Accelerating Sparse Neural Networks on GPUs

Alexander Ertl[*]
Supervised by: Markus Steinberger,[†] Mathias Parger[‡]

Institute of Computer Graphics and Vision
Graz University of Technology
Graz / Austria

Abstract

Ever larger networks with parameters in the order of the hundreds of millions are required to fit increasingly complex and expansive datasets. In conjunction with ubiquitous machine learning applications on mobile or embedded platforms, this makes efficiency a vital property of artificial neural networks. Therefore we build upon work on replacing fully connected dense layers with trainable, evolving sparse layers in CSR encoding. This allows us to train networks at sparsity levels of up to 97% while considerably reducing the memory footprint as well as the number of computations thereby indicating that GPU accelerated sparse layers are a viable alternative to dense layers.

Keywords: GPU Acceleration, Deep Learning, Sparse Matrices

1 Introduction

The abundance of neural networks and the ever-increasing complexity of challenges they face has created the demand for faster and more efficient approaches. Specifically, architectures like VGGNet [11] require training parameters in the hundreds of millions. The increased usage of neural networks on mobile platforms with tightly constrained resources makes efficient networks, both in terms of computational resources as well as memory usage, even more desirable. To this end, this thesis will contrast the accuracy and efficiency of sparse neural networks to their dense counterparts as well as presenting a GPU accelerated sparse network implementation. Furthermore, we show that it is possible to train sparse neural networks from scratch with a sparsity of up to 97%, suffering only small penalties in accuracy while reducing the number of computations as well as the memory footprint during training by a considerable factor.

[*]ertl@student.tugraz.at
[†]steinberger@icg.tugraz.at
[‡]mathias.parger@icg.tugraz.at

1.1 Sparse Neural Networks

The bulk of the parameters comes from the weights. To be specific, there are $N_{in} \cdot N_{out}$ weights per layer, where N_{in} is the number of input features and N_{out} is the number of neurons i.e. outputs, whereas there are only N_{out} biases. This means, that reducing the number of biases in a network has little to no effect on the performance, however introducing sparsity to the weights can greatly improve efficiency. Since the weights are centred around zero and a large number of weights are very close to zero, many of them can be removed, i.e. set to zero, without greatly affecting the output of the neural network [12]. Pruning-based approaches, which focus on training dense networks and gradually removing weights close to zero have been shown to achieve sparsity levels of up to 95% while maintaining accuracy on par with dense networks [1]. Unfortunately, pruning, although resulting in a sparse network, does not actually reduce the cost of training. This hinders the increasingly common objective of online training on systems with fewer resources. Training sparse architectures from scratch has however proven more difficult, yet Mocanu et al. [9] have shown that networks with quadratically fewer parameters can be trained without suffering any penalty to accuracy and that sparsity can act as a form of regularisation to prevent overfitting during training. They achieve this by allowing the network topology to evolve after every epoch.

1.1.1 Sparse Matrix Representations

A sparse matrix is defined as a matrix where most of its elements are equal to zero. Sparse matrix representations store only values not equal to zero, are however only efficient at high sparsity levels. In order to reduce the memory footprint during training, we store the sparse matrices in compressed sparse row (CSR) form, which consists of three separate arrays: the row pointers, column indices and the values themselves. As illustrated in Figure 1, the row pointer at index i is simply an offset into the column index and values arrays and the number of elements in that row is defined as $n\_elements = row\_pointer_{i+1} - row\_pointer_i$. The consequence of this format is that high levels of sparsity are necessary for it to be effective. While the format requires sparsity levels above 50% to save memory since

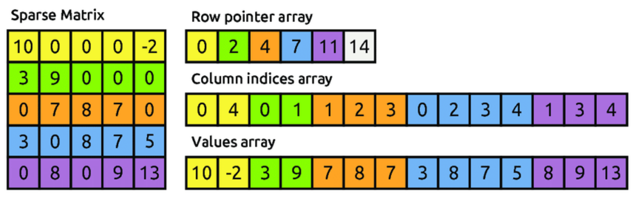


Figure 1: A sparse matrix in CSR form and the corresponding row pointer, column indices and values arrays [5].

every value requires both the value itself and a column index, our empirical results show that in order to reduce inference durations we require at least 30% and to reduce training duration at least 82% to compensate for the worse access patterns created by this format.

1.2 Our Contributions

We present a GPU accelerated sparse neural network with evolving topologies. This entails:

- An entirely sparse implementation of inference as well as backpropagation and weight updates.
- Advanced evolving topologies based upon SET [9].

This also enables us to directly compare GPU accelerated dense and sparse networks and examine the levels of sparsity required in order to reach a speedup as well as a reduction of memory.

2 Related Work

Numerous recent papers focus on regularisation techniques such as sparse variational dropout [10], L_0 regularisation [8] or sensitivity-driven approaches [12] to reduce the number of parameters during training. Molchanov et al. [10] used an adaptation of variational dropout to create sparse networks and were able to achieve state-of-the-art sparsity on LeNet-300-100 on MNIST handwritten digits while maintaining a similar accuracy to dense or pruned networks. Louizos et al. [8] on the other hand, solve a complex optimisation problem in order to minimise the L_0 norm, i.e. the number of parameters during training thus reducing the number of FLOPs performed, resulting in a speedup, while also maintaining similar levels of accuracy. Finally, Tartaglione et al. [12] use a sensitivity term to represent how sensitive the network's output is to specific parameters, i.e. how much a parameter influences the output of the network, in order to nudge less sensitive parameters towards zero, thus also resulting in sparse weight matrices. While all of these approaches have shown to produce highly sparse networks with good performance, none them achieve front-to-back sparsity while training, making training large networks on single GPUs, or systems with otherwise constrained resources difficult.

On another front, Chen et al. [3] utilise advances in algorithms for maximum inner product search (MIPS), to create SLIDE, an intelligent algorithm for training networks on CPUs. This approach uses MIPS sampling based on locality sensitive hashing to efficiently sample large activations that need to be computed. Every forward and backward pass, only a small set of active neurons is used for calculation of activations, costs and gradients. While this reduces the number of necessary computations it does not reduce the memory footprint.

Evci et al. [4] discuss the difficulties of training sparse neural networks and the tendency of sparse nets with fixed sparsity patterns to converge towards "bad" local minima or saddle points and indicate that allowing for changing network topologies during training might be necessary in order to find sparse solutions with a performance similar to dense solutions. A similar conclusion is reached by Alford et al. [1]: while pruning may in fact increase accuracy, pruning acting as a form of regularisation, training sparse networks with fixed topologies from scratch makes the training process unstable, i.e. convergence is unreliable and same accuracy levels can often not be reached, since training then becomes very dependent on initialisation. This is once again verified by Mocanu et al. [9] who therefore introduce sparse evolutionary training (SET). With SET, the network is initialised with a random sparse topology, however in addition to normal training procedure, after every epoch, a fraction of the smallest positive and largest negative weights are removed and replaced by a same number of new random weights, allowing the network to better fit the data.

Finally Wang et al. [13] examine the performance of sparse very deep neural networks on GPUs using cuSparse. They also use CSR to store the sparse matrices, however they train their networks with a fixed topology which has been shown to achieve relatively poor performance at higher levels of sparsity when compared to evolutionary variants [9]. Based upon GPU acceleration as well as SET, we attempt to provide more detailed insights into the behaviour and performance of evolving sparse neural networks.

3 Sparse Neural Networks

Our sparse neural networks consist of multiple sparse layers which in turn are composed of a dense array of biases and the sparse weight matrices in CSR form.

3.1 Inference

Sparse inference is identical to dense inference aside from the number of weights. Since the weights not present in the CSR representation are by definition zero, performing inference with a fully connected network with a sparse matrix or performing inference with a sparse network with the same sparse matrix represented in CSR format, are equivalent.

3.2 Training

The essence of training our sparse networks for classification challenges remained the same as for dense networks. After performing a sparse forward pass we calculated the costs and updated the weights with gradients computed by backpropagation. However in addition to these steps performed every iteration, after a certain number of epochs the network topology is allowed to evolve: Weights close to zero are thresholded and replaced by introducing the same number of the highest gradient values as new connections as illustrated in Figure 2. This evolution step improves convergence behaviour and achieved performance.

3.2.1 Initialisation

The number of weights per layer is calculated as $N_w = N_{in} \cdot N_{out} \cdot (1 - \text{sparsity})$. The weights were distributed evenly over all neurons of the layer, every neuron receiving exactly

$$n_w = \frac{N_w}{N_{out}}$$

weights. Every neuron's inputs were then partitioned into n_w equally sized partitions and the n_w weights distributed randomly in their respective partition.

The values arrays of the CSR representations were initialised using Xavier initialisation [6]: a normal distribution with a zero mean and

$$\sigma = \frac{2}{\sqrt{N_{in} + N_{out}}}$$

However, our results show that a slightly higher variance is beneficial to initial convergence. Therefore we modified the Xavier initialisation, to encompass the level of sparsity by changing σ to

$$\sigma = \frac{2}{\sqrt{(N_{in} + N_{out}) \cdot (1 - \text{sparsity})}}$$

3.2.2 Backpropagation and Gradient Calculation

The backpropagation step is completely sparse and we only calculate the gradients required to update the currently active weights. Since the number of required gradients is equal to the number of weights in the layer, the reduction of computations and the savings in memory regarding gradient computation and storage are proportional to the sparsity, e.g. a sparsity level of 95% results in a reduction of 95% of the necessary computations and memory. It should be noted that it is not necessary to store the gradients in CSR form, i.e. we only require the values array.

3.2.3 Weight Updates

Since the calculated gradients are already in the correct form, i.e. they are in the same order as the weights in the CSR values array, updating them is a simple vector add-multiply: $\boldsymbol{w}_{t+1} = \boldsymbol{w}_t - \eta \boldsymbol{\nabla}$ where η is the learning rate.

3.2.4 Evolving the Network Topology

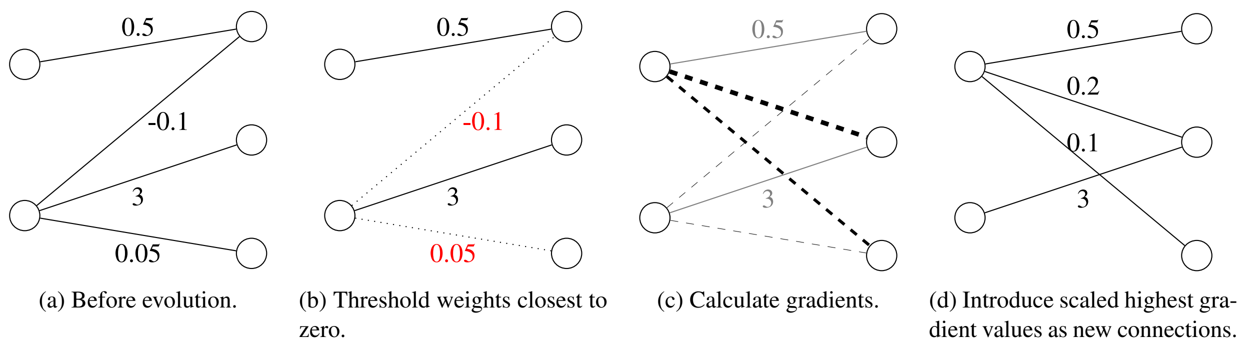
The key to successful sparse training is to allow for changing network topologies. SET [9] accomplishes this by thresholding a fraction ζ of weights close to zero in every layer and replacing them with random new connections initialised to zero. We have implemented a more precise algorithm, leading to faster convergence. As illustrated in Figure 2, while we also threshold a percentage of the smallest positive and largest negative weights every layer, instead of replacing these connections with random new connections, we instead insert the highest values of the gradient as new connections.

The intuition behind this approach is that the largest values of the gradient are the connections we should add to most reduce the training error. Since the weights not present in the CSR are zero, initialising these new weights to $w = -\eta \cdot \nabla$ is equivalent to performing dense training on these weights: $w_{t+1} = w_t - \eta \cdot \nabla$ since $w_t \equiv 0$. If we interpret the gradient as the "direction and rate of fastest increase" and are only allowed to select a fixed number of the gradient values as new connections, then we can minimise the cost function most quickly by selecting the largest values of the gradient. In this sense, this step is comparable to dense training while only updating the most important weights and setting less important weights to zero.

Instead of consistently thresholding a set percentage of the highest weights in every layer which would require sorting, we use an adaptive threshold τ. By comparing the target percentage of weights to threshold to the actual percentage computed after the thresholding step, we can adjust the adaptive threshold by multiplying or dividing it by a small factor. It is important to initialise τ as well as the adjustment factor to sufficiently conservative values, since an initially large τ could lead to an actual thresholding percentage close to 100% and an adjustment factor far from 1 could lead to a sudden blowup of the actual thresholding percentage during training which would lead to instabilities.

Once the number of thresholded weights has been determined, we perform quickselect to retrieve the same number of large gradient values. It is worth noting, that it is possible to increase or decrease sparsity levels in this step by selecting more or fewer gradient values than the number of thresholded weights.

The steps for calculating the costs in every layer are the same as in normal training, see sections 3.1 and 3.2.2. However, determining the fraction ζ of highest values of the gradients would require computing as well as sorting



0.5
-0.1
3
0.05
(a) Before evolution.
0.5
-0.1
3
0.05
(b) Threshold weights closest to zero.
0.5
3
(c) Calculate gradients.
0.5
0.2
0.1
3
(d) Introduce scaled highest gradient values as new connections.

Figure 2: Network topology evolution. The absolute gradient values in Figure c are represented by the line thickness.

the entire gradient vector, which as well as being computationally expensive would also render the so far reduced memory footprint useless. Since the gradient for the *ith* neuron is however computed as $\nabla_i = \boldsymbol{v}_i \cdot c_i$ where $\boldsymbol{v}_i$ is the input vector and c_i is the *ith* cost, we have found that sorting the costs and computing the gradient vectors for the k highest cost neurons is a good approximation. The ideal value for k strongly depends on the size of the layer, the sparsity as well as the target threshold percentage since the number of weights added to every neuron is given by $\frac{N_t}{k}$ and $N_t = N_{in} \cdot N_{out} \cdot (1 - \text{sparsity}) \cdot \text{TTP}$ where N_t is the number of thresholded values and TTP is the target threshold percentage. Since the number of thresholded weights per evolution step is constant (disregarding small oscillations induced by the adaptive threshold), setting

$$k = \sqrt{N_t} \tag{1}$$

strikes a balance between distributing new weights over as many neurons as possible and keeping the overhead for gradient calculation small as illustrated by Figure 3. Also, empirical analysis has shown that increasing k only seems to increase the percentage of highest gradient values found in the k highest cost neurons significantly until equation 1 holds. Furthermore, our results illustrated in Figure 4 show that increasing k too much or even using the entire gradient can be detrimental to convergence.

3.2.5 Update CSR

The final step is to update the CSR by removing the small weights and adding the already computed highest gradient values. This step is unfortunately difficult to parallelise since the arrays of the CSR format are stored in consecutive memory and removing a single weight would shift both the entire column index array as well as the values array; therefore, this step is implemented on the CPU. For efficient insertion, we pre-sort the highest gradient values according to their index in the new CSR. After sorting this step consists of simply copying values and their corresponding column index from the currently active CSR to a new CSR if the absolute value is larger than the threshold τ and inserting the negative highest gradient values multiplied by η as new connections.

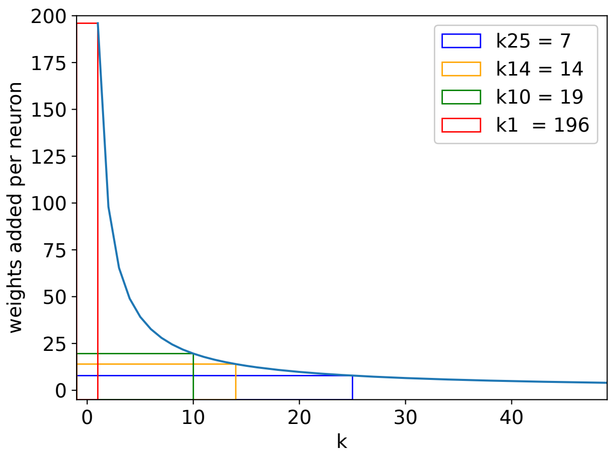


Figure 3: Setting $k = \sqrt{N_t} = \sqrt{196} = 14$ strikes a balance between distributing new weights over as many neurons as possible and keeping the overhead for gradient calculation small.

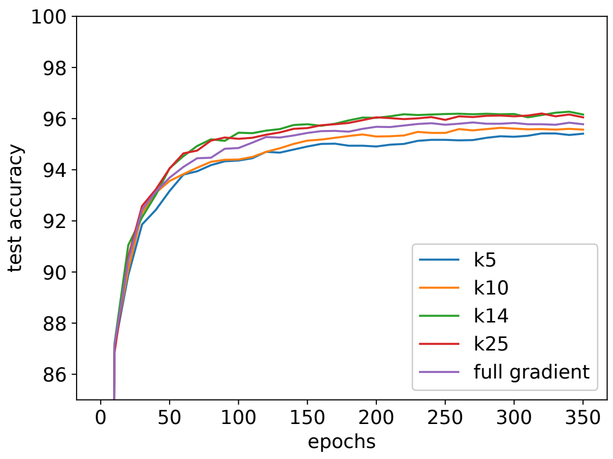


Figure 4: These results obtained on a 256x256x10 network with a sparsity of 97% and TTP = 0.1 suggest that calculating k according to equation 1, while reducing the overhead for gradient calculation, also benefits convergence.

Table 1: An overview of the datasets used for our evaluations including the number of classes and the number of training and test samples.

Name	Classes	Training	Test
MNIST	10	60k	10k
MNIST fashion	10	50k	10k
HIGGS	2	500k	100k

Table 2: The configuration and hyperparameters used during the evaluations.

Global Parameters	
Batch Size	512
Dense Parameters	
η	10^{-4}
Sparse Parameters	
η	$\frac{1}{2 \cdot \text{batchsize}}$
Target Threshold Percentage (TTP)	0.1%
τ	0.01
k	see equation 1

4 Evaluations

4.1 Datasets

The evaluations were performed on the MNIST [7], fashion MNIST [14] and the HIGGS [2] datasets, also listed in Table 1, with both the dense and sparse architectures with varying levels of sparsity as well as a dense CUBlas implementation. Both the MNIST and MNIST fashion datasets are image-based with 10 output classes and 28 by 28 feature vectors. The HIGGS dataset is a binary classification challenge with 28 input features of which the first 21 are based on measurements by particle detectors and the remaining 7 are derived features to facilitate classification. Although the HIGGS dataset consists of 11.000.000 samples, due to a lack of computational resources, we only used the first 600.000 samples split into 500.000 training and 100.000 test samples.

4.2 Configuration

We used the ReLU activation function for all hidden layers, the sigmoid activation for the final output layer and the mean squared error (MSE) cost function. The activation functions and cost function were kept consistent throughout all evaluations.

We performed mini batch gradient descent with a batch size of 512 and varying learning rates for dense and sparse networks. While all of our sparse networks performed well with $\eta = \frac{1}{2 \cdot \text{batchsize}}$, dense networks did not converge so we had to reduce the learning rate. For a summary of all hyperparameters see Table 2.

Table 3: A comparison of the evaluated networks when examining only the time per iteration as a factor.

Type/Sparsity	Inference		Training	
	ms	Speedup	ms	Speedup
Dense	296	-	9.48	-
Dense CUBlas	291	1.0x	-	-
Sparse 90%	63	4.7x	5.83	1.6x
Sparse 97%	45	6.6x	2.87	3.3x
Sparse 99%	39	7.6x	2.21	4.3x

4.3 Results

Our sparse networks generally performed well and while reducing both the duration per iteration of training and inference and the memory footprint were also able to achieve high levels of accuracy.

4.3.1 Computational Resources

Table 3, Figure 5 and Figure 6 contrast a dense network to sparse networks with different levels of sparsity regarding inference and training durations. For inference we measured the ms until all 10k test samples had been classified and the training time specifies the amount of time required for a single iteration in training. Inference times were measured on a 512-512-512-512-512-10 network and training times were measured on a 256-256-256-10 network; both measurements were performed on the fashion MNIST dataset.

While the dense CUBlas implementation was only marginally faster than our own dense implementation, using sparse networks we were able to achieve a speedup of almost 8x in inference and 4x in training at a sparsity level of 99%. In Figure 6 we also plotted a sparse network with no CSR updates which led to an interesting finding: not performing CSR updates actually increases training times. This is likely due to our initialisation where we distribute the weights over all neurons evenly. However, during evolution, some neurons are entirely dropped, which although not reducing the total amount of computations, makes our GPU implementation more efficient.

In terms of memory consumption, the savings of sparse networks are even more noticeable. Memory requirements for samples, caching mechanisms or output were not considered since these remain constant regardless of the network's sparsity. The results displayed in Figure 7 were normalized and show the memory usage in relation to a dense layer. Our sparse layers reach break-even at a sparsity level of around 30% and sparsity levels of 99% result in a 56 times smaller memory footprint than that of a dense layer.

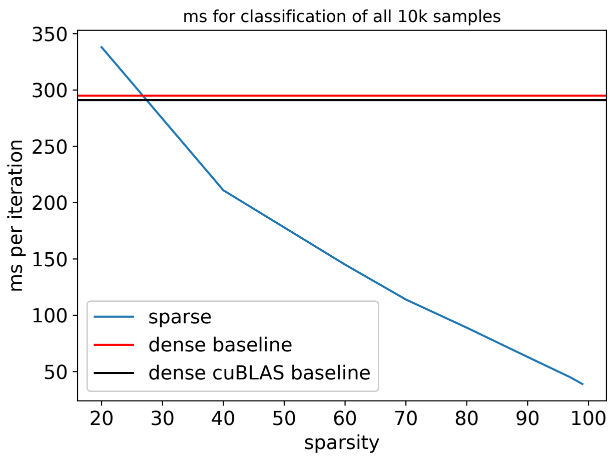


Figure 5: Inference: ms for classification of all 10k test samples.

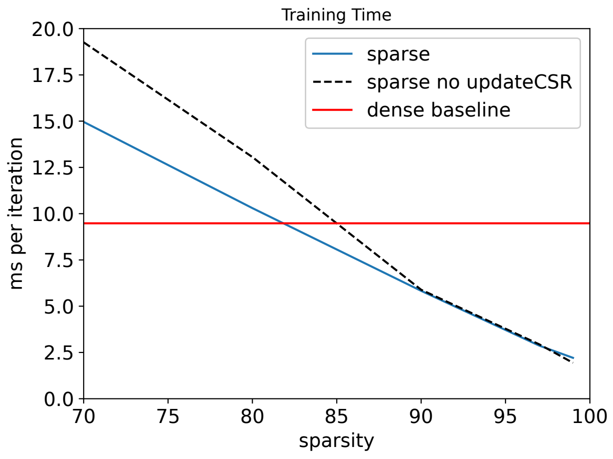


Figure 6: Training: ms for one iteration of training with a batch size of 512.

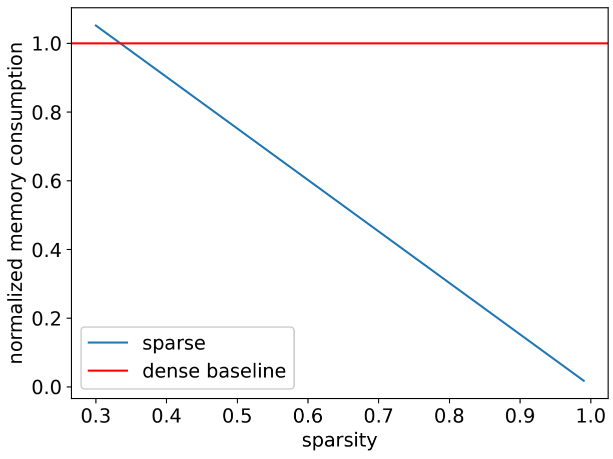


Figure 7: The memory required for the weights, biases and gradients by a 512x512 layer during training, normalized by the dense layer.

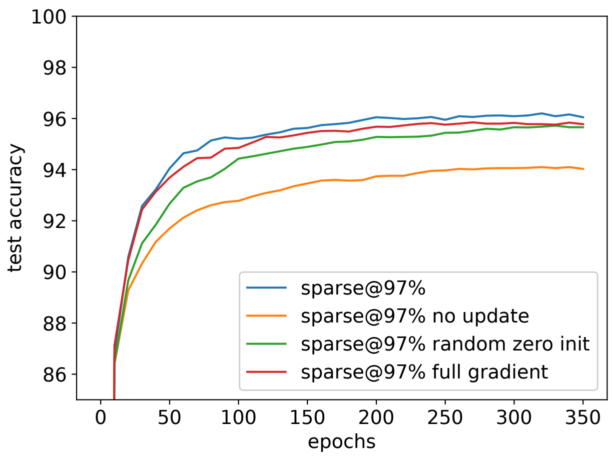


Figure 8: Training performed at a sparsity level of 97% with varying CSR update methods (our algorithm in blue).

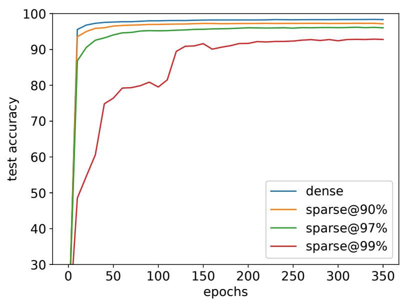
4.3.2 Accuracy

Figure 8 clearly shows the benefits of using the gradient to determine new connections rather than randomly adding new connections initialised to zero or not allowing for evolution at all.

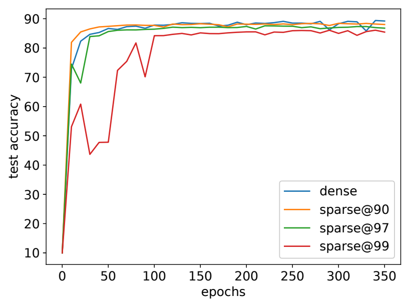
Although our sparse networks were able to converge consistently at the tested levels of sparsity, the dense network was able to achieve higher accuracies on both image classification datasets. However the sparse networks at 90% sparsity displayed faster convergence on the fashion MNIST and HIGGS datasets. This behaviour was especially pronounced on the HIGGS dataset as can be seen in Figure 9c where the sparse network was also able to achieve an overall higher accuracy. The precise results are all listed in Table 4.

The networks at 97% sparsity converged at accuracies approximately 1% lower than the networks at 90% and networks at 99% already show considerable drops in accuracy. Furthermore, figure 9 shows that higher sparsities also result in strong oscillations in both training error and test accuracy as the network evolves during the initial phase of training. The oscillations however flattened out on all networks as training progressed resulting in a smooth curve.

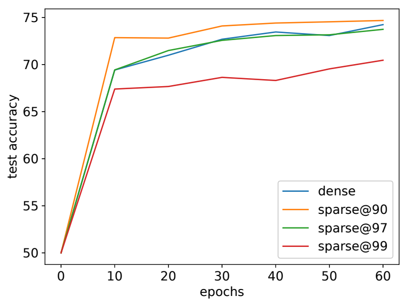
To highlight the importance of network evolution during sparse training, we trained multiple sparse networks on the MNIST dataset at the sparsity levels 90%, 97% and 99% while either using our algorithm, or not allowing for the network topology to change. As illustrated in Figure 10 our algorithm at 90% was only able to beat no evolution at the same sparsity by 0.25%, however while becoming more noticeable at 97%, at 99% the not evolving network converged extremely poorly but our algorithm was still able to achieve good results.



(a) MNIST test accuracy



(b) Fashion MNIST test accuracy



(c) HIGGS test accuracy

Figure 9: Convergence behaviour on the MNIST, fashion MNIST and HIGGS datasets. While the sparse networks with a sparsity of 90% always came within 1% of dense networks and in fact outperformed the dense network on the HIGGS testset and 97% was sufficient to come close, the 99% sparse network was not able to achieve similar performance.

Table 4: The results of our evaluations performed on the test sets with the respective sparsity levels and "-" for dense networks.

Dataset	Sparsity	Accuracy
MNIST	-	**98.34%**
	90%	97.30%
	97%	96.16%
	99%	92.88%
MNIST fashion	-	**89.40%**
	90%	88.43%
	97%	87.58%
	99%	86.08%
HIGGS	-	74.24%
	90%	**74.68%**
	97%	73.74%
	99%	70.46%
	95-99-99-90%	72.64%

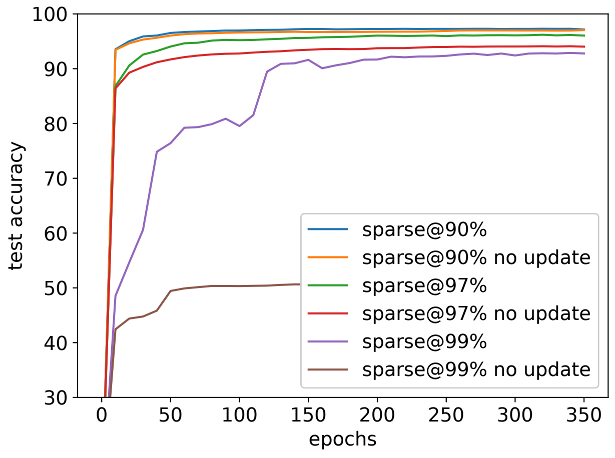


Figure 10: A comparison of our algorithm and not allowing the network topology to change on the MNIST dataset. While not allowing for evolution at lower sparsity does not greatly affect performance, it is vital at higher levels.

5 Conclusion and Future Work

We have shown that sparse layers are a viable alternative to dense layers, both increasing performance by lowering the number of necessary computations as well as reducing the memory footprint while maintaining a high accuracy. Furthermore, sparse layers can successfully be accelerated on the GPU which opens up the possibility of larger networks which are currently in demand.

5.1 Future Work

Techniques such as pruning have already shown that very sparse networks can reach accuracies on par with their dense counterparts. While the accuracies our sparse networks reached are respectable, further improvements to the weight replacement algorithm could likely achieve even better results. An interesting problem illustrated in Figure 11 may also have contributed to the deterioration of performance at higher levels of sparsity: large weights of a neuron that has no further connections to the following layer are essentially "lost" since the cost for the neuron will remain zero and the weight is too large to be removed by simple thresholding. This problem could be approached by removing weights of neurons whose cost is exactly zero during backpropagation.

Our sparse networks were trained without optimizers e.g. ADAM, which of course increased the number of epochs until convergence. Implementing such an optimizer while keeping the benefits of GPU accelerated training would further increase the usability of sparse layers in practical applications. Simply keeping the second moment of a dense gradient in memory however defeats part of the purpose of sparse layers: namely reducing the memory footprint during training.

References

[1] Simon Alford, Ryan Robinett, Lauren Milechin, and Jeremy Kepner. Training behavior of sparse

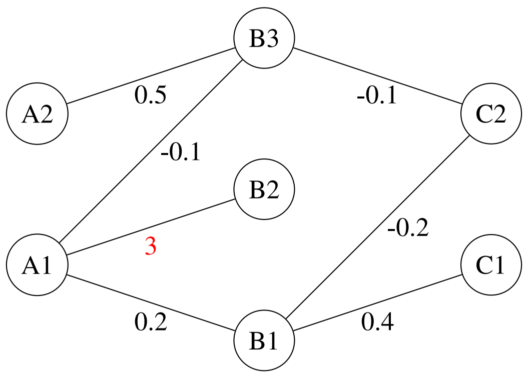


Figure 11: This graphic illustrates a problem random initialisation can lead to: The weight from A1 to B2 does not contribute to the output of the network, will however never be thresholded, thus leading to a "lost" weight.

neural network topologies. *2019 IEEE High Performance Extreme Computing Conference (HPEC)*:1–6, September 2019. DOI: 10.1109/HPEC.2019.8916385. arXiv: 1810.00299. URL: http://arxiv.org/abs/1810.00299.

[2] P. Baldi, P. Sadowski, and D. Whiteson. Searching for exotic particles in high-energy physics with deep learning. *Nature Communications*, 5(1):4308, July 2, 2014. ISSN: 2041-1723. DOI: 10.1038/ncomms5308. URL: https://www.nature.com/articles/ncomms5308. Number: 1 Publisher: Nature Publishing Group.

[3] Beidi Chen, Tharun Medini, James Farwell, Sameh Gobriel, Charlie Tai, and Anshumali Shrivastava. SLIDE : in defense of smart algorithms over hardware acceleration for large-scale deep learning systems. *arXiv:1903.03129 [cs]*, February 29, 2020. arXiv: 1903.03129. URL: http://arxiv.org/abs/1903.03129.

[4] Utku Evci, Fabian Pedregosa, Aidan Gomez, and Erich Elsen. The difficulty of training sparse neural networks. *arXiv:1906.10732 [cs, stat]*, July 17, 2019. arXiv: 1906.10732. URL: http://arxiv.org/abs/1906.10732.

[5] Figure 4.9: a sparse matrix and its corresponding CSR row pointer,... ResearchGate. URL: https://www.researchgate.net/figure/A-sparse-matrix-and-its-corresponding-CSR-row-pointer-column-indices-and-values-arrays_fig11_274379571 (visited on 09/17/2020).

[6] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*. Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics, pages 249–256. JMLR Workshop and Conference Proceedings, March 31, 2010. URL: http://proceedings.mlr.press/v9/glorot10a.html. ISSN: 1938-7228.

[7] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, November 1998. ISSN: 1558-2256. DOI: 10.1109/5.726791. Conference Name: Proceedings of the IEEE.

[8] Christos Louizos, Max Welling, and Diederik P. Kingma. Learning sparse neural networks through l0 regularization. *arXiv:1712.01312 [cs, stat]*, June 22, 2018. arXiv: 1712.01312. URL: http://arxiv.org/abs/1712.01312.

[9] 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, December 2018. ISSN: 2041-1723. DOI: 10.1038/s41467-018-04316-3. URL: http://www.nature.com/articles/s41467-018-04316-3.

[10] Dmitry Molchanov, Arsenii Ashukha, and Dmitry Vetrov. Variational dropout sparsifies deep neural networks. *arXiv:1701.05369 [cs, stat]*, June 13, 2017. arXiv: 1701.05369. URL: http://arxiv.org/abs/1701.05369. version: 3.

[11] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv:1409.1556 [cs]*, April 10, 2015. arXiv: 1409.1556. URL: http://arxiv.org/abs/1409.1556.

[12] Enzo Tartaglione, Skjalg Lepsøy, Attilio Fiandrotti, and Gianluca Francini. Learning sparse neural networks via sensitivity-driven regularization:11, 2018.

[13] Jianzong Wang, Zhangcheng Huang, Lingwei Kong, Jing Xiao, Pengyu Wang, Lu Zhang, and Chao Li. Performance of training sparse deep neural networks on GPUs. In *2019 IEEE High Performance Extreme Computing Conference (HPEC)*. 2019 IEEE High Performance Extreme Computing Conference (HPEC), pages 1–5, September 2019. DOI: 10.1109/HPEC.2019.8916506. ISSN: 2643-1971.

[14] Han Xiao, Kashif Rasul, and Roland Vollgraf. Fashion-MNIST: a novel image dataset for benchmarking machine learning algorithms. *arXiv:1708.07747 [cs, stat]*, September 15, 2017. arXiv: 1708.07747. URL: http://arxiv.org/abs/1708.07747.