



Neural Networks with Block Diagonal Inner Product Layers

Amy Nesky^(✉) and Quentin F. Stout

Computer Science and Engineering, University of Michigan,
Ann Arbor, MI 48109, USA
{anesky,qstout}@umich.edu

Abstract. We consider a modified version of the fully connected layer we call a block diagonal inner product layer. These modified layers have weight matrices that are block diagonal, turning a single fully connected layer into a set of densely connected neuron groups. This idea is a natural extension of group, or depthwise separable, convolutional layers applied to the fully connected layers. Block diagonal inner product layers can be achieved by either initializing a purely block diagonal weight matrix or by iteratively pruning off diagonal block entries. This method condenses network storage and speeds up the run time without significant adverse effect on the testing accuracy.

Keywords: Neural networks · Block diagonal · Structured sparsity

1 Introduction

Ideally, efforts to reduce the memory requirements of neural networks would also lessen their computational demand, but often these competing interests force a trade-off. Fully connected layers are unwieldy, yet they continue to be present in the most successful networks [13, 23, 28]. Our work addresses both memory and computational efficiency without compromise. Focusing our attention on the fully connected layers, we decrease network memory footprint and improve network runtime.

There are a variety of methods to condense large networks without much harm to their accuracy. One such technique that has gained popularity is pruning [3, 4, 21], but traditional pruning has disadvantages related to network runtime. Most existing pruning processes slow down network training, and the resulting condensed network is usually significantly slower to execute [3]. Sparse format operations require additional overhead that can greatly slow down performance unless one prunes nearly all weight entries, which can damage network accuracy.

Localized memory access patterns can be computed faster than non-localized lookups. By implementing block diagonal inner product layers in place of fully connected layers, we condense neural networks in a structured manner that

speeds up the final runtime and does little harm to the final accuracy. Block diagonal inner product layers can be implemented by either initializing a purely block diagonal weight matrix or by initializing a fully connected layer and focusing pruning efforts off the diagonal blocks to coax the dense weight matrix into structured sparsity. The first method reduces the gradient computation time and hence the overall training time. The latter method retains higher accuracy and supports the robustness of networks to *shaping*. That is, pruning can be used as a mapping between architectures—in particular, a mapping to more convenient architectures. Depending on how many iterations the pruning process takes, this method may also speed up training.

We have converted a single fully connected layer into an ensemble of smaller inner product learners whose combined efforts form a stronger learner, in essence boosting the layer. These methods also bring artificial neural networks closer to the architecture of biological mammalian brains, which have more local connectivity [6].

2 Related Work

There is an assortment of criteria by which one may choose which weights to prune. With any pruning method, the result is a sparse network that takes less storage space than its fully connected counterpart. Han et al. iteratively prune a network using the penalty method by adding a mask that disregards pruned parameters for each weight tensor [4]. This means that the number of required floating point operations decreases, but the number performed stays the same. Furthermore, masking out updates takes additional time. Han et al. report the average time spent on a forward propagation after pruning is complete and the resulting sparse layers have been converted to CSR format; for batch sizes larger than one, the sparse computations are significantly slower than the dense calculations [3].

More recently, there has been momentum in the direction of structured reduction of network architecture. Node pruning preserves some structure, but drastic node pruning can harm the network accuracy and requires additional weight fine-tuning [5, 25]. Other approaches include storing a low rank approximation for a layer’s weight matrix [22] and training smaller models on outputs of larger models (distillation) [7]. Group lasso expands the concept of node pruning to convolutional filters [14, 26, 27]. That is, group lasso applies L_1 -norm regularization to entire filters. Sidhawani et al. propose structured parameter matrices characterized by low displacement rank that yield high compression rate as well as fast forward and gradient evaluation [24]. Their work focuses on toeplitz-related transforms of the fully connected layer weight matrix. However, speedup is generally only seen for compression of large weight matrices. According to their Fig. 3, for displacement rank higher than 1.5×10^{-3} times the matrix dimension the forward pass is slowed down, and backward pass is slowed down for displacement rank higher than 9×10^{-4} times the matrix dimension.

Group, or depthwise separable, convolutions have been used in recent CNN architectures with great success [2, 8, 29]. In group convolutions, a particular filter

does not see all of the channels of the previous layer. Block diagonal inner product layers apply this idea of separable neuron groups to the fully connected layers. This method transforms a fully connected layer into an ensemble of smaller fully connected neuron groups that boost the layer.

3 Methodology

We consider two methods for implementing block diagonal inner product layers:

1. We initialize a layer with a purely block diagonal weight matrix and keep the number of connections constant throughout training.
2. We initialize a fully connected layer and iteratively prune entries off the diagonal blocks to achieve a block substructure.

Within a layer, all blocks have the same size. Method 2 is accomplished in three phases: a dense phase, an iterative pruning phase and a block diagonal phase. In the dense phase a fully connected layer is initialized and trained in the standard way. During the iterative pruning phase, focused pruning is applied to entries off the diagonal blocks using the weight decay method with L_1 -norm. That is, if W is the weight matrix for a fully connected layer we wish to push toward block diagonal, we add

$$\alpha \sum_{i,j} |\mathbb{1}_{i,j} W_{i,j}| \tag{1}$$

to the loss function during the iterative pruning phase, where α is a tuning parameter and $\mathbb{1}_{i,j}$ indicates whether $W_{i,j}$ is off the diagonal blocks in W . The frequencies of regularization and pruning during this phase are additional hyperparameters. During this phase, masking out updates for pruned entries is more efficient than maintaining sparse format. When pruning is complete, to maximize speedup it is best to reformat the weight matrix once such that the blocks are condensed and adjacent in memory.¹ Batched smaller dense calculations for the blocks use cuBLAS strided batched multiplication [20]. There is a lot of flexibility in method 2 that can be tuned for specific user needs. More pruning iterations may increase the total training time but can yield higher accuracy and reduce overfitting.

4 Experiments: Speedup and Accuracy

Our goal is to reduce memory storage of the inner product layers while maintaining or reducing the final execution time of the network with minimal loss in accuracy. We will also see the reduction of total training time in some cases. All experiments are run on the Bridges’ NVIDIA P100 GPUs through the Pittsburgh Supercomputing Center.

¹ When using block diagonal layers, one should alter the output format of the previous layer and the expected input format of the following layer accordingly, in particular to row major ordering.

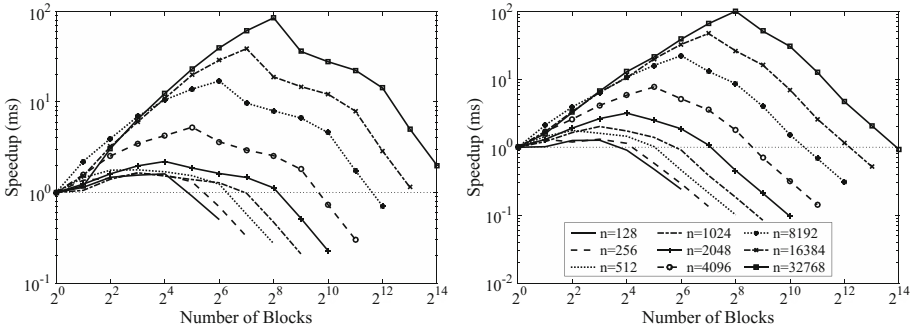


Fig. 1. Speedup when performing matrix multiplication using an $n \times n$ weight matrix and batch size 100. (Left) Speedup when performing only one forward matrix product. (Right) Speedup when performing all three matrix products involved in the forward and backward pass in gradient descent. Both images in this figure share the same key.

For speedup analysis we timed block diagonal multiplications using $n \times n$ matrices with varying dimension sizes and varying numbers of blocks; we considered the forward pass and gradient updates. We also calculate an upper bound on the ratio of the number of pruning iterations to the number of pure block iterations that will yield speedup when using block diagonal method 2. For accuracy results, we ran experiments on the MNIST [16] dataset using a LeNet-5 [15] network, and the SVHN [19] and CIFAR10 [10] datasets using Krizhevsky’s cuda-convnet [11]. Cuda-convnet does not produce state-of-art accuracies for SVHN or CIFAR10, but demonstrates the performance differences between our methods and others. We implement our work in Caffe, which provides these architectures; Caffe’s MNIST example uses LeNet-5 and cuda-convnet can be found in Caffe’s CIFAR10 “quick” example.

4.1 Speedup

Figure 1 shows the speedup when performing matrix multiplication using an $n \times n$ weight matrix and batch size 100 when the weight matrix is purely block diagonal. The speedup when performing only the forward-pass matrix product is shown in the left pane, and the speedup when performing all gradient descent products is shown in the right pane. As the number of blocks increases, the overhead to perform cuBLAS strided batched multiplication can become noticeable; this library is not yet well optimized for performing many small matrix products [17]. However, with specialized batched multiplications for many small matrices, Jhurani et al. attain up to 6 fold speedup [9]. Using cuBLAS strided batched multiplication, maximum speedup is achieved when the number of blocks is $1/2^7$ times the matrix dimension. When only timing the forward pass, the speedup is always greater than 1 when the number of blocks is at most $1/2^5$ times the matrix dimension. When timing the forward and backward

pass, the speedup is always greater than 1 when the number of blocks is at most $1/2^6$ times the matrix dimension.

For a given inner product layer, using block diagonal method 2 we would see speedup during training if

$$\frac{T(\text{FC}) - T(\text{Block})}{T(\text{Prune})} > \frac{y}{x} \quad (2)$$

where $T(\cdot)$ is the combined time to perform the forward and backward passes of an inner product layer in the input state, x is the number of pure block iterations, and y is the number of pruning iterations. $T(\text{Prune})$ is the time to regularize and apply a mask to the off diagonal block layer weights, which happens once in a training iteration. Figure 2 plots the upper bound in ratio 2 against the number of blocks for a layer with an $n \times n$ weight matrix and batch size 100.

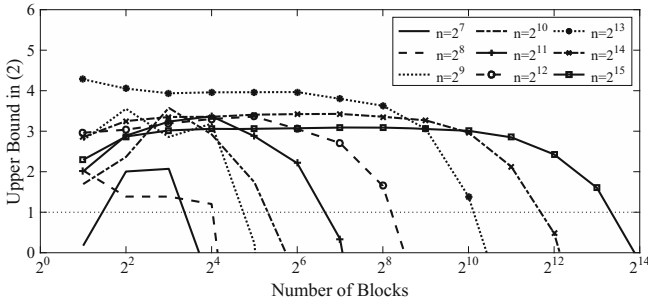


Fig. 2. Using batch size 100, upper bound on the ratio of the number of pruning iterations to the number of pure block iterations that will result in an overall training speedup when using block diagonal method 2.

Figure 3 shows timing results for the inner product layers in Lenet-5 (Left) and cuda-convnet (Right), which both have two inner product layers. We plot the forward time per inner product layer when the layers are purely block diagonal, the combined forward and backward time to do the three matrix products involved in gradient descent training when the layers are purely block diagonal, and the runtime of sparse matrix multiplication with random entries in CSR format using cuSPARSE [20]. For brevity we refer to a block diagonal network architecture as (b_1, \dots, b_n) -BD; $b_i = 1$ indicates that the i^{th} inner product layer is fully connected. FC is short for all inner product layers being fully connected. The points at which the forward sparse and forward block curves meet in each plot in Fig. 3 indicate the fully connected dense forward runtimes for each layer; these are made clearer with dotted, black, vertical lines. In Lenet-5 (Left), the first inner product layer, ip1, has a 500×800 weight matrix, and the second has a 10×500 weight matrix, so the (b_1, b_2) -BD architecture has $(800 \times 500)/b_1 + (500 \times 10)/b_2$ nonzero weights across both inner

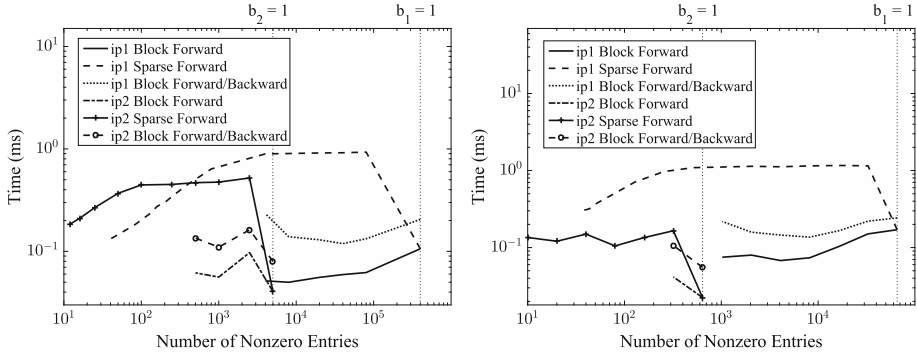


Fig. 3. For each inner product layer in Lenet-5 (Left) and cuda-convnet (Right): forward runtimes of block diagonal and CSR sparse formats, combined forward and backward runtimes of block diagonal format. Lenet-5 uses batch size 64, and cuda-convnet uses batch size 100.

product layers. There is $\geq 1.4\times$ speedup for $b_1 \leq 50$, or 8000 nonzero entries, when timing both forward and backward matrix products, and $1.6\times$ speedup when $b_1 = 100$, or 4000 nonzero entries, in the forward only case. In cuda-convnet (Right), the first inner product layer, ip1, has a 64×1024 weight matrix, and the second has a 10×64 weight matrix. The (b_1, b_2) -BD architecture has $(1024 \times 64)/b_1 + (64 \times 10)/b_2$ nonzero entries across both inner product layers. In the ip1 layer, there is $\geq 1.26\times$ speedup for $b_1 \leq 32$, or 2048 nonzero entries, when timing both forward and backward matrix products, and $\geq 1.65\times$ speedup for $b_1 \leq 64$, or 1024 nonzero entries, in the forward only case. In both plots we see sparse format performs poorly until there are less than 50 nonzero entries.

4.2 Accuracy Results

All hyperparameters and initialization distributions provided by Caffe’s example architectures are left unchanged. Training is done with batched gradient descent using the cross-entropy loss function on the softmax of the output layer. In our experiments we performed only manual hyperparameter tuning of new hyperparameters introduced by block diagonal method 2 like the coefficient of the new regularization term (see Eq. 1) and the pruning modulus cutoff.

In ShuffleNet, Zhang et al. note that when multiple group convolutions are stacked together this can block information flow between channel groups and weaken representation [29]. To correct for this, they suggest dividing the channels in each group into subgroups, and shuffling the outputs of the subgroups in this layer before feeding them to the next layer. Applying this approach to block inner product layers requires either moving entries in memory or doing more, smaller matrix products. Both of these options would hurt efficiency. Using pruning to achieve the block diagonal structure, as in method 2, also addresses information flow. Pruning does add some work to the training iterations, but, unlike the

ShuffleNet method, does not add work to the final execution of the trained network. After pruning is complete, the learned weights are the result of a more complete picture; while the information flow has been constrained, it is preserved like a ghost in the remaining weights. Another alternative is to randomly shuffle whole blocks each pass like in the “random sparse convolution” layer in the CNN library *cuda-convnet* [12]. We found that for the inner product layers in LeNet-5 and Krizhevsky’s Cuda-convnet, the ShuffleNet method did not show as much improvement in accuracy as randomly shuffling the whole blocks, so we do not include results using the ShuffleNet method.

Table 1 shows the accuracy results for block diagonal method 1, method 1 with random block shuffling, method 2 and traditional iterative pruning, which uses the penalty method to prune weight entries not subject to any confinement or organization. We show accuracy results for the most condensed net with block diagonal inner product layers and the net with the fastest speedup in the inner product layers.

Table 1. Accuracy results on MNIST, SVHN, and CIFAR10 datasets.

	Method 1	Rand. shuff	Method 2	Trad. it. prune
<i>MNIST (99.11% accurate when using FC)</i>				
(10, 1)-BD	98.83%	98.81%	99.02%	99.04%
(100, 10)-BD	98.39%	98.42%	98.65%	98.55%
<i>SVHN (91.96% accurate when using FC)</i>				
(8, 1)-BD	91.39%	91.46%	91.88%	91.15%
(64, 2)-BD	89.21%	89.69%	90.02%	90.93%
<i>CIFAR10 (76.29% accurate when using FC)</i>				
(8, 1)-BD	75.07%	75.09%	76.05%	75.64%
(64, 2)-BD	72.7%	73.45%	74.81%	75.18%

MNIST. We experimented on the MNIST dataset with the LeNet-5 framework [15] using a training batch size of 64 for 10000 iterations. LeNet-5 has two convolutional layers with pooling followed by two inner product layers with ReLU activation. FC achieves a final accuracy of 99.11%. In all cases testing accuracy remains within 1% of FC accuracy.

Using traditional iterative pruning with L_2 regularization, as suggested in [4], pruning until 4000 and 500 nonzero entries survived in ip1 and ip2 respectively gave an accuracy of 98.55%, but the forward multiplication was more than 8 times slower than the dense fully connected case (See Fig. 3 Left). Implementing (100, 10)-BD method 2 with pruning using 15 dense iterations and 350 pruning iterations gave a final accuracy of 98.65%. (10, 1)-BD yielded $\approx 1.4\times$ speedup for all gradient descent matrix products in both inner product layers after any pruning is complete, and (100, 10)-BD condensed the inner product layers in LeNet-5 ≈ 81 fold.

SVHN. We experimented on the SVHN dataset with Krizhevsky’s cuda-convnet [11] using batch size 100 for 9000 iterations. Cuda-convnet has three convolutional layers with ReLu activation and pooling, followed by two fully connected layers with no activation. (8, 1)-BD yielded $\approx 1.5\times$ speedup for all gradient descent matrix products in both inner product layers when purely block diagonal, and (64, 2)-BD condensed the inner product layers in Cuda-convnet ≈ 47 fold.

Using FC we obtained a final accuracy of 91.96%. Table 1 shows all methods stayed under a 2.5% drop in accuracy. Using traditional iterative pruning with L_2 regularization until 1024 and 320 nonzero entries survived in the final two inner product layers respectively gave an accuracy of 90.93%, but the forward multiplication was more than 8 times slower than the dense fully connected computation. On the other hand, implementing (64, 2)-BD method 2 with pruning, which has corresponding numbers of nonzero entries, with 500 dense iterations and <1000 pruning iterations gave a final accuracy of 90.02%. This is ≈ 47 fold compression of the inner product layer parameters with only a 2% drop in accuracy when compared to FC.

CIFAR10. We experimented on the CIFAR10 dataset with Krizhevsky’s cuda-convnet [11] using batch size 100 for 9000 iterations. Using FC we obtained a final accuracy of 76.29%. Table 1 shows all methods stayed within a 4% drop in accuracy. Using traditional iterative pruning with L_2 regularization until 1024 and 320 nonzero entries survived in the final two inner product layers gave an accuracy of 75.18%, but again the forward multiplication was more than 8 times slower than the dense fully connected computation. On the other hand, implementing (64, 2)-BD method 2 with pruning, which has corresponding numbers of nonzero entries, with 500 dense iterations and <1000 pruning iterations gave a final accuracy of 74.81%. This is ≈ 47 fold compression of the inner product layer parameters with only a 1.5% drop in accuracy. The total forward runtime of ip1 and ip2 in (64, 2)-BD is 1.6 times faster than in FC. To achieve comparable speed with sparse format we used traditional iterative pruning to leave 37 and 40 nonzero entries in the final inner product layers giving an accuracy of 73.01%. Thus implementing block diagonal layers with pruning yields comparable accuracy and memory condensation to traditional iterative pruning with faster final execution time.

Whole node pruning decreases the accuracy more than corresponding reductions in the block diagonal setting. Node pruning until ip1 had only 2 outputs, i.e. a 1024×2 weight matrix, and ip2 had a 2×10 weight matrix for a total of 2068 weights between the two layers gave a final accuracy of 59.67%. (64, 2)-BD has a total of 1344 weights between the two inner product layers and had a final accuracy 15.14% higher with pruning.

The final accuracy on an independent test set was 76.29% on CIFAR10 using the FC net while the final accuracy on the training set itself was 83.32%. Using the (64, 2)-BD net without pruning, the accuracy on an independent test set was 72.49%, but on the training set was 75.63%. With pruning, the accuracy of

(64, 2)-BD on an independent test set was 74.81%, but on the training set was 76.85%. Both block diagonal methods decrease overfitting; the block diagonal method with pruning decreases overfitting slightly more.

5 Conclusion

We have shown that block diagonal inner product layers can reduce network size, training time and final execution time without significant harm to the network performance.

While traditional iterative pruning can reduce storage, the scattered surviving weights make sparse computation inefficient, slowing down both training and final execution time. Our block diagonal methods address this inefficiency by confining dense regions to blocks along the diagonal. Without pruning, block diagonal method 1 allows for faster training time. Method 2 preserves the learning with focused, structured pruning that reduces computation for speedup during execution. In our experiments, method 2 saw higher accuracy than the purely block diagonal method. The success of method 2 supports the use of pruning as a mapping from large dense architectures to more efficient, smaller, dense architectures. Both methods make larger network architectures more feasible to train and use since they convert a fully connected layer into a collection of smaller inner product learners working jointly to form a stronger learner. In particular, GPU memory constraints become less constricting.

There is a lot of room for additional speedup with block diagonal layers. Dependency between layers poses a noteworthy bottleneck in network parallelization. With structured sparsity like ours, one no longer needs a full barrier between layers. Additional speedup would be seen in software optimized to support weight matrices with organized sparse form, such as blocks, rather than being optimized for dense matrices. For example, for many small blocks, one can reach up to 6 fold speedup with specialized batched matrix multiplication [9]. Hardware has been developing to better support sparse operations. Block format may be especially suitable for training on evolving architectures such as neuromorphic systems. These systems, which are far more efficient than GPUs at simulating mammalian brains, have a pronounced 2-D structure and are ill-suited to large dense matrix calculations [1, 18].

Acknowledgments. This material is based upon work supported by the National Science Foundation Graduate Research Fellowship under Grant No. DGE-1256260. This work used the Extreme Science and Engineering Discovery Environment (XSEDE), which is supported by National Science Foundation grant number OCI-1053575. Specifically, it used the Bridges system, which is supported by NSF award number ACI-1445606, at the Pittsburgh Supercomputing Center (PSC).

References

1. Boahen, K.: Neurogrid: a mixed-analog-digital multichip system for large-scale neural simulations. *Proc. IEEE* **102**(5), 699–716 (2014)
2. Chollet, F.: Xception: deep learning with depthwise separable convolutions. [arXiv:1610.02357](https://arxiv.org/abs/1610.02357) (2017)
3. Han, S., et al.: Deep compression: compressing deep neural networks with pruning, trained quantization and Huffman coding. In: *ICLR* (2015)
4. Han, S., et al.: Learning both weights and connections for efficient neural networks. In: *NIPS*, pp. 1135–1143 (2015)
5. He, T., et al.: Reshaping deep neural network for fast decoding by node-pruning. In: *IEEE ICASSP*, pp. 245–249 (2014)
6. Herculano-Houzel, S.: The remarkable, yet not extraordinary, human brain as a scaled-up primate brain and its associated cost. *PNAS* **109**(Supplement 1), 10661–10668 (2012)
7. Hinton, G., et al.: Distilling the knowledge in a neural network. In: *NIPS* (2014)
8. Ioannou, Y., et al.: Deep Roots: improving CNN efficiency with hierarchical filter groups. In: *CVPR* (2017)
9. Jhurani, C., et al.: A GEMM interface and implementation on NVIDIA GPUs for multiple small matrices. *J. Parallel Distrib. Comput.* **75**, 133–140 (2015)
10. Krizhevsky, A.: Learning multiple layers of features from tiny images. Technical report, Computer Science, University of Toronto (2009)
11. Krizhevsky, A.: Cuda-convnet. Technical report, Computer Science, University of Toronto (2012)
12. Krizhevsky, A.: Cuda-convnet: high-performance C++/CUDA implementation of convolutional neural networks (2012)
13. Krizhevsky, A., et al.: Imagenet classification with deep convolutional neural networks. In: *NIPS*, pp. 1106–1114 (2012)
14. Lebedev, V., et al.: Fast convnets using group-wise brain damage. In: *CVPR* (2016)
15. LeCun, Y., et al.: Gradient-based learning applied to document recognition. *Proc. IEEE* **86**(11), 2278–2324 (1998)
16. LeCun, Y., et al.: The MNIST database of handwritten digits. Technical report
17. Masliah, I., et al.: High-performance matrix-matrix multiplications of very small matrices. In: Dutot, P.-F., Trystram, D. (eds.) *Euro-Par 2016. LNCS*, vol. 9833, pp. 659–671. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-43659-3_48
18. Merolla, P.A., et al.: A million spiking-neuron integrated circuit with a scalable communication network and interface. *Science* **345**(6197), 668–673 (2014)
19. Netzer, Y., et al.: Reading digits in natural images with unsupervised feature learning. In: *NIPS* (2011)
20. Nickolls, J., et al.: Scalable parallel programming with CUDA. *ACM Queue* **6**(2), 40–53 (2008)
21. Reed, R.: Pruning algorithms—a survey. *IEEE Trans. Neural Netw.* **4**(5), 740–747 (1993)
22. Sainath, T.N., et al.: Low-rank matrix factorization for deep neural network training with high-dimensional output targets. In: *IEEE ICASSP* (2013)
23. Simonyan, K., et al.: Very deep convolutional networks for large-scale image recognition. [arXiv:1409.1556](https://arxiv.org/abs/1409.1556) (2014)
24. Sindhvani, V., et al.: Structured transforms for small-footprint deep learning. In: *NIPS*, pp. 3088–3096 (2015)

25. Srinivas, S., et al.: Data-free parameter pruning for deep neural networks. [arXiv:1507.06149](#) (2015)
26. Wen, W., et al.: Learning structured sparsity in deep neural networks. In: NIPS, pp. 2074–2082 (2016)
27. Yuan, M., et al.: Model selection and estimation in regression with grouped variables. *J. Royal Stat. Soc. Ser. B* **68**(1), 49–67 (2006)
28. Zeiler, M.D., et al.: Visualizing and understanding convolutional networks. [arXiv:1311.2901](#) (2013)
29. Zhang, X., et al.: ShuffleNet: an extremely efficient convolutional neural network for mobile devices. [arXiv:1707.01083](#) (2017)