Learning Optimal Prolongation and Restriction Operators for Multigrid PDE Solvers

Philipp Suffa
Learning Optimal Prolongation and Restriction Operators for Multigrid PDE Solvers

Philipp Suffa
Masterarbeit

Aufgabensteller: Prof. Dr. Harald Köstler
Betreuer: M. Sc. Jonas Schmitt
Bearbeitungszeitraum: 07.07.2020 – 18.02.2021
Erklärung:

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Der Universität Erlangen-Nürnberg, vertreten durch den Lehrstuhl für Systemsimulation (Informatik 10), wird für Zwecke der Forschung und Lehre ein einfaches, kostenloses, zeitlich und örtlich unbeschränktes Nutzungsrecht an den Arbeitsergebnissen der Masterarbeit einschließlich etwaiger Schutzrechte und Urheberrechte eingeräumt.

Erlangen, den 18. Februar 2021
Abstract

Multigrid methods are an excellent tool for solving partial differential equations. A crucial part of Multigrid solvers are the prolongation and restriction operators, heavily influencing the Multigrid method’s efficiency. In practice, it is laborious to construct prolongation and restriction operators for new problems. Therefore a trainable Multigrid solver is developed in this work. It is shown how to implement a fully functional Multigrid V-Cycle in the machine learning framework Tensorflow, where every layer of a neural network corresponds to a Multigrid operation, such as a relaxation step, the residual equation, or the prolongation and restriction. This neural network is then capable of learning optimized prolongation and restriction operators. The learned operators are tested on two-dimensional diffusion problems with varying coefficients. The convergence factor of the deep Multigrid solver is compared to a geometric Multigrid solver with and without the Galerkin coarsening, as well as to an algebraic Multigrid solver. In addition, the influence of various learning parameters is discussed, and the operators are tested in terms of generalization.
1 Motivation

The rising star in computer science these days is machine learning. There are a lot of different applications for machine learning. The most common examples are computer vision using machine learning for image recognition or natural language processing, utilizing it for speech recognition. But also bioinformatic operates with machine learning, for example in tumour detection and DNA sequencing. The energy sector utilizes machine learning for predictions of energy load and prices, and the finance sector for Credit Scoring and algorithmic trading. [1]

Even everyday life is influenced by machine learning. If it is your virtual personal assistant Siri learning your preferences, Netflix recommending series adapted to you or Google refining its search engine by tracking your surf behaviour. [2]

In this work, machine learning is used to optimize partial differential equation (PDE) solvers. PDEs are crucial for mathematically-oriented scientific fields. They are utilized to model physical and engineering problems such as sound, heat, diffusion, electrostatics and fluid dynamics. Because most PDEs cannot be effectively solved analytically, iterative solvers are used to numerically approximate their solution. An established iterative solver is the Multigrid method. Its convergence relies significantly on its prolongation and restriction operators. The approach of this work is to create a trainable Multigrid solver as neural network, where each operation of a classical Multigrid scheme is one layer of the network. This trainable Multigrid solver is then capable of learning optimized prolongation and restriction operators for various two-dimensional diffusion problems.

1.1 Related work

This work is inspired by "Learning to Optimize Multigrid PDE Solvers" by Greenfeld et al. [3]. It proposes a framework, which learns a single mapping from discretized PDEs to prolongation operators for a broad class of 2D diffusion problems, using a neural network and an unsupervised loss function. A detailed description of the paper and a comparison to this work’s approach is given in subsection 7.2.

Another paper, which is based on learning prolongation/restriction operators for Multigrid is "Deep Multigrid: learning prolongation and restriction matrices" by Katrutsa et al. [4]. It also follows the approach of considering every operation of a Multigrid solver as a layer of a deep neural network, while only the layers of the prolongation and the restriction are trainable and thus be optimized through Backpropagation. More details are shown in subsection 7.2. The successor of this paper is "Black-box learning of multigrid parameters" [5], which is based on the approach of its predecessor, extended for two-dimensional problems and a recursive coarse grid correction instead of the two-grid method.

In "Learning neural PDE solvers with convergence guarantees" by Hsieh et al. [6] not the prolongation/restriction is the focus of learning, but a trainable error correction term is introduced to make both the Jacobi method and the Multigrid method converge faster.

There are a variety of papers related to solving PDEs with the help of neural networks. While all the previous papers are related to Multigrid methods, there are many interesting approaches of using neural networks for solving PDEs not referred to Multigrid, for example [7–13].

In general, finding good prolongation and restriction operators for Multigrid methods without utilizing neural networks is studied a lot, for example in [14–16]. For general literature about the Multigrid method, see [16, 17].
2 Multigrid background

The Multigrid method is typically used for boundary value problems. Boundary value problems are differential equations in combination with special conditions on the boundaries, where the solution of the differential equation is found by fulfilling the conditions on the boundaries. These boundary value problems arise in several branches of physics. For example, consider a two-point boundary value problem that describes the steady-state temperature distribution in a long uniform rod in one dimension. This can be expressed as second-order differential equation

\[-u(x)'' = f(x), \quad 0 < x < 1,\]
\[u(0) = u(1) = 0.\]

This problem is called the Poisson equation. It could be solved analytically as well, but this can often be complex and lead to errors. A superior approach is a numerical method, such as the finite difference method. For this method the domain is divided into \( n \) sub-intervals to transform the continuous problem to a discrete one. So we introduce \( v_j \) as approximation of the exact solution of \( u(x_j) \). So the components of the vector \( \mathbf{v} = (v_1, ..., v_{n-1})^T \) satisfy

\[-v_{j-1} + \frac{2v_j - v_{j+1}}{h^2} = f(x_j), \quad 1 \leq j \leq n - 1,\]
\[v_0 = u_n = 0.\]

This system of linear equations can also be written in matrix form as \( A\mathbf{v} = \mathbf{f} \) or

\[
\begin{pmatrix}
2 & -1 \\
-1 & 2 & -1 \\
& . & . & . \\
& & . & . & . \\
& & & . & . \\
-1 & 2 & & & \\
& & & & & -1 \\
\end{pmatrix}
\begin{pmatrix}
v_1 \\
v_2 \\
. \\
. \\
. \\
v_n-1 \\
\end{pmatrix} =
\begin{pmatrix}
f_1 \\
f_2 \\
. \\
. \\
. \\
f_n-1 \\
\end{pmatrix}
\]

with the right-hand side values \( \mathbf{f} = (f(x_1), ..., f(x_{n-1}))^T = (f_1, ..., f_{n-1})^T \). The matrix \( A \) is tridiagonal, symmetric and positive-definite. As it is expensive to store the whole matrix \( A \) with size \((n - 1)^2\) for \( n - 1 \) inner grid points, a stencil notation can be useful. For the one dimensional example this leads to

\[ A = \frac{1}{h^2} \begin{bmatrix} -1 & 2 & -1 \end{bmatrix}. \]

A one-dimensional application is rare, since most physical problems need to be defined at least in two dimensions. The problem of (1) for two dimensions yields

\[-u_{xx} - u_{yy} = f(x, y), \quad 0 < x < 1, \quad 0 < y < 1. \]

As before, we reconsider this problem on discrete grid points \((x_i, y_j) = (ih_x, jh_y)\). Also, we only employ uniform cartesian grids, which means that the same number of grid points in \( x \) as well as in \( y \) direction is used, so \( n_x = n_y = n \) and \( h_x = h_y = h = 1/n \). Replacing (2) by second-order finite differences leads to the system of linear equations

\[ \frac{4v_{i,j} - v_{i-1,j} - v_{i+1,j} - v_{i,j-1} - v_{i,j+1}}{h^2} = f_{i,j}, \]
\[ v_{i,0} = v_{i,n} = v_{0,j} = v_{n,j} = 0, \quad 1 \leq i \leq n - 1, \quad 1 \leq j \leq n - 1. \]

If we write the system \( A\mathbf{v} = \mathbf{f} \) again in matrix notation, we get

\[
\begin{pmatrix}
B & -aI \\
-aI & B & -aI \\
& . & . & . & . \\
& & . & . & . & . \\
& & & . & . & . \\
& & & & . & . \\
& & & & & & B \\
\end{pmatrix}
\begin{pmatrix}
v_1 \\
v_2 \\
. \\
. \\
. \\
v_{n-1} \\
\end{pmatrix} =
\begin{pmatrix}
f_1 \\
f_2 \\
. \\
. \\
. \\
f_{n-1} \\
\end{pmatrix},
\]
where $B$ is the same matrix as for the one dimensional case again with size $(n-1)^2$ but with 4 in the diagonal, $a = \frac{1}{17}$ and $I$ is the identity matrix with size $(n-1)^2$. The whole matrix $A$ then has the size $(n-1)^2 	imes (n-1)^2$, and $A$ is generally symmetric ($A = A^T$) and sparse (a great amount of entries are zeros). Because of the size of $A$ and the sparse property, it is again useful to only store the stencil instead of the whole matrix, which here is

$$A = \frac{1}{h^2} \begin{bmatrix} 0 & -1 & 0 \\ -1 & 4 & -1 \\ 0 & -1 & 0 \end{bmatrix}.$$ 

For the boundaries, the Dirichlet boundary condition is used in this work. For Dirichlet boundaries, there is an extra layer of outer points around the inner points, so for one dimension, there are $n-1$ inner points and 2 boundary points.

### 2.1 Iterative solvers

An exemplary two dimensional boundary value problem is presented in (3). Now there are two possibilities to solve this system of linear equations. The first one is by direct methods, for example Gaussian elimination. These methods are very efficient, needing only $O(n^2 \log(n))$ operations for a $n \times n$ grid size, but are also rather specialized and restricted primarily to systems that arise from separable self-adjoint boundary value problems. The second possibility to solve these systems of linear equations are iterative methods. Their goal is to improve the current approximation of the exact solution with multiple simple update steps. Iterative methods start with an initial guess, and after a number of iterations, the method converges close enough to the exact solution and stops. In this thesis, we focus on iterative methods, in particular the Jacobi relaxation.

To simplify (3), it is multiplied by $h^2$ to achieve

$$4v_{i,j} - v_{i-1,j} - v_{i+1,j} - v_{i,j-1} - v_{i,j+1} = h^2 f_{i,j},$$

$$v_{i,0} = v_{i,n} = v_{0,j} = v_{n,j} = 0, \quad 1 \leq i \leq n-1, \quad 1 \leq j \leq n-1.$$ 

Now the equation can be reformulated to only have $v_{i,j}$ on the left side to get the component notation of the Jacobi relaxation iteration scheme

$$v_{i,j}^{(1)} = \frac{1}{4}(h^2 f_{i,j} + v_{i-1,j}^{(0)} + v_{i+1,j}^{(0)} + v_{i,j-1}^{(0)} + v_{i,j+1}^{(0)}).$$

While the matrix $\mathbf{v}^{(0)}$ contains the current values of the approximate solution, $\mathbf{v}^{(1)}$ contains the new values that need to be computed. This iteration scheme is proceeded for every point in the grid excluding the boundary points, to update all points to a new value, that is expected to converge to the exact solution. The term \textit{component notation} means the equation, which is formulated as components with index declaration. It is also possible to express the Jacobi relaxation scheme in matrix notation, which is the opposite of the component notation. The matrix $A$ can be split in the form

$$A = D - L - U,$$

where $D$ is the diagonal of $A$, and $-L$ and $-U$ are the strictly lower and upper triangular parts of $A$, respectively. Now the term $Au = f$ can be written as

$$(D - L - U)u = f$$

and reformulated to

$$u = D^{-1}(L + U)u + D^{-1}f.$$ 

This matrix notation is going to be useful when the Jacobi method needs to be implemented in Tensorflow, because only full matrix operations are allowed in the machine learning framework.

The original Jacobi method can be extended by a weighting or damping factor to achieve better convergence. Therefore the result of (4) is defined as a temporary value

$$v_{i,j}^* = \frac{1}{4}(h^2 f_{i,j} + v_{i-1,j}^{(0)} + v_{i+1,j}^{(0)} + v_{i,j-1}^{(0)} + v_{i,j+1}^{(0)}).$$
and it is damped by a weighting factor $\omega$ and added to its previous value multiplied by $(1 - \omega)$ to get

$$u_{i,j}^{(1)} = \omega u_{i,j}^{(0)} + (1 - \omega) u_{i,j}^{(0)}.$$ 

A damping factor of $\omega = 1$ results in the standard Jacobi method. In this work, the damping factor of $\omega = 0.6$ is used in all following computations.

To understand how well the Jacobi relaxation iterations perform, a metric is needed to identify the convergence of our solver. The error of the equation system $A\mathbf{v} = \mathbf{f}$ is

$$\mathbf{e} = \mathbf{u} - \mathbf{v}.$$ 

The magnitude of the error can be calculated with the Euclidean norm

$$||\mathbf{e}||_2 = \sqrt{\sum_{j=1}^{n^2} e_j^2}.$$ 

The error is as inaccessible as the exact solution, so an other metric is needed. The residual also represents how well $\mathbf{v}$ approximates $\mathbf{u}$ and is calculated by

$$\mathbf{r} = \mathbf{f} - A\mathbf{v}.$$ 

It defines the amount by which the approximation $\mathbf{v}$ fails to satisfy the original problem $A\mathbf{u} = \mathbf{f}$. Notice that a small residual does not always necessarily imply a small error. The residual is also an important component for the Multigrid method. If we combine $A\mathbf{u} = \mathbf{f}$ with $\mathbf{r} = \mathbf{f} - A\mathbf{v}$ and $\mathbf{e} = \mathbf{u} - \mathbf{v}$, the relation between the error and the residual yields

$$A\mathbf{e} = \mathbf{r}.$$ 

2.2 Multigrid idea

Relaxation methods like the Jacobi method are useful for reducing the high-frequency part of the error, but they perform poorly on low-frequency parts. The idea of the Multigrid algorithm is, that the smoothed low-frequency part of the error is of higher frequency on a coarser grid. In the following, we use the notation $\Omega^h$ as fine grid and $\Omega^{2h}$ as coarser grid, with $\Omega^{2h}$ having $n/2$ grid points for one dimension, and $(n/2)^2 = n^2/4$ grid points on two dimensions. A smooth error on $\Omega^h$ is of higher frequency on $\Omega^{2h}$, so relaxation methods are more effective on coarser grids. Also, it is cheaper to smooth on a coarser grid, as there are less grid points to be updated.

For relaxing on the error directly, we need the residual equation already presented in subsection 2.1

$$A\mathbf{e} = \mathbf{r} = \mathbf{f} - A\mathbf{v}.$$ 

Notice, that relaxation on the original equation $A\mathbf{u} = \mathbf{f}$ is the same as relaxing on the residual equation $A\mathbf{e} = \mathbf{r}$ with specific initial guess $\mathbf{e} = 0$. This is another good reason to use the residual equation for Multigrid methods.

Putting all parts together results in a procedure called correction scheme:

- Relax on $A\mathbf{u} = \mathbf{f}$ on $\Omega^h$ to obtain an approximation $\mathbf{v}$
- Compute the residual $\mathbf{r} = \mathbf{f} - A\mathbf{v}$
- Relax on the residual equation $A\mathbf{e} = \mathbf{r}$ on $\Omega^{2h}$ to obtain an approximation to the error $\mathbf{e}^{2h}$
- Correct the approximation obtained on $\Omega^h$ with the error estimate obtained on $\Omega^{2h}$:
  $$\mathbf{v}^h \leftarrow \mathbf{v}^h + \mathbf{e}^{2h}$$

Some serious parts of this algorithm are still missing. For instance, the question remains how to convert the calculated residual on the fine grid $\Omega^h$ to the coarse grid $\Omega^{2h}$, and how to convert the calculated error on $\Omega^{2h}$ back to $\Omega^h$.  

10
Let’s start with the second case, getting values from a coarse grid back to a fine grid. This procedure is called prolongation or interpolation, and the simplest but also an effective way is linear interpolation. The linear interpolation operator is denoted $I_{2h}^h$ and is applied as $I_{2h}^h v^{2h} = v^h$ with

$$
v_{2j}^h = v_j^{2h},
$$

$$
v_{2j+1}^h = \frac{1}{2}(v_{2j}^{2h} + v_{2j+1}^{2h}), \quad 0 \leq j \leq \frac{n}{2} - 1
$$

for one dimension. For every even-numbered fine grid point, its value is just taken from the corresponding coarse grid, for the odd fine grid points, the average of the two neighboring points on the coarser grid is taken. The matrix notation of the linear interpolation $I_{2h}^h$ for one dimension and $n = 8$ grid points (which means $n - 1 = 7$ inner points on $\Omega^h$ and $n/2 - 1 = 3$ inner points on $\Omega^{2h}$) is

$$
I_{2h}^h v^{2h} = \frac{1}{2} \begin{bmatrix}
1 & 2 & 1 & 2 \\
1 & 2 & 1 & 2 \\
\end{bmatrix}
\begin{bmatrix}
v_1 \\
v_2 \\
v_3 \\
v_{3j}^{2h}
\end{bmatrix}
= \begin{bmatrix}
v_1 \\
v_2 \\
v_3 \\
v_4 \\
v_5 \\
v_6 \\
v_7 \\
v_{7j}^h
\end{bmatrix}
= v^h.
$$

For two dimensions the prolongation looks similar as for one dimension:

$$
v_{2i,2j}^h = v_{i,j}^{2h},
$$

$$
v_{2i+1,2j}^h = \frac{1}{2}(v_{i,j}^{2h} + v_{i+1,j}^{2h}),
$$

$$
v_{2i,2j+1}^h = \frac{1}{2}(v_{i,j}^{2h} + v_{i,j+1}^{2h}),
$$

$$
v_{2i,2j+1}^h = \frac{1}{4}(v_{i,j}^{2h} + v_{i+1,j}^{2h} + v_{i,j+1}^{2h} + v_{i+1,j+1}^{2h}), \quad 0 \leq i,j \leq \frac{n}{2} - 1.
$$

Again, if the $i$ and $j$ index is even-numbered, the value for the fine grid is just taken from the same position on the coarse grid. For odd numbers, the average of neighbouring points is taken. If one index is odd, two neighbours are taken into account, if both indices are odd-numbered, four neighbours are taken into account.

The linear interpolation works well as long as the interpolated vector is smooth, but values can get wrong for oscillating vectors. Because we are relaxing on the error before interpolating it, the error should not be oscillating but smooth. Therefore linear interpolation is often a good choice for the prolongation operator.

Next, a transformation from a fine grid $\Omega^h$ to a coarse grid $\Omega^{2h}$ is needed, which is called restriction. The simplest restriction is injection, which means that the value of the fine grid is just injected at the same place as on the coarse grid. For $I_{2h}^h v^h = v^{2h}$ this results in

$$
v_j^{2h} = v_{2j}^h.
$$

Because not all information of the fine grid is used within injection restriction, the so called full-weighting restriction often leads to better results. Here the coarse grid point is calculated by considering all neighbouring fine grid points for one dimension as

$$
v_j^{2h} = \frac{1}{4}(v_{2j-1}^h + v_{2j}^h + v_{2j+1}^h), \quad 1 \leq j \leq \frac{n}{2} - 1.
$$

For the matrix notation with $n = 8$ again (so $I_{2h}^h$ transforms $n - 1 = 7$ to $n/2 - 1 = 3$ inner grid
points), we get

$$I_{2h}^h \mathbf{v}^h = \frac{1}{4} \begin{bmatrix} 1 & 2 & 1 & 2 & 1 \\ 1 & 2 & 1 & 2 & 1 \\ \vdots & \vdots & \vdots & \vdots & \vdots \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \\ v_3 \\ v_4 \\ v_5 \\ v_6 \\ v_7 \\ v_8 \\ v_9 \\ v_{10} \end{bmatrix} = \begin{bmatrix} v_1 \\ v_2 \\ v_3 \\ v_4 \\ v_5 \\ v_6 \\ v_7 \\ v_8 \\ v_9 \\ v_{10} \end{bmatrix} = \mathbf{v}^{2h}.$$  

The injection restriction can also outperform the full-weighting restriction for some problems, but the full-weighting restriction has an interesting property, which makes it more useful in this thesis. The full-weighting restriction operator is the transpose of the linear interpolation operator, multiplied with a factor $c$:

$$I_{2h}^h = c (I_{2h}^h)^T.$$  

So the matrix of the linear interpolation operator can be used as restriction matrix as well and vice versa. For one dimension, the factor $c = 2$, for two dimensions $c = 4$. For the full-weighting restriction in two dimensions, the coarse grid points are computed from its nine neighbors, where direct neighbors have more influence than diagonal neighbors:

$$v_{i,j}^{2h} = \frac{1}{16} (v_{i-1,j-1}^h + v_{i-1,j+1}^h + v_{i+1,j-1}^h + v_{i+1,j+1}^h + 2(v_{i-1,j}^h + v_{i+1,j}^h + v_{i,j-1}^h + v_{i,j+1}^h) + + 4v_{i,j}^h), \quad 1 \leq i, j \leq \frac{h}{2} - 1$$

Now that we know how to transfer information from fine grids to coarse grids and vice versa, using the prolongation and restriction operators, we can go back to the correction scheme. The two-grid correction scheme with integrated prolongation and restriction operators now looks like this:

\[ \mathbf{v}^h \leftarrow MG(\mathbf{v}^h, f^h) \]

- Relax $p_1$ times on $A^h u^h = f^h$ on $\Omega^h$ with initial guess $\mathbf{v}^h$.
- Compute residual $r^h = f^h - A^h \mathbf{v}^h$ and restrict it to $\Omega^{2h}$ by $r^{2h} = I_{2h}^h r^h$.
- Solve $A^{2h} e^{2h} = r^{2h}$ on $\Omega^{2h}$.
- Interpolate coarse-grid error to fine grid by $e^h = I_{2h}^h e^{2h}$ and correct fine-grid approximation by $\mathbf{v}^h \leftarrow \mathbf{v}^h + e^h$.
- Relax $p_2$ times on $A^h u^h = f^h$ on $\Omega^h$ with initial guess $\mathbf{v}^h$.

This is the same correction scheme as before, but now the prolongation and restriction operators are included. The number of pre- and postsmoothing steps $p_1$ and $p_2$ is usually 1, 2 or 3. In this thesis, 2 pre- and postsmoothing steps are continuously used. The superscripts $2h$ for $e^{2h}$ represents the error on the coarse grid $\Omega^{2h}$. All components of the algorithm are known, except the matrix $A^{2h}$ on $\Omega^{2h}$. We can set $A^{2h}$ to the same stencil as $A^h$ but with adapted preterm $\frac{1}{h^2} \rightarrow \frac{1}{(2h)^2}$, which is the same as discretizing the problem on $\Omega^{2h}$.

Now the only remaining question is, how to solve the coarse grid problem $A^{2h} e^{2h} = r^{2h}$. The trick is to rename the parts of the problem $A^{2h} e^{2h} = r^{2h}$. We can set the error $e^{2h}$ to $u^{2h}$, as it is just a solution vector, and $r^{2h}$ is called $f^{2h}$, as it is just a right-hand side. Now we have a similar problem as on the fine grid, namely $A^{2h} u^{2h} = f^{2h}$, and we can again apply our correction scheme to solve it. This results in a recursive structure, where the grid gets coarser and coarser, until there is only a small number of grid points left, which can be effectively solved. The full recursive algorithm, called V-Cycle scheme, looks as follows:
\[ v^h \leftarrow V^h(v^h, f^h) \]

1. If \( \Omega^h = \text{coarsest grid} \), Relax \( x \) times on \( A^h u^h = f^h \) with initial guess \( v^h \), then return.
   Else: Go to step 2.

2. Relax \( p_1 \) times on \( A^h u^h = f^h \) with initial guess \( v^h \).

3. Recursive call:
   \[ f^{2h} \leftarrow f^h - A^h v^h, \]
   \[ v^{2h} \leftarrow 0, \]
   \[ v^{2h} \leftarrow V^{2h}(v^{2h}, f^{2h}). \]

4. \[ v^h \leftarrow v^h + I^{2h}_h v^{2h}. \]

5. Relax \( p_2 \) times on \( A^h u^h = f^h \) with initial guess \( v^h \).

The input of the algorithm is the initial guess \( v^h \) and the right-hand side \( f^h \). First, it is checked if the coarsest grid is reached. If so, \( x \) relaxation steps are performed, and the algorithm returns to the next finer grid. Here \( x > 100 \), because it is useful to fully remove the low frequency part of the error, which is most dominant on the coarsest grid. Thereby, a good approximation on the coarsest grid is achieved, and it is cheap to proceed many relaxation steps here as there are very few grid points to be updated. If the coarsest grid is not reached, step 2 is processed, which performs \( p_1 \) relaxation steps on \( A^h u^h = f^h \). Then the residual is calculated, restricted and then set as right-hand side \( f^{2h} \) for the coarser grid. Also, the error is initialized to zero and set to the new initial guess \( v^{2h} \) for the coarser grid. Then the actual recursive call follows, it calls the V-Cycle with the calculated residual as right-hand side and the error as the initial guess. Step 4 does a prolongation of the calculated approximation \( v^{2h} \) to the next finer grid and adds it to the current approximation \( v^h \) on the finer grid. Lastly, \( p_2 \) post-smoothing steps are done.

In Figure 1 on the left, the name origin of the V-Cycle is observable. The V-Cycle algorithm reduces its grid points through restriction down to the coarsest grid, here at \( \Omega^{8h} \), and then increases its grid size through prolongation again. In Figure 1 on the right a W-Cycle is shown. It is attained by proceeding the recursive call two times on every grid level. Although the W-Cycle can have advantages over the V-Cycle, this thesis focuses on the V-Cycle for simplicity.

The V-Cycle needs to be extended to achieve a fully functional Multigrid method. It needs a termination condition when it should stop proceed V-cycles. The simplest one is to just run a fixed number of iterations, but usually, it is more useful to stop at a residual reduction threshold. Therefore, the initial residual of the problem needs to be stored first. Then a while-loop is necessary, in which the new residual at the end of each cycle is calculated. The loop stops when the initial residual divided by the current residual is smaller than a certain threshold, for example \( 10^{-10} \). For the calculation of the magnitude of the residual, the Euclidean norm is used. It can also be useful to implement a residual threshold for the coarsest grid, such that we perform relaxation steps until the initial residual of the coarsest grid divided by the current residual is smaller than a certain threshold. By this, unnecessary relaxation steps can be avoided at the lowest grid level.

Figure 1: V-cycle (left) and W-cycle (right) with four grid levels each.
2.3 Jumping coefficients

Classical Poisson problems like (2) can be extended by a coefficient function. This is useful for modeling diffusion in an inhomogeneous medium, where a very dense part provides very little diffusion. An example is the simulation of an oil reservoir, where the oil provides very little diffusion compared to gas or water. In this thesis, coefficient jumps are used for evaluating the approach of learning prolongation and restriction operators. Linear interpolation and full-weighting restriction are already good operators for the Poisson problem, so a more complex problem like jumping coefficients is needed, where the linear interpolation/full-weighting restriction operators are going to have problems achieving a good convergence. We consider the problem

\[-\nabla \cdot (c \nabla u) = f \tag{6}\]

with \(c(i, j)\) as discontinuous (jumping) coefficient function. \(c(i, j) = 1\) would lead to the classical Poisson equation again. We consider the coefficient function to be defined on the grid points and take care, that no coefficient jump exactly takes place at a grid line. We call a grid line one row or column of the grid matrix.

A coefficient function used in this thesis reads

\[c(i, j) = \begin{cases} 1.0 & \text{for } ih < 0.5 \\ 11.0 & \text{for } ih > 0.5 \end{cases},\]

where one coefficient jump right next to \(ih = 0.5\), but not on the grid line, so the grid line \(ih = 0.5\) is well defined. The new Jacobi relaxation step, including the coefficient function for (6) is

\[v_{i,j}^{(1)} = \frac{1}{4c_{i,j}} (h^2 f_{i,j} + c_{i-1,j} v_{i-1,j} + c_{i,j+1} v_{i,j+1} + c_{i-1,j} v_{i-1,j} + c_{i+1,j} v_{i+1,j}). \tag{7}\]

It can be seen, that the coefficient function \(c_{i,j}\) just gets multiplied to the approximation \(v_{i,j}\) at the same index. Later, also the component notation of the residual equation \(r = f - A v\) with the coefficient function is needed, which is

\[r_{i,j} = f_{i,j} - \frac{1}{h^2} (4c_{i,j} v_{i,j} - c_{i-1,j} v_{i-1,j} - c_{i,j+1} v_{i,j+1} - c_{i-1,j} v_{i-1,j} - c_{i+1,j} v_{i+1,j}). \tag{8}\]

Using classical Multigrid methods for jumping coefficients will lead to worse convergence factors, the larger the jump is. Also multiple jumps lead to poor convergence factors, which is shown later in section 5. A remedy can be the use of operator dependent prolongations and restrictions as shown in [16]. But in this thesis, the prolongation and restriction are going to learn their own operators, so this method is not used.

2.4 Galerkin coarsening

Another idea to improve the convergence factor of a Multigrid solver, that has to deal with jumping coefficients, is Galerkin coarsening. The poor convergence factors of simple Multigrid methods dealing with jumping coefficients can result from an inappropriate discretization of the problem including the coefficient function. Earlier, we set our matrix \(A^{2h}\) for the coarser grid \(\Omega^{2h}\) to the same discretization stencil as for \(\Omega^h\) (2), with only adjusting the pre-factor \(\frac{1}{\tau} \rightarrow \frac{1}{(2h)^k}\). But \(A^{2h}\) can be defined better using the Galerkin coarsening

\[A^{2h} = I^h_{2h} A^h I^h_{2h},\]

which calculates \(A^{2h}\) by multiplying the restriction operator \(I^h_{2h}\) with the matrix \(A^h\) of the fine grid and then multiplying the result with the prolongation operator \(I^h_{2h}\). By this procedure, the coarse grid problem represents the problem on the fine grid more suitable and thus leads to better convergence. This is even more useful if the prolongation operator is the transpose of the restriction operator, so some important properties of \(A^h\) like symmetry or positive definiteness are automatically transmitted to \(A^{2h}\). For linear interpolation and full-weighting restriction, equation (2) yields for

\[A^{2h} = I^h_{2h} A^h I^h_{2h} = \frac{1}{(2h)^2} \begin{bmatrix} -0.25 & -0.5 & -0.25 \\ -0.5 & 3 & -0.5 \\ -0.25 & -0.5 & -0.25 \end{bmatrix}.\]
The Galerkin coarsening performs best in combination with operator dependent prolongation and restriction, as shown in [16]. However, it can also be utilized for our learned Prolongation and restriction operators as we are going to see later.

The performance of our Multigrid solver with the Galerkin coarsening is also going to be discussed in section 5. The disadvantages of the Galerkin coarsening are, that some extra computational time is needed to initialize the matrix $A$ at every level. In addition, the stencil of $A$ gets larger, it has only five entries in $A^h$, but nine entries for coarser grids in two dimensions. This difference is even more significant in three dimensions, where the fine grid $A$ stencil has only seven stencil entries, but coarser grid stencils have 27 entries.

For more information about iterative methods, Multigrid and its extensions see [17] and [16].
3 Machine learning

The Oxford dictionary defines machine learning as "the use and development of computer systems that are able to learn and adapt without following explicit instructions, by using algorithms and statistical models to analyse and draw inferences from patterns in data " [18]. In this chapter, we are looking at the different types as well as the functionality of machine learning. For more information about machine learning, see for example [19] and [20].

3.1 Types of machine learning

Machine learning algorithms can be separated into three different approaches:

The first approach is supervised learning. The main goal of supervised learning is to train a model with labelled training data to make predictions about future data. In the supervised learning approach, the training data always consists of a labelled input/output pair, where the output is also called the supervisory signal. The algorithm learns a general rule for mapping the input to the output through optimization of an objective function. This learning approach is mostly used for classification problems like deciding between a cat and a dog on an input picture or recognizing handwritten letters, but it is also useful for regression and active learning.

Reinforcement learning is about software agents taking actions in order to maximize some notion of cumulative reward. Therefore no input training data is needed in advance, but the agents are training in a suitable simulation scenario by trial-and-error. For positive actions a positive reward is given, for negative actions a negative award. This approach is used for instance for autonomous vehicles or in learning to play games like chess or StarCraft [21].

Unsupervised learning does not work with labelled training data or agents. It requires only the input data to find structures in it like grouping or clustering of data points. The main application of those algorithms is for example density estimation in statistics. [19]

In this work, supervised learning is utilized to find optimal prolongation and restriction operators for a Multigrid solver. Therefore, it is used as an optimization tool. The functionality of supervised learning is explained in the following.

3.2 Functionality of neural networks

To understand how a neural network is working and learning, we first have to understand how its basic element looks like, namely the neuron. The simplest neuron is the perceptron, which contains a certain amount of inputs and one output as shown in Figure 2. Mathematically, the perceptron consists of weights \( w_j \), which are weighting the inputs \( x_j \), and a bias \( b \), which adds a certain value to the summed inputs. The perceptron has a binary output, it can fire, represented by a 1, or can not fire, represented by 0, depending on the inputs, weights and the bias. The equation describing this behaviour is

\[
\text{output} = \begin{cases} 
0 & \text{if } w \cdot x + b \leq 0, \\
1 & \text{if } w \cdot x + b > 0,
\end{cases}
\]  

(9)

where \( w \cdot x \) is the dot product of the weights vector and the input vector, so \( w \cdot x = \sum_j w_j x_j \).

![Figure 2: Perceptron with inputs x1, x2, x3 and one output.](image)

A more complex but also more useful neuron is the Sigmoid neuron. It works similar to the perceptron, with the difference, that the output is not binary, but instead represented by the sigmoid function \( \sigma(z) \), where

\[
\sigma(z) = \frac{1}{1 + e^{-z}}.
\]  

(10)
with \( z = wx + b \). The sigmoid function appears just to be the smoothed version of the step function, which is the function of the perceptron. So for very large values of \( wx + b \), the sigmoid function still results in 1, and for very small values it still results in 0, it only differs for moderate values. This is useful for the training process. In the training the weights as well as the bias are getting adjusted, so for the perceptron a small change in one weighting can result in a whole different output (a switch from 0 to 1 for example). For the Sigmoid neuron on the other hand, if one weight is adjusted, the output only slightly changes, whereby more exact training is possible.

The architecture of an exemplary network is shown in Figure 3, where multiple neurons are combined. It is common to also represent the inputs and outputs as neurons. All layers between input and output layers are called hidden layers. The structure of the network depends on the application it is built for. A neural network classifying handwritten digits, for example, could have one input neuron for each pixel of the digit picture, with the greyscale of each pixel as input value. For the output layer, ten output neurons could be used, one for each digit from zero to nine. The number of hidden layers, as well as the number of neurons in each hidden layer, is harder to determine and one of the challenging parts in creating a neural network.

After the creation of a neural network, we can start with the training. First, a set of training data is needed. For supervised learning, each training data consists of a training input and a desired output. For the example of the handwritten digits, the input would be a picture of a digit and the desired output would be the digit itself. We also need a cost function, which quantifies the effectiveness of the neural network. It does so by comparing the desired output \( y(x) \) with the output \( a \), which is created by inserting the input \( x \) into the network. A common example for a cost function is the quadratic cost function, also called mean squared error,

\[
C(w, b) = \frac{1}{2n} \sum_x ||y(x) - a||^2,
\]

where \( w \) and \( b \) are the weights and biases of the network, \( n \) is the number of training data, \( a \) is the output of the network after inserting the input \( x \), and \( y(x) \) is the desired output of the corresponding input \( x \). The aim of the training process is to minimize the cost function by adjusting the weights and biases of the network. Stochastic gradient descent is a good basic tool to satisfy this task. [20]

![Figure 3: Neural network with one input layer, multiple hidden layers and one output layer.](image)

### 3.2.1 Stochastic gradient descent algorithm

For the explanation of the gradient descent, we leave the neural network context and simplify the cost function to \( C(v) \), where \( v \) could have several dimensions, which is also able to represent our weights \( w \) and biases \( b \). For better imagination we first look at \( C(v) \), that only depends on two variables, \( v_1 \) and \( v_2 \). An exemplary cost function relying on \( v_1 \) and \( v_2 \) is shown in Figure 4, where it is relatively easy to see where the cost function has its global minimum. Nevertheless, it is much harder to find the global minimum if \( C \) would be a function of many more variables. The analytic approach of finding the global minimum would be to calculate the derivatives of \( C \), to find
the extrema. This approach is computationally very expensive for \( C \) depending on many variables, which is common for neural networks, because every neuron has multiple weights for multiple inputs as well as a bias. Therefore the analytic approach is not suitable to find the global minimum of the cost function.

A better approach is to deploy an iterative method, where the goal in every step is to decrease the cost function just a little bit, to finally reach the global minimum. Mathematically, this means to have a negative change of the cost function \( \Delta C \) in every step. The cost function \( C \) depending on only two variables then holds:

\[
\Delta C \approx \frac{\partial C}{\partial v_1} \Delta v_1 + \frac{\partial C}{\partial v_2} \Delta v_2.
\]  

(12)

With the gradient of the cost function as

\[
\nabla C = \begin{pmatrix} \frac{\partial C}{\partial v_1} \\ \frac{\partial C}{\partial v_2} \end{pmatrix}^T
\]

(13)

and the change of variables \( \Delta v = (\Delta v_1, \Delta v_2)^T \), (12) can be generalized to

\[
\Delta C \approx \nabla C \cdot \Delta v.
\]  

(14)

Here the gradient of the cost function \( \nabla C \) relates changes in \( v \) to changes in \( C \). Remember, that it is still our goal to have a negative \( \Delta C \) by choosing the right change of the variables \( \Delta v \). Therefore the equation

\[
\Delta v = -\eta \nabla C
\]  

(15)

is used, where \( \eta \) is a small positive parameter, called the learning rate. Combining (14) with (15) results in

\[
\Delta C \approx -\eta \nabla C \cdot \nabla C = -\eta ||\nabla C||^2,
\]  

(16)

which is very useful, because \( ||\nabla C||^2 \geq 0 \). This guarantees \( \Delta C \leq 0 \), which means that \( C \) is always decreasing, if we change \( v \) according to (15). So our resulting update rule is

\[
v \rightarrow v' = v - \eta \nabla C.
\]  

(17)

By applying this update rule step by step, we get closer and closer to the global minimum of the cost function.

A decisive parameter in the update rule is the learning rate \( \eta \). It needs to be small enough, such that (14) is a good approximation. On the other side, a too small \( \eta \) would lead to tiny changes in \( v \), making our gradient descent algorithm very slow and computationally expensive.
The gradient descent algorithm above is explained for the cost function $C(v)$ depending on two variables, but also works perfectly for more variables, by only changing the gradient of $C$ to

$$\nabla C = \left( \frac{\partial C}{\partial v_1}, \ldots, \frac{\partial C}{\partial v_m} \right)$$

(18)

and the vector $\Delta v$ in (14) to $\Delta v = (v_1, \ldots, v_m)^T$.

So after we have described the functionality of the gradient descent algorithm, we can apply it to our neural network. Remember $w$ and $b$ being the weights and biases of our network, respectively. The variable $v$ in (17) can be replaced by the weights or biases to achieve the update rules of the descent gradient algorithm in terms of components

$$w_k \rightarrow w'_k = w_k - \eta \frac{\partial C}{\partial w_k}$$

(19)

for every $w_k$ in all weights of the network $w$, and

$$b_l \rightarrow b'_l = b_l - \eta \frac{\partial C}{\partial b_l}$$

(20)

for every $b_l$ in all biases $b$.

The "stochastic" in stochastic gradient descent is about decreasing the computational cost of the training. Remember, that the cost function in (11) averages over all costs of the various training inputs $X$. To compute the gradient $\nabla C$, the gradient $\nabla C_x$ has to be computed for every training input $x$. This can be very costly for a large number of training inputs. To avoid this, only a small amount of training inputs is chosen at random. These few training inputs are also called a mini-batch. For all inputs in the mini-batch, the gradient $\nabla C_x$ is computed and averaged. By doing so, the true gradient $\nabla C$ is approximated well enough and the computational costs are significantly lower. Afterwards, a new mini-batch is randomly selected and trained with, until the whole training data is exhausted.

3.2.2 Backpropagation

In the last chapter, we have discussed how the stochastic gradient descent algorithm adjusts the weights and biases of our neural network in a way that reduces the cost function depending on the input training data. But we still miss how to compute the gradient of the cost function $\nabla C$. For this approach Backpropagation is used. Backpropagation is about understanding how changing the weights and biases in a network changes the cost function. This is achieved by finding $\nabla C$, which can be written in component form as the derivatives $\frac{\partial C}{\partial w}$. As described before, the total gradient of the cost function $\nabla C$ is not computed as one, but the gradient of the cost function $\nabla C_x$ for several training inputs is calculated and then averaged. But how to compute $\nabla C_x$ for one training input?

First some notation basics are needed. We can express the weighted input $z = wx + b$ in detailed component form as

$$z^l_j = \sum_k w_{jk}^l a^{l-1}_k + b^l_j$$

(21)

To explain the indices, $l$ represents the current layer, $j$ the neuron in layer $l$ and $k$ the neuron of the layer before. The notation $w_{jk}^l$ represents the weight of neuron $k$ of the previous layer $l-1$ to the neuron $j$ of the current layer $l$. $a^{l-1}_k$ is the activation of the neuron $k$ of the previous layer, and thus is the input of the actual layer $l$. The activation $a^l$ can be expressed as

$$a^l = \sigma(z^l) = \sigma(w^l a^{l-1} + b^l),$$

(22)

where $\sigma$ is an activation function as in (10).

For the Backpropagation, first an intermediate quantity $\delta^l_j$ is introduced, which is called the error of neuron $j$ in layer $l$. We define the error $\delta^l_j$ as

$$\delta^l_j = \frac{\partial C}{\partial z^l_j}.$$ 

(23)
There are four equations needed for Backpropagation. The first equation is used to compute the error \( \delta^L_j \) of the last/output layer \( L \) by

\[
\delta^L_j = \frac{\partial C}{\partial a^L_j} \sigma'(z^L_j). \tag{24}
\]

The first term \( \frac{\partial C}{\partial a^L_j} \) measures the influence of an output neuron \( a^L_j \) on the cost function \( C \). The second term \( \sigma'(z^L_j) \) measures how rapidly the activation function \( \sigma \) is changing at \( z^L_j \). The matrix form of (24) is

\[
\delta^L = \nabla_a C \odot \sigma'(z^L), \tag{25}
\]

which is more suitable for the actual Backpropagation algorithm later. The \( \odot \) represents an element-wise vector multiplication, also called the Hadamard product.

The next equation needed for Backpropagation is an expression for the error \( \delta^l \), which depends on the error of the next layer \( \delta^{l+1} \). In particular

\[
\delta^l = ((w^{l+1})^T \delta^{l+1}) \odot \sigma'(z^l). \tag{26}
\]

Suppose we know the error \( \delta^{l+1} \). By multiplying it with the transposed weights \( (w^{l+1})^T \) of layer \( l + 1 \), we move the error backwards through our network, to achieve a measure for the error at the output of layer \( l \). If we then take the Hadamard product \( \odot \sigma'(z^l) \), the error is moved backwards through the activation function, leading us to the error \( \delta^l \). By combining (25) and (26), the error can be computed at every layer, starting with the output layer \( L \), and moving the error backwards through the network.

Next, we need an equation for the rate of change of the cost with respect to any bias in the network, which is given by

\[
\frac{\partial C}{\partial b^l_j} = \delta^l_j. \tag{27}
\]

So the error \( \delta^l_j \) is exactly the rate of change \( \frac{\partial C}{\partial b^l_j} \). Finally we need an equation for the rate of change of the cost in respect to any weight in the network, which is given by

\[
\frac{\partial C}{\partial w^l_{jk}} = a^{l-1}_k \delta^l_j. \tag{28}
\]

So (27) and (28) deliver exactly what was needed at the start of this chapter, namely a solution for calculating the gradient of the cost function \( \nabla C = (\frac{\partial C}{\partial b^l_j}, \frac{\partial C}{\partial w^l_{jk}})^T \).

To summarize, the Backpropagation algorithm looks like the following:

1. **Input**: Set the corresponding activation \( a^1 \) for input layer.
2. **Feedforward**: For each \( l = 2, ..., L \) compute \( z^l = w^l a^{l-1} + b^l \) and \( a^l = \sigma(z^l) \).
3. **Output error** \( \delta^L \): Compute the vector \( \delta^L = \nabla_a C \odot \sigma'(z^L) \).
4. **Backpropagate the error**: For each \( l = L-1, L-2, ..., 2 \) compute \( \delta^l = ((w^{l+1})^T \delta^{l+1}) \odot \sigma'(z^l) \)
5. **Output**: The gradient of the cost is given by \( \frac{\partial C}{\partial b^l_j} = \delta^l_j \) and \( \frac{\partial C}{\partial w^l_{jk}} = a^{l-1}_k \delta^l_j \).

The steps always start or end at \( l = 2 \), because \( l = 1 \) is the input layer, where it is not useful to calculate the error, as no weights can be modified here.

Lastly, we can combine stochastic gradient descent with the Backpropagation algorithm to write down the actual process to adjust the weights and biases of our network according to the gradient of the cost function:
1. Input a set of training examples.

2. For each training example $x$: Set corresponding input activation $a^{x,1}$ and perform the following steps:

   - Feedforward: For each $l = 2, 3, ..., L$ compute $z^{x,l} = w^l a^{x,l-1} + b^l$ and $a^{x,l} = \sigma(z^{x,l})$.
   - Output error $\delta^{x,L}$: Compute the vector $\delta^{x,L} = \nabla_C a_x \odot \sigma'(z^{x,L})$.
   - Backpropagate the error: For each $l = L - 1, L - 2, ..., 2$ compute $\delta^{x,l} = (\delta^{x,l+1} T (w^{l+1}) \odot \sigma'(z^{x,l}))$.

3. Gradient descent: For each $l = L, L - 1, ..., 2$ update weights according to the rule $w^l \rightarrow w^l - \frac{\eta m}{m} \sum_x \delta^{x,l} (a^{x,l-1})^T$, and biases to $b^l \rightarrow b^l - \frac{\eta}{m} \sum_x \delta^{x,l}$.

Here $m$ is the number of training inputs in one mini-batch. As mentioned before, the gradient of the cost function is not calculated for every training input, but only for a random subset and then averaged to approximate the total gradient. Therefore (20) is modified in the algorithm, so it sums over all $\frac{\partial C_x}{\partial b^l_j} = \delta^{x,l}_j$ in a mini-batch and then averages them by $\frac{1}{m}$. The same holds for the weights in (19). Missing in the algorithm is, of course, one outer loop for generating mini-batches, and another for iterating over multiple training epochs, to make the training procedure complete.

### 3.2.3 Types of neural network layers

To build a neural network suitable for a specific problem, one has to choose from various types of neural layers.

The convolutional layer is utilized to only weight information in the neighbourhood of the current data. This is achieved by using one trainable filter/kernel for all input data. The filter could be of size $3 \times 3$ for instance, taking only the direct neighbours of the current neuron into account. The convolutional layer is often used for feature detection, for example to find edges in image recognition applications. Nevertheless, the kernel also reminds us of the stencils used in section 2. Therefore, the convolutional layer is utilized to enable the Jacobi relaxation as well as the residual equation for our Multigrid solver in this work.

Another typical layer in neural networks is the pooling layer. It is used to reduce the information of the input data, by reducing the amount of pixels for instance. A pooling layer reduces the number of pixels by merging a certain amount of neighbouring pixels into one pixel. For max-pooling, the highest pixel value defines the new pixel value. For mean-pooling, the average of this set of pixels is taken. At first, the pooling layer seems like a good possibility to implement our restriction as neural layer, but as it is not able to learn, it is not used in this work.

The last layer presented is the fully-connected layer. Here every neuron of the previous layer is connected to every neuron of the actual layer. The fully-connected layer is normally used to reduce the amount of neurons to a specific number of output neurons. In this work, the approach of a fully-connected layer is used to represent the prolongation as well as the restriction of our Multigrid solver. This is useful, because we want one prolongation/restriction stencil for every grid point. The exact implementation of these layers is shown in section 4.

By combining some of the presented layers, a feedforward neural network (FNN) is created. Data in a feedforward neural network always propagates in one direction, which means that nodes cannot form a cycle. The opposite is the recurrent neural network (RNN), where cycles are allowed. Because of these cycles, the RNN has a internal state, which can be seen as memory. This is useful for handwriting or speech recognition tasks. Nevertheless, to optimize the prolongation and restriction weights of a Multigrid solver, a feedforward neural network is used in the following.

### 3.3 Machine learning in Tensorflow

In this work, Tensorflow is used for machine learning, which takes care of the learning procedure, including stochastic gradient descent and the backpropagation process.
Tensorflow [22] is an open-source library for numerical computations and machine learning. It was developed by the Google Brain team in 2015, first only for Google internal usage. Tensorflow APIs are available in several programming languages, including C++, Java, JavaScript, Go and Swift, but the Python API is the most commonly used one. Nevertheless, the entire Tensorflow source code is executed in high-performance C++ or CUDA, if the code runs on Nvidia graphic processing units. Thereby, the comparatively slow programming language Python has no influence in the performance of Tensorflow operations.

Tensorflow applications are able to run on most devices such as local machines, clusters and mobile devices.

Tensorflow 2.0 was released in 2019 and has several extensions to his predecessor. A big advantage is the high-level Keras API, which makes creation and training of Tensorflow models even simpler. Another advantage is the eager execution, which makes the debugging of Tensorflow models a lot easier, because each graph node can be evaluated separately instead of building the whole graph. Both of the extensions were heavily used to develop the neural network for this work.

There are some other machine learning frameworks, such as PyTorch [23], the Microsoft Cognitive Toolkit(CNTK) [24] or Apache MCNet by Amazon [25], but Tensorflow 2.0 is used for the machine learning tasks in this work.
4 Implementation of the Multigrid method in Tensorflow

The idea of this thesis is about implementing a fully functional Multigrid V-Cycle in a machine learning framework, similar to [5] and [4]. In this work we use Tensorflow as machine learning framework. The developed neural network represents a V-Cycle consisting of multiple layers, where every layer corresponds to a Multigrid operation, such as the relaxation step, the residual computation, or the prolongation and restriction. But writing those operations in Tensorflow is not trivial, because data in the different layers always gets processed as matrices, so simple index accessing is not possible. In this section, the implementation of all important operations of a Multigrid V-Cycle in Tensorflow is described.

The mathematical correctness of the Tensorflow V-Cycle is tested by comparing predicted results of the V-Cycle in Tensorflow with the Exastencils framework [26]. Further, a simple Multigrid solver written in Python has been developed for double-checking the results of the Multigrid Tensorflow solver.

4.1 Jacobi relaxation

One of the main components of a Multigrid V-Cycle is the smoothing step. In this work, we use a damped Jacobi smoother, as described in subsection 2.1. In Figure 5, the dataflow graph of the damped Jacobi in Tensorflow is shown. The inputs of the Jacobi relaxation is the right-hand side \( f \), the coefficient matrix \( c \), the current approximation of the exact solution \( v_0 \) and the geometry matrix \( g \). The geometry matrix \( g \) looks like a bitmap, marking the inner points of the grid with ones and the boundary points with zeros.

In contrast to section 3, where a network was consisting of layers of neurons, in this section the layers represent whole input matrices as inputs layers and mathematical operations such as multiplications or additions as hidden layers. We are working with a higher-level view of our neural network, which we call dataflow graph. In the following, Tensorflow is also used as a math library. Only a few layers in our network are capable of being trained, but they work with neurons as described in section 3.

Following the path of \( v^0 \) to \( v^1 \) in Figure 5 results in

\[
v^1 = (((conv2d(c \cdot v^0) + (h^2 \cdot f))/4 \cdot c) \cdot g + (1 - g) \cdot v^0) \cdot w + (1 - w) \cdot v^0.
\]

where

\[
conv2D - stencil = \begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix}, \tag{30}
\]
In the following the similarity of (29) and the basic equation of the Jacobi relaxation extended by the coefficient function in (7) is shown, focusing on the Jacobi step part of (29). Recall (7) as
\[
v_{i,j}^{(1)} = \frac{1}{4c_{i,j}} (h^2 f_{i,j} + c_{i,j-1}v_{i,j-1}^{(0)} + c_{i,j+1}v_{i,j+1}^{(0)} + c_{i-1,j}v_{i-1,j}^{(0)} + c_{i+1,j}v_{i+1,j}^{(0)}).
\]
In Figure 5, first the current approximation \(v^0\) and the coefficient matrix \(c\) is multiplied. Then they are sent through a 2D convolutional layer with the filter stencil shown in (30). Thereby, the four nearest neighbours of the current point are added and the term \((c_{i,j-1}v_{i,j-1}^{(0)} + c_{i,j+1}v_{i,j+1}^{(0)} + c_{i-1,j}v_{i-1,j}^{(0)} + c_{i+1,j}v_{i+1,j}^{(0)})\) from (7) is achieved. Utilizing a convolutional layer for summing up the neighbouring points is a good workaround for not being able to access indices manually in Tensorflow. The next step is to add the right-hand side \(f\) multiplied with the quadratic step width \(h^2\) to the previous term. Now only the term \(\frac{1}{c_{i,j}}\) is missing, which is achieved by dividing the previous term by 4e. It also makes sense to compare the Jacobi step part of (29) with the Jacobi step formulated in matrix notation in (5). To see the similarity it first has to be reformulated to
\[
u = D^{-1}((L + U)u + f).
\]
Remembering that \(D, U\) and \(L\) are the diagonal, upper and lower parts of the matrix \(A\), respectively. Considering the coefficient function as well as the stencil of (2) included in the parts of \(A\) and \(h^2\) included in \(f\), the similarity to the Jacobi step part of (29) gets clear.

The part fix boundaries from (29) is needed, because the filter of the 2D convolutional layer (30) iterates over every point, including the boundary points. Instead of throwing an error when convolving over the left boundary, since there exists no point left to the left boundary, it assumes the point - which does not exist - to be zero. This behaviour can be modified by changing the argument padding from the Tensorflow layer Conv2D from same to valid, but then the boundaries of the input matrix of the convolutional layer would always get cut off and the matrix would get smaller for every smoothing step, because no values for the boundaries are calculated. For the padding argument same, which means that the output size stays the same as the input size in the convolution process, the boundary values only get wrong, and the fix boundaries part is there to fix them.

```python
# Kernel initialization for Conv2D layer
def kernel_init(shape, dtype=None):
    k = tf.constant([
        [0, 1.0, 0],
        [1.0, 0, 1.0],
        [0, 1.0, 0],
    ], dtype=tf.float64)
    kernel = tf.reshape(k, [3, 3, 1, 1])
    return kernel

# Conv2D layer creation with kernel init and padding argument same
self.conv2d = tensorflow.keras.layers.Conv2D(filters=1, kernel_size=(3, 3),
                                          kernel_initializer='kernel_init', padding='same', trainable=False, dtype='float64')
```

Code Listing 1: Initialization of the Tensorflow Conv2D layer for Jacobi relaxation, with focus on the stencil initialization and the padding argument.

The fix boundaries part of (29) fixes the boundaries by first multiplying the current approximation \(v^0\) with the inverse geometry matrix \(g\). \(g\) is a matrix with the same size as the current approximation \(v^0\), but it only consists of ones for points not at the boundary, and zeros for points at the boundary, like a bitmap. The inverse \(inv\) layer here means \((1 - g)\), so the boundary points are ones, and all inner points are zeros. So by element-wise multiplying the current approximation \(v^0\) with the inverse geometry \(inv(g)\), only the boundary values of \(v^0\) are retained, while the inner points are set to zero. Next, the original geometry matrix is element-wise multiplied with the solution matrix of the Jacobi step to get only the inner points of the new solution. Last the boundary and the inner points are added together, to get the real solution matrix with fixed boundaries.

The last part is the weighting/damping part of the Jacobi method. The new solution with fixed boundaries is multiplied with a damping factor \(w\). Meanwhile, the old solution \(v^0\) is multiplied by \((1 - w)\). Lastly, both are added together to achieve the final new solution \(v^1\).

The Jacobi relaxation is finally implemented, expressed by Tensorflow layers. Here, Tensorflow is used as math library, because no layer is capable of learning weights or biases.
4.2 Residual equation

The next step is to represent the residual equation as Tensorflow layers. Again, the input matrices are the coefficients $c$, the current approximation $v^0$, the right-hand side $f$ and the geometry $g$.

Figure 6: Dataflow graph of the residual equation in Tensorflow with $r$ as resulting residual, $v^0$ as current approximation, $f$ as right-hand side, $c$ as coefficient matrix, $g$ as geometry bitmap, $h$ as step width and $conv2d$ is a 2D convolution with the stencil shown in (33).

Following the path from the inputs to $r$ leads to

$$r = (f - \frac{1}{h^2} \cdot conv2d(c \cdot v^0)) \cdot g$$

with

$$convolution2D\_stencil = \begin{bmatrix} 0 & -1 & 0 \\ -1 & 4 & -1 \\ 0 & -1 & 0 \end{bmatrix}. \quad (33)$$

Again the similarity to the basic residual equation with coefficient function in (8) is noticeable. Remember (8) as

$$r_{i,j} = f_{i,j} - \frac{1}{h^2}(4c_{i,j}v_{i,j} - c_{i-1,j}v_{i-1,j} - c_{i+1,j}v_{i+1,j} - c_{i,j-1}v_{i,j-1} - c_{i,j+1}v_{i,j+1}). \quad (34)$$

Similar to in the Jacobi step, the coefficient matrix $c$ is multiplied with the current approximation $v^0$ first. Afterwards the result is again sent through a 2D convolutional layer, but this time the filter stencil is equal to the discretization stencil of matrix $A$ (see (33)). Multiplying the result by $\frac{1}{h^2}$ leads to the rear part of (8). The result is subtracted from the right-hand side and lastly the geometry is multiplied to set the boundaries to zero, which again have been modified by the convolutional layer.

4.3 Implementation of the Galerkin coarsening in Tensorflow

To implement the Galerkin coarsening in our V-Cycle, as it is described in subsection 2.4, some changes have to be done to the Jacobi layer as well as to the residual equation layer. Starting with the Jacobi layer, it can be seen in Figure 7, that instead of having a coefficient matrix as input as in Figure 5, here the whole discretization matrix $A$ is given. That is the case, because $A$ is built before the initialization, and already contains the coefficients as well as the discretization stencil.

Following the path in Figure 7 from the current approximation $v^0$ to the resulting approximation $v^1$ leads to the equation

$$v^1 = (\frac{(f - (A \cdot v^0 - \text{diag}(A \cdot v^0))/\text{diag}(A))}{\text{Diag}(A)} \cdot g + (1 - g) \cdot v^0) \cdot w + (1 - w) \cdot v^0. \quad (35)$$

This is the result of the Jacobi step, followed by the Galerkin step, fixing the boundaries, and finally weighted Jacobi.
Figure 7: Dataflow graph of the Jacobi step with Galerkin coarsening in Tensorflow with $v^1$ as resulting approximation, $v^0$ as current approximation, $f$ as right-hand side, $A$ as iteration matrix, $g$ as geometry bitmap, $w$ is the weighting factor for weighted jacobi, operation $\text{inv}$ means $(1 - g)$ here and $\text{diag}$ extracts only the diagonal of the matrix $A$.

Again the goal is to reach similarity to (7) to assess whether the Tensorflow Galerkin Jacobi layer works correctly. So first the current approximation $v^0$ is multiplied with $A$ containing the discretization stencil and the coefficients. To exclude the middle points, the result is subtracted with the diagonal of $A \cdot v^0$. Then the current term is subtracted from the right-hand side $F$. Lastly, the result is divided by the diagonal of $A$, what corresponds to the front term $\frac{1}{\sigma_{u_{v'}}}$ of (7). If we look at (31) the similarity gets even more clear, as $A \cdot v^0 - \text{diag}(A \cdot v^0)$ corresponds to $(L + U)u$.

The part of fixing the boundaries and the weighting stays the same as for the Jacobi layer without the Galerkin coarsening. Although no convolutional layer is used, the fix-boundary part is still needed, because the multiplication of $v^0$ and $A$ also takes place at the boundaries, which modifies them unintentionally.

Figure 8: Dataflow graph of the residual equation in Tensorflow with $r$ as resulting residual, $v^0$ as current approximation, $f$ as right-hand side, $A$ as iteration matrix and $g$ as binary geometry matrix.

For the residual equation, the changes are similar. From Figure 8 we obtain the equation

$$r = (f - A \cdot v^0) \cdot g.$$  

(36)

Again the coefficient matrix as well as the discretization stencil is included in $A$. By multiplying the current approximation $v^0$ with $A$, the rear part of (8) is already achieved. So the result only has to be subtracted from the right-hand side $f$ and then multiplied by the geometry $g$ to fix the boundaries.
4.4 Prolongation and restriction

The remaining parts for obtaining a fully functional Multigrid V-Cycle are the prolongation and restriction layers. These layers are more straightforward than the Jacobi relaxation layer for instance. This is because reducing the size of a grid from $n \times n$ to $\frac{n^2}{4} \times \frac{n^2}{4}$ in the case of the restriction is just about multiplying the $n \times n$ grid with a matrix of size $(n^2 \times \frac{n^2}{4})$.

![Dataflow graph of the restriction layer](image)

Figure 9: Dataflow graph of the restriction layer, with $v^h$ as current approximation, $v^{2h}$ as approximation on the next lower grid, $R$ as restriction weights, $cut_b$ cuts all boundary points of the matrix, and $add_b$ adds zero boundaries to the matrix.

The dataflow graph of the restriction layer is shown in Figure 9, the full Tensorflow code of the restriction is presented in Code Listing 2. First, the restriction matrix has to be created and initialized with the full-weighting restriction operators known from section 2. The creation is not part of the computation layer but has to be done before. Due to the size of the restriction, it is not useful to save it as a dense matrix. A restriction matrix is a sparse matrix with size $(n^2 \times \frac{n^2}{4})$, but for the dimension of size $n^2$, only 9 entries for the full-weighting restriction operators are needed. So instead of storing $(n^2 \times \frac{n^2}{4})$ values, the SparseTensor Tensorflow library can be used for storing only the $(9 \times \frac{n^2}{4})$ needed entries.

These sparse tensors consist of an array of indices, an array of values and the shape of the sparse matrix. The problem of these Tensorflow sparse tensors is, that there is no option to mark these objects as trainable. A workaround is used, which stores the three parts of a sparse tensor separately, marking the values array as trainable and create the real sparse tensor object only when it is needed for mathematical operations (see Code Listing 2). This way, the values of the sparse tensors are recognized by the Backpropagation process and can be trained.

```plaintext
values = tf.Variable(initial_value=values, trainable=True, dtype='float64')
```

So after the creation of the restriction matrix, the actual layer can be described. The inputs are the three parts of the restriction sparse tensor and the input matrix, which information should be transferred to a coarser grid. First, the parts of the sparse tensor are compounded to a sparse tensor object. Then the boundaries of the input matrix are cut off, as only the inner points are getting restricted. The cutoff of the boundaries – the layer is called $cut_b$ in Figure 9 – can be achieved by a 2D convolutional layer with a $3 \times 3$ Filter consisting of zeros, excluding the middle point, which is 1. Further, the padding argument has to be valid, so the filter has no effect on the inner values, but cuts off the boundaries, as described before in subsection 4.1. Next, the input matrix is reshaped from two dimensions to one, so it fits the restriction matrix. The actual restriction is only a matrix-vector multiplication of the shaped input matrix with the restriction sparse tensor. Lastly, the outcome is reshaped to two dimensions, and its boundaries are added again through the Tensorflow concat method.

The restriction sparse tensor is also used as prolongation matrix, it only has to be transposed, and the values have to get multiplied by 4. So the prolongation layer looks exactly the same as the restriction layer, besides the inverse dimension sizes of the sparse tensor.

For the initialization of the prolongation and restriction layer, linear operators, namely the linear interpolation and the full-weighting restriction are used, so the Tensorflow V-Cycle can also be used to solve problems without training, similar to a Multigrid solver written in Python. The more significant advantage is that the initial operators are a good starting point for learning optimal prolongation/restriction operators for various problems, so training needs less time/training steps to find them, compared to prolongation/restriction operators starting with a zero or random initialization.
class Restriction(Layer):
    def __init__(self, **kwargs):
        super(Restriction, self).__init__(**kwargs)

    def build(self, input_shape):
        self.size = input_shape[0][1]

        # initialize helper layers
        self.cut_b = Cut_Boundaries()
        self.add_b = Add_Boundaries()

        super(Restriction, self).build(input_shape)

    def call(self, x):
        size = self.size
        u = x[0]
        indices, values, shapes = x[1]

        # build sparse tensor
        R_sparse = tf.sparse.SparseTensor(indices, values, shapes)
        R_sparse = tf.sparse.reorder(R_sparse)

        # cut boundaries
        u = self.cut_b([u])
        size = size - 2
        u = tf.reshape(u, [1, size * size])

        # actual restriction
        u = tf.sparse.sparse_dense_matmul(u, R_sparse)
        size = size // 2
        u = tf.reshape(u, [1, size, size])

        # add boundaries
        u = self.add_b([u])
        size = size + 2
        u = tf.reshape(u, [1, size, size, 1])
        return u

Code Listing 2: Full restriction layer Tensorflow code.

4.5 Prediction and training

For the actual V-Cycle, all layers described before - first decide for one type of Jacobi relaxation and residual equation, with or without the Galerkin coarsening - can just be put together to obtain a functional V-Cycle, as shown in Figure 10. It must be considered, that features like stopping the coarse grid solver after a certain residual reduction threshold are not possible in this Tensorflow V-Cycle. It is not allowed to implement if-conditions in Tensorflow models, which are not definite at the initialization of the model. That is because Tensorflow creates dataflow graphs before the actual calculation, and no extra layer can be added or deleted on the fly. So for this example, the exact amount of iteration steps for coarse grid solving has to be set initially.

So for solving and learning with the V-Cycle, Tensorflow models need to be created. In this thesis, two models are created, one for solving and one for training.

4.5.1 The solve model

The solve model initializes all layers needed for each Multigrid level. The solve model call function, later used to predict the results, has as inputs only the current approximation $v^0$ and the right-hand side $f$. Then it calls the created V-Cycle and returns the new solution $v^1$. So the solve model performs only one step of V-Cycle in Tensorflow by calling the predict function of the solve model. The corresponding pseudo code for the solve model is shown in Code Listing 3. To obtain a functional Multigrid solver, the missing things like initializing the approximation and right-hand side or the loop over the V-Cycle and the termination at a residual reduction threshold are implemented in Python, not in the Tensorflow API.
class Solve_model:
    def init():
        for all levels of Multigrid:
            initialize Jacobi relaxation layer
            initialize residual equation layer
            initialize prolongation layer
            initialize restriction layer
    def call(inputs v,f):
        v = VCycle(v,f)
        return v

Code Listing 3: Pseudocode for the solve model.

4.5.2 The train model
The train model proceeds the same initialization as the solve model. The difference is in the call
function. The inputs are also the current approximation and the right-hand side. Further, it also
performs exactly one V-Cycle step. The difference is, instead of returning the new solution, it
calculates the residual of the new solution and returns it.

class Train_model:
    def init():
        ...
    def call(inputs v,f):
        v = VCycle(v,f)
        r = calculate_residual(v,f)
        return r

Code Listing 4: Pseudocode for the train model.

For the actual training, the train model has to be initialized first (see Code Listing 5). Then it
has to be compiled with an optimizer and a loss function. In this thesis, the Adam [27] optimizer
is used, which is based on the stochastic gradient descent method explained in subsubsection 3.2.1.
The quadratic cost function, also called mean-squared-error, is used as the loss function. Next, a
target output is created. Because our train model returns the residual, we define the target output
as a zero matrix, which represents the residual, that has to be minimized. After these preparation
steps, the actual training is done by calling the fitting function of the train model. Its inputs are
the initial solution, the right-hand side and the target output. The fitting function first predicts the
output with the given inputs by calling the created V-Cycle and then returning the residual of the
resulting approximation v. Then it calculates the loss function of the predicted output (residual)
and the target output (zero residual), and then adjusts the weights of the prolongation/restriction
layers depending on the cost/loss function, as explained in section 3. Because we want multiple
training steps, the initial solution is initialized randomly in every iteration, while the right-hand
side and the target output stays the same.

After training, the trained model is saved, and the learned operators of the prolongation and
restriction can be transferred in the solve model. The solve model with the learned operators is
evaluated on a problem with a new random initial solution. Then it can be evaluated by comparing
the learned operators with linear ones. The metrics are the required V-Cycles for reaching a certain
threshold and the average convergence factor.

#initialize Models
calc_model = Calc_Model()
train_model = Train_Model()

#compile train_model
opt = optimizers.Adam(learning_rate = 0.001)
loss = tf.keras.losses.MeanSquaredError()
train_model.compile(optimizer = opt, loss = loss, metrics=['accuracy'])

#initialize inputs and target output
u,f = init()
target_res = tf.convert_to_tensor(np.zeros((1,N+2,N+2,1)),dtype=tf.float64)
Code Listing 5: Python Code of the initialization, training and running of the train and solve model.

```python
# actual training
for i in range(TRAIN_SET):
    train_model.fit(x=[u,f], y=target_res, epochs=1, use_multiprocessing=True)
    u = random_init_u()

# set weights of train_model to solve_model
weights = train_model.get_weights()
calc_model.set_weights(weights)

# run V-Cycle with learned prolongation and restriction
cycles, average_conv_rate = calc(calc_model)
```

Figure 10: Dataflow graph of the V-Cycle in Tensorflow with output $v^1$ as resulting approximation and inputs $v^0$ as current approximation, $f$ as right-hand side, $c$ as coefficient matrix and $g$ as geometry bitmap.
5 Learned prolongation and restriction operators

In the previous section, the train model, as well as the solve model, are described. We developed a fully functional Multigrid solver implemented as a neural network, where every operation of a classic Multigrid solver corresponds to a neural layer. Therefore the trainable Multigrid solver implemented in Tensorflow is called deep Multigrid (DMG) from now on. Everything is prepared for the first training run, but there are training parameters influencing the training results.

The first one is the size of the training set, which indicates, over how much iterations we learn. If the used training set is chosen too small, the model would not be finished finding the perfect optimized operators, but greater training sets, of course, increase computation time. So the goal is to find the training set size, where the model has finished learning, and the loss is close to previous training steps. More about training set sizes is going to be discussed in subsubsection 5.2.1.

The next parameter is the learning rate. It regulates the adjustment of the operators in the stochastic gradient descent method, as already explained in subsubsection 3.2.1. A small learning rate needs a greater training set to reach the global minimum of the loss function, while a larger learning rate can jump over minima and even oscillate around them. The influence of various learning rates in practice is shown in subsubsection 5.2.2.

The last parameter is the number of learning iterations. It indicates, how much cycles are performed in one forward step - which is the prediction of the models output - before the residual of the predicted output is calculated and the weights of the models are adjusted through Backpropagation. One learning iteration specifies, that exactly one V-Cycle is predicted in each training step, for multiple learning iterations, the result of multiple V-Cycles is predicted. The impact of the learning iteration number is going to be discussed in subsubsection 5.2.3.

5.1 Evaluation of the learned prolongation and restriction operators

For the comparison of our deep Multigrid, some test cases are needed. They are called configurations in this thesis, and this chapter is about six different Multigrid configurations to get a wide range of application cases. All configurations have some parameters in common. They all have a grid size of $31 \times 31$ inner points, as we only look at 2D examples. They all have the initial solution initialized – in every training step and in the final solving – with a Gaussian distribution with centre 0.0 and a deviation of 0.5. Further, they all use two pre- and post-smoothing Jacobi steps with a damping factor of 0.6. Lastly, all configurations proceed 300 Jacobi steps for solving on the coarsest grid, and they all stop solving at a reduction of $10^{-10}$ of the initial residual.

All configurations:

- Grid size = $31 \times 31$
- Initial solution $v_0$ = Gaussian distribution (center = 0.0, deviation = 0.5)
- Jacobi damping $\omega$ = 0.6
- Pre-/Postsmoothing steps $p_0, p_1$ = 2
- Coarse grid smoothing steps = 300
- Target residual reduction = $10^{-10}$

In this section, the deep Multigrid is always trained and evaluated for exact one right-hand side, one boundary condition and one coefficient function. The only input, which is changing during the training process is the initial approximation $v_0$, which is a new random approximation in every training step. In subsection 5.4 it is going to be discussed, how the prolongation and restriction operators, which are learned for one right-hand side, boundary and coefficient function, perform for other right-hand side, boundaries and coefficient functions.

The first problem configuration is a very simple example with only two grids (maximum depth = 1), a right-hand side of 0.0 and Dirichlet boundaries are all set to 1.0. As it is the main part of this thesis to handle jumps in the coefficient function by learning new prolongation and restriction

31
operators, this simple configuration is useful to focus on the coefficient jump and see, how the deep Multigrid solver can handle it. The coefficient function in configuration 1 is

\[ c = \begin{cases} 
1.0 & \text{for } x \leq 0.5 \\
11.0 & \text{for } x > 0.5 
\end{cases} \]

so there is a coefficient jump right after \( x = 0.5 \), splitting the left and the right side of the domain (see Figure 11a). The jump is chosen so that the untrained Multigrid solver is not converging any more.

**Configuration 1:**
- Maximum depth = 1
- Right-hand side \( f = 0.0 \)
- Boundaries \( b = 1.0 \)
- Coefficient function \( c = \begin{cases} 
1.0 & \text{for } x \leq 0.5 \\
11.0 & \text{for } x > 0.5 
\end{cases} \)

<table>
<thead>
<tr>
<th>Solver</th>
<th>Iterations</th>
<th>( \varnothing ) Convergence factor</th>
</tr>
</thead>
<tbody>
<tr>
<td>GMG</td>
<td>( \infty )</td>
<td>1.041 ( \pm ) 0.001</td>
</tr>
<tr>
<td>GMG + Galerkin</td>
<td>&gt;50</td>
<td>0.709 ( \pm ) 0.001</td>
</tr>
<tr>
<td>AMG</td>
<td>&gt;50</td>
<td>0.564 ( \pm ) 0.004</td>
</tr>
<tr>
<td>Trained DMG</td>
<td>16</td>
<td>0.246 ( \pm ) 0.016</td>
</tr>
</tbody>
</table>

Table 1: Results for Configuration 1, trained with training set of 2000, learning rate of 0.001 and learning iterations of 1.

The deep geometric Multigrid (DMG) is compared to the untrained Multigrid as well as to the untrained Multigrid with implemented Galerkin coarsening and to an algebraic Multigrid solver (AMG). Algebraic Multigrid solvers are developed for solving PDE problems without having an underlying grid, or if grid locations are unknown or highly unstructured. However, it is also useful for handling problems with jumping coefficients, so it makes sense to also compare our deep Multigrid results to the AMG. For more information about AMG see [28] and [29]. For better differentiation from AMG, the Multigrid presented in section 2 is called geometric Multigrid (GMG).

For the GMG as well as the GMG + Galerkin, the Tensorflow implementations are chosen to compare their results with the DMG, as they are fully functional and mathematical correct Multigrid solvers. Their prolongation and restriction layers are set to linear interpolation and full-weighting restriction, respectively. For the algebraic Multigrid solver, the Python library PyAMG [30] is used. More precisely the adaptive smoothed aggregation solver is used, as it performs best for jumping coefficient functions, compared to the classical AMG provided by PyAMG. For better comparison to our GMG and DMG, the AMG solver is modified to also use the Jacobi method with a damping factor of 0.6 for the two pre- and post-smoothing steps as well as the 300 Jacobi steps for the coarse grid solving.

The metrics we use for the comparison of our deep Multigrid solver are the iterations needed for reaching the target residual reduction and the average convergence factor

\[ \varnothing \text{Convergence factor} = \frac{1}{N} \sum_{i=1}^{N} \frac{|r_i|}{|r_{i-1}|} \]

Here \( N \) is the number of iterations and \( r_i \) is the residual of the current iteration. The convergence factor is calculated by the residual of the current step divided by the residual of the last step, so it represents the residual reduction. A lower convergence factor means better convergence.
The average convergence factor shown in the following tables is the mean of multiple runs ± the empirical standard deviation

$$s = \sqrt{\frac{1}{N-1} \sum_{i=1}^{N} (x_i - \bar{x})^2}. \tag{37}$$

If we look at Table 1, we can see that the GMG is diverging for configuration 1. The coefficient jump of 1 for $x \leq 0.5$ and 11 for $x > 0.5$ is chosen the way that the GMG is diverging, with an average convergence factor close to one. The GMG + Galerkin can handle the coefficient jump better, because the Galerkin coarsening can depict the problem better on the coarse grid. Nevertheless, GMG with the Galerkin coarsening is still not optimal. It is not able to reduce the residual to the target threshold $10^{-10}$ in less than 50 iterations. The same applies to the AMG, but it has an even better convergence factor. The deep Multigrid, however, performs best with only 16 iterations to solve and an average convergence factor of 0.246. So for this simple example, the DMG found very good prolongation and restriction operators to outperform the GMG + Galerkin as well as the AMG.

The standard deviation in the GMGs comes from the changing of the initial solution over multiple runs. It is mostly less than 0.005. The greater deviation of the DMG, on the other hand, results from multiple training runs, with every training run has 2000 training iteration, which means 2000 random initial solutions, leading to a more significant deviation. In addition, training is a stochastic process, as it is described in subsubsection 3.2.1, therefore a comparison of the deviation of the unlearned solvers and the DMG is not perfect. Nevertheless it is important to show the stability of the DMG in form of the standard deviation.

Configuration 2:

- Maximum depth = 2
- Right-hand side $f = 0.0$
- Boundaries $b = 1.0$
- Coefficient function $c = \begin{cases} 1.0 & \text{for } x \leq 0.5 \\ 11.0 & \text{for } x > 0.5 \end{cases}$

<table>
<thead>
<tr>
<th>Solver</th>
<th>Iterations</th>
<th>∅</th>
<th>Convergence factor</th>
</tr>
</thead>
<tbody>
<tr>
<td>GMG</td>
<td>∞</td>
<td>2.815 ± 0.001</td>
<td></td>
</tr>
<tr>
<td>GMG + Galerkin</td>
<td>∞</td>
<td>1.261 ± 0.001</td>
<td></td>
</tr>
<tr>
<td>AMG</td>
<td>&gt;50</td>
<td>0.644 ± 0.001</td>
<td></td>
</tr>
<tr>
<td>Trained DMG</td>
<td>&gt;50</td>
<td>0.775 ± 0.009</td>
<td></td>
</tr>
</tbody>
</table>

Table 2: Results for Configuration 2, trained with training set of 2000, learning rate of 0.001 and learning iterations of 1.

The next configuration is similar to configuration 1, as it only differs in the number of grids. For configuration 2, the maximum depth is two, so the problem is solved on three grids. As the problem is not as well defined on lower grids for the GMG, it performs badly with an average convergence factor of 2.815, far from converging (see Table 2). The GMG + Galerkin has a greater convergence, compared to the GMG, again the problem is defined better on coarser grids because of the Galerkin coarsening, but it is also not converging. The AMG shows the best performance here. It seems to have no problems with more than one coarser grid. The trained DMG also manages to converge, but with a worse rate than the AMG but still a great improvement compared to the geometric Multigrid. A possible reason for the DMG not performing as well as for configuration 1 could be, that there are now two prolongation and restriction layers instead of one, so there are more weights influencing the cost function, making the learning process more complex. In Figure 13b it is observable, that the loss is still decreasing after 2000 training steps, so more training could improve the convergence factor of configuration 2.
Configuration 3:
- Maximum depth = 1
- Right-hand side $f = 0.0$
- Boundaries $b = 1.0$
- Coefficient function $c = \begin{cases} 
  1.0 & \text{for } x < 0.5 \text{ and } y < 0.5 \\
  10.0 & \text{for } x < 0.5 \text{ and } y > 0.5 \\
  1.0 & \text{for } x > 0.5 \text{ and } y > 0.5 \\
  10.0 & \text{for } x > 0.5 \text{ and } y < 0.5 
\end{cases}$

<table>
<thead>
<tr>
<th>Solver</th>
<th>Iterations</th>
<th>$\varnothing$Convergence factor</th>
</tr>
</thead>
<tbody>
<tr>
<td>GMG</td>
<td>$\infty$</td>
<td>$1.070 \pm 0.002$</td>
</tr>
<tr>
<td>GMG + Galerkin</td>
<td>29</td>
<td>$0.456 \pm 0.001$</td>
</tr>
<tr>
<td>AMG</td>
<td>$&gt;50$</td>
<td>$0.564 \pm 0.005$</td>
</tr>
<tr>
<td>Trained DMG</td>
<td>18</td>
<td>$0.285 \pm 0.020$</td>
</tr>
</tbody>
</table>

Table 3: Results for Configuration 3, trained with training set of 2000, learning rate of 0.001 and learning iterations of 1.

For configuration 3 we go back to the two grid method, which means one fine grid and one coarse grid. The right-hand side is still zero and boundaries are set to one, but a more complex coefficient function is chosen. It consists of four areas, with area top-left set to 1.0, top-right is 10.0, bottom-left is 10.0 and bottom-right is 1.0 (see Figure 11b), so the Multigrid methods need to handle four jumps. This coefficient function is a common example for coefficient jumps, already presented in [31]. A similar coefficient function but with four different values for the four areas is presented in [16].

If we look at Table 3, the geometric Multigrid is again diverging for configuration 3, with a convergence factor close to one. The geometric Multigrid extended by the Galerkin coarsening performs much better with a convergence factor of 0.456±0.001 and needs only 29 iterations to reach the residual reduction threshold. Interestingly the AMG performs worse than the GMG + Galerkin here. For all our tests the algebraic Multigrid never performs better than a convergence factor of 0.55, on the other hand, it converges for every problem, with a convergence factor always between 0.55 and 0.65, so it seems to be the most stable Multigrid method presented here. The deep Multigrid however highly outperforms the AMG as well as the GMG with and without Galerkin with only 18 iterations needed and an average convergence factor of 0.285±0.020. The DMG seems to be able to learn very efficient prolongation and restriction operators for simple right-hand sides and boundaries, so it can focus on the coefficient function in the training process. This is not the case for more complex right-hand sides and boundaries, as we are going to see in the following.

Configuration 4:
- Maximum depth = 1
- $\kappa = 10.0$
- Right-hand side $f = 2\kappa \cdot ((x - x^2) + (y - y^2))$
- Boundaries $b = 1.0 - e^{\kappa(x-x^2)}(y-y^2)$
- Coefficient function $c = e^{\kappa(x-x^2)}(y-y^2)$
Table 4: Results for Configuration 4, trained with training set of 2000, learning rate of 0.001 and learning iterations of 1.

Configuration 4 introduces a right-hand side and boundaries not equal zero and one, respectively, and it has a smooth coefficient function without jumps. The coefficient function, as well as the right-hand side and the boundaries, are presented in [32]. Because of the smooth coefficient function, even the geometric Multigrid solver has no problem to solve this configuration in 14 iterations. As the Galerkin coarsening provides no real advantages for smooth coefficient functions, it has a quite similar convergence factor as the GMG and needs the same amount of iterations. The algebraic Multigrid solver is located in its stable but moderate convergence factor of $0.555 \pm 0.003$, and thus performs worst for this configuration. The trained DMG here performs a bit worse than the GMG with a convergence factor of $0.297 \pm 0.014$. It seems like the initial operators for the prolongation and restriction, namely linear interpolation and full-weighting restriction are already nearly optimal for this smooth coefficient function, and the training process does not manage to find better operators, but instead makes the DMG having worse convergence than for linear operators. This behaviour can also be seen in Figure 12d, where no decrease of the loss can be recognized.

Table 5: Results for Configuration 5, trained with training set of 2000, learning rate of 0.001 and learning iterations of 1.

Configuration 5 combines the right-hand side and the boundaries of configuration 4 with the coefficient function of configuration 3. It should show how the training process performs for right-hand sides and boundaries not equal zero or one, respectively, compared to the same coefficient function with simple $f$ and $b$ (configuration 3). The GMG has a quite similar convergence factor as for configuration 3. The same holds for the GMG+Galerkin and the AMG, as their convergence depends more on the coefficient function than on the right-hand side or the boundaries. Interestingly this is not true for the trained DMG. Its average convergence factor is much worse than for configuration 3, it decreases from $0.285 \pm 0.020$ for configuration 3 to $0.621 \pm 0.062$ for configuration 5, only because of the right-hand side and boundaries not equal zero and one, respectively. Therefore it still
performs better than the GMG, but worse than the GMG+Galerkin and the AMG. The solution for this problem is shown in autorefsec:generalization, where a better convergence factor for configuration 5 is obtained, when the DMG is trained with the simple right-hand side and boundaries of configuration 3. Nevertheless, more research is needed to understand why configuration 5 does not manage to learn good operators just caused by the differing right-hand side and boundaries.

Configuration 6:

- Maximum depth = 1
- Right-hand side \( f = 0.0 \)
- Boundaries \( b = 1.0 \)
- Coefficient function \( c = \begin{cases} 1.0 & \text{for } 0 \leq x \leq 0.25 \text{ or } 0.5 < x \leq 0.75 \\ 10.0 & \text{for } 0.25 < x \leq 0.5 \text{ or } 0.75 < x \leq 1.0 \end{cases} \)

<table>
<thead>
<tr>
<th>Solver</th>
<th>Iterations</th>
<th>∅Convergence factor</th>
</tr>
</thead>
<tbody>
<tr>
<td>GMG</td>
<td>&gt;50</td>
<td>0.933 ± 0.000</td>
</tr>
<tr>
<td>GMG + Galerkin</td>
<td>&gt;50</td>
<td>0.721 ± 0.001</td>
</tr>
<tr>
<td>AMG</td>
<td>&gt;50</td>
<td>0.612 ± 0.002</td>
</tr>
<tr>
<td>Trained DMG</td>
<td>17</td>
<td>0.273 ± 0.015</td>
</tr>
</tbody>
</table>

Table 6: Results for Configuration 6, trained with training set of 2000, learning rate of 0.001 and learning iterations of 1.

For configuration 6 the right-hand side, as well as the boundaries, are chosen simple again, and the coefficient function consists of four areas equally distributed in x-direction, as shown in Figure 11c. For this problem the GMG is converging, but with the unsatisfactory convergence factor of 0.933 ± 0.000. The GMG+Galerkin performs slightly better, with a convergence factor of 0.721 ± 0.001, but still far away from a good convergence. The AMG stays in its stable convergence factor region with 0.612 ± 0.002. The trained DMG, however, performs very well with an average convergence factor of 0.273 ± 0.015 and only 17 iterations needed. Because the DMG is allowed to train for simple right-hand sides and boundaries, it reaches similar performance as for configuration 1 and 3.

To summarize, the learned prolongation and restriction operators of the deep Multigrid solver perform very well for various coefficient jumps. It outperforms the geometric Multigrid solver extended by the Galerkin coarsening as well as the algebraic Multigrid solver, if it is allowed to train for simple right-hand sides and boundaries. For smooth coefficient functions, the initial prolongation and restriction operators already seem nearly optimal, so the learned operators can not outperform them.

Figure 11: Coefficient function for configuration 1 and 2 (a), for configuration 3 and 5 (b), and for configuration 6 (c).
5.2 Influence of varying learning parameters

After the evaluation of the learned prolongation and restriction operators for multiple Multigrid configurations, now the influence of various learning parameters for the convergence factor is discussed.

5.2.1 Training set size

First, we look at the training set size. As mentioned before the training set size represents the number of iterations the training proceeds. In every training iteration, a new initial approximation is created and then used as input. The other input is the constant right-hand side. The used target output is $r = 0$. For every new training step, the model should get closer to its perfect prolongation and restriction operators. If the training set is too small, the training process may have not finished yet, but greater sizes, of course, need more computational time and can even lead to overfitting.

In Figure 12, the losses over the training steps are plotted for all configurations in subsection 5.1. As we remember, the loss represents the discrepancy between the prediction of the trained model based on the input and the associated target output. As loss/cost function, we use the mean-squared-error as shown in (11).

First, the focus is on configuration 1 in Figure 12a, where it can be seen that the loss is decreasing rapidly over the first 100 training steps. In the region of 100 to 500 training steps, the loss is still decreasing, but with a significantly slower speed. After step 500 there is not much more improvement of the loss noticeable, as it can be shown in the zoomed plot of Figure 13a. There the majority of the losses is placed between a value of $0.3 \cdot 10^6$ and $0.45 \cdot 10^6$. For all configurations, the scattering of the loss values result from the random initial solution for each new training run. The learned prolongation/restriction operators perform better for some random initial solutions, and worse for others, so there is always a small scattering of the loss values.

The loss plot for configuration 2 looks different to configuration 1. The loss is not decreasing as fast as for configuration 1. Further, as it can be observed in Figure 13b, the loss is still decreasing after 500 training steps, and even after 1000+ steps. It seems that more than 2000 training steps could have been useful for configuration 2 to obtain a better convergence. After 2000 training steps, the majority of the losses takes place between $0.35 \cdot 10^6$ and $0.5 \cdot 10^6$.

The plots of configuration 3 and 6 look similar to that of configuration 1. Here the loss is also rapidly decreasing in the first 500 training steps, and after 1000 steps, no more significant decreasing is observable. The majority of losses for configuration 3 and 6 after 1000 steps take place between $0.3 \cdot 10^6$ and $0.4 \cdot 10^6$. Therefore, configuration 1, 3 and 5 having a good convergence factor of smaller 0.3 in common.

The loss over the training steps for configuration 4, however, has nothing in common with the losses described before. In Figure 12d it can be seen directly, that the loss is already on a very low level. That is caused by the smooth coefficient function of configuration 4, which can be handled well by the linear operator, which are the initialization of the prolongation and restriction operators before the training. The training itself has problems in learning better operators than the linear ones, so no decreasing of the loss is observable. The convergence factor is even slightly worse after training compared to linear operators in the GMG, so the training has a negative effect on the operators. The majority of losses after the training lies between $0.009 \cdot 10^6$ and $0.0012 \cdot 10^6$, which is much lower than for configuration 1, 3 or 6, while the convergence factor is quite similar. Therefore it can be said, that the convergence factor cannot be derived directly from the loss.

Lastly, we look at the loss of configuration 5 in Figure 12e and Figure 13e. Interestingly, the loss after the first training step is at $\approx 1.0 \cdot 10^6$ instead of $\approx 10.0 \cdot 10^6$ for configuration 1, 2, 3 and 6. Further, the majority of losses after 2000 training steps take place in between $0.25 \cdot 10^6$ and $0.35 \cdot 10^6$, which is slightly lower than the losses of configuration 1.3 and 6, although configuration 5 has a significantly worse convergence factor of $0.621 \pm 0.062$. Again the convergence factor cannot be derived directly from the loss.
Figure 12: Plots of losses over the training steps, all configurations trained with training set of 2000, learning rate of 0.001 and learning iterations of 1.
(a) Loss over train-steps for configuration 1.  (b) Loss over train-steps for configuration 2.

(c) Loss over train-steps for configuration 3.  (d) Loss over train-steps for configuration 4.

(e) Loss over train-steps for configuration 5.  (f) Loss over train-steps for configuration 6.

Figure 13: Plots of zoomed losses over the trainings steps 1000 to 2000, all configurations trained with training set of 2000, learning rate of 0.001 and learning iterations of 1.

5.2.2 Learning rate

The next learning parameter is the learning rate. It regulates the adjustment of the operators in the training process, as shown in subsubsection 3.2.1. A small learning rate needs a greater training set to reach its minimum, while a larger learning rate can jump over minima and even oscillate around them. So it is important to find a good middle value for our problem.

In Figure 14 various learning rates from 0.0001 to 0.004 are tested for configuration 1. They are evaluated by their average convergence factor. The error bars are the standard deviation of the average convergence factor over multiple runs. For a learning rate smaller than 0.0004, the
convergence factor looks very badly, the training process does not seem to find good operators yet. The learning rate is just too small, or the training set size is too small, because when the training set size is great enough, even the lowest learning rate should lead to good operators. In Figure 14, 1000 training steps are processed for all learning rates. If we look at the learning rates greater than 0.003, the average convergence factor appears poor again. Here it might be a problem of jumping over the minima. Also, if we look at the error bars, they are much higher for learning rates > 0.003. A reason for this could be the oscillation around the optimum operators as mentioned before. So the perfect learning rate for configuration 1 is in between 0.001 and 0.003. All values in this interval yield almost the same average convergence rate, so the exact value does not matter too much. It can be said that the learning rate of 0.001, which is the default learning rate for the Adam optimizer, is suitable for configuration 1.

It is not that simple for other problems, such as for configuration 5. In Figure 15 a less smooth plot is presented. Again the learning rates from 0.0001 to 0.004 are tested, but this time the learning rate of 0.001 is not the best choice. Here the best choice seems to be 0.0006 as it comes with the lowest convergence factor. The error bars for all the learning rates are a lot bigger than in Figure 14, that is because configuration 5 has to learn with right-hand side and boundaries not equal zero and one, respectively, and so the learning process struggles a lot more finding optimal prolongation and restriction operators. It is also interesting to notice that the error bars are greater for learning rates of 0.001 to 0.019 than for 0.002 to 0.004. In addition, the horseshoe-effect is less present for configuration 5, which means that even the lowest learning rate of 0.0001 and the highest learning rate 0.004 are performing moderately with convergence factors < 0.7 in contrast to configuration 1 where the lowest learning rate had a convergence factor of \( \approx 1.4 \) and the highest learning rate a convergence factor of \( \approx 0.8 \).

For better comparison, all configurations in subsection 5.1 are trained with a learning rate of 0.001, but to achieve the lowest possible convergence factor, a perfect learning rate for every configuration has to be found.

![Figure 14: Average convergence factors for various learning rates for configuration 1 with training size 1000 and 1 learn iteration, standard deviation as error bars.](image-url)
5.2.3 Multiple learning iterations

The last learning parameter we look at is the number of learning iterations, which the train model does in every training step. For one learning iteration, the train model predicts the output for one V-Cycle and also backpropagates through one V-Cycle. For multiple learning iterations, the train model performs multiple V-Cycles before calculating the loss function from the predicted output and the target output and then also backpropagates multiple times through the same V-Cycle in one training iteration.

The problem of one learning iteration is that only the first step of the resulting deep Multigrid solver has a highly optimized convergence factor. This behaviour can be seen in Table 7 for configuration 1. Here the first V-Cycle has a superior convergence factor, because the learned Multigrid solver is perfectly trained for the first step. For the next V-Cycles the convergence factor is still good, much better then an untrained Multigrid solver would perform for this problem, but it gets slightly worse the more V-Cycles are performed.

A different behaviour occurs for two learning iterations in Table 7, here the convergence factor is optimized for the second V-Cycle, which is logical if the model is trained for two learning iterations. But more interestingly, the convergence factor is good for all V-Cycles which are a multiple of two. The steps which are not a multiple of two, however, have a particularly bad convergence of > 4.0 excluding the first step.

For three learning iterations, every third V-Cycle iteration performs well, but here also the other steps perform moderately, much better than for two learning rates. The same holds for five learning iterations in Table 7, for four learning iterations the behaviour is comparable to that of two learning iterations.

In Figure 16 the average convergence factor of the presented numbers of learning iterations is plotted. Here it can be observed that one learning iterations still performs best. Also well performing are the odd numbers 3 and 5, for even numbers however the average convergence factor is very bad.

The perfect scenario would be, if it were possible to learn the train model for various learning iterations, for example in a range of one to ten. By this, the convergence should be optimized for every V-Cycle iteration, and the learned Multigrid solver could have an even better average convergence factor. This is challenging form a technical point of view, as the train model is created once before the training and can not be modified during the training process. So the number of V-Cycles is fixed through the whole training process. To be able to learn for different numbers of learning iterations, for each new number of learning iterations, the operators of the learned prolongation and restriction would have to be saved first. Then the train model would have to be created anew, loading the saved operators. Presumably, it would be more useful to not make all training steps for one learning iterations and then all steps for the next learning iteration, but changing them more frequently, so that the train model does not fit its operators for one number of learning iterations first and then for the other, but learning them all at once. Using this procedure, the model would have to be created even more often, which would result in a huge computational time, because the graph creation of the train model consumes a great amount of the training time.
as it is going to be presented later in subsubsection 5.7.1.

Nevertheless, it is reasonable to use only one learning iteration, because the input of every learning step is a random initial approximation. This random approximation can represent the approximation for the first Cycle $v^0$ of the learned Multigrid solver, as well as the approximation for the second ($v^1$) or third Cycle ($v^2$). Thereby the train model does not know for which step it is trained, and the convergence factor of V-Cycle iterations after the first is still good. Perhaps the rate is not as good as the first step because the model is trained for non-smooth input, but after the first V-Cycle the approximation is usually smooth.

<table>
<thead>
<tr>
<th>Cycle</th>
<th>Learn iterations</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.037 0.153 0.145 0.135 0.150</td>
</tr>
<tr>
<td>2</td>
<td>0.186 0.083 0.264 0.214 0.179</td>
</tr>
<tr>
<td>3</td>
<td>0.215 4.697 0.117 1.251 1.380</td>
</tr>
<tr>
<td>4</td>
<td>0.248 0.074 1.649 0.095 0.585</td>
</tr>
<tr>
<td>5</td>
<td>0.262 5.686 0.319 4.172 0.173</td>
</tr>
<tr>
<td>6</td>
<td>0.274 0.077 0.132 0.079 1.997</td>
</tr>
<tr>
<td>7</td>
<td>0.280 5.433 1.334 4.774 0.213</td>
</tr>
<tr>
<td>8</td>
<td>0.286 0.079 0.357 0.074 1.168</td>
</tr>
<tr>
<td>9</td>
<td>0.292 5.258 0.259 5.157 0.617</td>
</tr>
<tr>
<td>10</td>
<td>0.299 0.085 0.820 0.095 0.177</td>
</tr>
<tr>
<td>11</td>
<td>... ... ... ... ...</td>
</tr>
</tbody>
</table>

Table 7: Convergence factor for every cycle for different numbers of learning iterations for configuration 1, trained with training set size of 1000 and learning rate of 0.001.

Figure 16: Average convergence factor over various numbers of learn iterations for configuration 1 with training set size 1000 and learning rate 0.001, standard deviation as error bars.

5.3 Analysis of learned prolongation and restriction operators

We measured the performance of the learned prolongation and restriction operators of our deep Multigrid solver and discussed the influence of different learning parameters, but how do these learned operators look like? In Figure 17 we can see the nine-point stencil weights of the restriction for configuration 1. The top-left picture, for example, represents all top-left stencil weights for the restriction of the entire grid. For the prolongation, the operators just have to be multiplied by 4. The colour bars are chosen in order to have the initial operators, here full-weighting restriction, as reference value, to be able to highlight deviations in the upper and lower direction. Therefore, the reference value (green color) for the middle stencil point is 0.25, for direct neighbours 0.125 and for diagonal neighbours 0.06125. So for yellow colored points, the training adjusted operators to
a higher weighting compared to the initial linear operators, for blue points with lower weighting, and green colors indicate weightings close to the linear operators. The boundary points are zero, as only inner points are getting restricted.

Note that the colour/value at one point represents the weighting of one neighbouring grid point. Which neighbouring point to weight depends on the stencil position. So one colour/value of a point in the right stencil points refers to the weighting of its right neighbour.

Figure 17: Nine point stencil for learned restriction operators for Configuration 1, the top-left picture for example represents the top-left stencil weights for the whole grid.

The first highlight, that can be noticed in the restriction operators of configuration 1 in Figure 17, are the points next to the boundaries. It seems to be an advantage for the convergence to give points close to the boundary more weighting than full-weighting does. This holds for every point of the stencil, as every stencil point weights its points close to the boundary higher in its stencil direction. For example, the top-left points have highlights on their upper and left points next to the boundary.

The more deciding conspicuousness, of course, is the big dark-blue line for the top-right, right, and bottom-right stencil points. This line indicates the weights there to be close to zero. Remembering, that our coefficient function for configuration 1 was

\[
    c = \begin{cases} 
        1.0 & \text{for } x \leq 0.5 \\
        11.0 & \text{for } x > 0.5 
    \end{cases},
\]

so the coefficient value on the grid line \(x = 0.5\) is still one, but the value on the next right grid line is then eleven. So the dark blue line comes from the grid points next to the coefficient jump, where the coefficient on the line is still one, but the coefficient function on the next right grid line is eleven. Therefore, the training tries not to weight the right neighbouring points there. So a blue line appears, indicating the weights there close to zero. This behaviour is only noticeable on the right points of the stencil, as only they are in contact with the right neighbouring point and thus to the coefficient jump. Further, small irregularities can be found in all stencil point colour maps close to the coefficient jump, but they are much less clear than for the right points.
It is also noticeable, that the points on the low coefficient side are generally a bit darker (smaller) than on the high coefficient side, and especially for the diagonal stencil points the points on the high coefficient side are looking less smooth.

Figure 18: Nine point stencil for learned restriction operators for Configuration 3.

In Figure 18, the restriction stencil for configuration 3 is shown. As the coefficient function of configuration 3 is more complex, more highlights can be found in the stencil colour maps. First, the points close to the boundaries are noticeable. Again, the learning process decided to let them points close to the boundary have more influence in the restriction.

More interesting is the behaviour at the coefficient jumps. Remembering, that the coefficient function was divided into four areas, the bottom-left and the top-right area had the value one, and the bottom-right and top-left area were ten, resulting in four coefficient jumps (see Figure 11b). If we focus on the top stencil point first, a horizontal blue line across half the domain appears. Here we have the same case as for configuration 1, but now in y-direction. The small coefficient values in the bottom-left area close to the upper coefficient jump do not weight the other side of the jump with coefficient ten at all, so their operators there are close to zero. The same holds for the right stencil points, this time in x-direction. The top-right stencil point just combines the top and the right stencil point.

The left stencil points show a less expected behaviour, here a yellow line occurs, representing a higher weighting of the neighbouring points. It splits the top-right area with the low coefficient and the top-left area with the high coefficient. The yellow line is still on the low coefficient area, so here the left border points of the low coefficient area are weighting the other side of the jump with coefficient ten higher than other points. This behaviour holds for every coefficient jump from a low coefficient area to a high one, but only in the left and bottom direction. This is counter-intuitive, as for the right and the top direction the low coefficient area close to a jump always weights the other side of the jump very low. It is not clear why it seems to be an advantage for minimizing the residual to let the other side of the jump have nearly no influence in the restriction for two directions (top and right), but let it have influence above average for the other two directions (bottom and left).
It is also interesting, that there is much less emphasis at the jump from a high coefficient area to a low one, the learning process seems to pay much more attention to the jump direction low to high. The middle points of the stencil, however, stays nearly unchanged, only if we look more closely, we can see that the points at the high coefficient areas are a bit higher than the low coefficient areas.

It is also worth mentioning, that in contrast to full-weighting, the learned restriction operators for configuration 1 and 3 are not normalized, meaning that the nine stencil points do not sum to one any more. This is especially true for points close to the boundary as well as for points at the coefficient jumps.

5.4 Generalization of learned prolongation and restriction operators

In this chapter, the generalization of the trained prolongation and restriction operators is evaluated. Our deep Multigrid solver is always trained for exactly one right-hand side, one boundary and one coefficient function in subsection 5.1. The only training input changing in the training process is the initial approximation of the solution \(v_0\), which is a new random approximation for every training step. Thereby, the deep Multigrid solver provides good convergence for all possible initial approximations.

Interestingly it is more useful to learn for random initial approximation than for a constant approximation, even if the constant approximation is the only test data afterwards. For example, if the train model for configuration 1 is only learned for \(v_0 = 0\), and also executed with \(v_0 = 0\), it only results in an average convergence factor of 0.566 ± 0.000, while trained for random initial approximation and executed with \(v_0 = 0\) results in convergence factor of 0.246 ± 0.016. This seems counterintuitive first, but if we learn for \(v_0 = 0\), only the first iteration step has superior convergence factor (here 0.007), all other steps have a convergence factor > 0.5. This is again because the train model never saw other input approximations then \(v_0 = 0\), but after the first execute step, the approximation differs from zero. Therefore it is useful to learn with random approximation.

But how do the prolongation and restriction operators, which are trained for only one setup, perform for other right-hand sides, other boundaries and other coefficient functions?

5.4.1 Different right-hand sides

First, we take a look at the right-hand sides (RHS). In Table 8 the DMG is trained for configuration 1 with right-hand side \(f = 0\), but executed for different RHSs. All other properties of configuration 1 stay the same, only the RHS differs. It can be observed, that the deep Multigrid has its best convergence for the RHS it learned on, but the convergence for RHSs it did not learn on is also good with a convergence factor of < 0.3, which is much better than the convergence factor of the untrained solver, which is > 1.0 for all RHSs.

In Table 9 the DMG is trained for configuration 5 with RHS \(f = 2\kappa \cdot ((x - x^2) + (y - y^2))\) and again executed for different RHSs it did not learn on. Here it can be seen, that the bad learned prolongation and restriction operators, which result in a convergence factor of 0.621 ± 0.062 for configuration 5 also perform badly for other right-hand sides. Interestingly, the worst convergence factor is obtained for the RHS the model trained with, but also the other RHSs result in convergence factors > 0.5.

So to summarize, it can be said, that, if learned prolongation and restriction operators perform well for the RHS they learned on, they also perform well for other right-hand sides. At the same time, operators, which perform poorly for one RHS, also perform poorly for others.

Interestingly, to optimize the operators for configuration 5, it is more useful to train with the setup of configuration 3, as it has the same coefficient function, but other RHS and boundaries, than for configuration 5 itself. As mentioned before, the training process seems to have problems in training on RHS and boundaries not equal zero and one, respectively, although they have only a small amount of influence on the evaluated performance. So if we train for configuration 5 and execute for configuration 5, we get a convergence factor of 0.621 ± 0.062. But if the model is trained for configuration 3 and executed on configuration 5, the convergence factor highly improves to 0.283 ± 0.019. So it seems to be always a good idea to learn on the problem with the simple RHS, as long as the coefficient function stays the same.
5.4.2 Different boundaries

For the boundaries, the same procedure is done as for RHSs. The DMG is learned for one specific boundary value and then tested on other boundaries, while other properties like the RHS or the coefficient function stays the same. In Table 10, the DMG is again trained for configuration 1 with boundary \( b = 1.0 \). Similar to the right-hand sides the DMG performs best for the learned boundaries and only slightly worse for other boundaries with a convergence factor always \( > 0.3 \), while the convergence factor of the geometric Multigrid solver is \( \approx 1.0 \) for all boundaries.

Again, the learned operators for configuration 5, shown in Table 11, perform as bad as for the various RHSs of configuration 5, with a convergence factor always above \( 0.5 \).

So the same behaviour holds as for the right-hand sides, if good prolongation and restriction operators are learned, they perform well for all boundaries and right-hand sides, as long as the coefficient function stays the same. Again it is more useful to learn the operators for configuration 5 by training with the more simple RHS and boundaries of configuration 3. It can be said that the learned operators in the evaluation are mostly focusing on performing well for a specific coefficient function, while the right-hand side, as well as the boundaries, does not really come into effect. On the other hand, the right-hand side and boundaries do seem to have an influence on the success of the training process, as the poor convergence of configuration 5 showed us.

Table 8: Trained for Configuration 1 with RHS = 0, trainsize=2000, learn iterations = 1, train rate = 0.001, \( \kappa = 10 \).

<table>
<thead>
<tr>
<th>right-hand side ( f )</th>
<th>( \varnothing )Convergence factor</th>
<th>GMG</th>
<th>DMG</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.0</td>
<td>1.041 ± 0.001</td>
<td>0.246 ± 0.016</td>
<td></td>
</tr>
<tr>
<td>1.0</td>
<td>1.040 ± 0.001</td>
<td>0.281 ± 0.014</td>
<td></td>
</tr>
<tr>
<td>10.0</td>
<td>1.041 ± 0.001</td>
<td>0.273 ± 0.013</td>
<td></td>
</tr>
<tr>
<td>( \pi^2 \cdot \cos(\pi \cdot x) - 4\pi^2 \cdot \sin(2\pi \cdot y) )</td>
<td>1.040 ± 0.001</td>
<td>0.279 ± 0.012</td>
<td></td>
</tr>
<tr>
<td>( 2\kappa \cdot ((x - x)^2 + (y - y)^2) )</td>
<td>1.041 ± 0.001</td>
<td>0.282 ± 0.006</td>
<td></td>
</tr>
</tbody>
</table>

Table 9: Trained for Configuration 5 with RHS = \( 2\kappa \cdot ((x - x)^2 + (y - y)^2) \), trainsize=2000, learn iterations = 1, train rate = 0.001, \( \kappa = 10 \).

<table>
<thead>
<tr>
<th>right-hand side ( f )</th>
<th>( \varnothing )Convergence factor</th>
<th>GMG</th>
<th>DMG</th>
</tr>
</thead>
<tbody>
<tr>
<td>( 2\kappa \cdot ((x - x)^2 + (y - y)^2) )</td>
<td>1.061 ± 0.005</td>
<td>0.621 ± 0.062</td>
<td></td>
</tr>
<tr>
<td>0.0</td>
<td>1.059 ± 0.005</td>
<td>0.560 ± 0.018</td>
<td></td>
</tr>
<tr>
<td>1.0</td>
<td>1.051 ± 0.020</td>
<td>0.599 ± 0.054</td>
<td></td>
</tr>
<tr>
<td>10.0</td>
<td>1.060 ± 0.001</td>
<td>0.594 ± 0.037</td>
<td></td>
</tr>
<tr>
<td>( \pi^2 \cdot \cos(\pi \cdot x) - 4\pi^2 \cdot \sin(2\pi \cdot y) )</td>
<td>1.026 ± 0.009</td>
<td>0.555 ± 0.020</td>
<td></td>
</tr>
</tbody>
</table>

Table 10: Trained for Configuration 1 with boundaries = 1.0, trainsize=2000, learn iterations = 1, train rate = 0.001.
Table 11: Trained for Configuration 5 with boundaries $1.0 - e^{\kappa (x-x^2) (y-y^2)}$, $\kappa = 10$, train-size=2000, learn iterations = 1, train rate = 0.001.

5.4.3 Different coefficient functions

Different right-hand sides as well as different boundaries are tested, but what happens if our solver is learned for one coefficient function and executed on another? Again the solver is learned for configuration 1 with coefficient function

$$c = \begin{cases} 
1.0 & \text{for } x \leq 0.5 \\
11.0 & \text{for } x > 0.5 
\end{cases}$$

and executed for different coefficient functions. In Table 12 the convergence factors for various coefficient functions are shown. The first one is the coefficient function already learned on, so the decent convergence factor of $0.262$ is no surprise.

The second function is a coefficient jump at the same position as the learned one, but the jump size of one to two is much smaller than the original jump from one to eleven. Because of the small jump, even the geometric Multigrid solver can handle the problem well with a convergence of $0.209$. The deep Multigrid, on the other hand, performs worse on this small jump with a convergence of $0.606$ – the learned operators do not seem to fit for this small jump.

The next coefficient function again contains one jump at the same position as the original one, but this time the jump is much greater, with one side being one and the other one hundred. The GMG solver can not handle this huge jump, resulting in a convergence factor of $11.108$. The DMG solver, however, performs better compared to the unlearned one, but is still diverging with a convergence of $2.440$.

Next, a jump with the same height as the original one is tested, but this time the jump is in the y-direction, instead of the x-direction. Here the GMG has the same convergence as for the original jump in x-direction. The DMG has a worse convergence of $1.413$. Of course, the learned prolongation and restriction weights are not optimized for a jump in the wrong direction, so the convergence is poor.

The last coefficient function is that of configuration 3 and 5, also shown in Figure 11b. Here the deep Multigrid solver performs slightly worse than the GMG, both with a convergence factor greater than one. One could have expected, that the convergence is better, because at least one of the four jumps is learned by the training – in specific the jump from area bottom-left to bottom-right, because there the jump is the same as in configuration 1 – but this does not seem to have a great impact in the convergence.

In Table 13 the solver is learned for configuration 5, with the coefficient function shown in Figure 11b and executed on other coefficient functions. Overall, there is no real surprise. Excluding the coefficient function, that the solver has learned on, the DMG performs badly for all other coefficient functions, even worse than the geometric Multigrid solver.

Generally, it can be said that it is not useful to learn for one specific coefficient function and then solve other coefficient functions with the same solver. A solution for this problem could be, to define the coefficient function as learning input and to vary it throughout the training process, so that the learned solver is more flexible for various coefficient functions. But it could also make the solver too generalized, being able to handle all coefficient functions, but having only a moderate convergence for all of them. Nevertheless, this subject would lead too far to be discussed further in this thesis.
5.5 Limits to coefficient functions

In subsection 5.1, we saw that the deep Multigrid solver has a mostly good convergence for the presented problems. In this section, its limits are tested, by first maximizing the height of the coefficient jump, and then maximizing the jump frequency.

5.5.1 Maximize coefficient jump height

We start with the height of the coefficient jump. The setup will be the same as in configuration 1, only the coefficient function is modified. The location of the coefficient jump is the same, but the jump height is different. In Figure 19, the convergence factor for various jump heights is presented. In contrast to the previous section, the DMG is trained with the same coefficient function it executed with. Take care of the non-linearity of the x-axis of the graph, as the jump size increases with a step-size of 10 for the first half of the graph, and with 100 for the second half.
As expected, the convergence factor is the best for the smallest jump height of 1-10. The notation 1-10 means, that the coefficient factor is 1 for the left half of the domain and 10 for the right half. For a coefficient jump size of 1-20, the convergence becomes worse, and even worse for 1-30. Interestingly, at 1-30, there seems to be some sort of boundary, from there on the convergence does not increase anymore, it even slightly decreases until to the jump from 1 to 300. So it appears that to make a difference if the jump is of height 1-30 or 1-300, the convergence factor stays at ≈ 0.8. This is not a desirable convergence at all, but the solver is at least converging. This is not the case for jump heights equal or greater 1-400, here the solver does not manage to converge most of the time. In addition, the error bars are significantly larger than of a coefficient jump ≤ 1-300, the training process seems to have a much harder time finding optimal prolongation and restriction operators for such large coefficient jumps.

It can be concluded, that to reach a moderate convergence factor < 0.5, the coefficient jump for configuration 1 has to be less or equal to 1-20. If it is sufficient for the solver to converge at all, the coefficient jump can be of size 1 to ≤ 300.

![Figure 19: Convergence factor over various jump sizes for configuration 1 with training-size 2000, learning rate 0.001 and 1 learning iteration, standard deviation as error bars.](image)

5.5.2 Maximize coefficient jump quantity

After discussing the maximal height of the jump, we can now focus on maximizing the jump frequency. How many coefficient jumps of jump size 1-10 can the deep Multigrid solver handle until it is diverging? The DMG is trained with the same numbers of jumps as it is tested. Again the test set is configuration 1 but with modified coefficient function. In Figure 20, the test results for various numbers of coefficient jumps are displayed, all equally distributed along the x-axis. So one coefficient jump would lead to a domain split in half as shown in Figure 11a but with coefficient 10 on the right side, and three jumps would lead to four areas in the domain as shown in Figure 11c. The coefficient function alternates from one to ten and vice versa for every coefficient jump, starting with one.

As expected, the best convergence in Figure 20 is observable for one coefficient jump. More surprising is the very bad convergence for two jumps, where the average convergence factor is above 0.8. It is not clear why the convergence for two jumps is that bad, while for three jumps the convergence is again nearly on the same level as for one jump, with a convergence < 0.3. The DMG seems to have problems to learn fitting prolongation and restriction operators here, but the reason is unknown.

For more than three coefficient jumps the convergence gets continuously worse, for four jumps the DMG would still converge, but with a bad convergence of ≈ 0.8. Numbers of jumps greater 15 do not really make sense, because the size of the fine grid in the x-direction is N = 31, whereby the coarser grid (with then N = 15) does not even have enough grid lines to represent more than 15 coefficient jumps. It is also noticeable, that there are similar abnormalities as with 2 jumps. The convergence for 6, 11 and 25 coefficient jump is even worse relative to the other high jump numbers.
5.6 Learned Multigrid with Galerkin coarsening

In this section, we evaluate our DMG, which is extended by the Galerkin coarsening. How the Galerkin coarsening is added to the deep Multigrid solver in Tensorflow was described previously in subsection 4.3. Now we can check if it is useful to train our model with Galerkin coarsening in comparison to the model without Galerkin implemented. In Table 14, the results of the deep Multigrid with Galerkin coarsening are compared to the results of the geometric Multigrid with Galerkin and the deep Multigrid without Galerkin. The DMG with Galerkin is trained with the same training parameters as the DMG without Galerkin, namely with 2000 training-steps, a learning rate of 0.001 and one learning iteration.

For configuration 1 it can be observed, that the training was not successful for the DMG+Galerkin. With a convergence of $0.830 \pm 0.004$, it performs worse than the GMG+Galerkin, and much worse than the DMG without Galerkin. The same holds for configuration 2. Here the convergence is on the same level as the GMG+Galerkin. Additionally, for configuration 3 and 6, the convergence for the DMG+Galerkin become worse than the untrained one, and the DMG without Galerkin is performing significantly better here. For configuration 4 the convergence is at least comparable to the DMG without Galerkin, but that is mostly because the problem has a smooth coefficient function and thus the GMG+Galerkin performs best.

Configuration 5 is an interesting case, here the DMG+Galerkin performs better than the GMG+Galerkin and also better than the DMG without Galerkin. As mentioned before, the DMG without Galerkin always has problems learning for configuration 5, which were caused by the right-hand side and boundaries not equal zero and one, respectively. The Galerkin DMG does not seem to have a problem with training for configuration 5 and outperforms the DMG without Galerkin with a good convergence factor of $0.309 \pm 0.014$.

Nevertheless, it can be said, that in general the deep Multigrid solver with Galerkin coarsening is harder to train, and the deep Multigrid solver without Galerkin should be preferred for learning optimal prolongation and restriction operators. The exception seem to be scenarios with right-hand sides and boundaries not equal zero and one, respectively, such as configuration 5. Here the Galerkin DMG outperforms the one without implemented Galerkin coarsening. Again, further research is needed to understand, why the DMG has problems to learn operators for configuration 5, and why these problems do not occur for the DMG with Galerkin coarsening.
/ | Trained DMG | GMG + Galerkin | Trained DMG + Galerkin
---|:---:|:---:|:---:
Config 1 | 0.246 ± 0.016 | 0.709 ± 0.001 | 0.830 ± 0.004
Config 2 | 0.775 ± 0.009 | 1.261 ± 0.001 | 1.242 ± 0.004
Config 3 | 0.285 ± 0.020 | 0.456 ± 0.001 | 0.870 ± 0.007
Config 4 | 0.297 ± 0.014 | 0.204 ± 0.005 | 0.242 ± 0.005
Config 5 | 0.621 ± 0.062 | 0.438 ± 0.013 | 0.309 ± 0.014
Config 6 | **0.273 ± 0.015** | 0.721 ± 0.001 | 0.837 ± 0.006

Table 14: Comparison of the trained deep Multigrid solver to the GMG with Galerkin and to the trained DMG with Galerkin for all configurations with 2000 training-steps, learning rate 0.001 and 1 learning iteration.

### 5.7 Scalability of the deep Multigrid

In this chapter, the scalability of the deep Multigrid solver is tested by evaluating the training times of the neural network.

#### 5.7.1 Training times

In Figure 21, the training times, as well as the graph creation times, for grid sizes from $15^2$ to $127^2$ are shown. The neural network is trained on configuration 1 with 2000 training steps. For all configurations, the training takes the same amount of time, because all configurations have the same number of layers. An exception is configuration 2, where two prolongation and restriction operators have to be learned, so the training process, as well as the graph creation, takes more time. So while Figure 21 presents the training and graph creation time of configuration 1, it holds for all configurations excluding configuration 2.

Graph creation is the process of constructing a computation graph in Tensorflow, where all the layers, such as the Jacobi relaxation layer or the prolongation or restriction layer, are merged together. This process is done in the first fitting step of the training.

In Figure 21, we can see that the graph creation process needs a big amount of the total training time. The red line here is the training time without the graph creation, while the blue line represents the time of the graph creation before training. As expected, both graphs show a quadratic curve, caused by the quadratic grid sizes.

The problem here is, that for the grid size $15^2$, the graph creation time is only $\approx 1/10$ of the training time, while for grid size $127^2$ the graph creation time heavily increases to $\approx 1/3$ of the training time. This trend is increasing for larger grid sizes. This occurs, because the graph creation seems to be very inefficient for large unrolled neural networks [33]. In our case, the large unrolled neural network is caused by the coarse grid smoother consisting of 300 Jacobi relaxation layers. In addition, the graph creation in Tensorflow is computed only on a single node, so it can not be accelerated by Multicore architecture. The training itself, on the other hand, is optimized for parallel computation, and therefore is very efficient for larger grid sizes compared to the graph creation. This is especially a problem for running the whole training process on graphic cards (GPUs), there the graph creation needs even more time due to the inefficiency of a single GPU core. The testing for Figure 21 is done on an Intel(R) Xeon(R) CPU E7-4830 (2.13 GHz – 2.4GHz) with 32 cores. This CPU is not capable of Advanced Vector Extensions (AVX), making the whole training process in Tensorflow a lot slower than for modern CPUs, taking nearly an hour to train the deep Multigrid with grid size $127^2$ for 2000 training steps.
Figure 21: Training times and Graph creation times for grid sizes $15^2$, $31^2$, $63^2$ and $127^2$ for configuration 1 with training set size of 2000.
6 Integration of the learned prolongation and restriction operators in Exastencils

ExaStencils is a code generation framework, which is build for generating high-performance stencil code. It utilizes a domain-specific language, called ExaSlang, which is stratified into four layers of abstraction. At every layer, ExaSlang expresses not only computational directives, but also domain knowledge of the problem and the platform, which is useful for code optimization. The most abstract layer specifies the problem as a set of partial differential equations. On the most concrete layer, low-level details can be specified for an efficient implementation. ExaStencils is focusing on stencil code for geometric Multigrid algorithms, what makes it suitable for the utilization it this thesis. [34]

The idea is to define a Multigrid solver in ExaStencils with the same configurations, as it is used for the training of the prolongation and restriction operators in Tensorflow. By this, the learned operators can be utilized to speed up computations in ExaStencils. Therefore, the operators – which were previously trained in Tensorflow – have to be written in separate files, one for the prolongation and one for the restriction. The operators are sorted in nine blocks, where one block contains all operators for one stencil direction, similar to Figure 17. The blocks are ordered in the directions bottom-left, bottom, bottom-right, left, mid, right, top-left, top, top-right.

Next, ExaStencils has to be configured through its domain-specific language ExaSlang. On the second layer, the initial solution is defined, as well as the right-hand side and the boundaries. Further, the Laplace stencil is defined and the solution equation is set. On the third layer, the details of the smoother as well as the coarse grid solver are implemented, as shown in Code Listing 6. There, the smoother is set to the Jacobi smoother with two pre- and postsmoothing steps and a damping factor of 0.6. In addition, the target residual reduction is defined and the coarse grid solver is set to 300 steps of the Jacobi smoother. The interesting part is at the bottom of Code Listing 6, where the standard prolongation(correction) and restriction function are replaced by new ones, which allow a separate weighting stencil for every grid point. The `InitVarRestriction` and the `InitVarCorrection` just read in the files with the trained restriction and prolongation operators, respectively.

```plaintext
generate solver for Solution in SolEq with {

    # set target residual reduction
    solver_targetResReduction = 1e-10

    # set Jacobi smoother with 2 pre-/postsmoothing steps and damping factor
    solver_smoother_jacobiType = true
    solver_smoother_numPre = 2
    solver_smoother_numPost = 2
    solver_smoother_damping = 0.6
    solver_smoother_coloring = "None"

    # set coarse grid solver to Jacobi smoother
    solver_cgs = "Smoother"
    solver_cgs_maxNumIts = 300

} modifiers{

    # set restriction and prolongation read in
    prepend to 'solver' {
        InitVarRestriction@((finest - 1)) ( )
        InitVarCorrection@((finest - 1)) ( )
    }

    # replace restriction and prolongation by custom ones
    replace 'restriction' {
        VarRestriction ( )
    }

    replace 'correction' {
        VarCorrection ( )
    }
}
```

Code Listing 6: ExaSlang code for third layer, with definitions of the smoother and coarse grid solver, and exchange of the prolongation/restriction functions.
In the fourth ExaSlang layer, all needed custom functions are implemented. For example, the coefficient function is set. But also the functions \textit{InitVarRestriction} and \textit{InitVarCorrection} for reading in the trained operators are defined there, as well as the custom restriction and prolongation functions \textit{VarRestriction} and \textit{VarCorrection} itself. In addition, one can modify there, which block of the trained operators corresponds to which stencil direction.

Lastly, the dimension as well as the discretization type and the number and size of the grid levels are set in another file, called the knowledge file.

After the configuration of the ExaStencils setup in the explained way, the trained operators of our deep Multigrid in Tensorflow can be successfully used to speed up ExaStencils computations. As expected, the numerical results are similar to the DMG, but because ExaStencils is a highly efficient framework, computations are significantly faster than in the slow Tensorflow framework. For instance, while the deep Multigrid solver needs $24.34 \pm 5.96$ seconds for the execution of configuration 1 without training, the generated ExaStencils code only needs $0.26 \pm 0.01$ seconds for the same configuration with the same number of iterations.
7 Comparison to related work

7.1 "Deep Multigrid: Learning Prolongation and Restriction Matrices"

The paper "Deep Multigrid: Learning Prolongation and Restriction Matrices" of Katrutsa et al. [4] follows a relatively similar approach to this thesis. The authors also reformulated their geometric Multigrid solver as a neural network, where every operation of a classic Multigrid solver is considered as a neural layer to learn optimized prolongation and restriction operators. In addition, they managed to include the Jacobi relaxation damping factor \( \omega \) in the training process. The paper only focuses on the two-grid method in one dimension, while in our work the deep Multigrid solver is also capable of more coarse grids, as shown in configuration 2, and of course two dimensions. In contrast to this thesis, the authors of the paper formulated a loss/objective function, which measures the spectral radius of one iteration of the deep Multigrid solver. More specific, they used a stochastic estimation of the spectral radius, which they derived from the Gelfand’s formula, because the computation of the spectral radius itself in every iteration would have been too costly. Furthermore, their deep Multigrid is written in Autograd instead of Tensorflow. The used optimizer for minimizing the loss function is the Adam optimizer, similar to our work. For the initialization of the prolongation and restriction operators before training, they used the homotopy approach instead of linear operators. Also, the authors of the paper are always using the Galerkin coarsening for their deep Multigrid as well as for their geometric Multigrid.

As testing experiments, they considered the Poisson equation as well as the Helmholtz equation, but no variable coefficient functions, whereby the results are hard to compare to ours. Nevertheless, they managed to improve the convergence (spectral radius) for the Poisson equation by their deep Multigrid solver compared to linear operators. Sadly the paper has no specific information about the configuration of the Poisson problem they solved besides the grid size, so the exact problem could not be tested with our deep Multigrid solver.

The successor of the presented paper is "Black-box learning of multigrid parameters" [5] by the same authors. Here they extended their deep Multigrid solver by the capability of using more than two grids and being able to solver problems in two dimensions, while the structure of the deep Multigrid solver is very similar to the preceding paper.

The authors used a similar approach as our deep Multigrid solver, but they did not test their solver for variable coefficients. It would be very interesting to see, if their solver is able to learn optimized prolongation/restriction weights for jumping coefficients.

7.2 "Learning to Optimize Multigrid PDE Solvers"

In the Paper "Learning to Optimize Multigrid PDE Solvers" by Greenfeld et al. [3], the authors are following a different approach for learning optimized prolongation/restriction operators for Multigrid optimization. They do not use a full Multigrid cycle implemented in a machine learning framework like [4] and we did, but they managed to learn a mapping from the discretization matrix to the prolongation matrix directly.

Therefore, they used a neural network consisting of 100 fully connected layers of width 100 with rectifier activation function. The input of the network is the 3 × 3 stencil of the fine grid point coinciding with the coarse point plus the stencil of its four nearest neighbours. The outputs are the four nearest neighbours for one column of the prolongation matrix. The middle point for the prolongation is always set to one, and the remaining diagonal neighbours are computed such that any grid function \( u \) obtained by prolongation from the coarse grid satisfies \( Au = 0 \) at these four grid points. This process is repeated for all fine grid points of the discretization matrix to achieve all columns of the prolongation matrix.

The network is trained multiple times with 163840 diffusion problems with sizes 16 × 16 and 32 × 32. The goal is to minimize the spectral radius of the error propagation matrix, but the spectral radius is again too costly to compute, so the Frobenius norm is used for approximation, bounding the spectral radius from above. For efficient training on large problems, a specialized Fourier analysis technique is used.

While the training is processed only on small grids with circulant blocks and periodic boundary conditions, the resulting prolongation operators are successfully tested for large grids (32 × 32 to 1024 × 1024), Dirichlet boundary conditions, non-periodic coefficient functions and even other do-
mains. The test results are compared to the well known Black Box Multigrid scheme by Dendy [35]. The learned prolongation operators outperform the Black Box Multigrid scheme for all problems.

To summarize the developed framework is very flexible in terms of grid size, boundary conditions and more, because it does not learn for a specific problem, but learns compact rules for constructing solvers for many different problems.

Again the results of the paper are hard to compare to ours. The authors also tested their learned prolongation operators for variable coefficients, but the coefficient function is sampled from a log-normal distribution. This results in a large amount of small coefficient jumps, while the coefficient functions tested in this thesis consist of a small number of larger coefficient jumps. Further, no particular information about the mean value or the standard deviation of the log-normal distribution is shown in the paper, so it is hard to reproduce and compare the results.
8 Future work

After presenting papers with a similar topic, some possible improvements for our Deep Multigrid can be discussed. It would be useful to utilize the spectral radius – or some approximation of it – as the loss/cost function of the training process, as it is used in [4] and [3].

It could also be interesting to learn other parts of the Deep Multigrid than the prolongation/restriction, because the cycle is already implemented in a machine learning framework, and the different layers only have to be marked as trainable to be able to get modified by the backpropagation process. For example, an optimized weighting factor for the damped Jacobi relaxation could be learned, as it is done in [4], or even some correction term for the Jacobi relaxation, as it is done in [6].

The initialization of prolongation and restriction operators before training could be changed from linear operators to a homotopy approach as is it also done in [4], but this would be more useful for other diffusion problems than the Poisson problem, for example for Helmholtz equation. Generally, it would be interesting to extend our Deep Multigrid for various diffusion problems.

It could also be interesting to train the neural network for more than one coefficient function, so it is able to produce prolongation and restriction operators working for multiple coefficient function. But this could result in operators, which are moderately working for all coefficient function, but not optimized for one specific one, ending up in similar convergences like linear operators.

Also, the prolongation and restriction stencil could be extended from $3 \times 3$ to $5 \times 5$ for example, taking more neighbouring grid points into account. This could increase the convergence factor, but also more weights have to be stored, and a problem would occur at the boundary points, because the stencil is going to access points outside of the domain.

Another idea is to learn for a W-cycle instead of a V-cycle, which could also increase the convergence for some problems, but will probably lead to higher training times.

For better convergence, another smoother could be implemented for the DMG. Especially for the coarse grid smoothing, 300 Jacobi steps – as it is implemented now – are not efficient. But smoothers like the Gauss-Seidel relaxation are not trivial to implement in Tensorflow. Further, larger grid sizes in addition to more coarse grid levels should be tested.

Lastly, it would be interesting to implement the deep Multigrid solver in another machine learning framework than Tensorflow, to be able to compare training times and efficiency. Also, maybe the problems we encountered with the graph creation would not occur in other frameworks like PyTorch or CNTK.
9 Conclusion

In this work, a deep Multigrid solver is presented. It is shown how to implement a fully functional V-Cycle in the machine learning framework Tensorflow, where every layer of a neural network corresponds to a Multigrid operation, such as a relaxation step, the residual equation, or the prolongation and restriction. This neural network is then capable of learning optimized prolongation and restriction operators for two-dimensional diffusion problems. In this work, mostly problems with jumping coefficients are chosen.

The convergence factor of the deep Multigrid solver is compared to a geometric Multigrid solver with and without the Galerkin coarsening, as well as to an algebraic Multigrid solver. The deep Multigrid solver converges better than the geometric Multigrid solvers for all problems with jumping coefficients, and most of the time better than the algebraic Multigrid solver. The learned prolongation and restriction operators are evaluated, and the influence on the convergence of different learning parameters like the learning rate or the size of the training set is explained. It is shown, that learned prolongation/restriction operators, which are trained for one problem, also work for other right-hand sides and boundaries, but diverging for other coefficient functions. Then, the limits of the deep Multigrid in terms of the jump height and the jump quantity are tested, as well as the scalability of the deep Multigrid solver in terms of grid size. The conclusion of this work is a comparison to the related papers [4] and [3] and a discussion of possible improvements for the presented deep Multigrid approach.
References


