In particular are matrix (or more generally tensor) multiplication operations which must be computed for nearly all machine learning applications, and general scientific computation. Basic matrix-matrix multiplication algorithms operate on the $O(N^3)$, but both numerical and analytical approaches have been found to reduce the power to below 2.5, albeit lower values keeping significant overhead due to the hidden constant factor [2, 27, 3]. Strassen's algorithm is a cornerstone algorithm in the early days of optimizing matrix multiplication [1]. In general, the space of matrix multiplication algorithms has been shown to be vast [25], and an present ongoing challenge to achieve useful speedup using improved algorithms alone.

Important implementation of modern methods for accelerated matrix multiplication are using the Basic Linear Algebra Subroutines library and the associated General Matrix Multiply (GEMM) method [6], and subsequent extensions to hardware accelerators.

Hardware accelerators have been found to fit the parallel and structured nature of matrix multiplication very effectively, to the point at which many operations can be simultaneously computed. As a result, graphical processing units (GPU) and similar devices are notable examples that have grown in popularity for scientific computing involving matrix multiplications. However, these may not be necessarily need or being very compatible with traditional accelerated algorithms, and thus require additional adaptation.

Matrix multiplication is at the heart of machine learning, and due to our speed-up capabilities it presents a convenient medium for executing math operations in machine learning. In important example is the adaptation of matrix-matrix multiplication for convolutions using *im2col* methods [7]. Other notable examples exist across all architecture types, including linear layers and attention heads [28].

Bringing together algorithmic and hardware acceleration acceleration is a critical part of the current state of accelerating machine learning. In particular, sparse matrices present fruitful opportunities for optimizations. Pruning and clamping present effective methods for inducing this reduction in size and compute requirements [4]. However, despite pruning reducing the number of parameters, this does not guarantee a reduction in parameter storage requirements, as they require adapted data structures (essentially evolved forms of adjacency lists) [8]. For example the coordinate list (COO) compressed matrix format requires 3 values per non-zero value, with compressed sparse row format offering a moderate reduction. Evidently, sparsity in excess of 50% will be required to manifest a reduction in storage requirements.

Achieving speedups for inference and training with matrix multiplication is an area which has gained traction to provide deployment benefits to pruning and sparse learning theory. NVIDIA has developed a solution for fine-structured sparsity which is the area in which we are interested in, in the form of a *Sparse Tensor Core* hardware acceleration solution which we investigate [21]. In essence, they use low level multiplexing (MUX) and predetermined matrix sparsity patterns to initiate up to 2x speedup of GEMM (see Figure 1). This is currently available via the proprietary NVIDIA API cuSPARSELt.

2.2 Neural Network Pruning

Neural network pruning has been an active area of research for a number of decades. The motivations for pruning include; reducing the storage requirements, reducing the computational cost of inference, and mitigating the potential for over-fitting. Early methods employed the second-order derivatives to determine which parameters to prune [5]. A commonly used approach identifies parameters

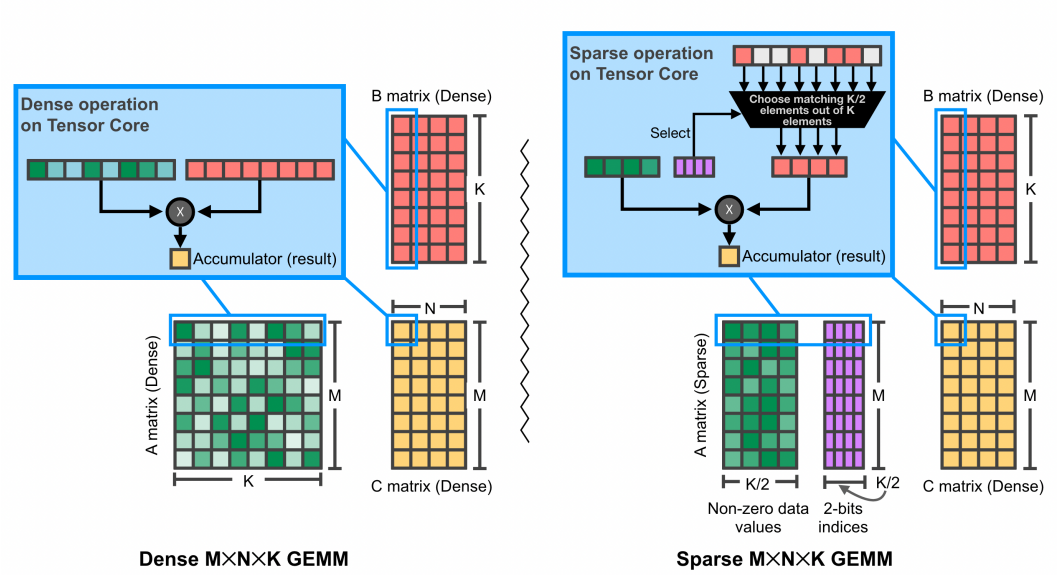


Figure 1: **Sparse Tensor Core acceleration strategy for GEMM.** (Adapted from [21])

through their magnitudes, with the intuition that smaller magnitude weights have less influence on the activation produced [**han**]. Another proposed method uses the empirical fisher information to determine the sensitivity of the output distribution to the parameters [14].

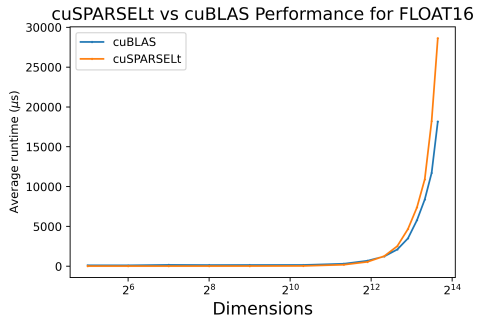
Until recently, it was believed that training a pruned neural network from random initialisation would result in inferior performance [12]. However, the work of Carbin et. al demonstrated this is to be untrue with the Lottery Ticket Hypothesis [16]. Their methodology involves iteratively training a network, pruning the parameters, and applying this pruning mask to re-train the network from its original initialisation. The requirement of retraining multiple times to achieve the higher levels of sparsity incurs a large computational cost, motivating research into efficient methods for identifying 'winning' lottery tickets [26]. However, later work questioned the optimally of lottery ticket methods on larger datasets, in favour of steadily increasing sparsity during training [17].

Often, the objective of pruning is to increase the efficiency of a network. However, many pruning methods induce *unstructured* sparsity; this lack of structure presents obstacles for hardware acceleration. This has motivated research into structured pruning. Examples include channel pruning [12, 13], node pruning [19], and block pruning [24]. The support of sparsity in NVIDIA Ampere Tensor Core has increased interest in structured sparsity. Mishra et. al conducted a comprehensive study of 2:4 sparsity in deep learning, applying a two-phase procedure in which a network is trained from initialisation, pruned, and then continued to train from these pruned parameters without returning to initialisation [21]. A further enhancement can be made by permuting the rows and columns of the weight matrices to minimise the pruning of large (and potentially important) parameters [22]. Another work applied a straight through estimator to learn 2:4 sparse neural networks [23].

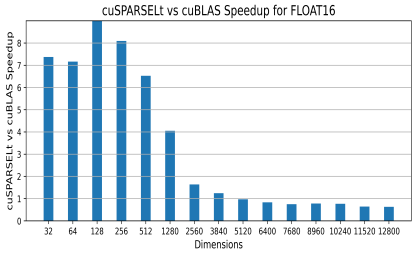
3 cuSPARSELt and Sparse Tensor Cores

3.1 Methodology

We first investigated the area of hardware accelerated *fine-structured sparse* matrix multiplication. This would enable us to interface with accelerated training and inference and to adapt it for arbitrary models and architectures. We chose to pursue 2:4 sparsity on NVIDIA's Ampere architecture, and the underlying Sparse Tensor Core design. Since it is a proprietary architecture with its implementations and source code being closed source, we were required to use the cuSPARSELt interface in order to access it. We did not have free interactive access with an NVIDIA A100 GPU for development, thus had to buy Google Colab Pro subscriptions to gain access through their Python kernel service.



(a) Overlapping performance



(b) Speedup comparison

Figure 2: **Comparison of the performance of cuSPARSELt and cuBLAS interfaces for 2:4 sparse matrix multiplication.** cuSPARSELt is designed to take advantage of Sparse Tensor Core multiplexing capabilities for fine-structured sparse matrices whereas cuBLAS is only utitlizing existing sparse matrix optimizations for regular Tensor Cores. We see that the performance gain is not visible through our experiments, in particular with behaviour trending just under 1x as fast for cuSPARSELt. We believe this is due to the additional optimizations for Sparse Tensor cores and additional cuSPARSELt specific steps which are taken within the corresponding. Error bars are too small to be shown.

The cuSPARSELt library is a basic environment introduced by NVIDIA around their release of the Sparse Tensor Core with the NVIDIA Ampere series. The core acceleration is provided by a multiplexing operation done at a low level 2:4 sparse matrices. Figure 1 illustrates the multiplexer operation and compares to traditional GEMM. The A matrix must be sparsified in a precise sparsity pattern where at most 2 of each 4 consecutive row elements must be zero. The metadata describing the pattern of sparsity for each row is then able to represented with 2-bits per group of 4 elements. This saves at least 44% for an INT8 representation, or 47% for FLOAT16 operations.

Our initial attempt at implementing a solution for benchmarking was through a PyTorch CUDA/C++ Extension. This would allow us to adequately make changes to the code, while still interacting effectively through the Colab environment and potentially take advantage of the PyTorch deep learning library [18]. Unfortunately, this failed due to a multitude of challenges with the extension framework and linking together with the cuSPARSELt library.

Instead, we worked to compile a third party library for integrating accelerations in Python named *cupy* with the cuSPARSELt as a backend. With this, we were able to access and use minimum set of the cuSPARSELt API, and run several benchmarks. cuBLAS's GEMM was our benchmarking comparison to make, which was natively integrated into the *cupy* library. We used floating point 16-bit data types, across numerous dimensions selected to overlap with the NVIDIA benchmarks and gave equal warmup and recorded runs for each experiment [21].

3.2 Results

The Sparse Tensor Core accelerated results can be seen in Figure 2. Overall, we do not see the speedup as expected from NVIDIA's publications. We tried modifying our experimental procedures to ensure we purely timed the actual matrix multiplication calls make to appropriate libraries, and increasing experimental runs. In the lower range of dimensions, cuSPARSELt was able to outperform the regular cuBLAS library. In the higher range of dimensions, we observe a degradation of performance, with cuSPARSELt dropping to almost 60% of the cuBLAS performance for the same parameters.

Within our resources, we also ran into errors when testing higher dimension ranges on the Colab machines, even while freeing memory when no longer needed. The library integration in *cupy* also prevented us from using different data types, exposing only cuSPARSELt's floating point 16 and integer 32 compute types, even while allowing a much wider range of types in its CUDA runtime.

3.3 Discussion

There are numerous potential causes for the results we see in Figure 2 and in our experimentation. We noted that the cuSPARSELt provided manipulation of operation handles, object descriptors, operation attributes directly by the user. We explicitly avoided timing any of these in our experiments. However, the cuBLAS implementation was able to directly take in sparse data and initiate its calls to the optimized sparse algorithms. The initial range of dimensions with higher cuSPARSELt performance suggest that the calls to the API were being made with reasonable efficacy, up to a certain point. The inversion is what is expected comparing traditional dense algorithms with matrices of low sparsity. As sparse matrices grow, the indirection in memory access means that contiguous memory loading of dense matrices can perform more efficiently. This leads to another possibility that the operation loading into its core, including registers and Sparse Tensor Core, is incurring additional overhead.

Additionally, there is a possibility the GPU allocated was not actually performing as expected. We had noticed issues running our experiments for high dimension parameters for the matrices being manipulated, to a point where we simply would cause memory errors that would only go away after Colab allocated us a different machine. We suspect that there could have been certain bottlenecking or other restrictions imposed and the runtimes we were seeing were *indirectly* caused by cuSPARSELt's differences with the cuBLAS setup.

It was challenging to interact with the A100 and its specialized 2:4 sparsity optimizations without direct access to an A100. With documentation providing little information about underlying detail and numerous bugs and flaws, even in installation instructions, it brought to question NVIDIA's interest in the proliferation of 2:4 structured sparsity with their hardware. We initially planned to do investigation with varying parameters such as data types, compute types, batches, bias, sparsity and others, and believe that with a more thorough investigation, direct access to NVIDIA support, and isolated GPU access, there is an opportunity to achieve better results. Once those are achieved and an interface is prepared for optimized access, we believe it would be greatly beneficial to train and run fine-structured sparse models and perform real-world studies.

4 2:4 Sparse Pruning

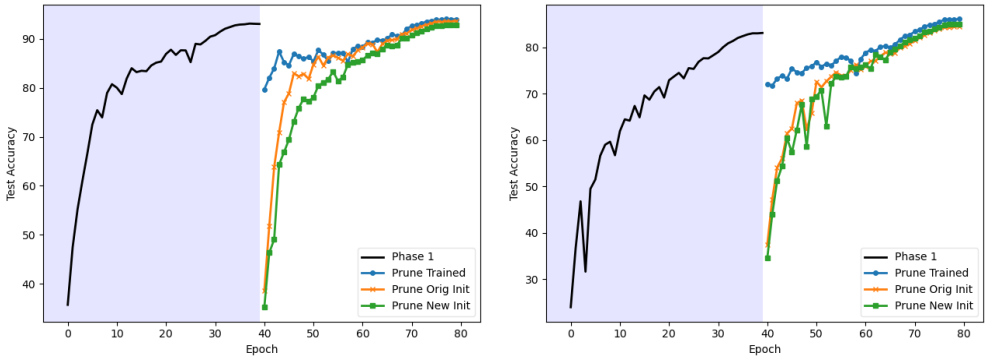
4.1 Methodology

We employ a similar methodology to [21]. However, following the first phase of training, and identification of a mask, we evaluate multiple strategies for further training.

1. We apply the mask to the trained parameters and continue to train. This is equivalent to the second phase of training in [21].
2. We return the parameters to their original initialisation, apply the mask, and train this masked network. This is a one-shot lottery ticket methodology [16].
3. We reinitialise the parameters from the same distribution, apply the mask, and train this masked network. This is similar to the work of [**empty citation**], and serves to ablate the lottery ticket method.

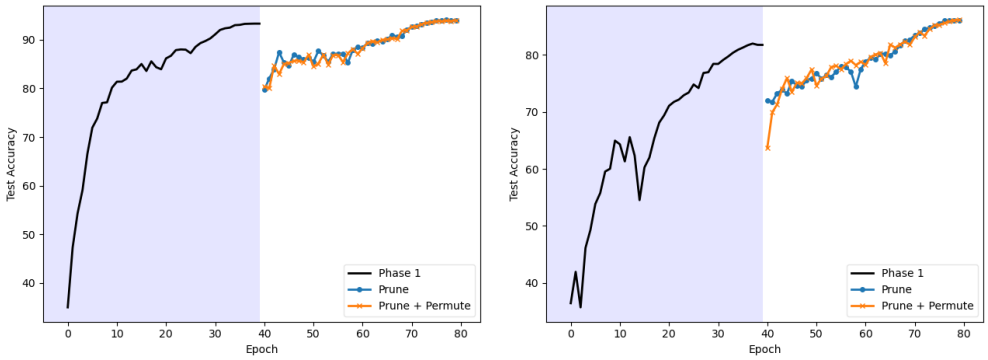
These experiments were carried out without channel permutations, since the lottery ticket method would require permuting the initialised mask, and we had difficulties extracting the permutation from the pruned and permuted networks. However, we further carry out similar experiments to those of [21], in which the use, or omission, of channel permutations is evaluated [**empty citation**]. These experiments simply train a network for a single phase, possibly permute the channels, apply a mask and continue to train.

We used two network architectures, ResNet18 [11] and EfficientNet-B0 [20] on the task classifying the CIFAR-10 dataset [10]. We implemented our experiments in PyTorch 1.13, using Google Colab with NVIDIA A100 GPUs. The NVIDIA Apex library was used for its ASP sub-module, which we used to prune our networks, and apply channel permutations. All experiments involved two phases of training, the same parameters were used for both phases, and kept consistent between experiments. Each phase consisted of 40 epochs, with batch size of 256. We used stochastic gradient descent with momentum of 0.9 and weight decay of 5e-4. The learning rate was scheduled with linear warm-up



(a) ResNet18 (b) EfficientNet-B0

Figure 3: **Pruning the trained parameters results in the highest accuracy:** the per epoch test accuracy is plotted for each of; pruning and continuing to train the trained parameters and applying the pruning mask to the original and random initialisation. Each plot mean across three random seeds.



(a) ResNet18 (b) EfficientNet-B0

Figure 4: **Applying channel permutations results has no significant impact:** the per epoch test accuracy is plotted for both permuted and non-permuted pruning, training continues from phase 1 trained parameters. Each plot mean across three random seeds.

and cosine annealing, with a peak value of 0.1. Every experiment was repeated three times, with the seeds {1,42,98}.

4.2 Results

The per epoch test accuracy for the main experiments is plotted in Figure 3. It is observed that all methods of post-pruning training exceed the accuracy of the phase 1 trained model. It is further observed that the continuing to train the pruned phase 1 weights achieves marginally superior accuracy for both networks, with random re-initialisation marginally inferior. However, for both networks both the highest accuracy and rate of convergence is comparable, despite the initialisation methods having much lower accuracy at first.

The per epoch test accuracy for the permutation experiments is plotted in Figure 4. It is observed that all both methods are highly comparable for both networks.

4.3 Discussion

The results of Figure 3 indicate that continuing to train the pruned phase 1 parameters is a good strategy for achieving 2:4 sparse networks, in agreement with [21]. The lack of significant difference between the two random initialisation is in contrast to the lottery ticket hypothesis [16], where we could expect re-initialisation to perform worse. However, it must be noted that we used *one-shot*

pruning, whereas the work of [16] demonstrates an iterative procedure to be superior, albeit with a higher computational cost. We are however surprised to see how similar these schemes performed, and would expect one-shot pruning to be effective at this moderate level of sparsity. The work of Gale et. al indicated that for larger datasets continuing to train whilst pruning was superior to retraining pruned networks [17], our results mildly support this conclusion for the specific case of 2:4 sparsity.

The results for the permutation experiments are also surprising, with no notable performance increase from using this technique. In [21] it was found that smaller networks had a notable performance increase from using permutations, due to their fewer parameters making them more sensitive to pruning. In particular, EfficentNet-BO achieved 1.3% higher test accuracy on the ImageNet dataset [9] when permutations were used. It is possible that the smaller and simpler dataset of CIFAR-10 does not present a sufficient challenge for permutations to demonstrate their superiority.

5 Conclusion

In this work, we demonstrate the application of mid-range sparsity in training and inference of deep learning models. We find that in the case of one-shot pruned 2:4 sparse matrices, lottery ticket initialization does not yield significant improvements over the use of random initialization. Our results provide support at low significance for the benefits of continual training while pruning instead of retraining, in 2:4 patterns. We highlight Sparse Tensor Cores and its potential for performance gains, although our experiments do not yield results indicating speedup over existing GEMM algorithms for NVIDIA GPUs. Future work should explore channel permutations with lottery tickets, more extensive Sparse Tensor Accelerated framework support, and overall comprehensive comparisons of mid-range sparsity with traditional high sparsity. As we have begun to demonstrate in this paper, we believe that mid-range sparsity has potential for improving upon or complementing existing deep learning sparsity factors, such as over-parametrization, bias and interpretability.

References

[1] Volker Strassen. "Gaussian elimination is not optimal". In: *Numerische Mathematik* 13.4 (Aug. 1, 1969), pp. 354–356. ISSN: 0945-3245. URL: https://doi.org/10.1007/BF02165411.

[2] Victor Pan. "How Can We Speed Up Matrix Multiplication?" In: *SIAM Review* 26.3 (1984). Publisher: Society for Industrial and Applied Mathematics, pp. 393–415. ISSN: 0036-1445. URL: https://www.jstor.org/stable/2031278.

[3] Costas S. Iliopoulos. "Worst-case complexity bounds on algorithms for computing the structure of finite abelian groups and Hermite and Smith normal forms of an integer matrix". In: *SIAM Journal Computing* 18 (1988), pp. 131–141.

[4] Steven A. Janowsky. "Pruning versus clipping in neural networks". In: *Physical Review A* 39.12 (June 1, 1989). Publisher: American Physical Society, pp. 6600–6603. URL: https://link.aps.org/doi/10.1103/PhysRevA.39.6600.

[5] Yann LeCun, John Denker, and Sara Solla. "Optimal Brain Damage". In: *Advances in Neural Information Processing Systems*. Vol. 2. Morgan-Kaufmann, 1989. URL: https://papers.nips.cc/paper/1989/hash/6c9882bbac1c7093bd25041881277658-Abstract.html.

[6] J. J. Dongarra et al. "A set of level 3 basic linear algebra subprograms". In: *ACM Transactions on Mathematical Software* 16.1 (Mar. 1, 1990), pp. 1–17. ISSN: 0098-3500. URL: https://doi.org/10.1145/77626.79170.

[7] Kumar Chellapilla, Sidd Puri, and Patrice Simard. "High Performance Convolutional Neural Networks for Document Processing". In: Tenth International Workshop on Frontiers in Handwriting Recognition. Suvisoft, Oct. 23, 2006. URL: https://hal.inria.fr/inria-00112631.

[8] Aydin Buluç et al. "Parallel sparse matrix-vector and matrix-transpose-vector multiplication using compressed sparse blocks". In: *Proceedings of the twenty-first annual symposium on Parallelism in algorithms and architectures*. SPAA 09: 21st ACM Symposium on Parallelism in Algorithms and Architectures. Calgary AB Canada: ACM, Aug. 11, 2009, pp. 233–244. ISBN: 978-1-60558-606-9. URL: https://dl.acm.org/doi/10.1145/1583991.1584053.

[9] Jia Deng et al. “ImageNet: A large-scale hierarchical image database”. In: *2009 IEEE Conference on Computer Vision and Pattern Recognition*. 2009 IEEE Conference on Computer Vision and Pattern Recognition. ISSN: 1063-6919. June 2009, pp. 248–255.

[10] A. Krizhevsky. “Learning Multiple Layers of Features from Tiny Images”. In: 2009. URL: https://www.semanticscholar.org/paper/Learning-Multiple-Layers-of-Features-from-Tiny-Krizhevsky/5d90f06bb70a0a3dced62413346235c02b1aa086.

[11] Kaiming He et al. *Deep Residual Learning for Image Recognition*. Dec. 10, 2015. arXiv: 1512.03385[cs]. URL: http://arxiv.org/abs/1512.03385.

[12] Hao Li et al. *Pruning Filters for Efficient ConvNets*. Mar. 10, 2017. arXiv: 1608.08710[cs]. URL: http://arxiv.org/abs/1608.08710.

[13] Jian-Hao Luo, Jianxin Wu, and Weiyao Lin. *ThiNet: A Filter Level Pruning Method for Deep Neural Network Compression*. July 19, 2017. arXiv: 1707.06342[cs]. URL: http://arxiv.org/abs/1707.06342.

[14] Lucas Theis et al. *Faster gaze prediction with dense networks and Fisher pruning*. version: 2. July 9, 2018. arXiv: 1801.05787[cs,stat]. URL: http://arxiv.org/abs/1801.05787.

[15] Pete Warden. *Speech Commands: A Dataset for Limited-Vocabulary Speech Recognition*. Apr. 9, 2018. arXiv: 1804.03209[cs]. URL: http://arxiv.org/abs/1804.03209.

[16] Jonathan Frankle and Michael Carbin. *The Lottery Ticket Hypothesis: Finding Sparse, Trainable Neural Networks*. Mar. 4, 2019. arXiv: 1803.03635[cs]. URL: http://arxiv.org/abs/1803.03635.

[17] Trevor Gale, Erich Elsen, and Sara Hooker. *The State of Sparsity in Deep Neural Networks*. Feb. 25, 2019. arXiv: 1902.09574[cs,stat]. URL: http://arxiv.org/abs/1902.09574.

[18] Adam Paszke et al. *PyTorch: An Imperative Style, High-Performance Deep Learning Library*. Dec. 3, 2019. arXiv: 1912.01703[cs,stat]. URL: http://arxiv.org/abs/1912.01703.

[19] Eran Malach et al. “Proving the Lottery Ticket Hypothesis: Pruning is All You Need”. In: *Proceedings of the 37th International Conference on Machine Learning*. International Conference on Machine Learning. PMLR, Nov. 21, 2020, pp. 6682–6691. URL: https://proceedings.mlr.press/v119/malach20a.html.

[20] Mingxing Tan and Quoc V. Le. *EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks*. Sept. 11, 2020. arXiv: 1905.11946[cs,stat]. URL: http://arxiv.org/abs/1905.11946.

[21] Asit Mishra et al. *Accelerating Sparse Deep Neural Networks*. Apr. 16, 2021. arXiv: 2104.08378[cs]. URL: http://arxiv.org/abs/2104.08378.

[22] Jeff Pool and Chong Yu. “Channel Permutations for N:M Sparsity”. In: *Advances in Neural Information Processing Systems*. Vol. 34. Curran Associates, Inc., 2021, pp. 13316–13327. URL: https://proceedings.neurips.cc/paper/2021/hash/6e8404c3b93a9527c8db241a1846599a-Abstract.html.

[23] Aojun Zhou et al. *Learning N:M Fine-grained Structured Sparse Neural Networks From Scratch*. Apr. 18, 2021. arXiv: 2102.04010[cs]. URL: http://arxiv.org/abs/2102.04010.

[24] Tianlong Chen et al. *Coarsening the Granularity: Towards Structurally Sparse Lottery Tickets*. June 9, 2022. arXiv: 2202.04736[cs]. URL: http://arxiv.org/abs/2202.04736.

[25] Alhussein Fawzi et al. “Discovering faster matrix multiplication algorithms with reinforcement learning”. In: *Nature* 610.7930 (Oct. 2022). Number: 7930 Publisher: Nature Publishing Group, pp. 47–53. ISSN: 1476-4687. URL: https://www.nature.com/articles/s41586-022-05172-4.

[26] Haoran You et al. *Drawing Early-Bird Tickets: Towards More Efficient Training of Deep Networks*. Feb. 16, 2022. arXiv: 1909.11957[cs,stat]. URL: http://arxiv.org/abs/1909.11957.

[27] Sara Robinson. “Toward an Optimal Algorithm for Matrix Multiplication”. In: ().

[28] Ashish Vaswani et al. “Attention is All you Need”. In: ().