PERFORATEDCNNS: ACCELERATION THROUGH ELIMINATION OF REDUNDANT CONVOLUTIONS

Michael Figurnov[1], **Dmitry Vetrov**[1,2], and **Pushmeet Kohli**[3]

[1]Skolkovo Institute of Science and Technology (Skoltech), Moscow, Russia
[2]National Research University Higher School of Economics, Moscow, Russia
[3]Microsoft Research, Cambridge, UK
michael@figurnov.ru, vetrovd@yandex.ru, pkohli@microsoft.com

ABSTRACT

We propose a novel approach to reduce the computational cost of evaluation of convolutional neural networks, a factor that has hindered their deployment in low-power devices such as mobile phones. Inspired by the loop perforation technique from source code optimization, we speed up the bottleneck convolutional layers by skipping their evaluation in some of the spatial positions. We propose and analyze several strategies of choosing these positions. Our method allows to reduce the evaluation time of modern convolutional neural networks by 50% with a small decrease in accuracy.

1 INTRODUCTION

The last few years have seen convolutional neural networks (CNNs) emerge as an indispensable tool for computer vision. This has been driven by their ability to achieve state-of-the-art performance for many challenging vision problems (Mnih et al., 2015; Schroff et al., 2015; Zheng et al., 2015). Modern CNNs have high computational cost of evaluation, with convolutional layers usually taking up over 80% of the time. For instance, VGG-16 network (Simonyan & Zisserman, 2015) for the problem of object recognition requires $1.5 \cdot 10^{10}$ floating point multiplications per image. These computational requirements hinder the deployment of such networks on systems without GPUs and in scenarios where power consumption is a major concern, such as mobile devices.

The problem of trading accuracy of computations for speed is well-known within the software engineering community. One of the most prominent methods for this problem is *loop perforation* (Misailovic et al., 2010; 2011; Sidiroglou-Douskos et al., 2011). In a nutshell, this technique isolates loops in the code that are not critical for the execution, and then reduces their computational cost by skipping some iterations. More recently, researchers have considered problem-dependent perforation strategies which involve identifying data-parallel patterns, such as "stencil", and applying pattern-specific approximations that exploit the structure of the problem (Samadi et al., 2014).

Inspired by the general principle of perforation, we propose to reduce the computational cost of CNN evaluation by exploiting the spatial redundancy of the network. Modern CNNs, such as AlexNet, exploit this redundancy through the use of strides in the convolutional layers. However, increasing convolutional strides changes the architecture of the network (intermediate representations size and number of weights in the first fully-connected layer), which might be undesirable. Instead of using strides, we argue for the use of interpolation (perforation) of responses in the convolutional layer. A key element of this approach is the choice of the perforation mask, which defines the output positions to evaluate exactly. We propose several approaches to select the perforation masks, and a method of choosing a combination of perforation masks for different layers. In order to restore the network accuracy, we perform fine-tuning of the perforated network. Our experiments show that this method can reduce the evaluation time of modern CNN architectures proposed in the literature by a factor of $2\times$ - $4\times$ with small decrease in accuracy.

2 RELATED WORK

Reducing the computational cost of CNN evaluation is an active area of research, with both highly optimized implementations and approximate methods being investigated.

Implementations that exploit the parallelism available in computational architectures like GPUs (cuda-convnet2 (Krizhevsky, 2014), CuDNN (Chetlur et al., 2014)) have allowed to significantly reduce the evaluation time of CNNs. Since CuDNN internally reduces the computation of convolutional layers to the matrix-by-matrix multiplication (without explicitly materializing the data matrix), our approach can potentially be incorporated into this library. In a similar vein, the use of FPFGAs (Ovtcharov et al., 2015) leads to better trade-offs between speed and power consumption. Several papers (Courbariaux et al., 2015; Gupta et al., 2015) showed that CNNs may be efficiently evaluated using low precision arithmetic, which is important for FPFGA implementations. Most approximate methods of decreasing the CNN computational cost exploit the redundancies of the convolutional kernel using low-rank tensor decompositions (Denton et al., 2014; Jaderberg et al., 2014; Lebedev et al., 2015; Zhang et al., 2015a;b). In most cases, a convolutional layer is replaced by several convolutional layers applied sequentially, which have a much lower total computational cost. Another direction of research is modifications of CNN architecture for speed gains (He & Sun, 2015; Jin et al., 2015; Romero et al., 2014). The resulting networks are usually CNNs, meaning that our approach could be combined with most of these methods.

For spatially sparse inputs, it is possible to exploit this sparsity to speed up evaluation and training (Graham, 2014b). While this approach is similar to ours in the spirit, we do not rely on spatially sparse inputs. Instead, we sparsely sample the outputs of convolutional layer, and interpolate the remaining values.

In a recent work, Lebedev & Lempitsky (2015) also decrease the CNN computational cost by reducing the size of the data matrix. The difference is that their approach reduces the kernel filter support, while our approach decreases the number of spatial positions in which the convolutions are evaluated. The two methods are complementary.

Several papers have demonstrated that it is possible to significantly compress the parameters of the fully-connected layers (where most CNN parameters reside) with a marginal error increase (Collins & Kohli, 2014; Yang et al., 2015; Novikov et al., 2015). Since our method does not directly modify the fully-connected layers, it is possible to combine these methods with our approach to obtain a fast and small CNN.

3 PERFORATEDCNNS

The section provides a detailed description of our approach. Before proceeding further we introduce the notation that will be used in the rest of the paper.

Notation. A convolutional layer takes as input a tensor U of size $X \times Y \times S$ and outputs a tensor V of size $X' \times Y' \times T$, $X' = X - d + 1$, $Y' = Y - d + 1$. The first two dimensions are spatial (height and width), and the third dimension is the number of channels (for example, for an RGB input image $S = 3$). The set of T convolution kernels K is given by a tensor of size $d \times d \times S \times T$. For simplicity of notation, we assume unit stride, no zero-padding and skip the biases. The convolutional layer output may be defined as follows:

$$V(x, y, t) = \sum_{i=1}^{d} \sum_{j=1}^{d} \sum_{s=1}^{S} K(i, j, s, t) U(x + i - 1, y + j - 1, s) \tag{1}$$

Additionally, we define the set of all spatial indices (positions) of the output $\Omega = \{1, \dots, X'\} \times \{1, \dots, Y'\}$. Perforation mask $I \subseteq \Omega$ is the set of indices in which the outputs are to be calculated exactly. Denote $N = |I|$ the number of positions to be calculated exactly, and $r = 1 - \frac{N}{|\Omega|}$ the *perforation rate*.

Reduction to matrix multiplication. In order to achieve high computational performance, many deep learning frameworks, including Caffe (Jia et al., 2014) and MatConvNet (Vedaldi & Lenc, 2014), reduce computation of convolutional layers to the heavily-optimized matrix-by-matrix multiplication routine of basic linear algebra packages. This process, sometimes referred to as *lowering*,

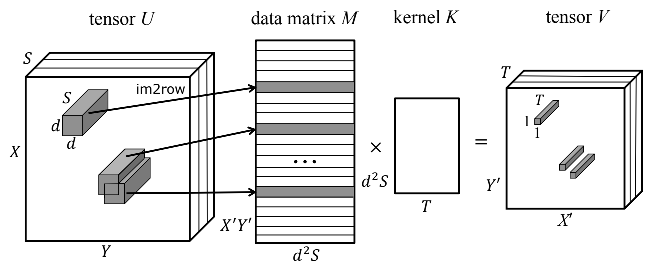


Figure 1: Reduction of convolutional layer evaluation to matrix multiplication. Our idea is to leave only a subset of rows (defined by a *perforation mask*) in the data matrix M and to interpolate the missing output values.

is illustrated in fig. 1. First, a *data matrix* M of size $X'Y' \times d^2S$ is constructed using `im2row` function. The rows of M are elements of patches of input tensor U of size $d \times d \times S$. Then, M is multiplied by the kernel tensor K reshaped into size $d^2S \times T$. The resulting matrix of size $X'Y' \times T$ is the output tensor V, up to a reshape. For a more detailed exposition, see (Vedaldi & Lenc, 2014).

The matrix multiplication is by far the most computationally expensive part of the convolutional layer evaluation. Suppose we removed all but N rows of the data matrix M. This reduces the number of operations required for the matrix multiplication by a factor of $\frac{X'Y'}{N}$, accelerating the evaluation. This speedup comes at a cost of losing information about the values of the outputs for some of the spatial positions *for all the channels*.

3.1 Perforated convolutional layer

Our first contribution is an accelerated version of the convolutional layer, called *perforated convolutional layer*. In a small fraction of spatial positions, the outputs of the proposed layer are equal to the outputs of a usual convolutional layer. The remaining values are interpolated using the nearest neighbor from the aforementioned set of positions. We evaluate other interpolation strategies in appendix A.

The perforated convolutional layer is a generalization of the standard convolutional layer. When the perforation mask is equal to all the output spatial positions, the perforated convolutional layer's output equals the conventional convolutional layer's output.

Formally, let $I \subseteq \Omega$ be the *perforation mask* of spatial output to be calculated exactly (the constraint that the masks are shared for all channels of the output is required for the reduction to matrix multiplication). The function $\ell(x, y) : \Omega \to I$ returns the index of the nearest neighbor in I according to Euclidean distance (with ties broken randomly):

$$\ell(x, y) = (\ell_1(x, y), \ell_2(x, y)) = \operatorname*{arg\,min}_{(x',y') \in I} \sqrt{(x - x')^2 + (y - y')^2}. \tag{2}$$

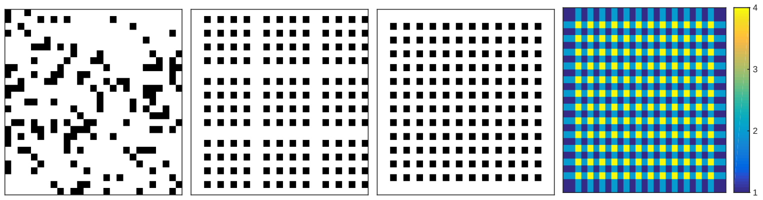
Note that the function $\ell(x, y)$ may be calculated in advance and cached.

The perforated convolutional layer output $\hat{V}$ is defined as follows:

$$\hat{V}(x, y, t) = V(\ell_1(x, y), \ell_2(x, y), t), \tag{3}$$

where $V(x, y, t)$ is the output of the usual convolutional layer, defined by (1). Since $\ell(x, y) = (x, y)$ for $(x, y) \in I$, the outputs in the spatial positions I are calculated exactly. The values in other positions are interpolated using the value of the nearest neighbor. In order to evaluate a perforated convolutional layer, we only need to calculate the values $V(x, y, t)$ for $(x, y) \in I$. This can be done efficiently by reduction to matrix multiplication. In this case, the data matrix M contains just $N = |I|$ rows, instead of original $X'Y' = |\Omega|$ rows. Perforation is not limited to this particular implementation of a convolutional layer, and can be combined with other implementations that support strided convolutions, such as the direct convolution approach used in cuda-convnet2 (Krizhevsky, 2014).

In our implementation, we only store the output values $V(x, y, t)$ for $(x, y) \in I$. The interpolation is performed implicitly by masking the reads of the following pooling or convolutional layer. This



(a) Uniform (b) Grid (c) Pooling structure (d) Weights $A(x, y)$

Figure 2: Perforation masks, AlexNet conv2, $r = 80.25\%$. Best viewed in color.

design choice has several advantages. Firstly, the memory size required to store the activations is reduced by a factor of $\frac{1}{1-r}$. Secondly, the following non-linearity layers and 1×1 convolutional layers are also sped up, since they are applied to a smaller number of elements. Finally, it makes interpolation computationally cheap by avoiding costly "scatter" operation. For example, when accelerating conv3 layer of AlexNet, the interpolation cost is transferred to conv4 layer. We observe no slowdown of conv4 layer when using GPU, and a 0-3% slowdown when using CPU.

To obtain derivatives of the perforated convolutional layer, we consider it as a composition of a convolutional layer and interpolation defined by (3). By the chain rule, it is sufficient to calculate the derivatives $\frac{\partial \hat{V}(x',y',t')}{\partial V(x,y,t)}$. If $\ell(x, y) = (x', y')$ and $t = t'$, this derivative is equal to one; otherwise, it is zero. In other words, during the backpropagation, the derivatives are summed over spatial regions which share the same interpolated values.

3.2 Perforation masks

We propose several ways of generating the perforation masks, or choosing $N = (1 - r)|\Omega|$ points from Ω. In order to visualize the perforation masks I, we use binary matrices, with black squares in positions indicated by the set I. We only consider the perforation masks that are independent of the input object and leave exploration of input-dependent perforation masks to the future work.

Uniform perforation mask is just N points chosen randomly without replacement from the set Ω. However, as can be seen from fig. 2a, for $N \ll |\Omega|$, the points tend to cluster. This is undesirable, because a more scattered set I would reduce the average distance to the set I.

The idea of **grid** perforation mask is to choose a set of scattered points. Define

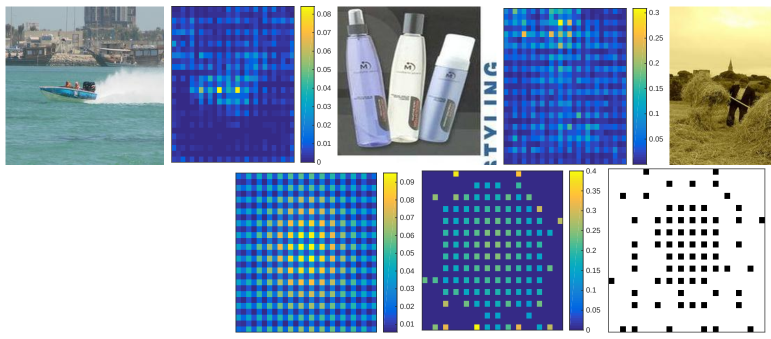
$$K_x = \left\lfloor \sqrt{N \frac{X'}{Y'}} \right\rfloor, K_y = \left\lfloor \sqrt{N \frac{Y'}{X'}} \right\rfloor. \tag{4}$$

A natural way to choose scattered points is to use a grid $I = \{a(1), \ldots, a(K_x)\} \times \{b(1), \ldots, b(K_y)\}$. If X' is divisible by K_x, then we can simply pick $a(i) = \frac{X'}{K_x} i$. For the general case, we adopt the *pseudorandom integer sequence generation* scheme from (Graham, 2014a). First, a random value $u \in (0, 1)$ from uniform distribution is generated. Then, the indices $a(i)$ are obtained as follows:

$$a(i) = \left\lceil \frac{X'}{K_x}(i - 1 + u) \right\rceil. \tag{5}$$

The indices $b(1), \ldots, b(K_y)$ are computed similarly. An example of grid mask is shown on fig. 2b.

Pooling structure mask exploits the structure of the overlaps of pooling operators. Denote by $A(x, y)$ the number of times an output of the convolutional layer is used in the pooling operators. An example of the values of $A(x, y)$ is shown on fig. 2d. The grid-like pattern is caused by the fact that the pooling has size 3×3 and is applied with stride 2 (these parameters are quite popular and are used in Network in Network and AlexNet). Intuitively, interpolating an output of the convolutional layer that is used in just one pooling operator is much better than interpolating an output that is used, say, four times. In other words, the higher the value of $A(x, y)$ is, the more important it is to include (x, y) in the set I. The pooling structure mask is obtained by picking top-N positions with the highest values of $A(x, y)$ and the ties broken randomly. The result is illustrated on fig. 2c.



(a) $B(x, y)$, original network (b) $B(x, y)$, perforated network (c) Impact mask, $r = 90\%$

Figure 3: **Top:** ImageNet images and corresponding values of impact $G(x, y; V)$ for AlexNet conv2 layer. **Bottom:** average impacts and impact perforation mask for AlexNet conv2. Best viewed in color.

Impact mask estimates the impact of perforation of each position on the CNN loss function, and then chooses the least important positions.

Denote by $L(V)$ the loss function of the CNN (such as class negative log-likelihood) as a function of the considered convolutional layer outputs V. Next, suppose V' is obtained from V by replacing one element (x_0, y_0, t_0) with a neutral value zero. We estimate the magnitude of change of the loss function (*impact* of the position) using first-order Taylor expansion:

$$|L(V') - L(V)| \approx \left| \sum_{x=1}^{X} \sum_{y=1}^{Y} \sum_{t=1}^{T} \frac{\partial L(V)}{\partial V(x,y,t)} (V'(x,y,t) - V(x,y,t)) \right| = \left| \frac{\partial L(V)}{\partial V(x_0,y_0,t_0)} V(x_0,y_0,t_0) \right|. \tag{6}$$

The value $\frac{\partial L(V)}{\partial V(x_0,y_0,t_0)}$ may be obtained using backpropagation. In case of a perforated convolutional layer, we calculate the derivatives with respect to the convolutional layer output V (not the interpolated output $\hat{V}$). This makes the impact of the previously perforated positions zero, and sums the impact of the non-perforated positions over all the outputs which share the value.

Since we are interested in the total impact of a spatial position $(x, y) \in \Omega$, we take a sum over all the channels and average this estimate of impacts over the training dataset:

$$G(x, y; V) = \sum_{t=1}^{T} \left| \frac{\partial L(V)}{\partial V(x,y,t)} V(x,y,t) \right| \tag{7}$$

$$B(x, y) = \mathbb{E}_{V \sim \text{training set}} \, G(x, y; V) \tag{8}$$

Finally, the impact mask is formed by taking the top-N positions with the highest values of $B(x, y)$. Examples of the values of $G(x, y; V)$, $B(x, y)$ and impact mask are shown on fig. 3. Note that the regions of high value of $G(x, y; V)$ usually contain the most salient features of the image. The averaged weights $B(x, y)$ tend to be higher in the center, since ImageNet's images usually contain a centered object. Additionally, a grid-like structure similar to the one seen in pooling structure mask is evident. Impact perforation mask tends to focus on the center of an image, but also places some points on the borders.

Since perforation of any layer changes the impacts of all the layers, in the experiments we iterate between increasing the perforation rate of a layer and recalculation of impacts. We find that this improves results by co-adapting the perforation masks of different convolutional layers.

Network	Dataset	Error	CPU time	GPU time	Mem.	Mult.	# conv
NIN	CIFAR-10	top-1 10.4%	4.6 ms	0.8 ms	5.1 MB	$2.2 \cdot 10^8$	3
AlexNet	ImageNet	top-5 19.6%	16.7 ms	2.0 ms	6.6 MB	$0.5 \cdot 10^9$	5
VGG-16		top-5 10.1%	300 ms	29 ms	110 MB	$1.5 \cdot 10^{10}$	13

Table 1: Details of the CNNs used for the experimental evaluation. Timings, memory consumption and number of multiplications are normalized by the batch size. Memory consumption is the memory required to store activations (intermediate results) of the network during the forward pass

3.3 Choosing the perforation configurations

For whole network acceleration, it is important to find a combination of per-layer perforation rates that would achieve high speedup at low error increase. To do this, we employ a simple greedy strategy. We use a single perforation mask type and a fixed range of increasing perforation rates. Denote by t the evaluation time of the accelerated network and by e the objective (we use class negative log-likelihood for a subset of training images). Let t_0 and e_0 be the respective values for the original (non-accelerated) network. At each iteration, we try to increase the perforation rate for each layer, and choose the layer for which this results in the minimal value of the cost function $\frac{e-e_0}{t_0-t}$. This cost function reflects that we would like to obtain high decrease of the running time with low increase of the objective.

4 Experiments

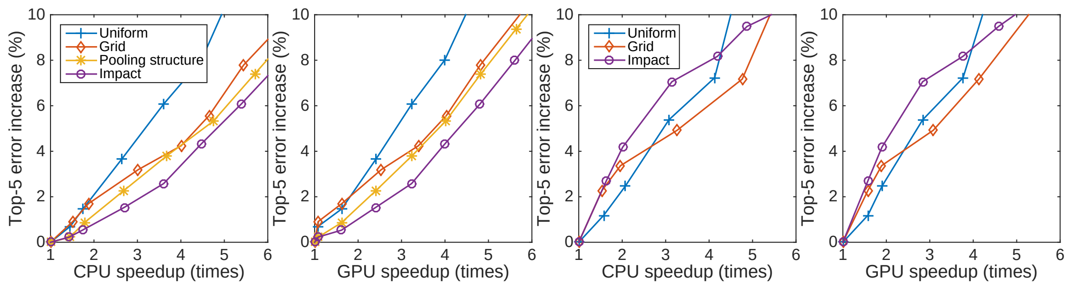
We use three convolutional neural networks of increasing size and computational complexity: Network in Network (Lin et al., 2014), AlexNet (Krizhevsky et al., 2012) and VGG-16 (Simonyan & Zisserman, 2015). See table 1 for more details. In all networks, we attempt to perforate all the convolutional layers, except for 1×1 convolutional layers of NIN. We perform timings on a computer with a quad-core Intel Core i5-4460 CPU, 16 GB RAM and a single NVidia Geforce GTX 980 GPU. The batch size used for timings is 128 for NIN, 256 for AlexNet and 16 for VGG-16. The networks are obtained from Caffe Model Zoo. For AlexNet, the Caffe reimplementation is used which is slightly different from the original architecture (pooling and normalization layers are swapped). We use a fork of MatConvNet framework[1] for all experiments, except for fine-tuning of AlexNet and VGG-16, for which we use a fork of Caffe [2].

We begin our experiments by comparing the proposed perforation masks in a common scenario used in the papers: acceleration of a single AlexNet layer. Then, we compare whole-network acceleration with the best-performing masks to baselines such as decrease of input images size and increase of strides. This requires retraining of the network, so we use a smaller NIN network. Finally, we show that perforation scales to large network by presenting the whole-network acceleration results for AlexNet and VGG-16.

4.1 Single layer results

We explore the speedup-error trade-off of the proposed perforation masks on the two bottleneck convolutional layers of AlexNet, conv2 and conv3, which consume 23.6% and 24.3% of the evaluation time on GPU, respectively. Conv2 is followed by a max-pooling, while conv3 is followed by another convolutional layer, meaning that the pooling structure perforation mask is only applicable to conv2. Figure 4 presents the results of this experiment. We see that impact perforation mask works best for conv2 layer, while grid mask performs very well for conv3. The standard deviation of results is small for all the perforation masks, with the exception of uniform mask for high speedups (where it is outperformed by the grid mask). The results are similar for both CPU and GPU, showing applicability of our method for both platforms. Note that if we consider the best perforation mask for each speedup value, then we see that conv2 layer is easier to accelerate than conv3 layer. We observe this pattern in other experiments: layers immediately followed by a max-pooling are easier

[1]`https://github.com/mfigurnov/perforated-cnn-matconvnet`
[2]`https://github.com/mfigurnov/perforated-cnn-caffe`



(a) conv2, CPU (b) conv2, GPU (c) conv3, CPU (d) conv3, GPU

Figure 4: Acceleration of a single layer of AlexNet for different mask types without fine-tuning. Values are averaged over 5 runs.

Method	CPU time ↓	Error ↑ (%)
Impact mask, $r = \frac{3}{4}$, 3×3 filters	**9.1×**	+1
Impact mask, $r = \frac{5}{6}$	5.3×	+1.4
Impact mask, $r = \frac{4}{5}$	4.2×	+0.9
(Lebedev & Lempitsky, 2015)	**10×**	top-1 +1.1
(Lebedev & Lempitsky, 2015)	5×	top-1 +0.4
(Jaderberg et al., 2014)	6.6×	+1
(Lebedev et al., 2015)	4.5×	+1
(Denton et al., 2014)	2.7×	+1

Table 2: Acceleration of AlexNet's conv2. **Top:** our results after fine-tuning, **bottom:** previously published results. Result of Jaderberg et al. (2014) provided by Lebedev et al. (2015). The experiment with reduced spatial size of the kernel (3×3, instead of 5×5) suggests that perforation is complimentary to the "brain damage" method of Lebedev & Lempitsky (2015) which also reduces the spatial support of the kernel.

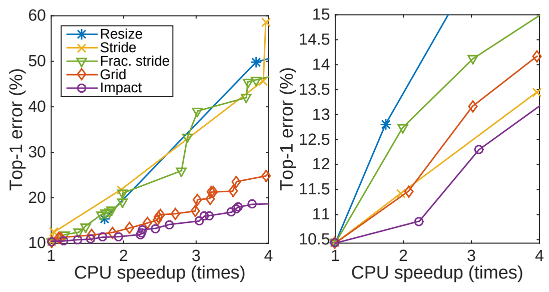
to accelerate than the layers followed by a convolutional layer. Additional results for NIN network are presented in appendix B.

We compare our results to the previously published results on acceleration of AlexNet's conv2 in table 2. Measured by a commonly used metric, CPU acceleration achieved for the 1% top-5 error increase, our method performs slightly worse than the state-of-the-art. However, perforation is complementary to those methods, since they are based on modifications of the filter kernel, while perforation exploits spatial redundancy of the output values. Motivated by the results of Lebedev & Lempitsky (2015) that the spatial support of conv2 convolutional kernel may be reduced with a small error increase, we reduce the kernel's spatial size from 5×5 to 3×3 and apply impact perforation mask. This leads to state-of-the-art $9.1\times$ acceleration for 1% top-5 error increase. Using the more sophisticated method of Lebedev & Lempitsky (2015) to reduce the spatial support may lead to further improvements.

4.2 Baselines

We compare PerforatedCNNs with the baseline methods of decreasing the computational cost of CNNs by exploiting the spatial redundancy. Unlike perforation, these methods decrease the size of the activations (intermediate outputs) of the CNN. For a network with fully-connected (FC) layers, this would change the number of CNN parameters in the first FC layer, effectively modifying the architecture. To avoid this, we use NIN network for CIFAR-10 dataset, which replaces FC layers with global average pooling (mean-pooling over all spatial positions in the last layer). Additionally, this network is fairly small, which allows to retrain quickly.

We consider the following baseline methods. **Resize**. The input image is downscaled with the aspect ratio preserved. **Stride**. The strides of the convolutional layers are increased, meaning that



(a) Original network (b) After retraining

Figure 5: Comparison of whole network perforation (grid and impact mask) with baseline strategies (resizing the input images, increasing the strides of convolutional layers) for acceleration of CIFAR-10 NIN network.

the convolutions are evaluated on a regular grid. The missing values are simply omitted from the output, decreasing the output size. For example, for stride value 2, every second output value in both horizontal and vertical direction is omitted, making the output four times smaller. **Fractional stride**. Stride of the convolutional layer can only take integer values, which might not be flexible enough. Motivated by fractional max-pooling Graham (2014a) and grid perforation mask, we introduce a modification of strides which evaluates convolutions on a non-regular grid (with varying step size), providing a more fine-grained control over the output size and speedup. We use grid perforation mask generation scheme to choose the output positions to evaluate.

We compare these strategies to perforation of all the layers with the two types of masks which performed best in the previous section: **grid** and **impact**. Note that "grid" is in fact equivalent to fractional strides, but with missing values being interpolated.

All the methods, except resize, require a parameter value per convolutional layer, leading to a large number of possible configurations. We use the original network to explore this space of configurations. For impact, we tune the perforation rates using the greedy algorithm. For stride, we evaluate all possible combinations of parameters. For grid and fractional strides, for each layer we consider the set of rates $\frac{1}{3}, \frac{1}{2}, \ldots, \frac{8}{9}, \frac{9}{10}$ (for fractional strides this is the fraction of convolutions calculated), and evaluate all possible combinations of such rates. Then, for each method, we build Pareto-optimal front of parameters which produced smallest error increase for a given CPU speedup. Finally, we train the network weights "from scratch" (starting from random initialization) for the Pareto-optimal configurations with accelerations close to $2\times, 3\times, 4\times$. For fractional strides, we had to use fine-tuning, since it performed significantly better than training from scratch.

The results are displayed on fig. 5. Impact perforation is the best strategy both for the original network and after training the network from scratch, with grid perforation being slightly worse. Interestingly, in the case of training the network from scratch, increasing the strides is a very competitive strategy. Convolutional strides are used in many CNNs, such as AlexNet, to decrease the computational cost of training and evaluation. Our results show that if changing the intermediate representations size and training the network from scratch is an option, then it is indeed a good strategy. Fractional strides perform poorly compared to strides, even though fractional strides is a more general strategy. A possible reason is that fractional strides "downsample" the outputs of a convolutional layer non-uniformly, making such outputs hard to process by the next convolutional layer.

4.3 Whole network results

We evaluate the effect of perforation of all the convolutional layers of three CNN models. To tune the perforation rates, we employ the greedy method described in section 3.3. We use twenty perforation rates: $\frac{1}{3}, \frac{1}{2}, \frac{2}{3}, \ldots, \frac{18}{19}, \frac{19}{20}$. For NIN and AlexNet we use the impact perforation mask, and for VGG-16 the grid perforation mask. We find that using the grid mask for VGG-16 greatly simplifies fine-tuning, and that using more than one type of perforation masks does not improve the results. Obtaining the perforation rates configuration takes about one day for the largest network we considered,

Network	Device	Speedup	Mult. ↓	Mem. ↓	Error ↑ (%)	Tuned error ↑ (%)
NIN	CPU	$2.2\times$	$2.5\times$	$2.0\times$	+1.5	+0.4
		$3.1\times$	$4.4\times$	$3.5\times$	+5.5	+1.9
		$4.2\times$	$6.6\times$	$4.4\times$	+8.3	+2.9
	GPU	$2.1\times$	$3.6\times$	$3.3\times$	+4.5	+1.6
		$3.0\times$	$10.1\times$	$5.7\times$	+18.2	+5.6
		$3.5\times$	$19.1\times$	$9.2\times$	+37.4	+12.4
AlexNet	CPU	$2.0\times$	$2.1\times$	$1.8\times$	+10.7	+2.3
		$3.0\times$	$3.5\times$	$2.6\times$	+28.0	+6.1
		$3.6\times$	$4.4\times$	$2.9\times$	+60.7	+9.9
	GPU	$2.0\times$	$2.0\times$	$1.7\times$	+8.5	+2.0
		$3.0\times$	$2.6\times$	$2.0\times$	+16.4	+3.2
		$4.1\times$	$3.4\times$	$2.4\times$	+28.1	+6.2
VGG-16	CPU	$2.0\times$	$1.8\times$	$1.5\times$	+15.6	+1.1
		$3.0\times$	$2.9\times$	$1.8\times$	+54.3	+3.7
		$4.0\times$	$4.0\times$	$2.5\times$	+71.6	+5.5
	GPU	$2.0\times$	$1.9\times$	$1.7\times$	+23.1	+2.5
		$3.0\times$	$2.8\times$	$2.4\times$	+65.0	+6.8
		$4.0\times$	$4.7\times$	$3.4\times$	+76.5	+7.3

Table 3: Full network acceleration results. Arrows indicate increase or decrease of the metric. Mult. is reduction of the number of multiplications in convolutional layers (theoretical speedup). Mem. is reduction of memory required to store the network activations. Tuned error is the error after training from scratch (NIN) or fine-tuning (AlexNet, VGG16) of the accelerated network's weights.

VGG-16. In order to decrease the error of the accelerated network, we tune the network's weights. We do not observe any problems with backpropagation, such as exploding/vanishing gradients. The results are presented in table 3. Perforation damages the network performance significantly, but network weights tuning restores most of the accuracy. All the considered networks may be accelerated by a factor of two on both CPU and GPU, with under 2.6% increase of error. Theoretical speedups (reduction of the number of multiplications) are usually close to the empirical ones. Additionally, the memory required to store network activations is significantly reduced by storing only the non-perforated output values.

The only paper we are aware of that accelerates whole VGG-16 model is (Zhang et al., 2015a), achieving a $2.9\times$ GPU speedup with $+0.3\%$ increase of error, which is superior to our results. However, their method deepens the network, making network training harder and increasing the memory required to store activations. Additionally, this result is achieved by a combination of a matrix decomposition method exploiting output channels redundancy and a method of Jaderberg et al. (2014) based on convolutional kernel spatial redundancy. Using just the second component, they report $+9.7\%$ increase of error for $2.9\times$ GPU acceleration, which is inferior to our results. As a direction of future research, it would be interesting to combine the channel redundancy elimination method with perforation.

5 Conclusion

We have presented PerforatedCNNs which exploit redundancy of intermediate representations of modern CNNs to reduce the evaluation time and memory consumption. Perforation requires only a minor modification of the convolution layer, and obtain speedups close to theoretical ones on both CPU and GPU. Compared to increasing the strides of convolutions, PerforatedCNNs achieve lower error, are more flexible and do not change the architecture of a CNN (number of parameters in the fully-connected layers and size of the intermediate representations). Retaining the architecture allows to easily plug in PerforatedCNNs into the existing computer vision pipelines and only perform fine-tuning of the network, instead of complete retraining.

An interesting direction of future work is using *several* sets of perforation masks with different running time for a single network (with the same parameters). This way, the same network can be run with varying budget of time without reloading of the parameters. This might be useful, for example, to handle peak loads in the datacenters. Such extension is not applicable to most of the existing approaches to acceleration of CNNs, since they typically modify the network parameters.

In this work, we have proposed and analyzed several ways to select input-independent perforation masks. Our experiments suggest that this choice is important to reduce the error introduced by perforation. In future, we plan to explore the connection between PerforatedCNNs and visual attention by considering perforation masks that are conditioned on the network's input, and can achieve higher speedups by only processing the salient parts of the input. Unlike recent works on visual attention (Mnih et al., 2014; Ba et al., 2015; Jaderberg et al., 2015) which consider rectangular crops of an image, PerforatedCNNs can process non-rectangular and even disjoint salient parts of the image by choosing appropriate perforation masks in the convolutional layers.

ACKNOWLEDGMENTS

We would like to thank Alexander Kirillov and Dmitry Kropotov for helpful discussions, and Yandex for providing computational resources for this project. This work was supported by RFBR project No. 15-31-20596 (mol-a-ved) and by Microsoft: Moscow State University Joint Research Center (RPD 1053945).

REFERENCES

Ba, Jimmy, Salakhutdinov, Ruslan R, Grosse, Roger B, and Frey, Brendan J. Learning wake-sleep recurrent attention models. In *Advances in Neural Information Processing Systems*, pp. 2575–2583, 2015.

Chen, Tianqi. Matrix shadow library. `https://github.com/dmlc/mshadow`, 2015.

Chetlur, Sharan, Woolley, Cliff, Vandermersch, Philippe, Cohen, Jonathan, Tran, John, Catanzaro, Bryan, and Shelhamer, Evan. cuDNN: Efficient primitives for deep learning. *arXiv preprint arXiv:1410.0759*, 2014.

Collins, Maxwell D. and Kohli, Pushmeet. Memory bounded deep convolutional networks. *arXiv preprint arXiv:1412.1442*, 2014.

Courbariaux, Matthieu, Bengio, Yoshua, and David, Jean-Pierre. Low precision arithmetic for deep learning. *ICLR*, 2015.

Denton, Emily L, Zaremba, Wojciech, Bruna, Joan, LeCun, Yann, and Fergus, Rob. Exploiting linear structure within convolutional networks for efficient evaluation. *NIPS 27*, pp. 1269–1277, 2014.

Graham, Benjamin. Fractional max-pooling. *arXiv preprint arXiv:1412.6071*, 2014a.

Graham, Benjamin. Spatially-sparse convolutional neural networks. *arXiv preprint arXiv:1409.6070*, 2014b.

Gupta, Suyog, Agrawal, Ankur, Gopalakrishnan, Kailash, and Narayanan, Pritish. Deep learning with limited numerical precision. *ICML*, 2015.

He, Kaiming and Sun, Jian. Convolutional neural networks at constrained time cost. *CVPR*, 2015.

Jaderberg, Max, Vedaldi, Andrea, and Zisserman, Andrew. Speeding up convolutional neural networks with low rank expansions. *BMVC*, 2014.

Jaderberg, Max, Simonyan, Karen, Zisserman, Andrew, et al. Spatial transformer networks. In *Advances in Neural Information Processing Systems*, pp. 2008–2016, 2015.

Jia, Yangqing, Shelhamer, Evan, Donahue, Jeff, Karayev, Sergey, Long, Jonathan, Girshick, Ross, Guadarrama, Sergio, and Darrell, Trevor. Caffe: Convolutional architecture for fast feature embedding. In *Proceedings of the ACM International Conference on Multimedia*, pp. 675–678. ACM, 2014.

Jin, Jonghoon, Dundar, Aysegul, and Culurciello, Eugenio. Flattened convolutional neural networks for feedforward acceleration. *ICLR*, 2015.

Krizhevsky, Alex. cuda–convnet2. `https://github.com/akrizhevsky/cuda-convnet2/`, 2014.

Krizhevsky, Alex, Sutskever, Ilya, and Hinton, Geoffrey E. Imagenet classification with deep convolutional neural networks. *NIPS 25*, pp. 1097–1105, 2012.

Lebedev, Vadim and Lempitsky, Victor. Fast convnets using group-wise brain damage. *arXiv preprint arXiv:1506.02515*, 2015.

Lebedev, Vadim, Ganin, Yaroslav, Rakhuba, Maksim, Oseledets, Ivan, and Lempitsky, Victor. Speeding-up convolutional neural networks using fine-tuned CP-decomposition. *ICLR*, 2015.

Lin, Min, Chen, Qiang, and Yan, Shuicheng. Network in network. *ICLR*, 2014.

Misailovic, Sasa, Sidiroglou, Stelios, Hoffmann, Henry, and Rinard, Martin. Quality of service profiling. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*, pp. 25–34. ACM, 2010.

Misailovic, Sasa, Roy, Daniel M, and Rinard, Martin C. Probabilistically accurate program transformations. In *Static Analysis*, pp. 316–333. Springer, 2011.

Mnih, Volodymyr, Heess, Nicolas, Graves, Alex, et al. Recurrent models of visual attention. In *Advances in Neural Information Processing Systems*, pp. 2204–2212, 2014.

Mnih, Volodymyr, Kavukcuoglu, Koray, Silver, David, Rusu, Andrei A, Veness, Joel, Bellemare, Marc G, Graves, Alex, Riedmiller, Martin, Fidjeland, Andreas K, Ostrovski, Georg, et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 2015.

Novikov, Alexander, Podoprikhin, Dmitry, Osokin, Anton, and Vetrov, Dmitry. Tensorizing neural networks. *NIPS*, 2015.

Ovtcharov, Kalin, Ruwase, Olatunji, Kim, Joo-Young, Fowers, Jeremy, Strauss, Karin, and Chung, Eric S. Accelerating deep convolutional neural networks using specialized hardware. *Microsoft Research Whitepaper*, 2015.

Romero, Adriana, Ballas, Nicolas, Kahou, Samira Ebrahimi, Chassang, Antoine, Gatta, Carlo, and Bengio, Yoshua. Fitnets: Hints for thin deep nets. *arXiv preprint arXiv:1412.6550*, 2014.

Samadi, Mehrzad, Jamshidi, Davoud Anoushe, Lee, Janghaeng, and Mahlke, Scott. Paraprox: Pattern-based approximation for data parallel applications. In *Proceedings of the 19th international conference on Architectural support for programming languages and operating systems*, pp. 35–50. ACM, 2014.

Schroff, Florian, Kalenichenko, Dmitry, and Philbin, James. Facenet: A unified embedding for face recognition and clustering. *CVPR*, 2015.

Sidiroglou-Douskos, Stelios, Misailovic, Sasa, Hoffmann, Henry, and Rinard, Martin. Managing performance vs. accuracy trade-offs with loop perforation. *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, pp. 124–134, 2011.

Simonyan, Karen and Zisserman, Andrew. Very deep convolutional networks for large-scale image recognition. *ICLR*, 2015.

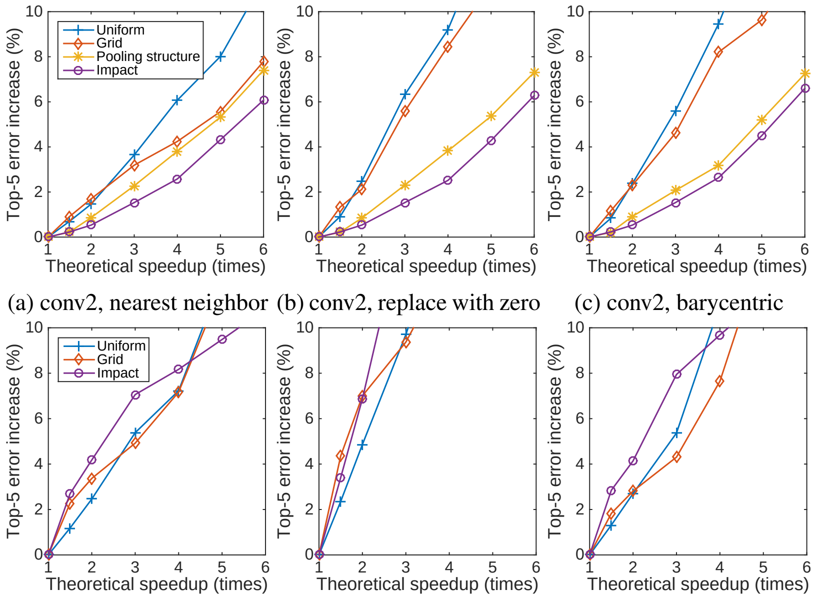
Vedaldi, Andrea and Lenc, Karel. MatConvNet – convolutional neural networks for MATLAB. *arXiv preprint arXiv:1412.4564*, 2014.

Yang, Zichao, Moczulski, Marcin, Denil, Misha, de Freitas, Nando, Smola, Alexander J., Song, Le, and Wang, Ziyu. Deep fried convnets. *ICCV*, 2015.

Zhang, Xiangyu, Zou, Jianhua, He, Kaiming, and Sun, Jian. Accelerating very deep convolutional networks for classification and detection. *arXiv preprint arXiv:1505.06798*, 2015a.

Zhang, Xiangyu, Zou, Jianhua, Ming, Xiang, He, Kaiming, and Sun, Jian. Efficient and accurate approximations of nonlinear convolutional networks. *CVPR*, 2015b.

Zheng, Shuai, Jayasumana, Sadeep, Romera-Paredes, Bernardino, Vineet, Vibhav, Su, Zhizhong, Du, Dalong, Huang, Chang, and Torr, Philip H. S. Conditional random fields as recurrent neural networks. *ICCV*, 2015.



(a) conv2, nearest neighbor (b) conv2, replace with zero (c) conv2, barycentric

(d) conv3, nearest neighbor (e) conv3, replace with zero (f) conv3, barycentric

Figure 6: Comparison of with different interpolation strategies for perforated pixels. AlexNet network

A Interpolation strategy

In the paper, perforated values are interpolated using the value of the nearest neighbor. We compare this strategy to two alternatives: replacing with a constant zero and barycentric interpolation. For the second option, we perform Delaunay triangulation of the non-perforated points set. If a perforated point is in the interior of a triangle, then it is interpolated by a weighted sum of the values in the three vertices, with barycentric coordinates used as weights. Exterior perforated points are simply assigned the value of the nearest neighbor.

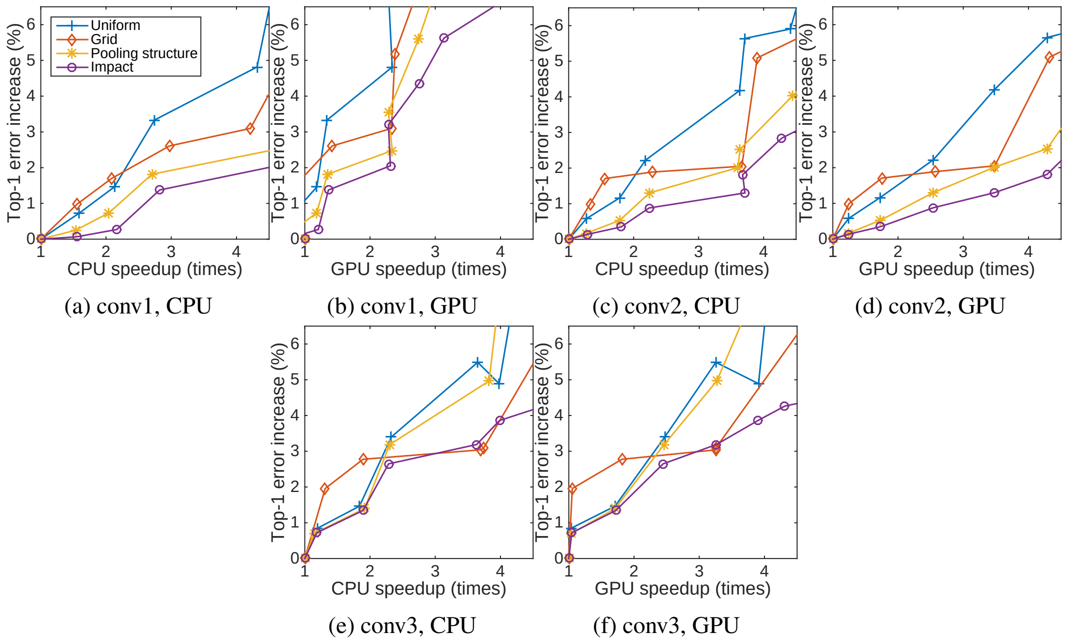
The results of comparison on AlexNet are presented on figure 6. We measure theoretical speedup (reduction of number of multiplications) to ignore the differences in implementations of the interpolation schemes. Replacing the missing values with zero is clearly not sufficient for successful acceleration of conv3 layer. Compared to nearest neighbor, barycentric interpolation slightly improves results for pooling structure mask in conv2 and grid interpolation mask in conv3 layer, but performs similarly or worse in other cases. Overall, nearest neighbor interpolation provides a good trade-off between complexity of the method (number of memory accesses per interpolated value) and the achieved error.

B Single layer results for NIN network

In section 4.1, we have considered single-layer acceleration of conv2 and conv3 layers of AlexNet. Here we present additional results for acceleration of the three non-1×1 convolutional layers of NIN network. Each convolutional layer is followed by two 1×1 convolutions (which we treat as a part of non-linearity) and a pooling operation. Therefore, pooling structure mask is applicable to all layers. The results are presented on figure 7. We observe a similar pattern to the one observed in AlexNet conv2 and conv3 layers: grid and impact perforation masks perform best.

C Empirical and theoretical speedups

As noted in (Denton et al., 2014), achieving empirical speedups that are close to the theoretical ones (reduction of the number of multiplications) is quite complicated. We find that our method generally



(a) conv1, CPU (b) conv1, GPU (c) conv2, CPU (d) conv2, GPU

(e) conv3, CPU (f) conv3, GPU

Figure 7: Acceleration of a single layer of CIFAR-10 NIN network for different mask types without fine-tuning. Values are averaged over 5 runs.

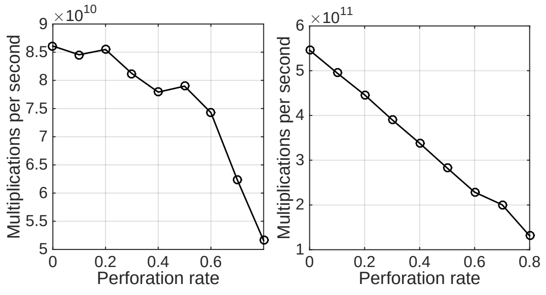
	NIN		AlexNet		VGG-16	
	CPU	GPU	CPU	GPU	CPU	GPU
conv1	$4.4\times$	$2.7\times$	$3.2\times$	$2.7\times$	$2.5\times$	$2.2\times$
conv2	$3.8\times$	$3.5\times$	$3.3\times$	$3.0\times$	$2.6\times$	$2.1\times$
conv3	$3.7\times$	$3.3\times$	$4.1\times$	$3.7\times$	$3.2\times$	$2.5\times$
conv4	-	-	$3.9\times$	$3.5\times$	$3.1\times$	$2.6\times$
conv5	-	-	$3.6\times$	$3.4\times$	$3.5\times$	$2.8\times$
conv6	-	-	-	-	$3.5\times$	$2.9\times$
conv7	-	-	-	-	$3.4\times$	$2.9\times$
conv8	-	-	-	-	$3.6\times$	$3.6\times$
conv9	-	-	-	-	$3.6\times$	$3.7\times$
conv10	-	-	-	-	$3.6\times$	$3.7\times$
conv11	-	-	-	-	$3.7\times$	$3.6\times$
conv12	-	-	-	-	$3.7\times$	$3.6\times$
conv13	-	-	-	-	$3.8\times$	$3.6\times$

Table 4: Per-layer empirical speedups for *uniform* perforation mask with $r = 0.75$. Theoretical speedup is $4\times$ in all cases. Results are averaged over 5 runs

allows to do that, see table 4. For example, for theoretical speedup $4\times$, AlexNet conv2 empirical acceleration is $3.8\times$ for CPU and $3.5\times$ for GPU. The results are below the theoretical speedup in almost all cases due to the additional memory accesses required. The perforation mask type does not seem to affect the speedup. The difference between the empirical speedups on CPU and GPU highlights that it is important to choose per-layer perforation rates for the target device.

D Implementation details

Convolutional layer is typically applied to each image of the mini-batch sequentially. Fig. 8 shows the number of multiplications per second achieved by a quad-core Intel CPU and NVidia Geforce



(a) CPU (b) GPU

Figure 8: Efficiency of matrix-by-matrix multiplication (measured in multiplications per second) of the data matrix M by the kernel matrix K, for different perforation rates. AlexNet, conv2 layer

GTX 980 GPU on the bottleneck operation of evaluation of the convolutional layer: matrix multiplication of the data matrix M by the kernel matrix K. We see that increasing the perforation rate reduces the efficiency of the operation, especially for GPU, which is as expected: GPUs work best for large inputs. Thus, for a fair comparison with the non-accelerated implementation, we stack $\lfloor \frac{1}{1-r} \rfloor$ images of the mini-batch, to match the size of the original data matrix. This requires a tensor transpose operation after the matrix multiplication, but we find that this operation is comparatively fast. The same idea is used in MShadow library (Chen, 2015). We also perform stacking of images for the baseline methods (resize, stride and fractional stride).

E PAPER REVISIONS

v1. Initial version.

v2. ICLR 2016 submission.

v3. Updated ICLR 2016 submission. Released the source code, expanded the conclusion, added new appendix B, and performed other small modifications according to the feedback from the reviewers.

v4. ICLR 2016 workshop style file.