Machine Learning-Aided Numerical Linear Algebra: Convolutional Neural Networks for the Efficient Preconditioner Generation

Markus Götz
Steinbuch Center for Computing
Karlsruhe Institute of Technology
76131 Karlsruhe, Germany
markus.goetz@kit.edu

Hartwig Anzt✉
Steinbuch Center for Computing
Karlsruhe Institute of Technology
76131 Karlsruhe, Germany
Innovative Computing Laboratory (ICL)
University of Tennessee
37996 Knoxville, USA
hartwig.anzt@kit.edu

***Abstract*—Generating sparsity patterns for effective block-Jacobi preconditioners is a challenging and computationally expensive task, in particular for problems with unknown origin. In this paper we design a convolutional neural network (CNN) to detect natural block structures in matrix sparsity patterns. For test matrices where a natural block structure is complemented with a random distribution of nonzeros (noise), we show that a trained network succeeds in identifying strongly connected components with more than 95% prediction accuracy, and the resulting block-Jacobi preconditioner effectively accelerating an iterative GMRES solver. Segmenting a matrix into diagonal tiles of size 128×128, for each tile the sparsity pattern of an effective block-Jacobi preconditioner can be generated in less than a millisecond when using a production-line GPU.**

***Index Terms*—Block-Jacobi Preconditioning, Convolutional Neural Networks, Multilabel Classification**

I. Introduction

In scientific computing, the process of iteratively solving a linear system of equations often strongly benefits from the use of a sophisticated preconditioner that carefully adapts to the linear system properties. A preconditioner is efficient if the convergence improvement rendered to the iterative solver compensates for the derivation of the preconditioner. In the context of high performance computing, the efficiency of the preconditioner specifically depends on the parallel scalability of both, the preconditioner generation prior to the start of the iterative solver and the preconditioner application at each step of the iteration process. Preconditioners based on Jacobi (diagonal scaling) and block-Jacobi (block-diagonal scaling) typically renders moderate improvements to the convergence of the iterative solver [18]. They are nevertheless attractive, as (block-)diagonal scaling introduces very small computational overhead to the solver iterations. Moreover, the application of a Jacobi-type preconditioner is inherently parallel and, therefore, highly appealing for massively parallel architectures. On the other hand, the preconditioner generation can be quite challenging. In a first step, it requires to map the characteristics of the system matrix to a block-structure reflecting clusters of strongly connected variables. In a second step, each block is inverted. With fast and scalable techniques for diagonal block inversion being available [4]–[6], the cost of the block-Jacobi preconditioner generation primarily boils down to the block pattern generation. In an era of growing hardware parallelism on a single node, it is important that the pattern generation also somewhat scales with the available resources—which is difficult to achieve for the conventional pattern generation tools based on graph analytics.

M. Götz is supported by the *Helmholtz Association Initiative and Networking Fund* under project grant ZT-I-0003. H. Anzt is supported by the *Helmholtz Association Initiative and Networking Fund* under project grant VH-NG-1241.

Machine learning (ML) [10] has recently gained a lot of attention in the scientific computing community due to the improved prediction quality and the efficient usage of high performance computing architectures. In particular deep learning using multi-layer neural networks (DNN) [22] can exploit the available compute power to improve the quality of the network architecture. The idea here is to use an extensive learning process to adjust weights connecting the network layers to efficiently classify input data [29].

In this paper, we attempt to bridge the gap between machine learning tools and classical linear algebra by employing DNN technology to quickly generate sparsity patterns for a block-Jacobi preconditioner. For this purpose, we design in Section III a convolutional network architecture that we train and evaluate in Section IV using a set of artificially creates test matrices. We start in Section II with providing some background about block-Jacobi preconditioning and convolutional neural networks. We also put this work into context with existing research efforts, and outline in Section V plans for future research.

II. Background and Related Work

A. Block-Jacobi preconditioning

In linear algebra, Block-Jacobi preconditioners are based on the idea of constructing a preconditioner matrix that propagates information locally, among variables that are adjacent in the system matrix [18]. In an iterative solution process, this is complemented by a top-level iterative method that propagates the information throughout the global system. More graphically, for a discretized partial differential equation, the block-Jacobi method corresponds (for an appropriate ordering of the unknowns) to the solution of independent sub-problems. The practical realization of this strategy encompasses the inversion of small diagonal blocks of the system matrix, and the subsequent composition of the inverted diagonal blocks into the preconditioner matrix, see Figure 1. Hence, for a coefficient matrix $A \in \mathbb{R}^{n \times n}$, the block-Jacobi method can be regarded as a straight-forward extension of its (scalar) Jacobi counterpart [4]. Instead of splitting the coefficient matrix as $A = L + D + U$ (with diagonal $D = (\{a_{ii}\})$, lower triangular $L = (\{a_{ij} \ : \ i > j\})$ and upper triangular $U = (\{a_{ij} \ : \ i < j\})$), the block-Jacobi variant gathers the diagonal blocks of A into

$$D = (D_1, D_2, \ldots, D_N), \quad D_i \in \mathbb{R}^{m_i \times m_i}, \quad i = 1, 2, \ldots, N,$$

with $n = \sum_{i=1}^{N} m_i$. The remaining elements of A are then partitioned into matrices L and U such that L contains the elements below the diagonal blocks while U comprises those above them [3].

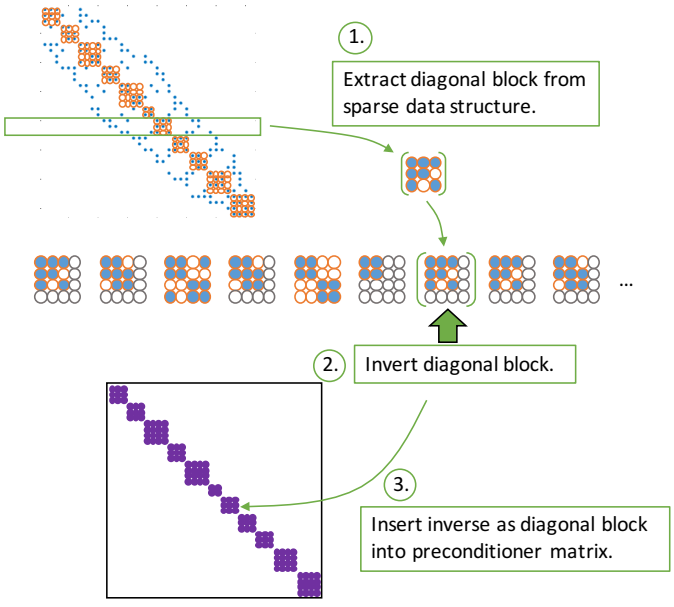

Fig. 1: Block-Jacobi preconditioning strategy [5].

Obviously, this approach works particularly well, if the diagonal blocks, to be inverted, accumulate multiple, strongly connected unknowns. Given an appropriate ordering of the components, these strongly connected unknowns appear as dense "natural blocks" in the, otherwise sparse, matrix. Fortunately, many linear systems exhibit some inherent block structure of this kind, for example because they arise from a finite element discretization of a partial differential equation (PDE) with multiple variables associated to each element [3]. The variables belonging to the same element usually share the same column sparsity pattern, and this set of variables is often referred to as a *supervariable*. If the underlying finite element discretization scheme is known, it is possible to analytically derive the size and arrangement of the blocks reflecting supervariables. Conversely, detecting natural blocks in a matrix coming from an unknown origin is often a tedious procedure. For finite element discretizations, *supervariable blocking* [12] is a strategy that exploits the property of variables of the same supervariable sharing the same column-nonzero-pattern. The blocking technique identifies adjacent columns with the same sparsity pattern, and accumulates those in a diagonal block. Depending on a pre-defined upper bound for the size of the Jacobi blocks, multiple supervariables adjacent in the coefficient matrix can be clustered within the same diagonal block [12]. However, in particular for matrices that do not come from a finite element discretization, the effectiveness of supervariable blocking in deriving a good preconditioner pattern is limited. Clustering techniques from graph analytics, and reordering of the unknowns combined with priority blocking can be used generate efficient preconditioners [12], but the high computational cost and the inferior scalability properties of these algorithms make their use questionable. In consequence, the preprocessing step of generating an efficient block pattern for a block-Jacobi preconditioner remains a challenging and computationally expensive task.

Once a suitable diagonal block pattern is identified, the block-Jacobi matrix $\hat{D} = D^{-1}$ derives from inverting the block-diagonal matrix $D = (D_1, D_2, \ldots, D_N)$. This is an inherently parallel process that can be realized efficiently using batched routines [3]–[6]. If $\hat{D}$ is well-defined, which is the case if all diagonal blocks $D_1, D_2, \ldots, D_N$ are non-singular, the block-Jacobi matrix can be used as preconditioner, transforming the original system $Ax = b$ into either the left-preconditioned system

$$D^{-1}Ax \quad = \quad c \quad (= \ D^{-1}b), \tag{1}$$

or the right-preconditioned system

$$AD^{-1}y \quad = \quad b, \quad \text{with } x = D^{-1}y. \tag{2}$$

Explicitly computing the block-inverse $\hat{D}_i = D_i^{-1}, i = 1, 2, \ldots, N$ allows to realize the application of a block-Jacobi preconditioner in (1) in terms of the multiplication with the block-Jacobi matrix [5]. If the block-Jacobi matrix is not available is explicit form, every preconditioner application requires the solution of the block-diagonal linear system (i.e., a linear system for each block D_i [4]). In general, pre-computing the block-inverse $\hat{D}$ is advisable if the preconditioner will be applied several times [5].

B. Deep convolutional neural networks

Convolutional neural networks (CNN [29]) are a subclass of traditional artificial neural networks (ANN [29]). Similarly to their classical counterparts, they compose of a number of weight- trainable neurons arranged in a layer-wise fashion. Typically, CNNs are tailored towards the pattern matching in two-dimensional matrices, with the goal of providing functionality analogous to the visual cortex enabling image perception in animals and humans. The main design characteristic of CNNs is the use of a convolutional layer (or multiple convolutional layers) consisting of very particular neurons. Each neuron is defined by a kernel (also referred to as *filter* [22]) that composes of a matrix of variable weights of size $l \times l$ (where $l < n$ for an input image of size $n \times n$). The kernel is convolved with the complete image, which means that a dot product of the kernel matrix and each $l \times l$ subregion of the image is computed. For an input image of size $n \times n$, this operation generates a matrix of the size $(n-l+1) \times (n-l+1)$.

The first convolution applied to the original image aims at matching basic patterns of the size $l \times l$. Subsequent convolutions then enable the matching of larger, or more complex patterns. The abstraction that is required to perceive particular entities in an image is achieved by stacking the patterns matched by layers of convolutions, often interchangeably with pooling — an operation collating the output of convolutions to enhance and accelerate the learning of visual generalizations.

A conventional ANN treats each pixel of the image as a distinct input feature. To preserve all information, this generally requires the network to contain at least one weight parameter per pixel of the image, per neuron. Through defining and reapplying the kernel, the convoulution re-uses the same neurons ("weight sharing"). This enables abstract pattern detection and reduces the number of weights required. A predictive CNN complements the obligatory input and output layer with at least one convolution and one dense layer. The term *deep neural network* is used for networks that compose of multiple layers and non-linear activation functions.

Deep neural network technology has recently made significant advances in a larger spectrum of applications [29]. This stems not only from enforced algorithm research efforts, but also from technological developments that allow for a quick learning process via the use of tensor cores in hardware architecture [19], [23], [25]. Also on the software side, machine learning has reached a state where ready-to-use open source software ecosystems like TensorFlow [1], PyTorch [27], or MXNet [9] (containing auto-gradient computation) are publicly available.

A primary application field of machine learning is image classification in areas like medicine [31], autonomous driving [15], and social network filtering [8]. As a result, there exist well-engineered deep neural network topologies for the pattern detection in visual images. Such topologies find use beyond image analysis, and successfully recognize more abstract patterns. In the field of scientific computing, for example, machine learning techniques have been employed to predict the distribution of the electric potential in a two- and three-dimensional electromagnetic simulation [32], and for solving ill-conditioned inverse problems [2]. For sparse numerical linear algebra, deep learning has facilitated the identification of a suitable storage format for sparse matrices [33]. Instead of selecting a suitable matrix storage format, we aim at using deep neural networks to generate sparsity patterns for efficient preconditioners.

III. Preconditioner Pattern Prediction Using Neural Networks

The identification of block patterns in two-dimensional square matrices can be considered as a series of binary classification problems. For each row i (respectively column) of the system matrix A, the task is to decide whether or not it should be considered the start, i.e. the upper (left) bounding edge, of a diagonal block. As elaborated in Section II, the diagonal blocks aim at reflecting clusters of strongly connected components, which implies that these blocks typically appear "denser" than the rest of the matrix. Hence, one approach is to view the matrix sparsity pattern as an image, and to detect the diagonal blocks with a combination of traditional image processing algorithms, such as edge detection filters, and thresholded scanline pixel accumulation [7]. As long as there is a reasonable difference between the "denser" diagonal blocks and the rest of the matrix, this strategy will achieve solid prediction results for arbitrary-sized blocks. The smaller the density differences are, the more "noise" has to be filtered by the neural network. While it is in principle possible to manually add and tailor additional filters to denoise the matrix, it is favorable to derive these automatically from data samples of the problem space.

Utilizing a convolutional neural network is particularly suitable for this task, as it is able to learn the necessary preprocessing and denoising features on its own. This training process is realized in a supervised fashion, i.e. the weights of the convolutional filters are inferred from examples where the block starts have previously been annotated ("labeled data" [29]). For a matrix A of size n, a label vector y with $y = \{0,1\}^n$ is used such that each row is assigned a separate and independent output neuron where 1 indicates the start of a block in the respective row. In the prediction process, the neural network then generates a prediction vector $\hat{y}$ where each $\hat{y}_i \in [0,1]$ indicates the probability of a block start in row i. The loss function for this multi-label classification problem is given as the sum of the binary cross-entropy across all the individual row predictions and available samples S:

$$\mathcal{L}(y,\hat{y}) = -\sum_{s=1}^{S}\sum_{i=1}^{n} y_{s,i} * log(\hat{y}_{s,i}) + (1-y_{s,i}) * log(1-\hat{y}_{s,i}).$$

The specific challenge of detecting diagonal blocks in a sparse matrix pattern allows for an optimization step that 1) reduces the amount of data that is processed; and 2) increases the prediction accuracy by mitigating the problem of false-correlations.

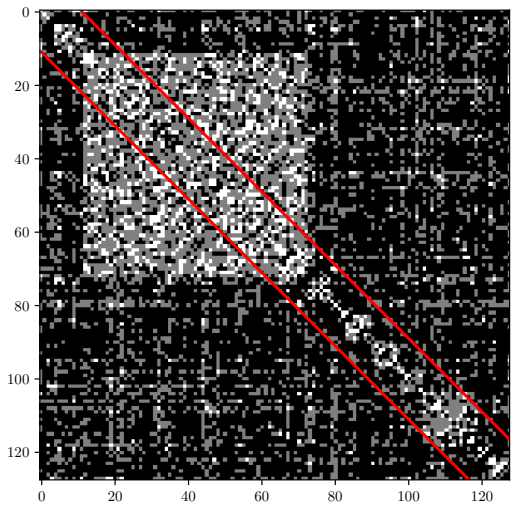

Fig. 2: Example of an artificially generated block-pattern matrix. The blocks are gap-free and corner-joint. The red lines mark the upper and lower boundary of the diagonal image.

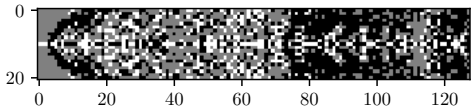

Fig. 3: Diagonal image extracted from the matrix presented in Fig. 2.

.

The idea is to crop the matrix parallel to the main diagonal, and to base the prediction process on the diagonal matrix band only, see Figure 2. This is motivated by the assumption that any block of strongly connected components is "denser" than the rest of the matrix also in the area close to the main diagonal [1]. For a diagonal band width w, the reduction of the sparse matrix image to a "diagonal image" is realized by cropping parallel to the main diagonal at the pre-defined distance w, and arranging all elements of the diagonal band row-wise in a new matrix that is right-aligned. The missing values on the left are filled with an arbitrary constant c, see Figure 3 representing the diagonal image extracted from the matrix image in Figure 2. Considering only the diagonal band of a matrix efficiently reduces the amount of pixels in the input images. In our experiments, we consider matrix images of size 128×128 and set $w = 10$. This results in diagonal images of size $(2 * w + 1) \times n = 21 \times 128$. The neural network we design in Figure 5 is a feed-forward convolutional neural network composed of three logical parts. The first major part of the network, is a modified residual network block [17] that aims at denoising the sparse matrix image. It consists of two two-dimensional convolutional layers with post-batch normalization and scaled exponential linear units (SELU) [21]

[1]This assumes that some non-block areas are included in the diagonal band.

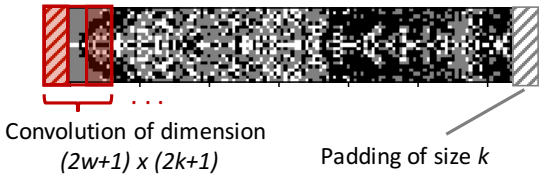

Fig. 4: Visualization of the convolution of size $(2w + 1) \times (2k+1)$ and the padding of size k left and right of the diagonal image.

as activations. The obtained filtered images are added to the original matrix image to generate a denoised copy.

In the second major block, consisting of non-standard, discrete convolutions, the image is reduced to a vector of length $2w + 1$. To that end, the convolution with mask height $2w + 1$ and width $2 * k + 1$ is applied to each matrix column, see Figure 4. To accommodate for the k left- and right-most columns, the diagonal image is horizontally padded with constant values (here: zeros), such that the convolution is applied to a matrix with dimensions $(2w + 1) \times (n + 2 * k)$. Each element within the resulting vector is activated using the $tanh$ function to model a binary choice between the two options "block start" or "no block start". Using a one-dimensional convolution, the choice is cross-correlated with the neighboring elements.

Finally, in the third and last part of the network, the actual prediction is derived using a fully-connected dense layer and output in the l-sized output. To prevent overfitting and increase out-of-sample accuracy, the convolutional layers are regularized with an $l2$-norm of 0.02, and the fully-connected layer with dropouts [30]. Applying the $argmax$ function to the prediction vector $\hat{y}$ we obtain the rows/column indices $i_0, i_1 \ldots$, where the diagonal blocks start.

Block patterns for matrices exceeding the size n can be generated by tiling the matrix along the main diagonal into sub-matrices of size $n \times n$ (potentially requiring the padding for the last sub-matrix). The diagonal block detection is then realized by predicting block starts for each of the tiles independently, and combining the results. For detecting blocks in a matrix containing non-adjacent blocks, i.e. "gaps" in-between diagonal blocks, it is necessary to additionally predict block ends. This can be realized by complementing the architecture with a second network, or by modifying the existing network to feature an additional vector of n output neurons.

IV. Experimental Evaluation

A. Dataset

Due to the unavailability of matrix data with annotated Jacobi block patterns [2], we use an artificially generated dataset for training and validating the neural network. In total, we consider 3,000 matrices of size 128×128. All matrices contain a set of randomly-sized diagonal blocks that are adjacent and

[2]The usage of automatically generated labels from other block detection algorithm is undesirable as they do not have a good prediction accuracy, and will set an upper bound to the prediction performance of the network.

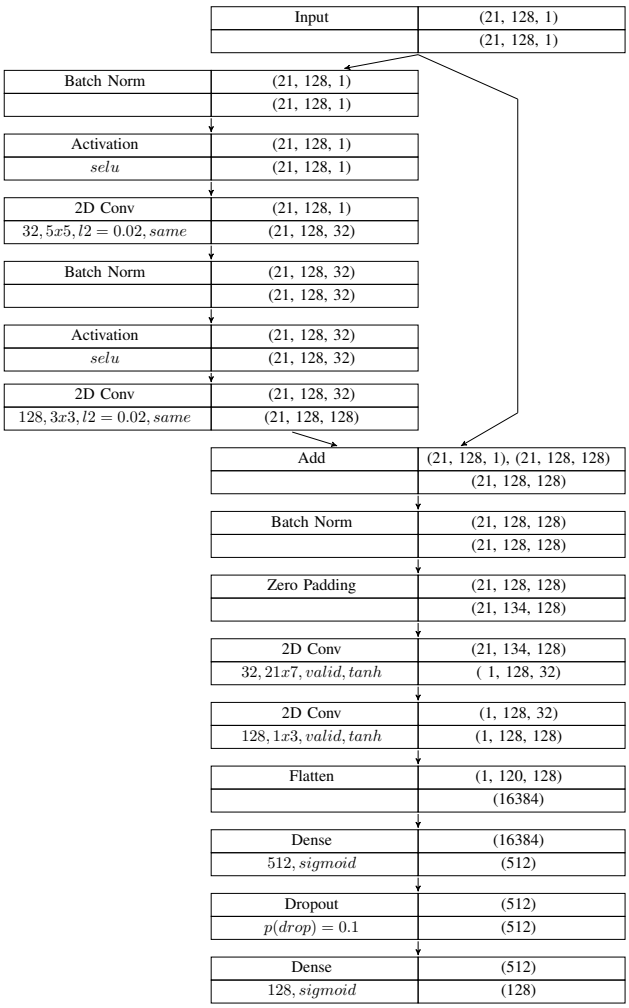

Fig. 5: Schematic representation of the convolutional neural network used for predicting the block patterns.

not overlapping in the matrix sparsity pattern. The average size of the diagonal blocks is 10. The test matrices are all generated in the following fashion:

1) First, the empty matrix is filled with "background noise", which is randomly scattered non-zero values. Across the test matrices we set the density of the background noise to values between 0 and 0.5 (30%- 50% nonzero values) following a uniform distribution.
2) Next, "noise blocks" of arbitrary size are added to the main diagonal. These noise blocks contain between 30% and 50% nonzero elements.
3) Finally, the diagonal blocks of arbitrary size with density values between 0.5 and 0.7 are generated on the main diagonal. This is realized by generating a vector of size 128, and then randomly selecting 10% of the entries to be vector block starts.

When using a trained network on other data, this data has to be normalized to the above described value range. Potentially, this also entails a statistically stable outlier removal, like Q1-Q3 interquartile-range-cutoff.

TABLE I: Hyper-parameter settings used for the neural network training.

Parameter	**Value**
Optimizer	Nadam
Initial learning rate	$1e-4$
Batch size	8
Class weights	0: 0.1, 1: 0.9
Training-test-distribution	80%-20%

For the generated set of test matrices, supervariable blocking will fail to generate useful diagonal blocks. The reason is that the supervariable blocking algorithm does not search for strongly connected components, but instead searches for similarities in the nonzero pattern of adjacent columns [12]. Obviously, when introducing noise by randomly inserting nonzero values, the chance of two adjacent columns sharing the same sparsity pattern is very low.

B. Experimental environment

The convolutional neural network architecture is available on the authors' source code repository [16]. It is derived using the following software packages: `Keras 2.1.6` [11], on top of `tensorflow 1.8.0` [1] as compute backend, `numpy 1.14.3` [26] for efficient multi-dimensional data structures, `h5py 2.7.1` [13] for accessing HDF5 files using `libhdf5 1.8.12`. The utilized compute node consists of 32 Intel Xeon i7 E5-2630 eight-core CPUs clocked at 2.4 GHz, eight AMD DIMM DDR4 16 GB (a total of 128 GB) RAM banks clocked at 2133 MHz and four Nvidia Tesla `K80` GPGPUs with 12 GB VRAM, driver version `396.26` and CUDA 9.0.176 [24].

C. Network training

The network has been trained in a supervised fashion by minimizing the loss function given in Section III. Due to the imbalanced label distribution of roughly 90%–10% "no-blocks" to "blocks," the individual classes have been additionally weighted by their inverse distribution, i.e. 0.1 for "no block" and 0.9 for "block". As optimizer, a modified version of the Adam algorithm [20] has been used, which additionally makes use of the Nesterov-momentum (hence the name Nadam [14]). The resulting learning curves, including the training and test loss, as well as prediction accuracy, are visualized in Figure 6. Training the network with more than 50 epochs results in over-fitting of the network, effectively reducing the training loss to zero, but not improving the test accuracy. The complete list of the training hyper-parameters we used is given in Table I.

To ensure the network has learned the right patterns and is not simply reacting to the regularities of the artificially generated noise, we visually investigate the output of the intermediary convolutional layers of the network. For the example problem given in Figure 3, we present three of the 128 resulting convolved images from the second convolutional layer of the first denoising residual network block.

The first image in Figure 7 corresponds to the desired output of the convolutional layer—a denoised version of

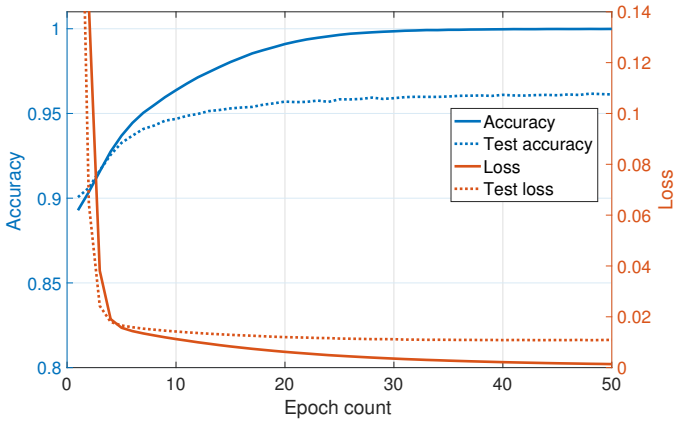

Fig. 6: Visualization of the training progress of the convolutional network. The prediction accuracy and the loss are given in blue and orange color, respectively.

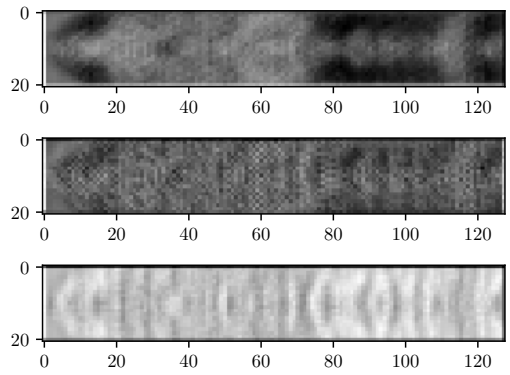

Fig. 7: Examples of the learned convolutional filters of the neural network taken from the second convolutional layer of the residual block.

the original diagonal image. The middle image contains the inverse content, an almost correctly separated image of the superimposed noise. The third image highlights potential block boundaries as vertical, sometimes slightly bent, darker lines.

A quantitative evaluation of the performance of *CNN blocking* is given in Table II. The center of the confusion matrix contains the absolute values of the true and false predictions of block starts and the lack of these. Additionally, the table lists the calculated precision and recall for both prediction classes. For a label vector y and the prediction $\hat{y}$, the precision is defined as:

$$precision(y, \hat{y}) = \frac{tp(y, \hat{y})}{tp(y, \hat{y}) + fp(y, \hat{y})}$$

where tp is true positive rate and fp the false positive rate of the prediction. Hence, the precision describes the rate of correctly predicted positive observations. Analogously, recall is defined as:

$$recall(y, \hat{y}) = \frac{tp(y, \hat{y})}{tp(y, \hat{y}) + fn(y, \hat{y})}$$

TABLE II: Evaluation matrix for the diagonal block predictions of the convolutional neural network on test data.

		Actual		
Acc.:	0.9617	no block	block	precision
Pred.	no block	68010	553	0.9919
	block	2389	5848	0.7100
	recall	0.9661	0.9136	**F1:** 0.7990

where fn is the false negative rate.

The recall for "no block" is relatively high in comparison to "block". This indicates a bias against offensively predicting blocks. The total accuracy of the block detection in the test is 96.17%, which indicates a solid prediction performance. However we note that due to a strong imbalance between the class labels the accuracy of 90% can be achieved by simply always predicting "no-block". The statistically more robust F1-metric [28] is defined as

$$F1(y, \hat{y}) = 2 * \frac{precision(y, \hat{y}) * recall(y, \hat{y})}{precision(y, \hat{y}) + recall(y, \hat{y})}.$$

This harmonic mean of precision and recall may be considered as a more representative performance indicator.

Considering all predicted diagonal blocks and all test matrices, the average size of the predicted diagonal blocks is 9.85.

TABLE III: Evaluation matrix for the diagonal block predictions of supervariable agglomeration with upper block size bound 10 on test data.

		Actual		
Acc.:	0.8261	no block	block	precision
Pred.	no block	62105	6458	0.9107
	block	6895	1342	0.1547
	recall	0.9001	0.1721	**F1:** 0.1673

TABLE IV: Evaluation matrix for the diagonal block predictions of supervariable agglomeration with upper block size bound 25 on test data.

		Actual		
Acc.:	0.8695	no block	block	precision
Pred.	no block	65872	2691	0.9608
	block	7328	909	0.1103
	recall	0.8998	0.2525	**F1:** 0.1535

As expected, supervariable blocking fails to actually detect any blocks for the artificially generated dataset. As no adjacent columns share the same sparsity pattern, supervariable blocking always accumulates blocks of size 1 until the upper block size bound is reached. Hence, supervariable blocking generates a sparsity pattern of uniform blocks on the main diagonal, each of them having the size of the upper bound. We consider upper bounds in the interval $[1, 25]$. For completeness, we provide in Table III and Table IV the confusion matrix for the block predictions generated by supervariable blocking using the upper bound of 10 and 25, respectively. We observe

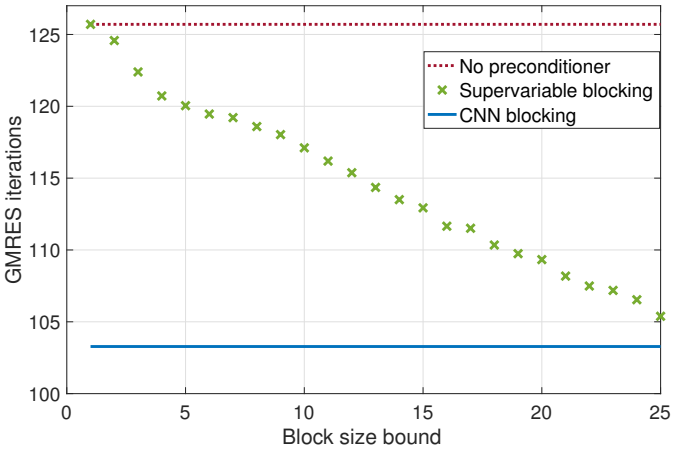

Fig. 8: GMRES iteration counts averaged over the set of test matrices. The GMRES solver uses no preconditioner, a block-Jacobi based on the sparsity pattern generated with the convolutional neural network, and a block-Jacobi preconditioner based on the sparsity pattern generated with supervariable blocking, respectively.

that the accuracy with a seemingly high values of 82.61% and 86.95% are notably smaller than the naïve accuracy-maximizing prediction of always forecasting "no-block", i.e. $\approx$ 90% accuracy. More indicative is the F1-metric taking values of 0.1673 and 0.1535, which is significantly below the 0.7990 achieved by the neural network.

To evaluate the quality of the preconditioner sparsity pattern generated by the convolutional neural network, we use the block structure to generate a block-Jacobi matrix, and deploy the preconditioner inside a GMRES iterative solver. To ensure non-singularity, we replace all nonzero entries in the test matrices with values in the range $(-1, 0)$ and set all diagonal values to 1.0. We derive linear systems by accompanying these matrices with a right-hand side of all-ones. We set the relative residual stopping criterion of the non-restarted GMRES solver to $1e-5$ and start the iterations with an zero-vector initial guess. The number of iterations necessary to reach the approximation accuracy differs for the distinct systems. Averaging over the 3,000 test systems, the block-Jacobi preconditioner based on the sparsity pattern generated by *CNN blocking* reduces the number of necessary iterations from 125.7 (GMRES without preconditioner) to 103.3 ($\approx 22\%$ reduction). In Figure 8 we visualize the GMRES iteration counts along with the results from a setup where the sparsity pattern is derived with supervariable blocking using different block size bounds. As elaborated, for the noise-containing test matrices, the supervariable blocking pads the diagonal with uniform-sized blocks having the size of the upper bound. Since the system matrices all have a unit-diagonal, an upper bound of size 1 (resulting in a scalar Jacobi preconditioner) does not render any improvements to the GMRES convergence. As expected, larger upper bounds allow for better preconditioners, effectively improving the GMRES convergence. For the upper bound of 10, the diagonal blocks generated by supervariable blocking are on average of the same size like the diagonal blocks generated by the convolutional neural network (labeled "CNN blocking" in Figure 8). The preconditioner quality is however inferior, providing only 7% convergence improvement to the plain GMRES. This indicates that supervariable blocking fails to detect the strongly coupled variables. Even for the largest upper bound of 25 (resulting in blocks that are on average more than twice larger than the CNN-generated blocks), the resulting block-Jacobi preconditioner is inferior to the preconditioner based on the sparsity pattern generated with CNN blocking. This indicates that CNN blocking succeeds in detecting strongly coupled components.

The convolutional network necessary to generate the CNN blocking encompasses a total of 9,107,812 parameters. If all of the parameters are stored in four-byte, tightly-packed binary format, the model size is 36 431 248 B or roughly 34.74 MB. For prototyping purposes we use the more verbose `Keras` model serialization format, which additionally stores training meta-data and the network's architecture. This results in a total memory footprint of 105 MB. The model's memory footprint is relevant as a numerical solver ecosystem utilizing CNN-based block pattern detection needs to take the additional memory requirement into account.

In the setting of the network being implemented in Python (using `Keras`) and running on an NVIDIA K80 GPU, it takes 2.5 s to predict the block starts for the set of 3.000 matrices when using a batch size of 1500. This boils down to a block-pattern generation time of about 1.66 ms per matrix.

V. Summary and Outlook

We propose to employ machine learning techniques to generate preconditioner sparsity patterns. In this work, we have shown that a convolutional neural network can efficiently detect strongly connected components in a matrix sparsity image, and that a therefrom derived block-Jacobi preconditioner is effective in accelerating the iterative solution process of the induced linear system. In future work, we plan to manually label a large set of test matrices taken from scientific applications which allows to move from artificially created test matrices to real-world problems. Furthermore, we will investigate the potential of using machine learning techniques for generating sparsity patterns for other problem-adapted preconditioners.

References

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems. *arXiv preprint arXiv:1603.04467*, 2015. [Online, Accessed: 2018-08-16 09:19 CEST].

[2] Jonas Adler and Ozan Öktem. Solving ill-posed inverse problems using iterative deep neural networks. *Inverse Problems*, 33(12):124007, 2017.

[3] Hartwig Anzt, Jack Dongarra, Goran Flegar, and Enrique S. Quintana-Ortí. Batched Gauss-Jordan elimination for block-Jacobi preconditioner generation on GPUs. In *8th Int. Workshop Programming Models & Appl. for Multicores & Manycores*, PMAM, pages 1–10, 2017.

[4] Hartwig Anzt, Jack Dongarra, Goran Flegar, and Enrique S. Quintana-Ortí. Variable-Size Batched LU for Small Matrices and Its Integration into Block-Jacobi Preconditioning. In *2017 46th International Conference on Parallel Processing (ICPP)*, pages 91–100, 2017.

[5] Hartwig Anzt, Jack Dongarra, Goran Flegar, and Enrique S. Quintana-Ortí. Variable-size batched gauss–jordan elimination for block-jacobi preconditioning on graphics processors. *Parallel Computing*, 2018.

[6] Hartwig Anzt, Jack Dongarra, Goran Flegar, Enrique S. Quintana-Ortí, and Andrés E. Tomás. Variable-Size Batched Gauss-Huard for Block-Jacobi Preconditioning. *Procedia Computer Science*, 108:1783 – 1792, 2017. International Conference on Computational Science, {ICCS} 2017, 12-14 June 2017, Zurich, Switzerland.

[7] John Canny. A computational approach to edge detection. *IEEE Transactions on pattern analysis and machine intelligence*, (6):679–698, 1986.

[8] Michael Chau and Hsinchun Chen. A machine learning approach to web page filtering using content and structure analysis. *Decision Support Systems*, 44(2):482 – 494, 2008.

[9] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. *CoRR*, abs/1512.01274, 2015.

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

[11] François Chollet et al. Keras. https://github.com/fchollet/keras, 2015. [Online, Accessed: 2018-08-16 09:17 CEST].

[12] Edmond Chow, Hartwig Anzt, Jennifer Scott, and Jack Dongarra. Using jacobi iterations and blocking for solving sparse triangular systems in incomplete factorization preconditioning. *Journal of Parallel and Distributed Computing*, 119:219 – 230, 2018.

[13] Andrew Collette. *Python and HDF5*. O'Reilly, 2013. ISBN: 978-1449367831.

[14] Timothy Dozat. Incorporating Nesterov Momentum into Adam.

[15] Lance B. Eliot. *Advances in AI and Autonomous Vehicles: Cybernetic Self-Driving Cars Practical Advances in Artificial Intelligence (AI) and Machine Learning*. LBE Press Publishing, 1st edition, 2017.

[16] Markus Götz and Hartwig Anzt. Block prediction. https://github.com/Markus-Goetz/block-prediction/, 2018. [Online, Accessed: 2018-08-28 09:95 CEST, Change set: 3f89d17].

[17] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Identity mappings in deep residual networks. In *European conference on computer vision*, pages 630–645. Springer, 2016.

[18] Markus Hegland and Paul E. Saylor. Block jacobi preconditioning of the conjugate gradient method on a vector processor. *International Journal of Computer Mathematics*, 44(1-4):71–89, 1992.

[19] Norman P. et al. Jouppi. In-datacenter performance analysis of a tensor processing unit. *SIGARCH Comput. Archit. News*, 45(2):1–12, June 2017.

[20] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[21] Günter Klambauer, Thomas Unterthiner, Andreas Mayr, and Sepp Hochreiter. Self-normalizing neural networks. In *Advances in Neural Information Processing Systems*, pages 971–980, 2017.

[22] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *nature*, 521(7553):436, 2015.

[23] Stefano Markidis, Steven Wei Der Chien, Erwin Laure, Ivy Bo Peng, and Jeffrey S. Vetter. NVIDIA tensor core programmability, performance & precision. *CoRR*, abs/1803.04014, 2018.

[24] NVIDIA. *NVIDIA CUDA TOOLKIT V9.0*, March 2018.

[25] NVIDIA Corp. Whitepaper: NVIDIA TESLA V100 GPU ARCHITECTURE, 2017.

[26] Travis E Oliphant. *A guide to NumPy*, volume 1. Trelgol Publishing USA, 2006.

[27] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch. 2017.

[28] David Martin Powers. Evaluation: from precision, recall and f-measure to roc, informedness, markedness and correlation. 2011.

[29] Jürgen Schmidhuber. Deep learning in neural networks: An overview. *Neural Networks*, 61:85 – 117, 2015.

[30] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *The Journal of Machine Learning Research*, 15(1):1929–1958, 2014.

[31] Deep Learning for Medical Image Analysis. Academic Press, 2017.

[32] W. Tang, T. Shan, X. Dang, M. Li, F. Yang, S. Xu, and J. Wu. Study on a poisson's equation solver based on deep learning technique. In *2017 IEEE Electrical Design of Advanced Packaging and Systems Symposium (EDAPS)*, pages 1–3, Dec 2017.

[33] Yue Zhao, Jiajia Li, Chunhua Liao, and Xipeng Shen. Bridging the gap between deep learning and sparse matrix format selection. *SIGPLAN Not.*, 53(1):94–108, February 2018.