Inference, learning and attention mechanisms that exploit and preserve sparsity in CNNs

Conference Paper

Author(s):
Hackel, Timo; Usvyatsov, Mikhail; Galliani, Silvano; Wegner, Jan D.; Schindler, Konrad

Publication date:
2018

Permanent link:
https://doi.org/10.3929/ethz-b-000320285

Originally published in:
11269, https://doi.org/10.1007/978-3-030-12939-2_41

Inference, Learning and Attention Mechanisms that Exploit and Preserve Sparsity in CNNs

Timo Hackel, Mikhail Usvyatsov, Silvano Galliani,
Jan D. Wegner, Konrad Schindler

Photogrammetry and Remote Sensing, ETH Zürich

Abstract. While CNNs naturally lend themselves to densely sampled data, and sophisticated implementations are available, they lack the ability to efficiently process sparse data. In this work we introduce a suite of tools that exploit sparsity in both the feature maps and the filter weights, and thereby allow for significantly lower memory footprints and computation times than the conventional dense framework, when processing data with a high degree of sparsity. Our scheme provides *(i)* an efficient GPU implementation of a convolution layer based on direct, sparse convolution; *(ii)* a filter step within the convolution layer, which we call *attention*, that prevents fill-in, i.e., the tendency of convolution to rapidly decrease sparsity, and guarantees an upper bound on the computational resources; and *(iii)* an adaptation of back-propagation that makes it possible to combine our approach with standard learning frameworks, while still exploiting sparsity in the data and the model.

1 Introduction

Deep neural networks are nowadays the most successful tool for a wide spectrum of computer vision task [18, 23, 31]. A main reason for their spectacular comeback, perhaps even the single most important factor, is the enormous gain in computational efficiency brought about by massively parallel computing on GPUs. Both the response maps (feature maps) within the neural network and the parameters (filter weights) of the network form regular grids that are conveniently stored and processed as tensors. However, while naturally suited for image processing, regular grids are a suboptimal representation for data such as line drawings or irregular 3D point clouds (Fig. 1). E.g., the latter are typically acquired with line-of-sight instruments, thus the large majority of points lies on a small number of 2D surfaces. When represented as 3D voxel grid they therefore exhibit a high degree of sparsity, as most voxels are empty; while at the same time 3D data processing with CNNs is challenged by high memory demands [3, 38, 24, 14].

A counter-measure is to make explicit the sparsity of the feature maps and store them in a sparse data representation, see Fig. 2. Moreover, it can also be beneficial to represent the CNN parameters in a sparse fashion to improve runtime and – perhaps more important for modern, deep architectures – memory footprint; especially if the sparsity is promoted already during training through appropriate

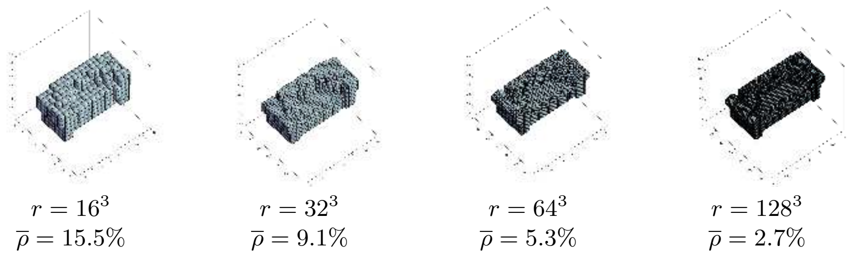
$r = 16^3$
$\overline{\rho} = 15.5\%$
$r = 32^3$
$\overline{\rho} = 9.1\%$
$r = 64^3$
$\overline{\rho} = 5.3\%$
$r = 128^3$
$\overline{\rho} = 2.7\%$

Fig. 1: Sparsity in 3D data analysis. The density $\overline{\rho}$ of occupied voxels is low in 3D data from Modelnet40, and decreases with increasing voxel resolution.

regularisation. It is obvious that, in a sufficiently sparse setting, a significant speed-up can be achieved by performing convolutions *directly*, incrementally updating a layer's output map only where there are non-zero entries in the input map as well as non-zero filter weights. This has recently been confirmed independently by two concurrent works [8, 27]. Direct convolution guarantees that only the minimum number of necessary operations is carried out. However, the selective updating only at indexed locations makes parallelisation harder. This may be the reason why, to our knowledge, no practical implementations with sparse feature maps exist. In this work we develop a framework to exploit both sparse feature maps and sparse filter parameters in CNNs. To that end *(i)* we provide a sparse *Direct Convolution Layer*, as well as sparse versions of the *ReLU* and *max-pooling* layers; *(ii)* we extend the back-propagation algorithm to preserve sparsity and make our sparse layers usable with existing optimisation routines that are available in modern deep learning frameworks, which have been designed for dense data; *(iii)* we propose to add a density-dependent regulariser that encourages sparsity of the feature maps, and a pruning step that suppresses small filter weights. This regularisation in fact guarantees that the network gets progressively faster at its task, as it receives more training. All these steps have been implemented on GPU as extensions of *Tensorflow*, for generic n-dimensional tensors. The source code is available at `https://github.com/TimoHackel/ILA-SCNN`. In a series of experiments, we show that it outperforms its dense counterpart in terms of both runtime and memory footprint when processing sufficiently sparse data.

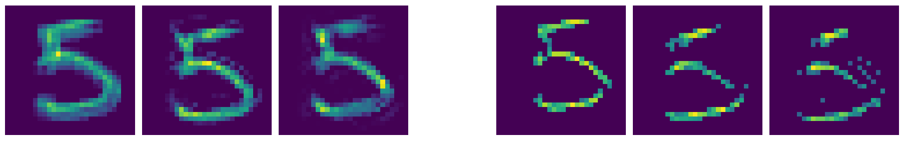

Fig. 2: With suitable computational mechanisms, sparsity in the input can be preserved throughout the CNN. Shown are the activations for one channel of the 1st, 2nd and 3rd convolution layers on MNIST for a dense network (left) and for a sparse network with upper bound $\rho_{up} = 15\%$ (right).

2 Related work

Dense CNN for sparse data

Neural networks, usually of the deep, convolutional network flavour, offer the possibility to completely avoid heuristic feature design and feature selection. They are at present immensely popular in 2D image interpretation. Recently, deep learning pipelines have been adapted to voxel grids [28, 19, 24, 38], RGB-D images [34] and video [17], too. Being completely data-driven, these techniques have the ability to capture appearance as well as geometric object properties. Moreover, their multi-layered, hierarchical architecture is able to encode a large amount of contextual information. A general drawback when directly applying 3D-CNNs to (dense) voxel grids derived from (originally sparse) point clouds is the huge memory overhead for encoding empty space. Computational complexity grows cubically with voxel grid resolution, but in fact high resolution would only be needed at object surfaces.

Data sparsity

Therefore, more recent 3D-CNNs exploit the *sparsity of occupied voxels* prevalent in practical voxel grids. In [9] a sparse CNN is introduced, which is however limited to small resolutions (in the paper, up to 80^3) due to decreasing sparsity in convolutional layers. Another strategy is to resort to an octree representation [32, 35]. Since the octree partitioning depends on the object at hand, an important question is how to automatically adapt to previously unseen objects. While [32] assume the octree structure to be known at test time, [35] learn to predict it together with the labels. In [13] a coarse-to-fine scheme is used to hierarchically predict the values of small blocks of voxels in an octree. Another strategy is to rely only on a small subset of discriminative points, while neglecting the large majority of less informative ones [21, 29, 30]. The idea is that the network learns how to select the most informative points and aggregates information into global descriptors of object shape via fully-connected layers. This allows for both shape classification and per-point labeling using only a small subset of points, resulting in significant speed and memory gains. Bilateral convolutional layers [16] map the data into permutohedral space, thus also exploiting sparsity in the data, but do not have a mechanism to exploit parameter sparsity. Recently [11, 10] advocate the strategy to perform convolutions only on non-zero elements in the feature map and find correspondences via hash tables. However, limiting activations to non-zero inputs can increase the error and slow down learning.

Parameter sparsity

Several works address the situation that the model *parameters* are sparse. Denil *et al.* [5] reduce the network parameters by exploiting low rank matrix factorisation. Liu *et al.* [22] exploit the decomposition of matrices to perform efficient convolutions with sparse kernel parameters. Some authors [15, 6] approximate convolutional filters to achieve a faster runtime, moreover it has been proposed to reduce the number of parameters by pruning connections [12] or imposing sparsity in an already trained network [37].

Direct convolutions

The works [27, 8, 26] are the most related ones to our approach, in that they also perform convolutions in a direct manner to efficiently exploit sparsity in network parameters and feature maps. While [27] use compressed rows as sparse format for the filter parameters, neither [8] nor [27] uses a sparse format for both filter parameters and feature maps. Parashar *et al.* [26] implement sparse convolutions on custom-designed hardware to achieve an energy- and memory-efficient CNN. Even though all three works follow a similar idea, only the latter exploits sparsity in both the parameters and the data, with compressed sparse blocks; but requires dedicated, non-standard hardware.

3 Method

It is a general theme of computing to speed up computations and reduce memory usage by exploiting sparsity in the data. In the following section, we propose a number of ways to do the same for the specific case of neural network layers, always keeping in mind the specific requirements and limitations of modern GPU architectures. Throughout, sparse tensors are represented and manipulated in a format similar to *Coordinate List*[1], which stores indices into the sparsely populated grid and the corresponding data entries in separate tensors, and is available in the "SparseTensor" implementation of *Tensorflow.* To minimise memory overhead, the indices of the form $\{batch, index_x, index_y, ..., channel\}$ are compressed into unique $1D$ keys and only expanded when needed.

To achieve coalesced memory access, which permits efficient caching, the tensors for feature maps are sorted w.r.t. batches and within each batch w.r.t. channels. Likewise, filter weights are sorted w.r.t. the output channels and within each channel w.r.t. the input channels. Compared to dense tensors, the sparse representation naturally adds some overhead. For instance, in our implementation we use 64 bit keys, and 32 bit depth for feature maps. Consequently, storing a dense feature map (100% density) required 3× more memory. For densities <33% the sparse representation is more efficient, and at low densities the savings can be quite dramatic, e.g., at density 1% it uses 97% less memory.

Sparse convolution

Our convolutional layer is designed to work with sparse tensors for both feature maps and filter weights. Feature maps are updated incrementally with *atomic*

[1] We have also experimented with other sparse formats, like compressed sparse blocks; but found none of them to work as well, in part due to limitations and idiosyncrasies of current GPU hardware.

Algorithm 1 Direct Sparse Convolution with Attention

```
1: decompress filter and data indices from 1D to $k$D
2: for $b \in [0 : batch\_count]$ do
3:     for $oc \in [0 : out\_channel\_count]$ do
4:         initialize dense $buffer$ with 0
5:         for $ic \in [0 : in\_channel\_count]$) do
6:             for $\{id, val\} \in \text{data}(b, ic)$ do
7:                 for $\{fid, fval\} \in filter(oc, ic)$ do
8:                     compute $uid$ with get_update_id($id, fid$)
9:                     atomically add $val \cdot fval$ to $buffer$ at $uid$
10:        get $non\text{-}zero$ entries from $buffer$
11:        add $bias$ to non-zero entries in $buffer$
12:        select $k$ largest responses from $non\text{-}zero$ entries
13:        compress ids of $k$ largest responses from $k$D to 1D
14:        write $k$ largest features and ids as sparse output
```

operations, c.f. Algorithm 1, where atomic operations are small enough to be thread-safe, even if no locking mechanism is used. In that respect it is similar to two concurrent works [26, 27]. In practice, the incremental update is limited by the current hardware design, since atomic operations are slightly slower than non-atomics: at present, off-the-shelf GPUs do not offer native support for atomic floating point operations in shared memory, although they do for more costly CAS instructions. Yet incremental updating is significantly faster, because it performs only the minimum number of operations necessary to obtain the convolution, while avoiding to multiply or add zeros.

The sparse convolution is computed sequentially per output channel and batch, but in parallel across input channels, features and filter weights. Its result is stored in a temporary, dense buffer with batch size and channel depth 1. This buffer increases quadratically for 2D images, cubically for 3D volumes, *etc.*. Still, it is in practice a lot smaller than a typical dense tensor with correct dimensionality for batches and channels, such that volumes up to 512^3 can be processed on a single graphics card (Nvidia Titan Xp, 12 GB).

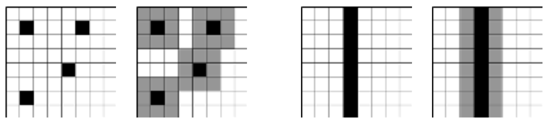

Fig. 3: Fill-in (loss of sparsity) due to convolutions depends on the data distribution. Uniformly distributed data is affected most strongly, e.g., in 3D every $3 \times 3 \times 3$ filter will increase the density by a factor of 27, until data is dense.

Algorithm 2 Backpropagation for convolutional layer

```
1: initialize $bp\_data$ with shape($input\_values$) and 0
2: initialize $bp\_filter$ with shape($filter\_weights$) and 0
3: decompress filter and data indices from 1D to $k$D
4: for $b \in [0 : batch\_count]$ do
5:     for $oc \in [0 : out\_channel\_count]$ do
6:         initialize dense $buffer$ with gradients($b, oc$)
7:         for $ic \in [0 : in\_channel\_count])$ do
8:             for $\{id, val\} \in$ data($b, ic$) do
9:                 for $\{fid, fval\} \in filter(oc, ic)$ do
10:                    compute $uid$ with get_update_id($id, fid$)
11:                    get gradient $g$ from $buffer$ at $uid$
12:                    atomically add $g \cdot fval$ to $bp\_data$ at $id$
13:                    atomically add $g \cdot val$ to $bp\_filter$ at $fid$
```

Preserving sparsity with attention

Convolution with kernels larger than (1×1) generates fill-in, i.e., it reduces the sparsity of a feature map, by construction. See Fig. 3. This "smearing out" of the sparse inputs usually only has a small influence on the output of the network, see the experiments. But it considerably increases memory consumption and runtime, especially when occurring repeatedly over multiple layers. In order to guarantee upper bounds on the memory footprint and runtime of the network, we apply a k-selection filter [2] on each output channel, keeping only the k strongest responses. This can be seen as an approximation of the exact convolution where small responses are suppressed, but using an adaptive threshold that suppresses only as many values as necessary to maintain the desired degree of sparsity.

The parameter k controls the sparsity, and thus the memory consumption, of the convolutional layers. Processes that aim to optimally direct and manage the limited resources available for some cognitive task are commonly referred to as *attention*. We have implemented two versions of our simple attention mechanism via k-selection: *(i)* acts on the raw responses, so it prefers large positive responses, making it similar to a rectified linear unit; *(ii)* picks the k values with the largest absolute values, expressing a preference for responses with large magnitude. The time complexity of this layer to convolve data of dimension k, resolution s_d and density ρ_d, with filters of size s_f and density ρ_f, is

$$O\big((\rho_d \cdot \rho_f \cdot s_f^k \cdot c_{in} + log(s_d^k)) \cdot s_d^k \cdot c_{out} \cdot b\big) \ , \tag{1}$$

with b the number of batches and c_{in}, c_{out}, the number of input and output channels, respectively.

Pooling layer

Our sparse pooling layer has three straight-forward stages. First, assign features to an output (hyper-)voxel, by dividing the data channels of their index by strides.

Second, sort the data w.r.t. voxels, so that responses within the same voxel are clustered together. Third, apply the pooling operator separately to each cluster. The time complexity for this is

$$O\big(\rho_d \cdot s_d^k \cdot log(\rho_d \cdot s_d^k) \cdot c_{in} \cdot b\big) \ . \tag{2}$$

Direct sparse backpropagation

Our target for back-propagation is again to skip operations that can be avoided due to sparsity. We must propagate error gradients only to all those features which have produced evidence, in the form of non-zero responses during the forward pass. Yet, the performance issues already discussed for the forward pass apply also to the backward pass: back-propagation through a convolution layer is itself a convolution that leads to fill-in, increasing memory use and runtime.

Contrary to the forward pass, it is not advisable to bound the fill-in with the k-selection technique, since this will not prevent the back-propagated error gradients from spreading to zero activations and vanishing, while smaller gradients flowing towards non-zero activations might be missed. It is evident that this effect could seriously slow down the training process. Hence, we propose to use a stricter back-propagation, which only propagates errors L to non-zero features x and model parameters w:

$$\frac{\partial L}{\partial x_i} = \begin{cases} 0 & \text{for } x_i = 0 \\ \frac{\partial L}{\partial y}\frac{\partial y}{\partial x_i}, \text{else} \end{cases} \quad , \quad \frac{\partial L}{\partial w_i} = \begin{cases} 0 & \text{for } w_i = 0 \\ \frac{\partial L}{\partial y}\frac{\partial y}{\partial w_i}, \text{else} \end{cases} \ . \tag{3}$$

Here, weights are considered equal to zero only if they have been explicitly removed by pruning, so as to avoid suppressing the gradients of weights that pass through $w_i = 0$ while changing sign. Note the similarity of our approximated back-propagation to back-propagation through any layer with $ReLU$ activation: Conventional back-propagation sets values to zero in function of the layer output y_i, whereas we do so in function of the input x_i.

Neglecting zero-elements slightly reduces the efficiency per learning iteration, since not all error gradients are propagated anymore. However, it has a number of advantages: *(i)* The tensors used for back-propagation have fixed size and shape. Therefore, one can still use optimisation frameworks that have been designed for dense data, and expect fixed and known array dimensions; *(ii)* By considering only gradients on non-zero elements of the forward pass, back-propagation can be implemented in a clean and transparent manner. E.g. for convolutional layers one obtains Algorithm 2, which is very similar to Algorithm 1; *(iii)* Once a filter weight has been set to zero, it will remain zero. Below, we will describe how this property can be used to guarantee that the network gets progressively faster at its task as the learning proceeds and it sees more training data.

Adaptive density regularisation

There is a computationally more efficient way to encourage sparsity of the feature maps, such that the sparsity thresholds are rarely exceeded in the first place, and the more costly k-selection step is avoided. The *ReLU* non-linearity used in most modern CCNs, by definition, truncates negative activations to zero while leaving positive ones unchanged. This means that we can include a regularisation that pushes down the values (not magnitudes) of filter weights and biases. By doing so, more weights will be driven into the negative region, where they are extinguished by the subsequent *ReLU*. Moreover, the same idea can be used to reduce sparsity when it is not needed, and optimally use the available resources: when too few activations are > 0, one drives the filter weights up, so that fewer of them are suppressed by the *ReLU*. To achieve the desired effect, we simply add a bias b to the L_2-regularisation, so that the regulariser becomes $\sum(w+b)^2$. The scalar b is positive when the density ρ is too large, and negative when it is overly small:

$$b = \begin{cases} o + b_1 \cdot (\rho - \rho_{up}) & \rho > \rho_{up} \quad \text{(exceeds available resources)} \\ -b_2 \cdot (\rho_{up} - \rho) & \rho \leq \rho_{up} \quad \text{(not using available resources)} \end{cases} \tag{4}$$

with ρ_{up} the upper bound implied by the k-selection filter, and o, b_1, $b_2 \geq 0$ control parameters. The offset o adds an additional penalty for exceeding the available resources, since this case requires the use of the k-selection filter and, hence, increases the computational load.

Parameter pruning

As explained above, our training algorithm has the following useful properties: *(i)* The regulariser encourages small model parameters. *(ii)* The sparse back-propagation ensures that, once set to zero, model parameters do not reappear in later training steps. Together, these two suggest an easily controllable way to progressively favour sparsity during training: At the end of every training epoch we screen the network for weights w that are very small, $|w_i| < \epsilon$. If the magnitude of a weight w_i stays low for two consecutive epochs (meaning that it was already close to zero before, and that did not change during one epoch of training) we conclude that it has little influence on the network output and prune it (one-warning-shot pruning). We note that a small weight should not be pruned when first detected, without warning shot: it could have a large gradient and just happen to be at its zero-crossing from a large positive to a large negative value (or vice versa) at the end of the epoch. On the contrary, it is very unlikely to observe a weight exactly at its zero-crossing twice in a row.

Since a weight, once set to zero, will not reappear with our sparse back-propagation, every pruning can only reduce the number of non-zero weights. It is thus guaranteed that the network become sparser, and therefore also faster at the task it is learning, as it sees more training data. Note, it is well documented that biological systems get faster at a task with longer training [33, 25].

4 Evaluation

In this section we evaluate the impact of density upper bounds and regularisation on runtime and classification accuracy. The sparse network structures were

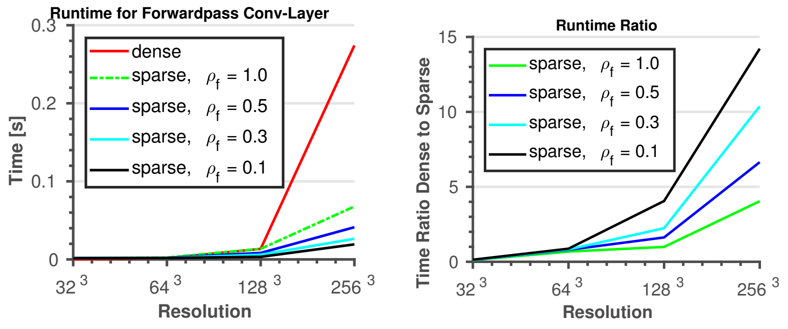

Fig. 4: Runtime [s] of a dense convolution layer in Tensorflow and of our sparse convolution layer, for different resolutions r^3. At high resolutions the sparse version is much more efficient.

implemented into the Tensorflow framework and programmed in C++/CUDA with a python interface. Our experiments were run on PCs with Intel Core i7 7700K processors, 64GB RAM and Titan Xp GPUs. Detailed specifications about the different CNN variants used in the experiments (in both sparse and dense versions) are given in the appendix.

To start with, we use a synthetic dataset of sparse random tensors to evaluate the memory footprint and runtime of our convolutional layer and to compare it against the dense layers of Tensorflow version 1.4 (compiled with Cuda 9.0 and CuDNN 6.0). We conduct different experiments to evaluate the effects of our sparse network on classification accuracy: First, the impact of upper bounds on classification is evaluated by performing a grid search on the upper bound ρ_{up} in the convolutional layers. For this experiment the MNIST data set [20] is used, as it is small enough to perform grid search in a reasonable amount of time and can be interpreted as sparse data (1D lines in 2D images). Second, the effects of pruning on runtime and classification accuracy are shown using the Modelnet data set [38], by varying the strength λ of the regularisation. Modelnet40 provides 3D CAD models of 40 different classes. Furthermore, the classification results of different baseline methods are compared on this data set. Modelnet40 is trained for 90 epochs with learning rate 0.001, using, using *adagrad* [7].

Resolution	32^3	64^3	128^3	256^3	512^3
Dense [GB]	0.04	0.27	2.15	17.18	137.28
Sparse 32 [GB]	$2 \cdot 10^{-3}$	$8 \cdot 10^{-3}$	0.03	—	—
Sparse 64 [GB]	$3 \cdot 10^{-3}$	0.013	0.05	0.2	0.8
Sparse Temp [GB]	$3 \cdot 10^{-4}$	0.002	0.016	0.13	1.07

Table 1: Memory consumption of a dense conv layer in Tensorflow and of our sparse conv layer, for different resolutions r^3, with $\rho_{up} = 1/r$, minibatch size 32 and output depth 8. At high resolutions the sparse version is much more efficient.

Runtime and Memory Footprint

For the evaluation of runtime, convolutions are performed on a sparse voxel grid filled with random numbers. The resolution of the voxel grid r^3 is varied between $r = 16$ and $r = 256$. To achieve the expected data density of a 2D surface in a 3D voxel grid, the data density ρ as well as the upper bound on the per-channel density ρ_{up} are set to $\rho = \rho_{up} = \frac{1}{r}$. To run dense convolution at resolution $r = 256$, the mini-batch size and channel depth had to be set to 1 (Protobuf limits each single tensor to 2 GB), while the number of output channels was set to 8. The density ρ_f of the filter weights is varied between 0.1 and 1. As baseline we use the convolutional layer of Tensorflow [1], which performs convolutions via the fast Fourier transform and batched general matrix-matrix multiplication from cuBlas, as front end to cuDNN [4]. We note that processing only a single input channel does not play to the strength of our sparse network. Moreover, Tensorflow is able to use the full capability of the GPU, while our implementation is limited to operating in global memory, due to the weak support for atomic floating point operations in shared memory. The effect of this limitation is particularly pronounced at small resolution and high density, whereas for high resolutions and low densities its influence fades. In particular, at $r = 256$ and $\rho_f = 0.1$, we are $14\times$ faster with strong density regularisation, so that the k-selection step is bypassed; and still $7\times$ faster including k-selection filtering. See Figure 4.

Table 1 shows the memory requirements for dense and sparse convolution layers at various resolutions r. Dense convolutions require only a single output tensor. The sparse implementation uses tensors for indices and data as well as a temporary buffer, which can be reused in all layers. For the experiment the data type is 32bit floating point, for the indices we consider both both 32bit and 64bit.[2] As expected, our sparse representation needs less memory at the sparsity levels of realistic 3D point cloud data. In particular, our sparse version makes it possible to work with large resolutions up to $r = 512^3$, which is impossible with the dense version on existing hardware.

Contribution of small feature responses

In the context of sparsity the question arises, whether zero-valued features contribute valuable information. Two recent works tried to answer this question. On the one hand, Graham *et al.* [11] found that they reach the same accuracies as dense networks for their application, while completely neglecting zero-valued features. On the other hand, Uhrig [36] concluded that for certain tasks zero-valued features may be beneficial. For our network it is possible to assess the importance of small feature responses (not limited to exact zero-values) by training neural networks with varying upper bounds. For this experiment, CNNs are trained on MNIST for 10 epochs without regularisation, using the *adagrad* optimiser and a learning rate of 0.01.

The pixels in MNIST were set to zero when their value $v \in [0, 255]$ was below a threshold of $v < 50$, to obtain a sparse dataset with average density $\overline{\rho_{in}} = 0.23$,

[2] 32bit indices can only be used for resolutions $r \leq 128^3$ due to buffer overflows.

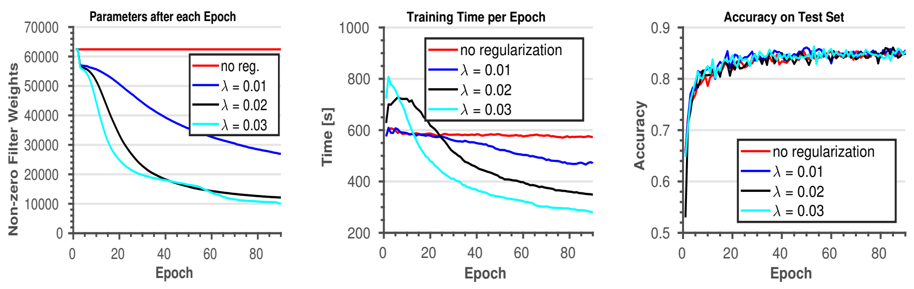

Fig. 5: Influence of adaptive density regularisation and pruning on *(left)* the number of non-zero filter weights, *(middle)* the runtime per training epoch, and *(right)* the accuracy on the Modelnet40 test set. Strong regularisation and pruning save a lot of memory and time without noticeable impact on accuracy.

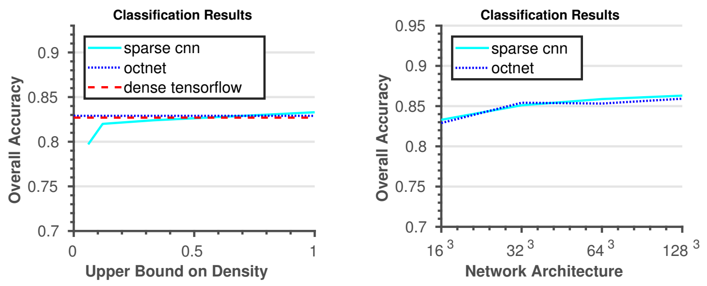

Fig. 6: Performance of sparse network on Modelnet40, compared to the equivalent dense network and Octnet. *(left)* accuracy for different upper density bounds; *(right)* accuracy for different input resolutions.

while the upper bound ρ_{up} ranges from $\rho_{up} = 0.035$ to $\rho_{up} = 0.095$. Note that even though letters can be interpreted as 1D lines in 2D images, the MNIST data has a low resolution of only 28×28 pixels. Hence, the data is still not extremely sparse. Lower upper bounds guarantee a small memory footprint, and also yield slightly faster runtime per epoch. The price to pay is slower convergence, because some gradients are lost during backpropagation; and a slight performance penalty for very strict bounds ($< 1.5\%$ for the strictest setting $\rho_{up} = 0.035$).

Regularisation and pruning

With our sparsity-inducing pruning and regularisation, we expect faster runtime. In order to verify this behaviour, neural networks are trained on Modelnet40 with varying regularisation scales $\lambda \in \{0, 0.1, 0.2, 0.3\}$. The bias for density-based regularisation is computed with $b_1 = b_2 = o = 0.1$. Stronger regularisation decimates the number of (non-zero) filter weights faster, as shown in Figure 5. It can also be seen that the number of parameters converges when only important weights are left. The drop in non-zero weights also reduces runtime. After 90 epochs, a network regularised with $\lambda = 0.3$ is 51% faster than one trained without regularisation and pruning, even though only the first nine out of twelve convolution layers are set to be sparse. Strong regularisation initially causes an increase in runtime, by driving up the number of non-zero weights to use the available resources via the bias term b_2. The classification accuracy for all tested regularisation scales quickly converges to practically identical values, as shown in Figure 5. We point out that pruning finds the most suitable sparsity pattern *for a given training set*. When using a pruned model for transfer learning, it may be safer to re-initialize the removed filter weights of the sparse representation with zeros before fine-tuning.

Classification performance on Modelnet40

Finally, we compare our upper-bounded neural network and modified back-propagation against a conventional net. To that end we run Octnet3, a dense network without octree structure, and a sparse version of the same network on Modelnet40, see Figure 6.

First, the input resolution is set to $r = 16^3$, while the upper bound on the density is varied between $\rho_{in} \in \{0.06, 0.12, 0.33, 1.0\}$. Both, the conventional dense network and Octnet converge to a similar overall accuracy of ≈ 0.83. For a trivial upper bound $\rho_{in} = 1.0$ the overall accuracy of our sparse network is also practically the same. Very low upper bounds up to $\rho_{in} = 0.12$ yield slightly worse results on the 16^3 inputs, for the lowest bound $\rho_{in} = 0.06$ the drop in performance reaches ≈ 3 percent points. Second, the resolution of the input is gradually increased: $r \in \{16^3,\ 32^3,\ 64^3,\ 128^3\}$. Both the sparse network and Octnet yield similar results, for all resolutions. Octnet performs slightly better on $r = 32^3$, while our bounded, sparse network has a small advantage at all other resolutions. The two experiments suggest that reasonable upper bounds and our sparse backpropagation do not reduce classification accuracy.

5 Conclusion

We have proposed novel neural network mechanisms which exploit and encourage sparseness in both feature maps and model parameters. At practically useful resolutions, our novel sparse layers and back-propagation rule significantly reduce *(i)* memory footprint and *(ii)* runtime of convolutional layers for sufficiently sparse data. Moreover, our approach guarantees upper bounds on the memory requirements and runtime of the network. For classification tasks the performance of our sparse network is comparable to its dense counterpart as well as Octnet. In future work, it will be interesting to employ sparsity also for other tasks. Our implementation is fully compatible with Tensorflow and has been released as open-source code. We hope, that hardware support for sparse convolutions will improve further on future consumer GPUs, as demonstrated by [26]; thus further boosting the performance of sparse, high-dimensional CNNs.

References

1. Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., et al.: Tensorflow: A system for large-scale machine learning. In: USENIX OSDI (2016)
2. Alabi, T., Blanchard, J.D., Gordon, B., Steinbach, R.: Fast k-selection algorithms for graphics processing units. Journal of Experimental Algorithmics **17** (2012)
3. Brock, A., Lim, T., Ritchie, J., Weston, N.: Generative and discriminative voxel modeling with convolutional neural networks. arXiv preprint 1608.04236 (2017)
4. Chetlur, S., Woolley, C., Vandermersch, P., Cohen, J., Tran, J., Catanzaro, B., Shelhamer, E.: CUDNN: Efficient primitives for deep learning. arXiv preprint arXiv:1410.0759 (2014)
5. Denil, M., Shakibi, B., Dinh, L., de Freitas, N., et al.: Predicting parameters in deep learning. In: NIPS (2013)
6. Denton, E.L., Zaremba, W., Bruna, J., LeCun, Y., Fergus, R.: Exploiting linear structure within convolutional networks for efficient evaluation. In: NIPS (2014)
7. Duchi, J., Hazan, E., Singer, Y.: Adaptive subgradient methods for online learning and stochastic optimization. Journal of Machine Learning Research **12** (2011)
8. Engelcke, M., Rao, D., Wang, D.Z., Tong, C.H., Posner, I.: Vote3Deep: Fast object detection in 3d point clouds using efficient convolutional neural networks. arXiv preprint 1609.06666 (2017)
9. Graham, B.: Spatially-sparse convolutional neural networks. arXiv preprint 1409.6070 (2014)
10. Graham, B., Engelcke, M., van der Maaten, L.: 3d semantic segmentation with submanifold sparse convolutional networks. arXiv preprint 1711.10275 (2017)
11. Graham, B., van der Maaten, L.: Submanifold sparse convolutional networks. arXiv preprint 1706.01307 (2017)
12. Han, S., Pool, J., Tran, J., Dally, W.: Learning both weights and connections for efficient neural network. In: NIPS (2015)
13. Häne, C., Tulsiani, S., Malik, J.: Hierarchical surface prediction for 3d object reconstruction. arXiv preprint 1704.00710 (2017)
14. Huang, J., You, S.: Point cloud labeling using 3d convolutional neural network. In: ICPR (2016)
15. Jaderberg, M., Vedaldi, A., Zisserman, A.: Speeding up convolutional neural networks with low rank expansions. arXiv preprint 1405.3866 (2014)
16. Jampani, V., Kiefel, M., Gehler, P.V.: Learning sparse high dimensional filters: image filtering, dense CRFs and bilateral neural networks. In: CVPR (2016)
17. Karpathy, A., Toderici, G., Shetty, S., Leung, T., Sukthankar, R., Fei-Fei, L.: Large-scale video classification with convolutional neural networks. In: CVPR (2014)
18. Krizhevsky, A., Sutskever, I., Hinton, G.E.: Imagenet classification with deep convolutional neural networks. In: NIPS (2012)
19. Lai, K., Bo, L., Fox, D.: Unsupervised feature learning for 3d scene labeling. In: ICRA (2014)
20. LeCun, Y., Bottou, L., Bengio, Y., Haffner, P.: Gradient-based learning applied to document recognition. Proceedings of the IEEE **86**(11) (1998)
21. Li, Y., Pirk, S., Su, H., Qi, C.R., Guibas, L.J.: FPNN: Field probing neural networks for 3d data. In: NIPS (2016)
22. Liu, B., Wang, M., Foroosh, H., Tappen, M., Pensky, M.: Sparse convolutional neural networks. In: CVPR (2015)

23. Long, J., Shelhamer, E., Darrell, T.: Fully convolutional networks for semantic segmentation. In: CVPR (2015)
24. Maturana, D., Scherer, S.: Voxnet: A 3d convolutional neural network for real-time object recognition. In: IROS (2015)
25. Nissen, M.J., Bullemer, P.: Attentional requirements of learning: Evidence from performance measures. Cognitive Psychology **19**(1) (1987)
26. Parashar, A., Rhu, M., Mukkara, A., Puglielli, A., Venkatesan, R., Khailany, B., Emer, J., Keckler, S.W., Dally, W.J.: SCNN: An accelerator for compressed-sparse convolutional neural networks. In: Int'l Symp on Computer Architecture (2017)
27. Park, J., Li, S., Wen, W., Tang, P.T.P., Li, H., Chen, Y., Dubey, P.: Faster CNNs with direct sparse convolutions and guided pruning. In: ICLR (2017)
28. Prokhorov, D.: A Convolutional Learning System for Object Classification in 3-D Lidar Data. IEEE Transactions on Neural Networks **21**(5), 858–863 (2010)
29. Qi, C.R., Su, H., Mo, K., Guibas, L.J.: PointNet: Deep learning on point sets for 3d classification and segmentation. In: CVPR (2017)
30. Qi, C.R., Yi, L., Su, H., Guibas, L.J.: PointNet++: Deep hierarchical feature learning on point sets in a metric space. arXiv preprint 1706.02413 (2017)
31. Ren, S., He, K., Girshick, R., Sun, J.: Faster R-CNN: Towards real-time object detection with region proposal networks. In: NIPS (2015)
32. Riegler, G., Ulusoy, A.O., Geiger, A.: OctNet: Learning deep 3d representations at high resolutions. In: CVPR (2017)
33. Robertson, E.M.: The serial reaction time task: implicit motor skill learning? Journal of Neuroscience **27**(38), 10073–10075 (2007)
34. Song, S., Xiao, J.: Deep sliding shapes for amodal 3d object detection in rgb-d images. In: CVPR (2016)
35. Tatarchenko, M., Dosovitskiy, A., Brox, T.: Octree generating networks: Efficient convolutional architectures for high-resolution 3d outputs. arXiv preprint 1703.09438 (2017)
36. Uhrig, J., Schneider, N., Schneider, L., Franke, U., Brox, T., Geiger, A.: Sparsity invariant CNNs. arXiv preprint 1708.06500 (2017)
37. Wen, W., Wu, C., Wang, Y., Chen, Y., Li, H.: Learning structured sparsity in deep neural networks. In: NIPS (2016)
38. Wu, Z., Song, S., Khosla, A., Yu, F., Zhang, L., Tang, X., Xiao, J.: 3d shapenets: A deep representation for volumetric shapes. In: CVPR (2015)

Appendix: Network architectures

Table 2 shows the network architectures of our experiments. Depending on the type of data and the goal of the experiment, we used the following network specifications (both for the dense and the sparse version, where applicable):

- For MNIST, we run OctNet3-24^2 with 10 output classes. ρ_{21} is set as specified in the paper text and $\rho_{22} = 2 \cdot \rho_{21}$.
- For Modelnet40 we have $c = 40$ different output classes and employ the following variants:
  1. For the regularisation experiment (Figure 5) we use OctNet3-64^3 with $\rho_{41} = 0.06$, $\rho_{42} = 0.14$, $\rho_{43} = 0.33$.
  2. Figure 6 *(left)* has been generated with OctNet3-16^3 with varying upper bounds $\rho_{11} = \{0.06, 0.12, 0.33, 1\}$, $\rho_{12} = \{0.12, 0.24, 0.33, 1\}$.
  3. For Figure 6 *(right)* the following networks were used: OctNet3-16^3 with $\rho_{11} = \rho_{12} = 1$; OctNet3-$32^3$ with $\rho_{31} = 0.14$, $\rho_{32} = 0.33$, $\rho_{33} = 0.66$; OctNet3-64^3 with $\rho_{41} = 0.06$, $\rho_{42} = 0.14$, $\rho_{43} = 0.33$; OctNet3-128^3 with $\rho_{51} = 0.02$, $\rho_{52} = 0.06$, $\rho_{53} = 0.14$.

OctNet3-16^3	OctNet3-24^2	OctNet3-32^3	OctNet3-64^3	OctNet3-128^3	OctNet3-256^3

conv(1, 8, ρ_{11})
conv(8, 8, ρ_{11})
conv(8, 8, ρ_{11})
maxPooling(2)
	conv(1, 8, ρ_{21})
	conv(8, 8, ρ_{21})
	conv(8, 8, ρ_{21})
	maxPooling(2)
		conv(1, 8, ρ_{31})
		conv(8, 8, ρ_{31})
		conv(8, 8, ρ_{31})
		maxPooling(2)
			conv(1, 8, ρ_{41})
			conv(8, 8, ρ_{41})
			conv(8, 8, ρ_{41})
			maxPooling(2)
				conv(1, 8, ρ_{51})
				conv(8, 8, ρ_{51})
				conv(8, 8, ρ_{51})
				maxPooling(2)
					conv(1, 8, ρ_{61})
					conv(8, 8, ρ_{61})
					conv(8, 8, ρ_{61})
					maxPooling(2)

conv(8, 16, ρ_{12})
conv(16, 16, ρ_{12})
conv(16, 16, ρ_{12})
sparseToDense()
	conv(8, 16, ρ_{22})
	conv(16, 16, ρ_{22})
	conv(16, 16, ρ_{22})
	sparseToDense()
		conv(8, 16, ρ_{32})
		conv(16, 16, ρ_{32})
		conv(16, 16, ρ_{32})
		maxPooling(2)
			conv(8, 16, ρ_{42})
			conv(16, 16, ρ_{42})
			conv(16, 16, ρ_{42})
			maxPooling(2)
				conv(8, 16, ρ_{52})
				conv(16, 16, ρ_{52})
				conv(16, 16, ρ_{52})
				maxPooling(2)
					conv(8, 16, ρ_{62})
					conv(16, 16, ρ_{62})
					conv(16, 16, ρ_{62})
					maxPooling(2)

		conv(16, 24, ρ_{33})
		conv(24, 24, ρ_{33})
		conv(24, 24, ρ_{33})
			conv(16, 24, ρ_{43})
			conv(24, 24, ρ_{43})
			conv(24, 24, ρ_{43})
			sparseToDense()
				conv(16, 24, ρ_{53})
				conv(24, 24, ρ_{53})
				conv(24, 24, ρ_{53})
				sparseToDense()
					conv(16, 24, ρ_{63})
					conv(24, 24, ρ_{63})
					conv(24, 24, ρ_{63})
					sparseToDense()

			maxPooling(2)
			conv(24, 32)
			conv(32, 32)
			conv(32, 32)
				maxPooling(2)
				conv(24, 32)
				conv(32, 32)
				conv(32, 32)
					maxPooling(2)
					conv(24, 32)
					conv(32, 32)
					conv(32, 32)

				maxPooling(2)
				conv(32, 40)
				conv(40, 40)
				conv(40, 40)
					maxPooling(2)
					conv(32, 40)
					conv(40, 40)
					conv(40, 40)

					maxPooling(2)
					conv(40, 48)
					conv(48, 48)
					conv(48, 48)

dropout(0.5)
fully-connected(1024)
fully-connected(c)					

Table 2: In our evaluation, we use different OctNet3 network architectures, similar to those also used by Riegler et al. [32]