Weight Redistribution Algorithms for Sparse Neural Networks

Paul Eibensteiner
Supervised by: Dipl.-Ing. Dr. techn. Markus Steinberger

ICG TU Graz

Abstract

Sparse neural networks are successfully used to speed up inference and reduce the memory requirements of fully trained networks. However, recently it has been shown that sparsity can also be employed in the training phase. In this work, we introduce two new methods to train sparse neural networks from scratch that alter the network's topology while training and maintain a global level of density. Then we compare them to recent state-of-the-art algorithms in a controlled setting on two different datasets. All algorithms are implemented in a GPU-accelerated framework and tested using the KMNIST and HIGGS datasets. The results show that global weight redistribution can significantly improve the network's accuracy without introducing significant overhead.

Keywords: Sparse Matrices, Deep Learning, Neural Networks

1 Introduction

Computational resources are commonly placing an upper bound on the performance of artificial neural networks (NNs). Thus, different ways to lower the training and inference cost have always been of interest. One way to achieve this is through sparse NNs, networks where not every pair of neurons of two consecutive layers is connected. This is in contrast to dense networks, where the layers are fully connected. Sparse NNs have been shown to perform equally or even outperform their dense counterparts due to their regularizing effect when overfitting is a problem [19].

So-called pruning algorithms that yield sparse NNs have been proposed in the research since the nineties [4, 13]. They take a fully (or partially) trained dense neural network and remove connections by some criterion, often in an iterative manner. The obtained sparse networks have great accuracy, even when more than 90% of the weights are removed. The downside is that training them takes the same or more resources than a fully connected network.

Pruning algorithms were further refined with the idea of retraining the network after pruning [22]. This led to sophisticated pruning-retraining cycles and gradual pruning, where weights are continually removed over the course of multiple epochs [12, 23]. All pruning algorithms typically increase training time by introducing additional retrain cycles.

In contrast to this, sparse training algorithms have appeared more recently. As the name suggests, they train with a sparsely connected model throughout the whole learning phase. They can be further subdivided into algorithms that keep the topology as initialized at the beginning and ones that allow the connections to change, respectively called static and dynamic sparse training in this work.

There are several potential advantages of sparse training. First, when sparse matrix operations are implemented efficiently, the training overhead can be reduced. In the most simple case this can lead to energy savings ("green AI"), but it can even improve training speed. Secondly, savings in storage can make training even bigger neural networks possible, especially when hardware access is restricted. Additionally, sparsity can prevent overfitting because fewer parameters are active but still allows for a high degree of generalization [11].

An essential part of every dynamic sparse training algorithm is the redistribution step, where some connections are removed and others are added. This work will focus on this step and compare different algorithms using an implementation of sparse matrix operations in CUDA [6]. This is especially interesting, as recent papers mainly focus on theoretical measures such as FLOPs to evaluate resource usage, even though on today's highly parallelized hardware, these measurements often do not correlate with the training speed. Additionally, two of the compared algorithms apply criteria that have only been used in pruning so far. To our knowledge, this is the first time they are used in the context of dynamic sparse training.

1.1 Dynamic Sparse Training

Dynamic sparse training emerged because static sparse training was quickly found not to be able to yield good performance when the topology is initialized randomly [9]. It was found that traversing the parameter space, i.e., dynamic sparse training, is necessary to avoid stationary points found in the static subspace [8, 10].

Sparse Evolutionary Training (SET) is one of the most cited dynamic sparse training algorithms [19]. It introduced the concept of an “evolutionary step” after each epoch, where connections are redistributed. This step simply removes the weights with the smallest absolute value per layer (like magnitude-based pruning) and adds new connections at random locations. The new weights are initialized using Gaussian noise.

The newest algorithm, RigL, also builds on the ideas of SET [7]. However, it introduces a new idea of adding the connections: They are created at the highest magnitude gradients of non-active connections and initialized to zero. The intuition behind this is that these are the locations where the weights are expected to change the most and therefore improve the loss the most.

Most other algorithms for dynamic sparse training that we are aware of either have to store the full dense weight matrices or compute the dense gradient in the backward pass of every iteration [5, 15, 17]. This makes them ill-suited for our declared goal of improving training efficiency.

For a broader overview of the topic, we refer the reader to the works of Blalock et al., Gale et al. and Hoefler et al. [1, 10, 14].

2 Redistribution Algorithms

From the number of different approaches in the literature we chose four existing dynamic sparse training algorithms and adapted them for the comparison. Additionally, we created two new methods based on pruning algorithms. The main criterion in the selection for the comparison was to identify algorithms that yield the best test accuracy without introducing too big of an overhead. Such an overhead would then render the gains in efficiency introduced by sparse layers in the network useless.

To keep the results comparable the algorithms considered only differ in two ways: Which weights they remove, and which weights they add. In contrast to some of the original algorithms, new weights are always initialized to zero (instead of e.g. random noise), which has been empirically found to improve the performance in multiple papers [7, 20]. The original algorithms usually also vary in other aspects, e.g. how many weights are relocated, when the relocation step occurs, etc. However, they are not integral to the redistribution itself and are kept equal for all the algorithms in this work.

The general idea of all the given algorithms, from an optimization perspective, is to remove the weights that lead to the smallest possible increase in loss. The only way to compute this expected increase exactly would be to remove every combination of weights and compute the corresponding loss [21]. This is, of course, infeasible for networks in the scale of current state-of-the-art classifiers. Therefore, different approximations are used, with the most common being the absolute value of the weight in question.

2.1 Redr-Random

The first and simplest algorithm is based on Sparse Evolutionary Training [19]. It removes the weights closest to zero per layer. Then it adds the same amount of weights at random locations in the layer. The algorithm resembles a phenomenon present in biological brains known as synaptic shrinking [19]. The algorithm has been shown to clearly outperform a sparse neural network with fixed topology [19]. A version optimized for CPU execution was implemented by Liu et al. [18].

2.2 Redr-Gradient

The second algorithm is based on RigL [7], which in turn builds on ideas of SET. It also removes the weights with the smallest absolute value per layer. It then adds the weights with the highest magnitude gradient per layer. The intuition behind this is that these are the connections expected to receive the most change in the following training epoch, therefore contributing the most to a decrease in the loss. The densities per layer are kept constant.

2.3 Redr-Gradient-Global

The third algorithm is based on RigL as well [7]. It uses the same criteria to remove (weight magnitude) and add (gradient magnitude) weights, but it applies them globally across the whole network. The given percentage of weights is removed per layer, and then the connections with the globally highest gradient are added. For weight insertion, the full gradient is only calculated for one layer at a time and then discarded. To globally find the k highest absolute partial derivatives, the maximum k of the previously found maxima and the current layer's absolute gradient are calculated iteratively for every layer. Their indices and values are stored. After iterating through all layers, the algorithm inserts the weights corresponding to the resulting indices. This avoids calculating (storing) and sorting the full dense gradient of the model.

2.4 Redr-Loss

This algorithm was adapted from the work of Ertl [6]. It also removes a set fraction of weights by magnitude per layer. It then adds weights by an approximation of the gradient: The absolute losses of one layer's neurons are sorted, and the top k are selected. For these top k

neurons, the gradient vectors are computed. Then for each neuron an equal amount of new weights is added at the highest absolute gradients. In our algorithm, k is set to 30% of the amount of neurons in the current layer.

2.5 Redr-Woodfisher and Redr-Woodtaylor

Finally, the last two algorithms are based on an approximation of the inverse Hessian matrix already used by Hassibi and Stork [13]. The intuition behind this is that its magnitude does not sufficiently approximate the significance of a weight. However, a first-order approximation would be problematic as well because a fully trained weight at a local minimum has a partial derivative of zero. Therefore, a second-order approximation of the loss induced by the removal is calculated. This might reduce the number of important weights that are removed from the NN. Singh and Alistarh [21] use an efficient approximation for the inverse Hessian they call Woodfisher. The approximation algorithm was introduced by Hassibi and Stork [13] and both papers show that it outperforms magnitude-based pruning in all tested settings.

In short, Singh and Alistarh [21] use the Fisher Information Matrix as a replacement for the Hessian. Their equivalence is given under the assumption that the conditional distribution of the model is equal to the conditional distribution of the data. Then they use the empirical Fisher as an approximation of the true Fisher, where the model distribution is replaced by the empirical training distribution. Finally, they save additional computational complexity by calculating the inverse empirical Fisher block-wise via the Sherman-Morrison formula (for mor details see Singh and Alistarh [21]).

The resulting approximation of the inverse Hessian is used to estimate the loss that is introduced when changing a parameter. This value can be calculated for every parameter and sorted to find the one with the smallest (also possibly largest negative) associated change in loss. By applying the corresponding optimal weight update to all parameters, it is possible to remove a weight resulting in the minimal loss possible as computed by the quadratic approximation.

2.5.1 Implementation as a redistribution algorithm

In the formulation of the optimization problem above, Singh and Alistarh [21] differentiate two different approaches: One which assumes the gradient to be zero (common in pruning) and one which incorporates it into the estimation of the loss. For the comparisons in this work, we implemented the two approaches in two separate redistribution algorithms. They are both used as criteria in the weight removal step: The parameters with the smallest expected loss are removed, and all others are updated according to the optimal weight updates. In both algorithms, new weights are added randomly inside the layer and in proportion to the layer's size.

This means that in contrast to the usage in pruning where the empirical Fisher is only calculated once, it has to be calculated for every redistribution. To still have a good performance while training, we made one significant adaptation. In the original work, all the necessary gradients were calculated in an additional loop before pruning. Therefore, another high number of forward and backward passes is necessary. In contrast to this, we use the gradients calculated anyways in the last epoch before redistribution. Therefore, another small approximation error is made because the model changes while calculating the empirical Fisher. The advantage is that there is no additional computational overhead for the gradient calculation and a low overhead per iteration for the calculation of the inverse empirical Fisher. A version that calculates the Fisher inverse in an additional loop was also tested and did not yield significantly better results.

To our knowledge, the two presented algorithms are the first ones using a second-order optimization for dynamic sparse training. We build on the theoretical foundations of Singh and Alistarh [21] but create a new algorithm by including the insertion of weights and by alternating the way the gradients for the Hessian are calculated.

3 Experimental Setup

We evaluated the algorithms with two different datasets, each with an adapted network architecture. Special care was taken to choose a network size small enough to prevent overfitting, as it was observed that in this case, random removal of connections can lead to improved test performance. This might be due to the random resets generating noise in the network, which prevents the remaining weights from overfitting.

The initialization of the topology happens as introduced by Ertl [6]. The active weights are added so that an equal amount n exists for every neuron of the layer. Additionally, the weights for one neuron are partitioned in n partitions, and in each partition, the active weight is chosen uniformly randomly. The values of the sparse model parameters are initialized using uniform distribution. For the weights the bounds are calculated as $\pm\frac{1}{k_i}$ where k_i is the number of input features of neuron i. The biases are initialized using the bounds $\pm\frac{1}{\sqrt{k_i}}$.

To evaluate the performance of every redistribution algorithm, the networks are trained with three different densities, namely 0.1, 0.06 and 0.03. Also, as a baseline, a dense model with the same number of neurons ("neuron-equivalent dense") and a dense model with (approximately) the same number of parameters ("parameter-equivalent dense") is trained. We trained all models with minibatch gradient descent and the Pytorch Adam optimizer [16].

3.1 Redistribution Schedule

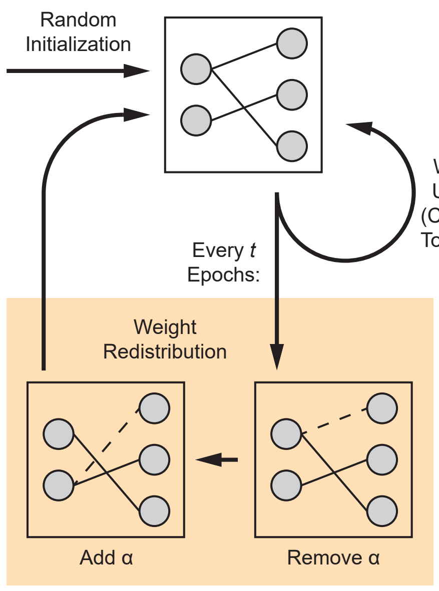

Figure 1: Simplified training schedule. Image adapted from [7].

The training schedule is similar to the work of Evci et al. [7], and mainly defined by the number of training epochs between the weight redistribution steps t and the number of redistributed weights α (see Fig. 1). In accordance with other work [5, 7, 19], t is set to one epoch, and α is initially set to $\frac{1}{2}$ of the current amount of active weights – per layer or globally depending on the algorithm. Also, multiple papers have shown the advantage of continuing training after the last redistribution step as a sort of fine-tuning [5, 7]. Therefore, we use cosine annealing so that alpha is continually reduced until the last redistribution step at $\frac{3}{4}$ the total training epochs [7].

3.2 KMNIST Dataset

KMNIST is an image dataset based on the popular MNIST dataset but harder to learn and therefore better suited to see the slight differences in accuracy [3]. Likewise, it consists of 70.000 28 by 28 grayscale input images categorized into ten classes. We used 60.000 images for training and 10.000 images for validation and apply random affine data augmentation on the training set to prevent overfitting.

We trained on the KMNIST dataset with a deep, simple multi-layer perceptron (MLP) inspired by Ciresan et al. with layer sizes 784-256-256-10, using the ReLU activation function on every layer [2]. The parameter-equivalent dense network has layer sizes 784-33-33-10, yielding slightly more parameters than the number of parameters in the sparse network at density 0.1. We set the batch size to 512 and the learning rate to the low value of 0.0005 to reduce noise in the resulting test and training accuracy, revealing the more subtle differences between the redistribution algorithms. We set the block size for Redr-Woodfisher and Redr-Woodtaylor to 34.

3.3 HIGGS Dataset

HIGGS is a binary classification dataset sampled from particle physics experiments. There are 28 features, but the last seven are human-defined functions of the first 21; thus we did not use them in our training. Due to lack of computational resources, we only use 500.000 of the original 11 million samples per training epoch and 100,000 for validation.

We trained on the HIGGS dataset with a simple MLP with layer sizes 21-256-256-256-1 for the sparse and the neuron-equivalent dense network. The input and all hidden layers use the ReLU activation function, the last layer the Sigmoid function. Also, since it is a binary classification problem, introducing sparsity in the last layer would mean discarding the output of neurons in the penultimate layer. Therefore, the last layer is kept dense in the sparse network to keep all neurons active.

The parameter-equivalent dense network uses layer sizes 21-79-79-79-1, which again yields slightly more than the number of parameters in the sparse network at a density of 0.1. We chose the network to be relatively deep to be able to see different performance levels between local and global redistribution algorithms. We set the batch size to 128, the learning rate to 0.001 and the total amount of epochs to 80 for reasonable performance and convergence. The block size for Redr-Woodfisher and Redr-Woodtaylor was again set to 34.

4 Results

The sparse networks generally performed well, at a density of 0.1 always outperforming the parameter-equivalent dense net in terms of test accuracy. Only the training speed was impacted negatively by sparse back propagation in general and additionally by the redistribution step.

4.1 Computational Resources

Unfortunately, the testing was not done in a controlled environment, so it is impossible to give exact measurements of the run times of the different models. Regardless, it can be said that the current implementation of sparse matrices does create a slow down of about $\frac{1}{3}$ compared to the dense networks. Furthermore, the redistribution algorithms again increase the runtime significantly (see Tbl. 1). The speed differences between redistribution algorithms were less than 10% and not measurable with sufficient precision in our setting.

We adapted the KMNIST architecture to have 4096 neurons in the hidden layers to measure memory consumption more realistically. This makes asymptotic changes that are important for real-world applications with bigger networks more noticeable. As seen in Tbl. 2, random and gradient-based redistribution algorithms have a lower memory usage than the neuron-equivalent dense network even at density 0.1. It should be noted that also Redr-Loss could be adapted to have equal or less memory requirements than Redr-Gradient.

In contrast, Redr-Woodfisher and Redr-Woodtaylor do use more memory in every context tested. As shown by Singh and Alistarh [21], the algorithm's performance deteriorates quickly when the block size hyperparameter is set to a low value. Therefore, for every sensible value of this parameter, the algorithm uses more memory than its dense counterpart. This makes the algorithm interesting only for frameworks that are less focused on memory. For example, in a framework where sparse training is faster, the slow down introduced by the redistribution step does not make such a big difference (as it only happens once every t epochs).

4.2 Test Accuracy

4.2.1 KMNIST Dataset

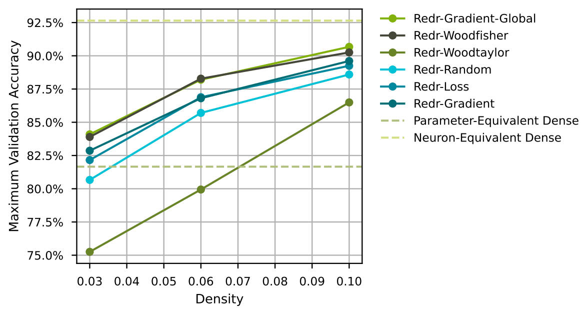

Figure 2: Test accuracy in percent while training using different redistribution algorithms and different densities on KMNIST dataset.

Evaluations on the KMNIST dataset clearly show the advantages of the sparse networks (see Fig. 2). They are able to match or outperform the parameter-equivalent network even at a density of 0.03. The neuron-equivalent dense network reaches the highest test accuracy and the accuracy of the sparse networks drops (by about 2% and 3%) as expected according to their sparsity. On all levels of sparsity, the global algorithms Redr-Woodfisher and Redr-Gradient-Global achieve the best accuracies. At a density of 0.1 Redr-Gradient-Global is less than 2% worse than the neuron-dense solution. The layer-local algorithms perform worse but also show a significant improvement from the Redr-Random baseline.

To measure the induced loss, the training loss was measured right before and after the redistribution step. Therefore, for the algorithms removing by magnitude, only the deletion of connections is measured. As expected they all perform similarly well. In contrast to this, the second-order-based methods additionally apply the weight pertubation to remaining weights, which improves the induced loss by a large margin in the case of Redr-Woodfisher (see Fig. 3).

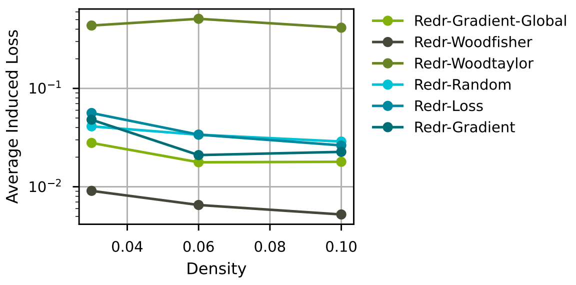

Figure 3: Training loss induced by the removal of weights by different algorithms (logarithmic scale).

4.2.2 HIGGS Dataset

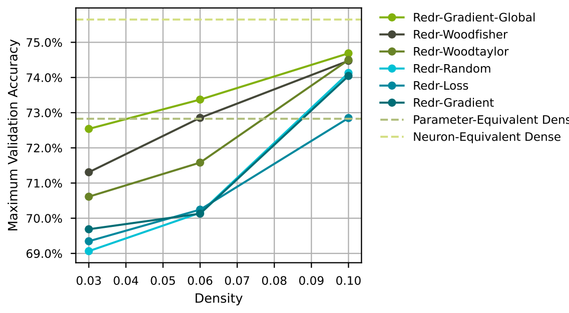

Figure 4: Test accuracy in percent while training using different redistribution algorithms and different densities on HIGGS dataset.

On the HIGGS dataset, the performance deteriorates

Table 1: Comparison of time needed for one training epoch on HIGGS dataset in seconds at a density of 0.1.

Dense neuron-equivalent	Dense parameter-equivalent	Sparse no redistribution	Redr-Random	Redr-Gradient-Global
9.80s	9.62s	13.64s	21.45s	22.20s

Table 2: Gpu storage used by different redistribution algorithms on the KMNIST dataset in megabytes (rounded). Highest and lowest in bold.

Network	Redr-Random	Redr-Loss	Redr-Gradient	Redr-Gradient-Global	Redr-Wood-Fisher	Redr-Wood-Taylor	No redistri-bution
Density 0.1	305.1	914.4	320.4	354.5	1,712.5	**1,712.7**	151.3
Density 0.06	265.9	903.4	271.2	297.5	1,041.4	1,042.5	129.1
Density 0.03	237.0	899.1	241.3	254.2	631.1	631.5	111.2
Dense parameter-equivalent	-	-	-	-	-	-	**44.6**
Dense neuron-equivalent	-	-	-	-	-	-	456.7

more quickly on low densities (see Fig. 4). Again the neuron-equivalent dense network reached the highest accuracy by a margin of less than 1% over Redr-Gradient-Global at a density of 0.1. The sparse networks at density 0.03 were not able to reach the performance level of the parameter-equivalent dense network at all. Differences between the redistribution algorithms are again bigger at lower densities, where global redistribution algorithms clearly outperform the local ones.

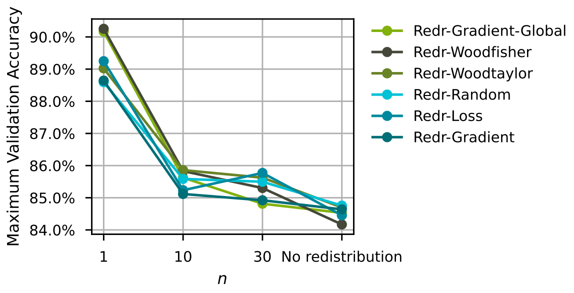

Figure 5: Validation accuracy in percent while training using different redistribution algorithms and at a density of 0.1 on KMNIST dataset.

4.3 Stability with regard to hyperparameters

4.3.1 Redistribution Frequency

We did a set of tests to investigate the effect of the frequency of redistributions. It can be seen that increasing t has the effect of decreasing the validation accuracy (Fig. 5). A high value of t means that the topology of the network changes less often. Therefore, the random initialization at the beginning plays a bigger role in the performance of the network. This can be seen in the validation accuracy of the networks varying randomly regardless of the redistribution algorithm used.

4.3.2 Network Depth

Two additional architectures were tested for the KMNIST dataset to evaluate the relation between network architecture and redistribution algorithm in detail (see Fig. 6). The only difference between these architectures is the number of layers and their respective number of neurons. The original architecture is 784-256-256-10, and the two other tested are 784-256-128-128-128-10 and 784-128-128-128-128-128-128-128-128-128-128-10. This yields about 269 thousand, 268 and 265 thousand weights respectively, i.e., a comparable amount of free parameters.

The better performance of second-order-based methods on deeper networks makes sense from a theoretical perspective. The more layers a network has, the lower the probability that a weight's magnitude is a good approximation of its importance, i.e., its effect on the training loss. Therefore, second-order-based methods are relatively better at removing the correct weights.

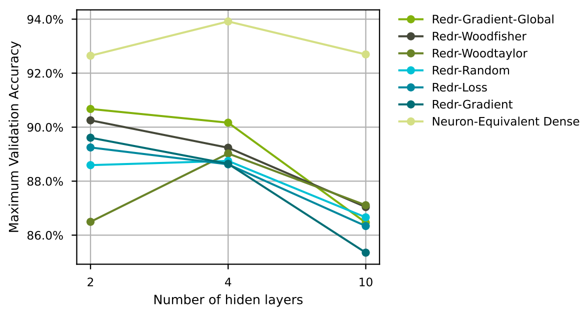

Figure 6: Validation accuracy while training using different redistribution algorithms and at a density of 0.1 on KMNIST dataset.

4.3.3 Woodtaylor performance and dampening factor

In our setting, the redistribution algorithms are applied while the network is still training. Therefore, the gradient can not be zero, and one would expect the Redr-Woodtaylor algorithm to yield better performance than Redr-Woodfisher, even more so since Singh and Alistarh [21] show improved performance even in a pruning context. Unfortunately, it seems that Redr-Woodtaylor is more sensitive to the choice of its two hyperparameters, the block size and the dampening term λ. A low block size impacts the performance of Redr-Woodtaylor more severely, reducing its validation accuracy by over 7%.

The sensitivity of the Woodtaylor algorithm is already mentioned by Singh and Alistarh [21]. A small ablation study was done on both datasets to evaluate the effect in our test setting. It shows that the network used for the HIGGS dataset does not seem to be affected much at all, while the network used for the KMNIST dataset does show a significantly increased test accuracy at $\lambda = 1$.

Ultimately, the missing fine-tuning of these parameters is the most probable cause of the bad performance of Redr-Woodtaylor in our testing.

5 Conclusion

All in all, it can be stated that global redistribution algorithms seem to give a clear advantage in test accuracy over layer-local methods. Especially redistributing via gradient information (Redr-Gradient-Global) results in consistent improvements without introducing an unreasonable overhead in computational complexity or memory.

The two new methods—Redr-Woodtaylor and Redr-Woodfisher–are harder to recommend despite their great performance. This is due to the introduced memory overhead and the additional hyperparameters that have to be fine-tuned for the given task. However, they might perform even better when paired with very deep networks, where the estimation of a weight's importance simply by its magnitude is less likely to be accurate. Moreover, these redistribution algorithms might be the best choice in applications, where memory consumption does not matter as much but where sparsity improves the execution speed.

For future projects we would like to investigate distribution algorithms on larger networks. Especially the performance on state-of-the-art image classifiers like convolutional neural networks and residual networks would be interesting. This would also make the evaluations more comparable to other papers.

In sum, all tested algorithms perform significantly better than a sparse network with fixed topology. We hope that the comparison provides a reference for future development or application of dynamic sparse training algorithms.

References

[1] Davis W. Blalock, Jose Javier Gonzalez Ortiz, Jonathan Frankle, and John V. Guttag. What is the State of Neural Network Pruning? *arXiv CoRR*, abs/2003.03033, 2020.

[2] Dan Cireşan, Ueli Meier, Luca Maria Gambardella, and Jürgen Schmidhuber. Deep, big, simple Neural Nets for Handwritten Digit Recognition. *Neural computation*, 22:3207–3220, December 2010.

[3] Tarin Clanuwat, Mikel Bober-Irizar, Asanobu Kitamoto, Alex Lamb, Kazuaki Yamamoto, and David Ha. Deep Learning for Classical Japanese Literature. *arXiv CoRR*, cs.CV/1812.01718, December 2018.

[4] Yann Le Cun, John S. Denker, and Sara A. Solla. Optimal Brain Damage. In David Touretzky, editor, *Advances in Neural Information Processing Systems*, volume 2. Morgan Kaufmann, 1990.

[5] Tim Dettmers and Luke Zettlemoyer. Sparse Networks from Scratch: Faster Training without Losing Performance. *arXiv CoRR*, abs/1907.04840, August 2019.

[6] Alexander Ertl. Accelerating Sparse Neural Networks on GPUs. *Bachelor Thesis*, 2021.

[7] Utku Evci, Trevor Gale, Jacob Menick, Pablo Samuel Castro, and Erich Elsen. Rigging the Lottery: Making All Tickets Winners. In *Proceedings of the International Conference on*

Machine Learning, 2020. URL http://proceedings.mlr.press/v119/evci20a.html.

[8] Utku Evci, Fabian Pedregosa, Aidan Gomez, and Erich Elsen. The Difficulty of Training Sparse Neural Networks. *arXiv CoRR*, abs/1906.10732, October 2020.

[9] Jonathan Frankle and Michael Carbin. The Lottery Ticket Hypothesis: Finding Sparse, Trainable Neural Networks. In *Proceedings of the International Conference on Learning Representations*, 2019. URL https://openreview.net/forum?id=rJl-b3RcF7.

[10] Trevor Gale, Erich Elsen, and Sara Hooker. The State of Sparsity in Deep Neural Networks. *arXiv CoRR*, abs/1902.09574, February 2019.

[11] Richard C. Gerum, André Erpenbeck, Patrick Krauss, and Achim Schilling. Sparsity through Evolutionary Pruning prevents Neuronal Networks from Overfitting. *Neural Networks*, 128: 305–312, 2020. ISSN 0893-6080.

[12] Song Han, Jeff Pool, John Tran, and William Dally. Learning both Weights and Connections for Efficient Neural Network. In *Proceedings of the International Conference on Neural Information Processing Systems*, 2015. URL https://proceedings.neurips.cc/paper/2015/file/ae0eb3eed39d2bcef4622b2499a05fe6-Paper.pdf.

[13] Babak Hassibi and David Stork. Second Order Derivatives for Network Pruning: Optimal Brain Surgeon. In S. Hanson, J. Cowan, and C. Giles, editors, *Advances in Neural Information Processing Systems*, volume 5. Morgan-Kaufmann, 1993.

[14] Torsten Hoefler, Dan Alistarh, Tal Ben-Nun, Nikoli Dryden, and Alexandra Peste. Sparsity in Deep Learning: Pruning and growth for efficient inference and training in neural networks. *arXiv CoRR*, abs/2102.00554, January 2021.

[15] Siddhant M. Jayakumar, Razvan Pascanu, Jack W. Rae, Simon Osindero, and Erich Elsen. Top-KAST: Top-K Always Sparse Training. *arXiv CoRR*, abs/2106.03517, June 2021.

[16] Diederik P. Kingma and Jimmy Ba. Adam: A Method for Stochastic Optimization. In *Proceedings of the International Conference on Learning Representations*, 2015. URL http://arxiv.org/abs/1412.6980.

[17] Junjie Liu, Zhe Xu, Runbin Shi, Ray C. C. Cheung, and Hayden K. H. So. Dynamic Sparse Training: Find Efficient Sparse Network from Scratch with Trainable Masked Layers. In *Proceedings of the International Conference on Learning Representations*, 2020. URL https://openreview.net/forum?id=SJlbGJrtDB.

[18] Shiwei Liu, Decebal Constantin Mocanu, Amarsagar Reddy Ramapuram Matavalam, Yulong Pei, and Mykola Pechenizkiy. Sparse evolutionary Deep Learning with over one million artificial neurons on commodity hardware. *Neural Computing and Applications*, 33(7), July 2020. ISSN 1433-3058.

[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*, 9(1):2383–2392, 2018. ISSN 2041-1723.

[20] Hesham Mostafa and Xin Wang. Parameter Efficient Training of Deep Convolutional Neural Networks by Dynamic Sparse Reparameterization. In *Proceedings of the International Conference on Machine Learning*, 2019.

[21] Sidak Pal Singh and Dan Alistarh. WoodFisher: Efficient Second-Order Approximation for Neural Network Compression. In H. Larochelle, M. Ranzato, R. Hadsell, M. F. Balcan, and H. Lin, editors, *Proceedings of the Conference on Neural Information Processing Systems*, volume 33, pages 18098–18109. Curran Associates Inc., 2020. URL https://proceedings.neurips.cc/paper/2020/file/d1ff1ec86b62cd5f3903ff19c3a326b2-Paper.pdf.

[22] Nikko Ström. Sparse Connection and Pruning in Large Dynamic Artificial Neural Networks. In *Proceedings of the European Conference on Speech Communication and Technology*, pages 2807–2810, Rhodes, Greece, 1997.

[23] Michael Zhu and Suyog Gupta. To prune, or not to prune: exploring the efficacy of pruning for model compression. In *Proceedings of the International Conference on Learning Representations*, 2018.