No. 581 **February 2018**

On the Prospects of Using Machine Learning for the Numerical Simulation of PDEs: Training Neural Networks to Assemble Approximate Inverses

H. Ruelmann, M. Geveler, S. Turek

ISSN: 2190-1767

On the Prospects of Using Machine Learning for the Numerical Simulation of PDEs: Training Neural Networks to Assemble Approximate Inverses

Hannes Ruelmann*, Markus Geveler†, Stefan Turek‡

Abstract

In an unconventional approach to combining the very successful Finite Element Methods (FEM) for PDE-based simulation with techniques evolved from the domain of Machine Learning (ML) we employ approximate inverses of the system matrices generated by neural networks in the linear solver. We demonstrate the success of this solver technique on the basis of the Poisson equation which can be seen as a fundamental PDE for many practically relevant simulations [Turek 1999]. We use a basic Richardson iteration applying the approximate inverses generated by fully connected feedforward multilayer perceptrons as preconditioners.

Keywords: machine learning, FEM, preconditioning, SPAI

1 Introduction

There is conclusive evidence that we are on the edge of a technical revolution driven by artificial intelligence. To be more precise Machine Learning is a class of methods that can solve a multitude of problems by storing knowledge to and inferring it from a knowledge base that had previously been created via a training process. These techniques can be seen as a black box framework since they are strong in providing classification or even regression when exploring and altering large (unstructured) datasets for example for pattern recognition in text-, image-, video- or – in general – signal processing [Goodfellow et al. 2016]. Due to its success the hardware industry and chip vendors adapt their roadmaps to satisfy an ever larger demand to computing hardware that is especially tailored for ML. For example, due to the fact that the underlying computations in many cases don't need high precision, low (e.g. half-) precision accelerators are hitting the hardware ecosystem like Intel Knights Mill and NVIDIA Pascal and Volta or new microarchitectures are developed like Google's TPUs.

However at the moment it is unclear in what way and in how far these comparatively new methods and – alongside with them the modern and future compute hardware – can be exploited to assist solving PDEs in technical simulations: In the course of discretising multidimensional PDEs at a certain point we have to deal with a high number of degrees of freedom leading to the global system matrix being large and sparse. Hence, iterative methods have to be chosen over direct ones. In the former everything breaks down to how clever the linear solver can adapt to the system to be solved and here using specially tailored solvers that are implemented in a target hardware-oriented way can be orders of magnitude faster than simple ones.

The idea of this paper is based on the observations, that (1) besides pattern recognition ML can also be used for *function regression* and (2) that preconditioners in linear solvers can be kind of underdetermined and yet yield a good preconditioner: In previous studies we were able to show, that Sparse Approximate Inverses (SPAI) are a good representative of such a preconditioner [Geveler et al. 2013]. The application of an approximate inverse can be broken down to sparse matrix vector multiply (SpMV) and with sophisticated storage formats SpMV kernels map decently to for example GPUs. In contrast to that usual implementations of SPAI algorithms to *assemble* the approximate inverse are (in spite of their good parallelisation properties) quite expensive. Hence the idea is to compute a rough draft of an explicitly stored preconditioner in a different way and therefore provide an alternative to SPAI: Use the system matrix as input to a trained neural network and render the result into another (sparse) matrix that is used as an approximation to its inverse. This way the output of the function regression process in the machine learning pipeline is a matrix like in many image processing cases the output of this process would be another (enhanced) image.

In order to pioneer into the fusing of FEM and ML in this paper we provide insight into how such a system could work and concentrate on providing evidence that the resulting inverses can numerically compete with other preconditioners.

2 A concise application of neural networks in a linear solver

2.1 Model problem and FEM discretisation

As a starting point we define the Poisson equation to be our model problem, which is posed as: Find $u : \Omega \to \mathbb{R}$ such that

$$-\Delta u = f \quad \text{in } \Omega, \qquad u = 0 \quad \text{on } \partial\Omega \,. \tag{1}$$

Following the guidelines of [Braess 2013] we can convert this problem by using the variational formulation of (1) and the well-known Galerkin method into a discrete problem:
Find $u_h \in V_h$ such that

$$a_h(u_h, v_h) = b_h(v_h) \quad \forall v_h \in V_h \,. \tag{2}$$

In our case V_h is the finite element space of linear polynomials, which are zero on the boundary. The domain Ω is the unit square $(0,1)^2$ discretized with regular triangles T_h and a conforming refinement at the midpoints of the edges.
The global system matrix can be written as

$$(\mathbf{A}_h)_{ij} = \sum_{m=1}^{N} \int_{K_m} \nabla\phi_j \cdot \nabla\phi_i dx = \sum_{m=1}^{N} \mathbf{A}_{ij}^{(m)} \,. \tag{3}$$

with a nodal basis $\{\phi_1, ..., \phi_M\}$ and the local element matrices $\mathbf{A}_{ij}^{(m)}$ on the element K_m. Analogously we can proceed with the right hand side as $(b_h)_i = \int_\Omega f\phi_i dx$.

*TU Dortmund, email: hannes.ruelmann@math.tu-dortmund.de
†TU Dortmund, email: markus.geveler@math.tu-dortmund.de
‡TU Dortmund, ECCOMAS co-chairman for Scientific Computing, email: ture@featflow.de

2.2 Training tensor and basic iteration

To solve the corresponding system of equations $\mathbf{A}_h u_h = b_h$, with a sparse matrix $\mathbf{A}_h \in \mathbb{R}^{n \times n}$ which satisfies the M-matrix property [Saad 2003], we want to use a neural network. Hence a mechanism is needed to bring up a sufficiently large training dataset (called a training tensor). For this purpose we construct instances of $\mathbf{A}_h$ by randomly shifting the inner nodes on the finest level by maximum half the grid step size.
As the solver we use the Richardson iteration, which reads in its fixpoint formulation as:

$$x^{(k+1)} = x^{(k)} + \omega \mathbf{M}(b_h - \mathbf{A}_h x^{(k)}) \tag{4}$$

Here $\mathbf{M}$ is an approximate inverse we generate with the neural network.

3 Constructing a Machine Learning framework for solving linear systems of equations

3.1 Neural Network design and preconditioner construction

The design space for Neural Networks is very large. Since this paper is meant as a starting point for the exploration of fusing FEM and ML we keep it as simple as possible and employ straightforward choices where possible. Therefore we use fully connected feed-forward multilayer perceptrons. Fully connected means that every neuron in the network is connected to each neuron of the next layer. Moreover there are no backward connections between the different layers (feed-forward). The evaluation of such neural networks is a sequence of chained-up matrix vector products. The entries of the system matrix are represented in the input layer

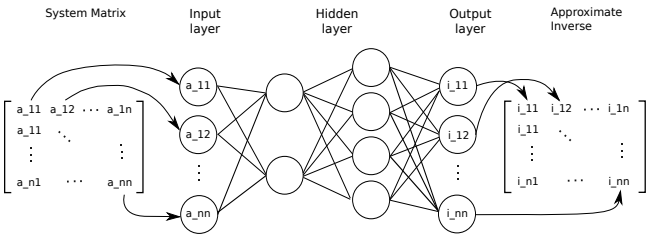

Figure 1: *Model of a neural network for matrix inversion*

vector-wisely (cf. Figure 1). In the same way, our output layer contains the entries of the approximate inverse. Between these layers we can add a number of hidden layers consisting of a bunch of hidden neurons. How many hidden neurons we need to create strong approximate inverses is a key design decision and we discuss this below.

3.2 Training and testing phase

In figure 2 we can see how we want to handle the neural network. First of all we use a pile of matrix pairs $(A_h)_i$ and its corresponding inverse $(A_h^{-1})_i$ to train the neural network via supervised learning. With some test data we can identify whether the neural network is able to generalise. This way we can determine how good the neural network works for approximating inverses of general matrices that are somehow similar but not identical to those used in training. Whether we are able to produce a suitable approximate inverse

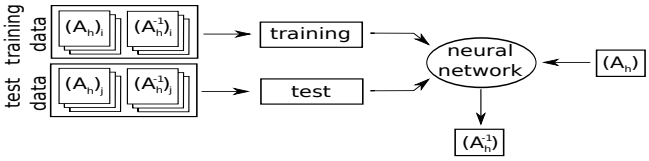

Figure 2: *Model of a neural network to generate an approximate inverse*

mainly depends on the structure of the neural network and the training algorithm.
In general our supervised training algorithm is called backward propagation with random initialisation. Alongside a linear propagation function

$$i_{\text{total}} = \mathbf{W} \cdot o_{\text{total}} + b$$

with the total (layer) net input i_{total}, the weight matrix $\mathbf{W}$, the vector for the bias weights b and the total output of the previous layer o_{total}, we use the rectified linear unit (ReLu) function as activation function $\alpha(x)$ [Goodfellow et al. 2016] and thus we can calculate the output y of each neuron as:

$$y := \alpha\left(\sum_{\text{j}} o_{\text{j}} \cdot w_{\text{ij}}\right)$$

Here o_{j} is the output of the preceding sending units and w_{ij} the corresponding weights between the neurons.
For the optimization we use the L2-error-function and employ for the update for the weights:

$$w_{\text{ij}}^{(t+1)} = w_{\text{ij}}^{(t)} + \gamma \cdot o_i \cdot \delta_{\text{j}}$$

with the output o_i of the sending unit, γ learning rate and δ_{j} symbolises the gradient decent method:

$$\delta_{\text{j}} = \begin{cases} f'(i_{\text{j}}) \cdot (\hat{o}_{\text{j}} - o_{\text{j}}), & \text{if neuron j is an output neuron} \\ f'(i_{\text{j}}) \cdot \sum_{k \in S} (\delta_k \cdot w_{kj}), & \text{if neuron j is a hidden neuron.} \end{cases}$$

With these definitions we can describe the training and testing phases with the pseudocode presented in Algorithm 1.

Algorithm 1 test and training phase

input: n_{hl}, n_{i}, n_{epoch}, n_{batch}, n_{train}, l, n_{test}, $n_{\text{testbatch}}$
· define the neural network (n_{hl}, n_{i})
· initialize weights
· initialize bias neurons b
· define error function and optimizer
start training:
for i in n_{epoch} **do**
A = load_training_matrices (n_{train})
A_inv = load_training_inverses (n_{train})
for j in n_{batch} **do**
x = A_batch$_j$
y = A_inv_batch$_j$
apply optimizer: (W, b) = opt(x, y, l)
test phase:
for i in $n_{\text{testbatch}}$ **do**
A = load_test_data (n_{test})
evaluate the neural network
apply error function or test scenario
output: W, b

4 Numerical experiments

4.1 Preconditioner quality

In order to get an impression on whether it is possible to gain suitable approximate inverses with neural networks we eliminate

for any complicated tweaks and start with networks for a fixed problem size (i.e. degrees of freedom) n and train it with pairs of (dense) system matrices and their inverses.

		# it			it down	κ	
lvl	n	$J_{\omega=0.7}$	GS	NN		before	after
2	9	49	17	8	2.13	5.9	1.6
3	49	273	95	21	4.52	29.8	2.9
4	225	1 323	463	66	7.02	127.3	7.8
5	961	5 879	2 057	39	52.74	516.0	23.4

Table 1: *Iteration and condition number κ, tol = $1.0 \cdot 10^{-5}$, 3 layer neural networks*

In Table 1 we deploy the results for using the approximate inverse of the system matrices with problem size-specific neural networks in Richardson iterations (labelled NN) in comparison to the damped Jacobi (J) and Gauss-Seidel (GS) defect correction methods. In addition to the iteration number the reduction of the condition number κ is shown for the neural networks. For the problem size levels 2, 3 and 4 the number of neurons in the 3 hidden layers equals the corresponding matrix dimension. The number of training epochs is set to 1 000.
As we can see from the data for every test configuration the neural network is able to generate a matrix that serves well as a preconditioner. The corresponding Richardson method needs less iterations than the damped Jacobi or the Gauss-Seidel method. Moreover the condition number is strongly reduced by the neural networks-generated preconditioner.
With neural networks it is a priori not possible to determine which number of hidden neurons and how many training epochs would work out best. Even other parameters like the learning rate and the matrix dimensions between the neurons have a large impact on the iteration numbers: The level 5 configuration needs even less iterations to converge than the previous level 4 configuration, because the parameters - fewer neurons with the same amount of training epochs and the online learning - fit better to that configuration. With error functions like the L2-loss function we can get a training accuracy and a rough idea of how good a neural network will be functional in the test and application phase, but a good enough accuracy for one network can be much too low for another case.

4.2 Time to solution and memory control

Initialisation and application The timings and the speedup between two different large neural networks and the Jacobi as well as the Gauss-Seidel method are shown for refinement level 4 in Figure 3. The underlying hardware is a Intel Xeon E5-2670 with 8 cores and a frequency of 2.60 GHz.
The noticeable differences between the methods are resulting from the initialisation time on the one hand and from the numbers of iterations on the other hand. While the Jacobi method needs only 3.59e-05 s to initialize the neural network with 50 neurons needs 3.44e-03 s and the network with 225 neurons needs 5.80e-03 s for the assembly. However the Gauss-Seidel defect correction needs 1.50e-02 s. The neural networks and the Gauss-Seidel are able to catch up with a lower number of iterations. For instance we get $it_J = 990$, $it_{GS} = 346$, $it_{NN50} = 32$ and $it_{NN225} = 23$ for a tolerance of 10^{-03} and $it_J = 2323$, $it_{GS} = 812$, $it_{NN50} = 106$ and $it_{NN225} = 72$ for a tolerance of 10^{-03}. Here it is noticeable that we do not need that many neurons even if the iteration number might decrease a bit.

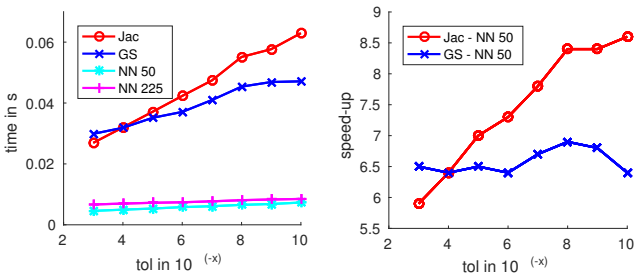

Figure 3: *Time and speed-up between Jacobi, Gauss-Seidel and the neural network preconditioned Richardson iteration*

Memory footprint Due to storage problems for larger matrices we use the online learning method in which we train our network with only one pair of system matrix and associated inverse in each training step instead of the batch learning with 100 pairs for the lower level. Moreover we reduce the number of the hidden neurons to 100 in each hidden layer for the level 5 configuration.
In addition we found that we do not need as many neurons in the hidden layer as in the input layer. This leads to a reduction of the weight matrices and therefore to decreased assembly and application times.
Since we use fully connected neural networks the structure of the matrix which depends on the node numbering is irrelevant. Hence for further simplification we can assume a banded system matrix and instead of using every matrix entry as input data, we save the matrix bands sequentially. By utilizing the matrix symmetry we can reduce the input data even more. Like for the unit square we have to store only 4 bands instead of 7. The resulting benefits are shown in table 2.

n	49	225	961	3 969	16 129
full	2 401	50 625	923.521	15 752 961	260 144 641
$\bar{n}$	289	1 457	6 481	27 281	111 889
diag	180	868	3 780	15 748	64 260
%	7.497	1.715	0.409	0.100	0.025

Table 2: *Storage of a $n \times n$ matrix with a dense (full) storage, the number of non-zeros ($\bar{n}$) and by utilizing the symmetry (diag), unit square*

Reducing time to solution and memory footprint We now employ a sparsity pattern which leads to less storage requirements and a faster application owing to smaller weight matrices. Due to the large matrix size the first matrix vector multiplication corresponding to the input layer and the last one to assemble the output matrix are the most expensive operations. By reducing the number of input values the first matrix vector multiplication can be reduced as well. Again we take a look at a matrix resulting from refinement level 4. By utilizing the sparsity pattern the dimension of the weight matrix is changed from 50.625×225 to 868×225 which equates to a factor of 58. For a new test matrix the assemble time is lowered from $4,378 \cdot 10^{-3}$ s to $2,474 \cdot 10^{-3}$ s. Since one of the two expensive operations is decreased massively the initialisation time is nearly halved.

Training Another benefit we generate from a reduced amount of input values is to simplify the training in general. Table 3 displays the iteration number of a Richardson iteration scheme with the approximate inverse of three different neural networks with the same

setting as above. The first one uses the sparse storage format and is trained with 10 000 pairs of input matrices and inverses in 1 000 training cycles. The other are trained on a dense storage format with a greater amount of 25 000 input data and 1 500 respectively 2 000 training epochs. With the dense storage the neural network

	NN_{sparse} (1 000)			NN_{full}(1 500)			NN_{full} (2 000)		
ω	0,6	0,7	0,8	0,6	0,7	0,8	0,6	0,7	0,8
1	33	27	23	118	110	88	29	24	37
2	37	31	26	110	94	82	31	25	20
3	35	29	25	109	93	81	29	24	22

Table 3: *Number of damped Richardson iterations with a sparse and a full storage for neural networks trained with different problem sizes*

got a greater weight matrix between the input and the first hidden layer. To adjust the greater amount of weights we need much more training data and epochs. Moreover we see the behaviour of the damped Richardson method with different damping parameters which is again difficult to optimise a priori.
On the other hand we lose the flexibility of the dense format and are bounded to a 'fixed' matrix structure. In most PDE-based simulations only sparse matrices with a predefined matrix structure due to the coupling of degrees of freedom and their numbering are used which neutralize this disadvantage. Moreover we can get more flexibility by adding zeros in those matrix locations where they are needed and take the benefits described above.

4.3 Designing sparse approximate inverses

To be competitive to SPAI and ILU preconditioners we have to fasten up the second large matrix vector multiplication and produce sparse output matrices. In general the inverse of a sparse matrix is not a sparse matrix. That is the reason why we use a filter method to reduce the approximate inverse of the matrix afterwards.
Table 4 contains the iteration number of the damped Richardson iteration method ($\omega = 0.8$) with an approximate inverse out of a neural network compared to the exact inverse after setting all entries smaller than ϵ to zero. With this filter method the matrix can be reduced to a sparse matrix. The underlying neural network is the same as above.

ϵ	# it (NN)	$\bar{n}$(NN)	%	# it (exact)	$\bar{n}$ (exact)
0.00	23	50 625	100.0	1	50 625
0.01	24	35 469	70.1	9	35 319
0.02	29	26 755	52.8	12	26 663
0.03	48	21 029	41.5	21	21 095
0.04	58	17 133	33.8	38	17 173
0.05	279	14 171	28.0	620	14 233

Table 4: *Number of iterations with the Richardson solver ($\omega = 0.8$) and different filtered approximate inverses in comparison to the filtered accurate inverse*

As we can see it is possible to reduce the approximate inverse by approximately $\frac{2}{3}$ and still get a converging method. In comparison to that the Gauss-Seidel method, which operates on nearly 50% of the matrix entries, needs 462 iterations to reach the same tolerance of 10^{-5}.

5 Conclusion and future work

We were able to demonstrate that it is at least possible to bring up simple learning systems that extrapolate strong approximate inverses for FEM matrices. Many of the descibed techniques are presented in detail in [Ruelmann 2017] and we are also preparing a follow-up publication that dives into many other aspects we abstained from presenting in this short introduction [Ruelmann et al. 2018]. The current state of our research triggers a lot of questions that have to be answered by future work, including

- Is it possible to bring up an optimised performance model describing the complete process from training a specific network for a problem over initialisation (aka assembly of the approximate inverse) up to the application?
- What is an optimal (or at least better) design for the neural network – since there are many screws to adjust e. g. the number of hidden layers and neurons as well as the size of the training data, the learning rate or the functions like activation, propagation and loss function in addition to the choice of the optimizer?
- Since our results indicate that the potential of the resulting approximate inverses is really big – how competitive is it with SPAI? A simple SPAI-1 method for example theoretically should speed up convergence in the order of magnitude of Gauss-Seidel. Note that in our results, the preconditioner is (much) better than GS.
- How beneficial will neural networks be as smoother or preconditioner in stronger solvers especially multigrid?
- How well will the neural network cope with larger alteration of the problem than modelled in the training tensor?
- What is the shape of a ML system for arbitrary matrices with arbitrary sizes, sparsity patterns, coefficients?

Acknowledgements

This work has been supported in part by the German Research Foundation (DFG) through the Priority Program 1648 Software for Exascale Computing (grant TU 102/48).

References

BRAESS, D. 2013. *Finite Elemente: Theorie, schnelle Löser und Anwendungen in der Elastizitätstheorie*. Springer-Verlag.

GEVELER, M., RIBBROCK, D., GÖDDEKE, D., ZAJAC, P., AND TUREK, S. 2013. Towards a complete FEM–based simulation toolkit on GPUS: Unstructured Grid Finite Element Geometric Multigrid solvers with strong smoothers based on Sparse Approximate Inverses. *Computers and Fluids 80* (July), 327–332. doi: 10.1016/j.compfluid.2012.01.025.

GOODFELLOW, I., BENGIO, Y., AND COURVILLE, A. 2016. *Deep Learning*. MIT Press. `http://www.deeplearningbook.org`.

RUELMANN, H., GEVELER, M., AND TUREK, S. 2018. Machine Learning-generated Sparse Approximate Inverses. under preparation.

RUELMANN, H. 2017. *Approximation von Matrixinversen mit Hilfe von Machine Learning*. Master's thesis, TU Dortmund, Dortmund, Germany.

SAAD, Y. 2003. *Iterative methods for sparse linear systems*. SIAM.

TUREK, S. 1999. *Efficient Solvers for Incompressible Flow Problems: An Algorithmic and Computational Approach*, vol. 6. Springer, Jan. 3-540-65433-X.