




STOFFTRANSPORTMODELLIERUNG IM HYDROGEOLOGISCHEN KONTEXT AUF BASIS EINES KÜNSTLICHEN NEURONALEN NETZES

eine
Dokumentation
der Projektarbeit im Modul
Programmierung für Data Science
Fassung vom 28.02.2020



Alexander Prinz
Matr.-Nr.: 4069949
Studiengang: Data Science (M. Sc.)
Fachbereich: Informatik und Sprachen

Hochschule Anhalt

Anhalt University of Applied Sciences

Eigenständigkeitserklärung

Hiermit bestätige ich, Alexander Prinz, geboren am 26.02.1986, die folgende Arbeit selbstständig verfasst und nur die mir gesetzlich zustehenden Mittel zum Verfassen wissenschaftlicher Arbeiten genutzt zu haben. Arbeiten anderer Autoren, welche mir beim Verfassen dieser Arbeit dienten, habe ich gemäß den Prinzipien des wissenschaftlichen Arbeitens und dem Urheberrecht entsprechend kenntlich gemacht.

Eingereicht am:

10.02.2020 via Mail (in nichtaktueller Fassung)

28.02.2020 via GitLab (in dieser Fassung)



Danksagung

Ich möchte mich an dieser Stelle ganz herzlich bei Herrn DR.-ING. CHRISTOPH STEUP als Dozent des Moduls „Programmierung für Data Science“ für die konstruktiven Beiträge und Anregungen sowie Hilfestellungen bzgl. dieser Projektarbeit und der vielen gelernten Dingen durch Vorlesung und Praktikum bedanken. Ich konnte in dem Modul sowohl außerhalb des Projektes als aber auch durch das Projekt selbst vieles in Punkto Programmierung lernen. Zudem hat mich dieses Projekt in ein Gebiet blicken lassen, mit dem ich mich zukünftig noch intensiver beschäftigen möchte.

„SIENCE IS A DIFFERENTIAL EQUATION.
RELIGION IS A BOUNDARY
CONDITION.“

~ ALAN TURING ~

Zusammenfassung

Kern des Projektes war die Modellierung der hydraulischen Stoffausbreitung in gesättigten porösen Grundwasserleitern unter Berücksichtigung der Advektion und Diffusion auf Basis eines künstlichen neuronalen Netzes. Ziel war das Schätzen der Konzentration des sich ausbreitenden Stoffes zu einem festen Zeitpunkt an einem bestimmten Ort unter einem gegebenen Anfangswert der Konzentration sowie der über das Modellgebiet vorherrschenden Permeabilität.

Konzeptionell wurde das Projekt in zwei hauptsächliche Bestandteile gegliedert: Zum einen in den Teil der Trainingsdatengeneration auf Basis der numerischen Lösung der Stofftransportgleichung, da keine realen Daten vorlagen. Im zweiten Teil wurden die so gewonnenen Trainingsdaten verwendet und damit ein künstliches neuronales Netz in Form eines feedforward-Netzes trainiert und getestet.

Diese Dokumentation dient der Erläuterung der einzelnen Softwarebausteine und deren Funktion im Projekt. Zum besseren Verständnis wird dabei jedoch auch der physikalische und mathematische Zusammenhang erläutert.

Abbildungsverzeichnis

<i>Abbildung I:</i>	Schwellenwertelement	S. 6
<i>Abbildung II:</i>	Programmablaufdiagramm	S. 11
<i>Abbildung III:</i>	Schema der Matrixskalierung	S. 12
<i>Abbildung IV:</i>	Schema des Datenformats der zusammengefassten und vorprozessierten Trainingsdaten.....	S. 14
<i>Abbildung V:</i>	Die Topologie des künstlichen neuronalen Netzes	S. 15
<i>Abbildung VI:</i>	Programmabschnitt der Netztopologiedefinition	S. 15
<i>Abbildung VII:</i>	Einige exemplarische Ergebnisse des Optimierungs- prozesses anhand von drei verschiedenen Modellen	S. 17
<i>Abbildung VIII:</i>	Die verwendeten 500×500 Graustufenpixelgrafiken als Vorlage für semistochastische Erzeugung der Permeabilitätsfelder	S. 19
<i>Abbildung IX:</i>	Verschiedene semistochastisch erzeugte Permeabilitätsfelder visualisiert als Heatmap.....	S. 20

Formelzeichen, Definitionen und Abkürzungen

$a_{i,j}$	Matrizelement an Stelle i, j
$b_{k,l}$	Matrizelement an Stelle k, l
$c, c(\vec{x}, t)$	Konzentration als Funktion von Ort und Zeit
D	Diffusionskoeffizient – im Fall dieses Projektes konstant über Ort und Zeit
∂	Del Operator
$h(\vec{x})$	Hydraulische Höhe als Funktion des Ortes \vec{x}
$\nabla h(\vec{x})$	Hydraulisches Potenzial als Funktion des Ortes \vec{x}
i	Index i
j	Index j
k	Index k
$\kappa, \kappa(\vec{x})$	(effektive) Permeabilität des Grundwasserleiters am Ort \vec{x}
λ	Skalar Lambda zur Skalierung der Permeabilitäts- und Konzentrationsmatrix
Δ	Laplace Operator
\mathbb{N}	Menge der natürlichen Zahlen
\mathbb{N}_G	Menge der natürlichen geraden Zahlen
n	Index n
m	Index m
R	Restterm als Repräsentant weiterer Terme
\mathbb{R}	Menge der reellen Zahlen
$\langle \vec{a}, \vec{b} \rangle$	Skalarprodukt der beiden Vektoren \vec{a} und \vec{b}
$\sum_{i=1}^N e_i$	Sigma, Summe über alle N Elemente e
t	Index t, Zeitpunkt t
v_A	Abstandsgeschwindigkeit
∇	Nabla Operator
\vec{x}	Ortsvektor
$\{ \}$	Menge
$rezipr(a)$	Reziproke der Zahl a
$dim(T)$	Dimension der Matrix T

Inhalt

1. Motivation.....	1
2. Methode	1
3. Methodik	2
4. Physikalischer Hintergrund der Stofftransportgleichung.....	3
5. Das künstliche neuronale Netz – eine kurze Definition	4
6. Numerisches Lösen der Stofftransportgleichung	6
7. Implementierung der Stofftransportgleichung	7
8. Die Klassen und ihre Aufgaben	9
8.1 Klasse <i>Solver</i>	9
8.2 Klasse <i>PermPatternGen</i>	10
9. Programmablauf der Hauptroutine <i>Data_Generator.py</i>	11
10. Training des künstlichen neuronalen Netzes	12
10.1 Programm <i>RESIZER.py</i>	12
10.2 Programm <i>Data_Merger.py</i>	14
10.3 Programm <i>NN_CUDA.py</i>	15
11. Modellergebnisse	17
Literaturverzeichnis	19
Anhang.....	i
Anhang 1	i
Anhang 2	ii

1. Motivation

Die Fragestellungen bzgl. Transportmodellen im Grundwasserbereich sind sehr vielschichtig und decken nach RAUSCH ET AL. (2002) z. B. folgende Gebiete ab:

- Interpolation von punktuell gemessenen Stoffkonzentrationen,
- Bilanzierung von Stoffflüssen,
- Auswertung von Tracer-Experimenten,
- Prognose der Schadstoffausbreitung,
- Kontrolle von in situ Sanierungsmaßnahmen,
- Abschätzung des natürlichen Schadstoffabbaus,
- Einsicht in die Entwicklung der Grundwasserbeschaffenheit

Jedoch gibt es weitere Modellierungsgebiete im hydrogeologischen Kontext, wie z.B. die CO₂-Speicherung in tiefen Gesteinsformationen (vgl. KÜHN ET AL., 2011) und gekoppelte Prozesse zur Abschätzung/Analyse von Prozessen in bodenmechanischen Bereichen, wie z.B. in Arbeiten von KEMPKA ET AL. (2014).

Jene Forschungsgebiete haben aufgrund der derzeitigen Lage des Weltklimas einen signifikanten Stellenwert innerhalb umweltrelevanter Fragenstellungen hinsichtlich unseres immer noch drastisch hohen CO₂-Emissionsvolumens.

Da die Modellierung auf das numerische Lösen komplexer Differenzialgleichungen basiert, sind teils sehr intensive computergestützte Lösungsverfahren nötig. Diese kosten Zeit, Geld und benötigen zudem auch nicht selten aufgrund der hohen Rechenaufwände große Energiemengen. Künstliche neuronale Netze könnten künftig die Modellierungsaufwendungen reduzieren und somit einen positiven Beitrag zur Vereinfachung von Arbeitsmethoden liefern sowie den Energiebedarf für die Lösungsaufwendungen reduzieren.

2. Methode

Das Konzept dieses Projektes sieht vor, ein künstliches neuronales Netz zu trainieren, um mit diesem anschließend eine Grundwassermodellierung im Bereich des Stofftransportes in porösen Grundwasserleitern betreiben zu können. Das Ziel ist, aus der im Modellgebiet vorherrschenden Permeabilität sowie dem Anfangskonzentrationswert eines sich im Grundwasserleiter ausbreitenden Stoffes auf die Konzentration dieses Stoffes nach einer bestimmten Zeit an einem bestimmten Ort schließen zu können.

Künstliche neuronale Netze verlangen zum Trainieren möglichst viele, diverse und mit wenigen Fehlern behaftete Daten. Im Kontext könnten diese Daten etwa aus verschiedenartigen geophysikalischen Untergrundmessungen und deren Implementierung in Untergrundmodelle gewonnen wurden sein.

Da solche Realdaten jedoch nicht vorliegen, soll in diesem Projekt mit virtuellen Daten gearbeitet werden. Diese virtuellen Daten sind zum einen randomisiert erstellte Permeabilitätsfelder sowie die daraus berechneten Konzentrationsfelder zu einem festgelegten Zeitpunkt. Dieses Konzentrationsfeld wird durch die Lösung der vereinfachten Stofftransportgleichung durch einen Gleichungssystemlöser berechnet (siehe dazu Kapitel 6). Die so erzeugten Datentupel, bestehend aus dem Permeabilitätsfeld und dem korrespondierenden Konzentrationsfeld, werden dann als Trainingsdaten zum Trainieren des künstlichen neuronalen Netzes genutzt.

3. Methodik

Das Generieren der Trainingsdaten erfolgt durch die Lösung der Stofftransportgleichung in einer stark vereinfachten Form. Sie wird auf einem äquidistanten Gitter hinsichtlich Ort und Zeit numerisch über das explizite EULER-Verfahren gelöst (siehe dazu im Kapitel 6). Als Eingabeparameter dienen das Stoffkonzentrationsfeld zum Modellanfangszeitpunkt sowie ein Feld über die Permeabilität im Modellgebiet. Orts- und Zeitdiskretisierung sowie Permeabilitätsminimum- und Maximumswerte wurden zuvor mit dem Ziel der numerischen Stabilität angepasst. Es wird also nicht mit realitätsnahen Größen gerechnet. Sofern dennoch numerische Instabilitäten auftreten, werden diese beim Generieren der Trainingsdaten durch numerische Prinzipien, wie Maximumsnormprinzip, registriert und mögliche invalide Trainingsdaten verworfen. Weitere Einzelheiten werden in den jeweiligen Kapiteln genauer erläutert.

Weitere Parameter sind Integrationszeitraum und Zeitschrittlänge und gehen jeweils als zeitinvarianter fester Wert in das Modell ein. Das künstliche neuronale Netz kann also nur die Lösung zu diesem Zeitpunkt schätzen.

Der Diffusionskoeffizient wird ebenfalls als konstant über Ort und Zeit behandelt, da einwirkende Parameter, welche diesen verändern können, nicht in die Modellierung eingehen. Die Ortsdiskretisierung ist äquidistant über das gesamte Modellgebiet. Bei dem Ortsgitter handelt es sich genauer um ein 50×50 Gitter mit gleichgroßen Gitterzellen in beide Ortsrichtungen. Das Permeabilitätsfeld, welches also demzufolge auf das 50×50 Gitter abgebildet werden kann, wird semistochastisch erzeugt und basiert auf computergestützt manipulierten Graustufen-Pixeldateien (siehe dazu in Kapitel 8.3).

Das Programm, welches die Trainingsdaten erzeugt, kann in mehreren unabhängigen Prozessen aufgerufen werden, um so zeiteffizient auf einer passenden Hardware Trainingsdaten zu generieren. Um die so generierten Trainingsdaten zusammenzufassen, erzeugt ein Programm eine XML-artige Datenstruktur mit allen Trainingsdaten (siehe dazu Kapitel 10.2). Diese Datei wird dann von dem Programm, welches dem Training des künstlichen neuronalen Netzes dient, eingelesen. Dieses Programm splittet die Daten zu je 50 % in Trainingsdaten und Testdaten und trainiert das künstliche neuronale Netz.

Das Netz ist ein einfaches *feed-forward*-Netz, dessen Einzelheiten bzgl. Topologie und Parametrisierung in Kapitel 10.3 genauer erläutert wird. Zum Prüfen der Modellgüte gibt das Programm die anhand von Testdaten errechneten Modellfehler sowie Modellwerte als CSV-Dateien aus, welche extern durch Visualisierungstools zum Evaluieren der Modellgüte ausgewertet werden können. Dazu ein paar Beispielabbildungen in Kapitel 11.

4. Physikalischer Hintergrund der Stofftransportgleichung

In der noch relativ einfachen Form der Stofftransportmodellierung im Grundwasserbereich dieses Projektes genügt das Modell der Lösung folgender partiellen Differentialgleichung:

$$\frac{\partial c}{\partial t} = \nabla(D \cdot \nabla(c)) + (-\nabla(v_A \cdot c)) + R \quad (1)$$

Nach Substitution der Abstandsgeschwindigkeit v_A durch den hydraulischen Gradienten $\nabla(h)$ multipliziert mit der effektiven Permeabilität κ ergibt sich die so modifizierte Stofftransportgleichung:

$$\frac{\partial c}{\partial t} = \nabla(D \cdot \nabla(c)) + (-\nabla(\kappa \cdot \nabla(h) \cdot c)) + R \quad (2)$$

Diese Gleichung beschreibt die zeitliche Veränderung der Konzentration c eines wässrig gelösten Stoffes als Gleichnis zu ihrer örtlichen Veränderung in Abhängigkeit der Diffusion $[\nabla(D \cdot \nabla(c))]$, Advektion $[-\nabla(\kappa \cdot \nabla(h) \cdot c)]$ sowie verschiedener weiterer Prozesse, welche durch R in die Gleichung eingehen. Zur Vereinfachung dieses Projektes wurde der Term R jedoch nicht mit einbezogen. Es sei daher nur am Rande erwähnt, dass es sich hierbei um Prozesse, wie Massenzu- und abflüsse, hydrochemische Reaktionen (Fällungsreaktionen etwa durch PH-Wertveränderung über Ort und Zeit), biogene Prozesse (Metabolismus von Kleinstlebewesen) sowie mechanisch- oder elektrochemisch induzierte Sorbtionsprozesse handeln kann (vgl. HOLZBECHER, 1996; HÖLTING ET AL. 2013).

Das Projekt bzw. dessen Modellierung bezieht R also nicht mit ein. Es sollte jedoch nicht unerwähnt bleiben, dass dieser Term bzw. vielmehr seine einzelnen Komponenten in der komplexen Grundwassermodellierung hinsichtlich des Stofftransportes einen starken Anteil - wenn nicht sogar den stärksten Anteil - an einer realitätsnahen Modellierung hat.

Mathematisch lässt sich die Stofftransportgleichung den parabolischen Differentialgleichungen erster oder unter gewissen Konfigurationen hinsichtlich mancher Komponenten aus R der zweiten Ordnung zuordnen (vgl. RAUSCH ET AL., 2002).

5. Das künstliche neuronale Netz – eine kurze Definition

Die Grundsteine neuronaler Netze als Imitation physiologisch neuronaler Interaktionen in Tier und Mensch wurden bereits in den 1940er Jahren gelegt (vgl. McCulloch & Pitts, 1943). So reagiert ein Lebewesen auf einwirkende Reize mit Reizantworten dadurch, dass die Reize über Rezeptoren registriert und an weitere Nervenzellen weitergegeben werden, sofern die Reizschwelle klein genug ist. Am Ende der Reizübertragung über viele Nervenzellen steht die Reizreaktion, welche etwa das instinktive Zurückziehen einer Hand in der Nähe einer starken Wärmequelle sein kann. Auch das Hören eines Schallereignisses stellt eine Reizübertragung dar. Dabei sind tierische Lebewesen in der Lage, Schallinformationen zu filtern, um relevante Signale im Konglomerat verschiedener überlagerter Schallereignisse zu registrieren und darauf reagieren zu können.

Die Idee, ein Nervensystem künstlich zu imitieren, besteht nun darin, ein logisches Element, genannt Neuron bzw. Schwellenwertelement, als Analogon zur Nervenzelle zu etablieren, welches Eingangssignale annimmt, auswertet und ein Signal ausgibt.

Welches Signal ausgegeben wird, hängt sowohl vom Schwellenwert als auch von den Eingangssignalen und der Sensibilität des Neurons auf die Eingangssignale ab. Die Sensibilität wird durch die Gewichte der Eingänge sowie dem Schwellenwert repräsentiert.

Mathematisch ist das Prinzip eines einzelnen Neurons relativ einfach zu verstehen. In Abbildung I ist der schematische Aufbau eines Neurons bzw. Schwellenwertelements gezeigt. Ein Neuron besteht aus N Eingängen, über die jeweils ein Reizwert x_i an das Neuron geleitet wird. Dieser Reizwert wird mit einem korrespondierenden Gewicht w_i multipliziert. Im weiteren Verlauf werden diese Produkte aus Eingangswerten und Gewichten aufsummiert. Da sowohl die Eingangswerte als auch die Gewichte jeweils zu Vektoren zusammengefasst werden können, entspricht diese Summe dem Skalarprodukt beider Vektoren (siehe Gleichung 3). Der Wert des Skalarproduktes beider Vektoren wird nun durch eine Schwellwertfunktion ausgewertet. Im einfachsten Fall kann das die Heaviside-Funktion sein. Diese gibt den Wert 1 zurück, wenn das Skalarprodukt beider Vektoren größer oder gleich dem Schwellwert ist, oder den Wert 0 für den komplementären Fall (siehe Gleichung 4).

Ein solches einzelnes Neuron ist in seiner Funktion sehr beschränkt. Jedoch kann der Zusammenschluss vieler solcher Neuronen mächtige und komplexe Aufgaben lösen. Der Zusammenschluss bzw. die Verbindungen können auf unterschiedliche Weise passieren. Im *Deep Learning* werden Schwellwertelemente in Schichten zusammengefasst, welche *Hidden Layer* genannt werden. Solchen *Hidden Layer* folgen weitere *Hidden Layer*. Die Anzahl dieser *Hidden Layer* definiert die Tiefe eines solchen tiefen neuronalen Netzes. Jede dieser Schichten repräsentiert bestimmte zu analysierende Merkmale des Eingangssignals (vgl. WARTALA, 2018).

Die Art und Weise, mit der die einzelnen Neuronen miteinander verbunden sind (Topologie), ist ein weiterer Parameter in der Architektur eines künstlichen neuronalen Netzes. Bei *Convolutional Neural Networks* etwa sind die Neuronen nicht unbedingt mit den Neuronen der folgenden Schicht verbunden, sondern können quasi Schichten „überspringen“. Auf Basis solcher künstlichen neuronalen Netze können etwa Bilderkennungen realisiert werden (vgl. WARTALA, 2018).

Eine weitere Netzart stellen *Recurrent Neural Networks* dar, welche Signale zwischen den einzelnen Neuronen rückkoppeln können.

Damit ein künstliches neuronales Netz funktionieren kann, muss es trainiert werden. Die Charakteristik bzw. Funktionalität zeichnet sich, wie bereits erwähnt, durch die Gewichte und Schwellenwerte aus. Diese Gewichte und Schwellenwerte werden durch Lernfehler-Rückführung auf Basis von unterschiedlichen mathematischen Optimierungsverfahren trainiert, indem das Netz Ausgangsdaten aus Eingangsdaten erzeugen muss.

Die Ausgangsdaten (Modellwerte) werden mit dem wahren Wert der Trainingsdaten (Zielwerte) verglichen und ein Fehler zwischen Modellwert und Zielwert berechnet. Durch das Optimierungsverfahren werden auf Basis der Modellfehlerminimierung die Gewichte und Schwellenwerte innerhalb des Lernzyklusses pro Lernepoche angepasst.

$$\langle \vec{x}, \vec{w} \rangle = \sum_{i=1}^N x_i \cdot w_i \quad (3)$$

$$f(\langle \vec{x}, \vec{w} \rangle) = \begin{cases} 0, & \langle \vec{x}, \vec{w} \rangle < \theta \\ 1, & \text{sonst} \end{cases} \quad (4)$$

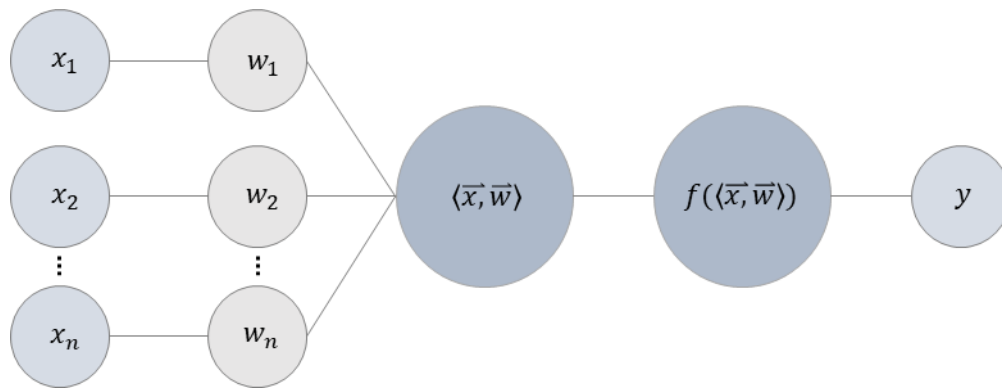


Abbildung 1: Schwellenwertelement. Die Abbildung zeigt ein Schwellenwertelement welches sinngemäß aus WARTALA R, 2018 entnommen wurde.

6. Numerisches Lösen der Stofftransportgleichung

Da eine analytische Lösung der Stofftransportgleichung auf realistische Modellparametrisierungen nicht möglich ist, muss sie durch geeignete numerische Verfahren gelöst werden. Dazu wird die Lösungsfunktion bzw. werden die Werte der Lösungsfunktion an diskreten Stellen approximiert. Das bedeutet, man bildet die Lösung hinsichtlich der Zeit sowie auch des Ortes auf endlich große Bereiche ab. Bezogen auf den Ort ist das als eine Art Gitter visuell gut vorzustellen. Wobei man nun die Lösung der Gleichung an der Stelle \vec{x} über die ganze Gitterzelle gemittelt auswertet und nicht an einem unendlich kleinen Punkt.

Um die Gleichung numerisch zu lösen, wird sie als lineares Gleichungssystem verstanden und dieses durch ein geeignetes Lösungsverfahren gelöst. Hierbei ist zu beachten, dass es sich dabei um ein Gleichungssystem handelt, welches aufgrund von unbekannten Fehlern keine exakten Lösungen bietet. Deshalb müssen solche Gleichungssysteme durch entsprechende Verfahren iterativ gelöst werden, etwa durch das JACOBI-Verfahren, GAUß-SEIDEL etc. (vgl. DAHMEN & REUSKEN, 2008). Diese Verfahren werden in untergeordnete Verfahren implementiert, um konkret solche Differentialgleichungen, wie die hier im Projekt behandelte Gleichung, numerisch zu lösen. Dazu existieren verschiedene Verfahren, welche sich hinsichtlich Rechenaufwand, numerische Stabilität und Genauigkeit unterscheiden, etwa dem RUNGE-KUTTA-Verfahren oder dem expliziten- sowie impliziten GAUß-Verfahren. Sie sind der Gruppe der Einschrittverfahren zuzuordnen (vgl. DAHMEN & REUSKEN, 2008).

Für die Lösung der Differentialgleichung in diesem Projekt wurde das explizite GAUß-Verfahren verwendet. Es zeichnet sich durch seinen geringen Rechen- und Implementierungsaufwand aus. Bei dem Verfahren wird vom vorhergehenden Zeitschritt und der dazu gehörenden örtlichen Lösung auf die örtliche Lösung des nächsten Zeitschrittes geschlossen (Einschrittverfahren zum Lösen von Anfangswertproblemen). Sofern also der Zustand zum Zeitpunkt $t = 0$ bekannt ist, lässt sich daraus der Zustand des Prozesses, der modelliert wird, zu jedem folgenden Zeitschritt approximieren. Ein wichtiger Faktor, der dabei zu beachten ist, ist die numerische Stabilität. So kann es durch zu große Zeitschritte und/oder zu große Ortsschritte im Verhältnis zur Modellparametrisierung zum Divergieren der Lösung kommen. Dies zeigt sich durch oszillierendes Verhalten der Lösung bzw. physikalisch unsinnige Ergebnisse. Eine wichtige Kenngröße hierzu ist das NEUMANN'sche Stabilitätskriterium, welches die Diskretisierung im Zusammenhang mit den Modellparametern untersucht. Weitere Kenngrößen zur Stabilitätsanalyse sind Maximumsprinzip und Massenerhaltungssatz (vgl. MALCHEREK, ANDREAS, Vorlesungsmaterial Hydromechanik; Universität der Bundeswehr München).

7. Implementierung der Stofftransportgleichung

Die Approximierung der Lösung der Differentialgleichung erfolgt über die Überführung der einzelnen Differentialquotienten in Differenzenquotienten. Im Folgenden werden die einzelnen Terme und ihre Größen in der Indexnotation dargestellt. Zu beachten ist dabei, dass das hochgestellte t über dem c somit kein Exponent, sondern der Zeitindex ist.

Für die partielle Ableitung der Konzentration nach der Zeit gilt folgende Approximation:

$$\frac{\partial c}{\partial t} \approx \frac{c_{i,j}^{t+1} - c_{i,j}^t}{\Delta t} \quad (5)$$

Zudem ist der Diffusionsterm aufgelöst ausgeschrieben:

$$\nabla(D \cdot \nabla(c)) = \nabla D \cdot \nabla(c) + D \cdot \Delta(c) \quad (6)$$

Da der Diffusionskoeffizient als konstant über den Ort betrachtet wird, ist $\nabla D \cdot \nabla(c) = 0$. Somit ist nur noch $D \cdot \Delta(c)$ zu betrachten. Wobei dann folgende Approximation gilt:

$$D \cdot \Delta(c) \approx \frac{D}{(\Delta x)^2} \cdot (c_{i+1,j}^t + c_{i-1,j}^t + c_{i,j+1}^t + c_{i,j-1}^t - 4c_{i,j}^t) \quad (7)$$

Der Advektionsterm $-\nabla(\kappa \cdot \nabla(h) \cdot c)$ löst sich auf zu:

$$\nabla(\kappa) \cdot \nabla(h) \cdot c - \kappa \cdot \Delta(h) \cdot c - \kappa \cdot \nabla(h) \cdot \nabla(c) \quad (8)$$

Wobei der Term $\kappa \cdot \Delta(h) \cdot c$ folgend approximiert wird:

$$\kappa \cdot \Delta(h) \cdot c \approx \frac{\kappa_{i,j} \cdot c_{i,j}^t}{4 \cdot (\Delta x)^2} \cdot (h_{i+1,j} + h_{i-1,j} + h_{i,j+1} + h_{i,j-1} - 4h_{i,j}) \quad (9)$$

Für $\nabla(\kappa) \cdot \nabla(h) \cdot c$ gilt:

$$\nabla(\kappa) \cdot \nabla(h) \cdot c \approx \frac{c_{i,j}^t}{4 \cdot (\Delta x)^2} \cdot (\kappa_{i+1,j} - \kappa_{i-1,j} + \kappa_{i,j+1} - \kappa_{i,j-1}) \cdot (h_{i+1,j} - h_{i-1,j} + h_{i,j+1} - h_{i,j-1}) \quad (10)$$

Für den letzten Term $\kappa \cdot \nabla(h) \cdot \nabla(c)$ gilt zudem noch:

$$\begin{aligned} & \kappa \cdot \nabla(h) \cdot \nabla(c) \\ & \approx \\ & \frac{\kappa_{i,j}}{4 \cdot (\Delta x)^2} \cdot (h_{i+1,j} - h_{i-1,j} + h_{i,j+1} - h_{i,j-1}) \cdot (c_{i+1,j}^t + c_{i-1,j}^t + c_{i,j+1}^t + c_{i,j-1}^t) \end{aligned} \quad (11)$$

Zunächst werden die einzelnen Terme folgend der Lesbarkeit halber durch Variablen ersetzt:

$$T_a := \frac{D}{(\Delta x)^2} \cdot (c_{i+1,j}^t + c_{i-1,j}^t + c_{i,j+1}^t + c_{i,j-1}^t - 4c_{i,j}^t) \quad (12)$$

$$T_b := \frac{\kappa_{i,j} \cdot c_{i,j}^t}{4 \cdot (\Delta x)^2} \cdot (h_{i+1,j} + h_{i-1,j} + h_{i,j+1} + h_{i,j-1} - 4h_{i,j}) \quad (13)$$

$$T_c := \frac{c_{i,j}^t}{4 \cdot (\Delta x)^2} \cdot (\kappa_{i+1,j} - \kappa_{i-1,j} + \kappa_{i,j+1} - \kappa_{i,j-1}) \cdot (h_{i+1,j} - h_{i-1,j} + h_{i,j+1} - h_{i,j-1}) \quad (14)$$

$$T_d := \frac{\kappa_{i,j}}{4 \cdot (\Delta x)^2} \cdot (h_{i+1,j} - h_{i-1,j} + h_{i,j+1} - h_{i,j-1}) \cdot (c_{i+1,j}^t + c_{i-1,j}^t + c_{i,j+1}^t + c_{i,j-1}^t) \quad (15)$$

Final wird die Gleichung so umgestellt, dass nach der gesuchten Größe $c_{i,j}^{t+1}$ aufgelöst wird. Es ist dann unter Berücksichtigung der Vorzeichen:

$$c_{i,j}^{t+1} = \Delta t \cdot (T_a - T_b - T_c - T_d) + c_{i,j}^t \quad (16)$$

8. Die Klassen und ihre Aufgaben

8.1 Klasse *Solver*

Computernumerisch wurde das Lösungsprinzip durch die Klasse *Solver* implementiert, welche in der PYTHON-Datei *LIB.py* bereitgestellt wird. Die Klasse wird in der Laufzeit der Hauptroutine, welche im Kapitel 9 erklärt wird, pro Iteration über alle zu erzeugenden Trainingsdatensätze initialisiert. Bei der Initialisierung bekommt sie dann das aktuelle Permeabilitätsfeld sowie optional ein Feld über eine mögliche Stoffquelle übergeben. Letztere ist aber nicht relevant und soll deshalb nicht weiter beschrieben werden.

Der hydraulische Gradient als weiteres Feld wird durch die externe Funktion *hydro_grad()* innerhalb der Klasse beim Initialisieren erzeugt. Die Funktion übernimmt dabei das Permeabilitätsfeld-Array, um daraus auf die erforderliche Dimensionierung des hydraulischen Gradientenfeldes zu schließen und gibt ein Gradientenfeld mit konstantem Anstieg zurück. Die Parametrisierung ist dabei hart kodiert in der Funktion definiert.

Der Diffusionskoeffizient ist ebenfalls konstant definiert und wird bei der Initialisierung eines Objektes durch einen festen Wert definiert, welcher sich als numerisch praktikabel hinsichtlich der

Stabilitätskriterien ergeben hat. Während des Lösungsprozesses wird über einen festgelegten Zeitraum von N Zeitschritten pro Zeitschritt die Methode `.set_init_cond()` der Klasse aufgerufen und ihr die Lösung des vorhergehenden Zeitschrittes in Form des Konzentrationsfeld-Arrays übergeben. Bei der Initialisierung der Zeititeration ist das das Konzentrationsfeld-Array, welches den Anfangswert repräsentiert. Sowohl Konzentrations- als auch Permeabilitätsfeld sind zweidimensionale NUMPY-Arrays.

Zudem wird der Methode auch der Ortschritt Δx übergeben. Um die Lösung, also das Konzentrationsfeld des gegenwärtigen Zeitschrittes der Iteration aus dem vorhergehenden Konzentrationsfeld sowie dem Permeabilitätsfeld zu erhalten, wird die Methode `.solve()` aufgerufen. Die Methode löst die Gleichung durch Iteration über das Ortsgitter und gibt die Lösung in Form des neuen Konzentrationsfeldes zurück. Zum Modellieren der Randbedingung (festgelegt als Nullgradientenbedingung) wird nach jeder abgeschlossenen Iteration über den Ort, der neu berechnete Randwert mit dem Wert des vorhergehenden Zeitschrittes überschrieben. Der Diffusionskoeffizient ist direkt in der Klasse als konstanter numerisch stabilisierender Wert definiert.

8.2 Klasse *PermPatternGen*

Die nötige Diversität der Trainingsdaten soll durch möglichst stark unterschiedliche Permeabilitätsfelder ermöglicht werden. In der Hauptroutine wird zur iterativen Erzeugung pro Trainingsdatentupel, bestehend aus einem Permeabilitätsfeld und dem korrespondierenden Konzentrationsfeld von der Klasse *PermPatternGen*, das Permeabilitätsfeld semistochastisch erzeugt. Die Klasse steht in der PYTHON-Datei *LIB.py* bereit. Pro zu erzeugendem Trainingsdatentupel wird ein Objekt dieser Klasse initialisiert, wobei der Pfad zu einem Verzeichnis übergeben wird. Dieses Verzeichnis enthält eine Anzahl sogenannter Eltern-Permeabilitätsmatritzen, welche durch 500×500 Pixelgraustufenbilder repräsentiert werden. Jedes dieser Pixelbilder zeichnet sich durch willkürliche Strukturen aus, welche eine realitätsnahe anisotrope Verteilung der Permeabilität abbilden sollen (siehe Anhang 1).

Die Methode `.load_im()` lädt aus diesem Verzeichnis per gleichverteiltem Zufallsprinzip eine der vorhandenen Pixeldateien und gibt diese Pixeldatei als NUMPY-Array zurück.

Über die Funktion `rand_sampling()` wird aus dem 500×500 Feld ein 50×50 Feld ausgeschnitten, wobei durch gleichverteilten Zufall eine Pixelkoordinate erzeugt wird, um die der Ausschnitt gewählt wird.

Die Funktion `rand_rot()` dreht das ihr übergebene Array mit gleicher Wahrscheinlichkeit um entweder 0° , 90° , 180° oder 270° und gibt es anschließend zurück. Um noch mehr Diversität zu erreichen, wird die Funktion `rand_smoother()` auf ein Array angewendet und über einen

GAUß'schen Filter geglättet, wobei mit gleicher Wahrscheinlichkeit eine Standardabweichung von entweder 1, 2 oder 3 verwendet wird.

Die letzte Funktion ist `normalizer()`, welche die Array-Elemente auf einen Maximalwert von 1 normalisiert. Dadurch werden Permeabilitätswerte zwischen 0 und 1 generiert, welche sich als numerisch stabil erwiesen und keine Oszillation bei der weiteren Parametrisierung induziert haben.

Da es sich bei allen Funktionen um lineare Abbildungen auf die stochastisch gewählte Eltern-Permeabilitätsmatrix handelt, können alle Funktionen direkt verschachtelt auf diese Matrix angewendet werden, was durch die Methode `.get_random_pattern_im()` passiert. In ihr sind die beschriebenen Funktionen definiert und werden durch die Methode zeitgleich aufgerufen. Im Anhang 2 sind exemplarisch einige so erzeugte Permeabilitätsfelder als Heatmaps zu sehen.

9. Programmablauf der Hauptroutine *Data_Generator.py*

Der Programmablauf zum Generieren der Trainingsdaten ist durch eine For-Schleife definiert, welche nach N Schritten terminiert. Pro Iterationsschritt wird durch die Klasse *PermPatternGen* aus einem Verzeichnis gleichverteilt eine Pixeldatei gewählt. Diese Pixeldatei wird durch weitere Funktionen/Methoden der Klasse zu einem 50×50 NUMPY-Array konvertiert, welches das Modellpermeabilitätsfeld repräsentiert. Dieses Array wird dann an ein Objekt der Klasse *Solver* übergeben und daraus der 3000ste Zeitschritt als Lösung der Stofftransportgleichung berechnet.

Jede zeitliche Lösung wird auf Validität geprüft. Sofern eine Lösung nicht stabil ist bzw. nicht konvergiert, wird die Lösung verworfen und neu begonnen. Wenn die Lösung valide ist, wird sie zusammen mit dem zugehörigen Permeabilitätsfeld jeweils in Form einer CSV-Datei gespeichert.

In der Abbildung II wird der Programmablauf dargestellt.

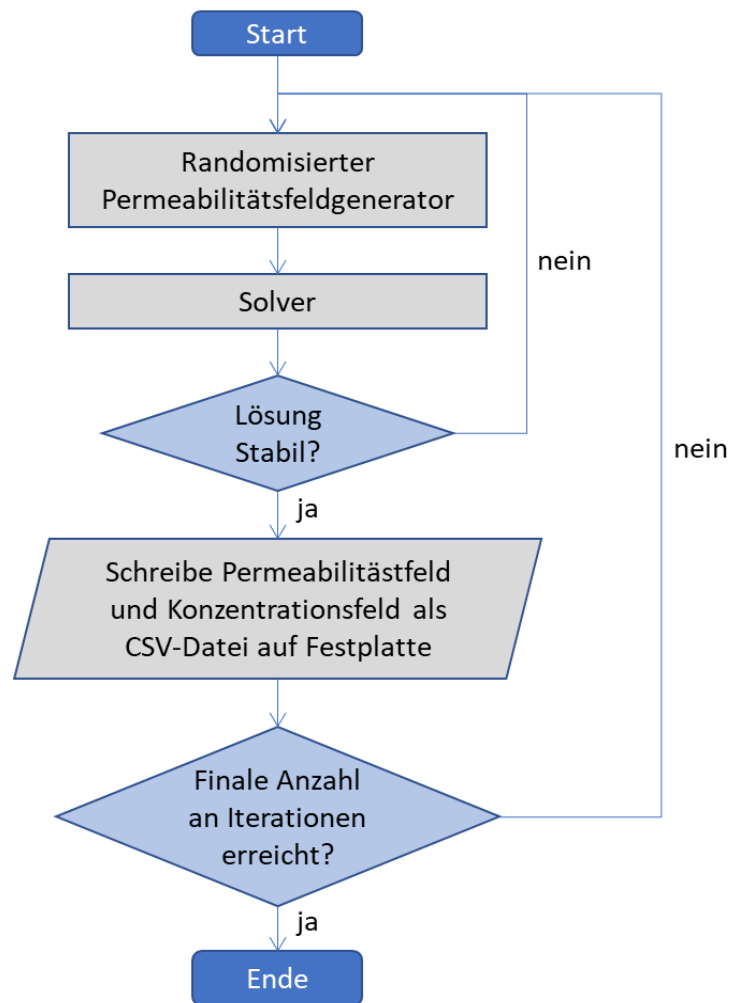


Abbildung II: Programmablaufdiagramm.
Beschreibt den iterativen Ablauf der Trainingsdatenerzeugung.

10. Training des künstlichen neuronalen Netzes

10.1 Programm *RESIZER.py*

Da die verwendeten Permeabilitätsfelder sowie Konzentrationsfelder als verwendete Trainingsdaten durch ihre Dimensionierung von 50×50 dem Training eines künstlichen neuronalen Netzes eine gewisse Zeit abverlangen, war eine Überlegung, die beiden Felder zu verkleinern, um erste Versuche mit zügigeren Ergebnissen durchzuführen. Nach einigem Suchen bzgl. passender Funktionen wurde schließlich eine eigene Lösung entwickelt und implementiert.

Sie basiert auf dem Verkleinern von $N \times N$ Matrizen mittels bilinearer Interpolation und wird folgend mathematisch definiert:

Sei die gegebene Skalierungsfunktion die Transformation einer Urbildmatrix auf eine Bildmatrix. Sei zudem λ jener Skalar, welcher die Skalierung parametrisch definiert. Dazu gelte für die Transformation die Abbildung $L: \mathbb{R}^{n \times n} \rightarrow \mathbb{R}^{m \times m}$, mit $m \neq n; n, m \in \mathbb{N}_G; m = \lambda n \Rightarrow \text{rezipr}(\lambda) \in \mathbb{N}$.

Dann ist die Transformation von der Urbildmatrix A auf die Bildmatrix B die sequenzielle Abbildung einer Teilmatrix T(A) von A mit der Dimensionierung $\dim(T) = \text{rezipr}(\lambda) \times \text{rezipr}(\lambda)$ auf den Wert $b_{k,l}$ der resultierenden Bildmatrix B.

Wobei die Vorschrift vereinfacht ausgedrückt der Mittelwert aller Elemente aus der Teilmatrix ist.

Die sequenzielle Abbildung selbst ist dann definiert durch die Vorschrift:

$$b_{k,l} = \lambda^2 \sum_{i=\frac{1}{\lambda}k}^{\frac{1}{\lambda}(k+1)-1} \sum_{j=\frac{1}{\lambda}l}^{\frac{1}{\lambda}(l+1)-1} a_{i,j} \quad (17)$$

Veranschaulicht für $\mathbb{R}^{4 \times 4} \rightarrow \mathbb{R}^{2 \times 2} \Rightarrow \lambda = 0,5$

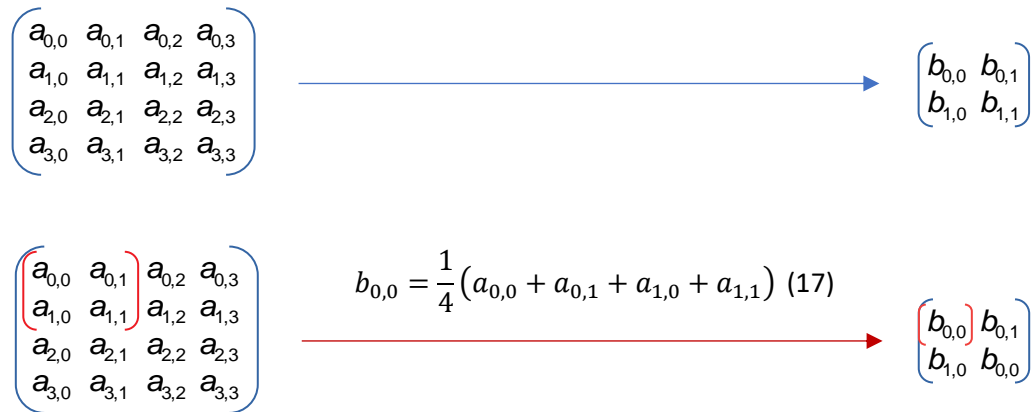


Abbildung III: Schema der Matrixskalierung

Für die Reskalierung der Permeabilitätsmatrizen sowie der korrespondierenden Lösungsmatrizen zur Vereinfachung des neuronalen Netzes genügt diese spezielle Vorschrift der Skalierung. Eine allgemeinere Lösungsvorschrift (für: $n \times m, m \neq n$) wird natürlich existieren, ist jedoch nicht nötig und wurde deshalb nicht entwickelt.

Das Programm funktioniert, indem es über den Inhalt des Verzeichnisses, das die Trainingsdatentupel enthält, iteriert und dabei jeweils sowohl die CSV-Datei des Permeabilitätsfeldes als auch die CSV-Datei des Konzentrationsfeldes lädt und nach dem beschriebenen mathematischen Prinzip verarbeitet. Dabei werden die Ursprungsdaten nicht überschrieben, sondern zusätzlich die skalierten Daten als weitere CSV-Dateien in das Ursprungsverzeichnis geschrieben.

10.2 Programm *Data_Merger.py*

Um möglichst schnell geeignete große Mengen an Trainingsdaten zu generieren, wurde das Programm *Data_Generator.py* parallel in mehreren Prozessen ausgeführt. So war es möglich, auf einem Rechner mit mehreren CPU-Kernen zeiteffizient Daten zu erzeugen. Diese Daten wurden in die den Prozessen zugeordneten eigenständigen Verzeichnisse geschrieben. Um diese Daten getrennt vom Trainingsprozess des künstlichen neuronalen Netzes zu einem ganzen Datensatz zusammenzuführen, wurde das Programm *Data_Merger.py* entwickelt.

Das Programm entnimmt dem Verzeichnis, in dem die Unterverzeichnisse der einzelnen Prozesse und deren Verzeichnisse, die die einzelnen Trainingsdatentupel verwalten, alle vorhandenen Trainingsdatentupel und extrahiert deren Inhalt und legt ihn in den Arbeitsspeicher ab. Anschließend erzeugt es eine XML-artige Datenstruktur aus den Trainingsdaten.

Da in dem Projekt nicht vorgesehen ist, dass das künstliche neuronale Netz das gesamte Konzentrationsfeld, sondern nur eine ausgewählte Gitterzelle berechnen soll, wird nur dieser im Programm festgelegte Gitterzellenwert des Konzentrationsfeldes in die Datei geschrieben. In der Abbildung IV wird das Prinzip der Datenstruktur gezeigt.

Im Bereich *meta* wird als erster Wert *input_length* die Länge jedes Permeabilitätsfeldvektors definiert. Des Weiteren definiert sich durch den Wert der Variable *batch_size* die Anzahl aller Trainingsdaten. Der letzte Wert *output_length* repräsentiert die Anzahl der Gitterzellen bzw. deren Werte hinsichtlich der Konzentration. Ziel war eine Parameterübergabe innerhalb eines automatisierten Programmablaufs beim Training des künstlichen neuronalen Netzes.

Der weitere Sektor *case_n* beinhaltet die beiden Subsektoren *permeability* und *sample_values*. *Case_n* definiert das n-te Trainingsdatentupel und die Felder *permeability* und *sample_value* beinhalten die entsprechenden Werte von Fall n.

```

001  <data>
002      <meta>
003          <input_length=2500, batch_size=4677, output_lenght=1>
004      </meta>
005      <case_n>
006          <permeability>
007              a1
008              a2
009              a3
010              a4
011              .
012              .
013              .
014              an
015          </permeability>
016          <sample_values>
017              b
018          </sample_values>
019      </case_n>
020 </data>

```

Abbildung IV: Schema des Datenformats der zusammengefassten und vorprozessierten Trainingsdaten. Die Datei enthält n Array mit dem Namen case_n in dem sich jeweils zwei weitere Arrays mit dem Namen permeability und sample_values befindet. In dem Array permeability werden die Permeabilitätswerte pro Fall gelistet. In dem Array sample_values werden die Werte der Konzentrationen für die ausgewählten Gitterzellen gespeichert.

10.3 Programm *NN_CUDA.py*

Das Programm dient dem Trainieren des künstlichen neuronalen Netzes. Der Suffix *CUDA* wurde dem Programmnamen später zusätzlich zugefügt, da mit dieser Version des Programms das Trainieren des künstlichen neuronalen Netzes auf einer *CUDA*-fähigen GPU möglich ist.

Das Programm lädt im ersten Schritt die Trainingsdaten, welche zuvor durch das Programm *Data_Merger.py* vorprozessiert und als XML-artige Datei bereitgestellt wurden. Um die Daten aus der Datei der gesamten Trainingsdaten zu extrahieren, wird die Klasse *Trainingsdata* benutzt, welche in der Datei *LIB.py* bereitgestellt ist. Bei der Initialisierung eines *Trainingsdata*-Objektes wird der Pfad zu der Datei der gesamten Trainingsdaten an die Klasse übergeben.

Die einzige Methode *get_tdata()* der Klasse extrahiert beim Aufruf dieser die einzelnen numerischen Daten aus der Datei, bereitet sie als *PYTORCH*-Tensor-Objekt für das Training vor und gibt sie zurück. Somit liegen zwei Tensoren *X* und *Y* vor, wobei *X* die Permeabilitätsfelddaten beinhaltet und *Y* die dazugehörigen Konzentrationsdaten. Die Daten werden dann in die eigentlichen Trainingsdaten *X_training* und *Y_training* sowie den Testdaten *X_valid*, *Y_valid* zu je 50 % Anteil gesplittet.

Mit der *to(device)*-Methode werden alle Daten in den Arbeitsspeicher der GPU oder auch CPU geladen. Wobei *device* ein *PyTorch*-Objekt ist, das eine Referenz zur benutzten Recheneinheit darstellt und Teil der *CUDA-API* ist. Mit *Number_Input_Neurons* wird die Anzahl der Eingangsneuronen definiert.

Mit *Number_Output_Neurons* wird die Anzahl der Ausgabeneuronen definiert.

Mit *Learning_Rate* wird die Lernrate definiert.

Mit *Number_Epochs* wird die Anzahl der Lernepochen definiert.

Mit *Model_Parameter_File* wird der Name der Ausgabedatei für die Gewichtematrix definiert.

Die Variable *model* repräsentiert das neuronale Netz als PYTORCH.*nn*-Objekt, wobei mit der Methode *.Sequential()* die Netztopologie definiert wird. *.Sequential()* erhält dazu pro Hiddenlayer ein Objekt das die Aktivierungsfunktionen aller Neuronen auf diesen Hiddenlayer abbildet. Das Objekt erhält als Parameter die Anzahl an Eingangsneuronen sowie die Anzahl der Ausgangsneuronen. In Abbildung IV ist die Netztopologie dargestellt.

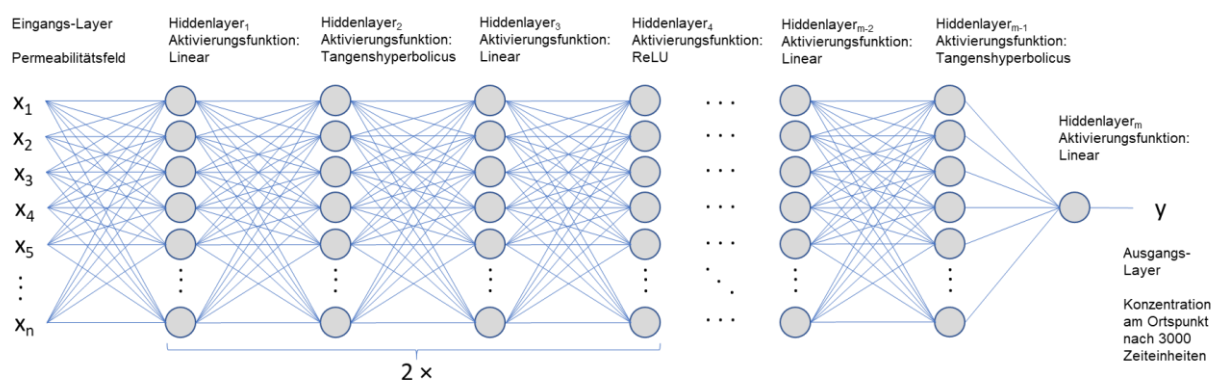


Abbildung V: Die Topologie des künstlichen neuronalen Netzes. Es liegt als ein Feed-Forward-Netz dar. Auf die Eingangsschicht mit $n = 2500$ Eingangsneuronen folgen 11 versteckte Schichten mit unterschiedlichen Aktivierungsfunktionen zu ihren Neuronen. Die Ausgangsschicht besteht aus nur einem Neuron welches der Konzentration am definierten Ort nach dem 3000-ten Zeitschritt entspricht.

In Abbildung VI wird der Programmteil gezeigt, in dem die Netztopologie definiert wird.

```
074 # definiere NN-Topologie
075 model = nn.Sequential(
076     nn.Linear(Number_Input_Neurons, Number_Input_Neurons),
077     nn.Tanh(),
078     nn.Linear(Number_Input_Neurons, Number_Input_Neurons),
079     nn.ReLU(),
080     nn.Linear(Number_Input_Neurons, Number_Input_Neurons),
081     nn.Tanh(),
082     nn.Linear(Number_Input_Neurons, Number_Input_Neurons),
083     nn.ReLU(),
084     nn.Linear(Number_Input_Neurons, Number_Input_Neurons),
085     nn.Tanh(),
086     nn.Linear(Number_Input_Neurons, 1)
087 ).to(device)
```

Abbildung VI: Programmabschnitt der Netztopologiedefinition. Alle versteckten Schichten haben die gleiche Anzahl an Neuronen welche der Anzahl der Eingangsneuronen entspricht. Lediglich die Ausgangsschicht besteht aus nur einem Neuron. Es wurden drei verschiedene Aktivierungsfunktionen (Linear, Tangenshyperbolicus und ReLU) verwendet. Auch bei diesem Datentyp wird über die *to(device)* festgelegt, dass die Daten im Arbeitsspeicher der GPU werden sollen und auf der GPU gerechnet werden soll.

Nach dem das Modell definiert wurde, wird im weiteren Programmablauf unter dem Objekt *criterion* die Verlustfunktion definiert, wobei die Minimierung der Summe der Fehlerquadrate als Verlustfunktion gewählt wurde. Zudem wird durch das Objekt *optimizer* die Modelloptimierungsmethode definiert.

Nachdem das Modell vollständig parametrisiert wurde, wird der Optimierungsprozess initialisiert und die Gewichte des künstlichen neuronalen Netzes anhand der Trainingsdaten iterativ über alle Lernepochen optimiert.

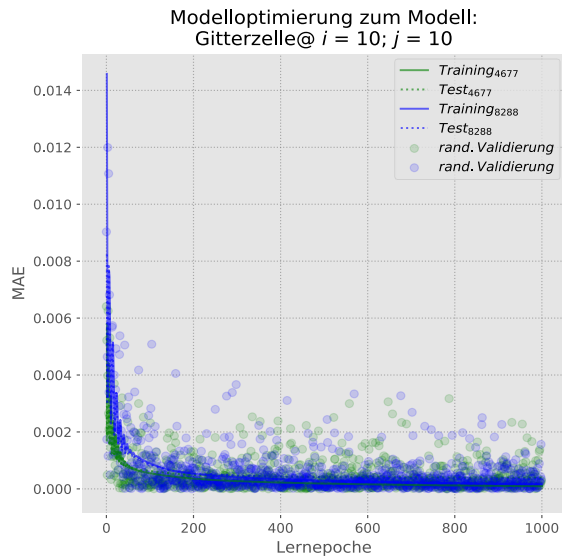
Zu Evaluierungszwecken werden die Modellergebnisse gegen die Testdaten verglichen und die Ergebnisse als CSV-Dateien ausgegeben (siehe dazu Kapitel 11).

11. Modellergebnisse

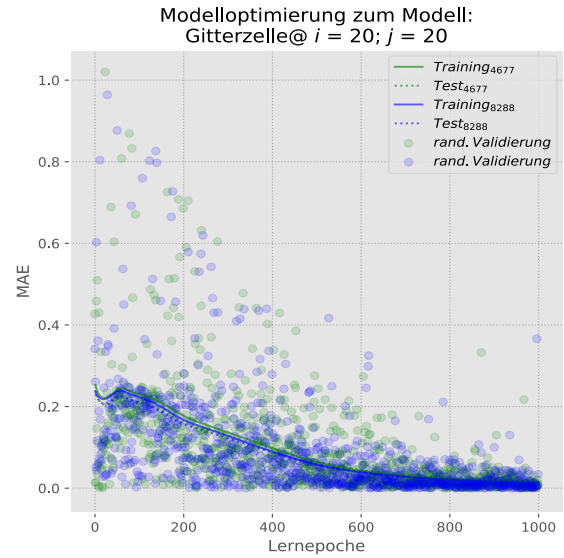
Die Modellergebnisse in Abbildung VII zeigen, dass die Erzeugung und Integration der Trainingsdaten in das Modell funktionieren. Der relative Modellfehler für alle 3 Modelle variiert zwischen 15 % und 400 %, wobei die 400 % dem Modell (Abbildung VII, c) zuzuordnen sind und die Modellfehlerkurve eine Verbesserung über weitere Trainingsepochen annehmen lässt. Die beiden ersten Modelle scheinen hingegen nur geringfügige Verbesserung des Modellfehlers durch weitere Trainingsepochen zuzulassen.

Zur Vereinfachung des Projektes und dem erstmaligen Umgang mit einem künstlichen neuronalen Netz wurde eine sehr einfache Netztopologie gewählt, welche nicht unbedingt optimal zu dem Modellierungszweck sein muss. Das heißt, andere Netzklassen und/oder -Topologien könnten die Modellierung noch erheblich verbessern.

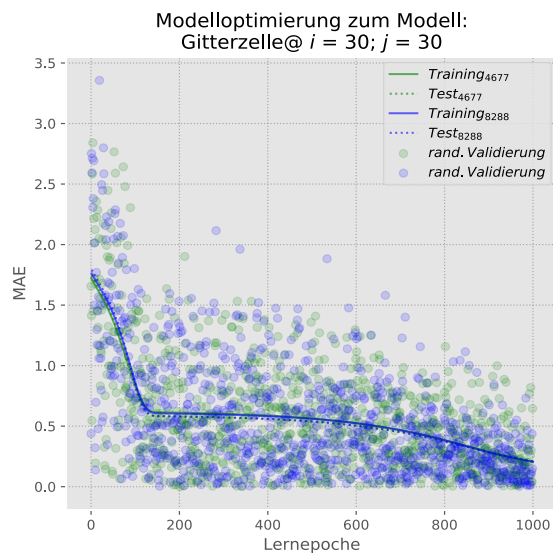
Die entwickelten Programme könnten zukünftig somit als Grundlage für tiefere Untersuchungen dieser Modellierungsmethode dienen, wobei dann auch andere Netz-Klassen untersucht werden sollten.



(a)



(b)



(c)

Abbildung VII: Einige exemplarische Ergebnisse des Optimierungsprozesses anhand von drei unterschiedlichen Modellen. Zu sehen ist das Modellfehlerverhalten über die Lernepochen während der Modelloptimierung. Der Modellfehler ist auf der Y-Achse aufgetragen und ist durch den arithmetischen Mittelwert der Summe der Absolutfehler über alle Trainingsdateien definiert. Die X-Achse repräsentiert die Lernepochen. Das künstliche neuronale Netz wurde mit 2 unterschiedlich großen Datensätzen trainiert, wobei der kleinere Datensatz eine Teilmenge des großen darstellt. Beide Modelltrainingsergebnisse werden durch die durchgehenden grünen bzw. blauen Grafen visualisiert. Es wurde parallel zum Training ein Modelltest mit Modellunabhängigen Testdaten vorgenommen. Der Modelltest wird für beide Datensätze in jeweils blau oder grünen gestrichelten Grafen dargestellt. Da sowohl Training als auch Test als mittlere Summe aller Modellfehler vorliegen wurde zur besseren Einschätzung über das Fehlerstreuverhalten pro Lernepoche eine randomisierte Modellvalidierung vorgenommen. Dazu wurde zufällig ein Trainingsdatentupel aus dem Datensatz gewählt und der Zielwert gegen den Modellwert auf Basis des Permeabilitätsfelds verglichen. Der absolute Fehler spricht der Betrag der Differenz zwischen Modellwert und Zielwert ist in Form der blauen bzw. grünen Punkte in jeder Grafik zu sehen. Die Modelle unterschieden sich zudem durch den Modellort. Abbildung (a) zeigt ein Modell zur Lösung der Konzentration der Gitterzelle $i = 10, j = 10$ des 50×50 Modells. Abbildung (b) zeigt ein Modell zur Lösung der Konzentration der Gitterzelle $i = 20, j = 20$. Abbildung (c) zeigt ein Modell zur Lösung der Konzentration der Gitterzelle $i = 30, j = 3$. Für alle Modelle sind unterschiedliche Modellfehlerverhalten zu sehen.

Literaturverzeichnis

- DAHMEN, W., REUSKEN, A. (2006), „*Numerik für Ingenieure und Naturwissenschaftler*“, Springer-Verlag Berlin Heidelberg, 2. Auflage, DOI 10.1007/978-3-540-76493-9
- HOLZBECHER, E., (1996), „*Modellierung dynamischer Prozesse in der Hydrologie – Grundwasser und ungesättigte Zone*“, Springer-Verlag Berlin Heidelberg 1996, DOI 10.1007/978-3-642-61073-8
- Hölting, B., Coldewey, W.G., (2013), „*Hydrogeologie – Einführung in die Allgemeine und Angewandte Hydrogeologie*“, Springer Spektrum, 8. Auflage, DOI 10.1007/978-3-8274-2354-2
- KEMPKA, T., DE LUCIA, M., KÜHN, M. (2014), „*Geomechanical integrity verification and mineral trapping quantification for the Ketzin CO2 storage pilot site by coupled numerical simulations*“, Elsevier Energy Procedia, Issue 1876-6102, DOI 10.1016/j.egypro.2014.11.36
- KÜHN, M., KEMPKA, T., LIEBSCHER, A., LÜTH, S., MARTENS, S., SCHMIDT-HATTENBERGER, C., (2011), „*Geologische CO2-Speicherung am Pilot-standort in Ketzin – sicher und verlässlich*“, System Erde 1, 2, DOI: 10.2312/GFZ.syserde.01.02.4
- MCCULLOCH, W.S., PITTS, W. A (1943). „logical calculus of the ideas immanent in nervous activity“. Bulletin of Mathematical Biophysics **5**, 115–133, DOI 10.1007/BF02478259
- RAUSCH, R., SCHÄFER, W., WAGNER, CH., (2002), „*Einführung in die Transportmodellierung im Grundwasser*“, Gebrüder Borntraeger Verlagsbuchhandlung, Berlin Stuttgart, ISBN 3-443-01048-2
- WARTALA, R., (2018), „*Praxiseinstieg Deep Learning*“, O'REILLY, 1. Auflage, ISBN 978-3-96009-054-0

Anhang

Anhang 1

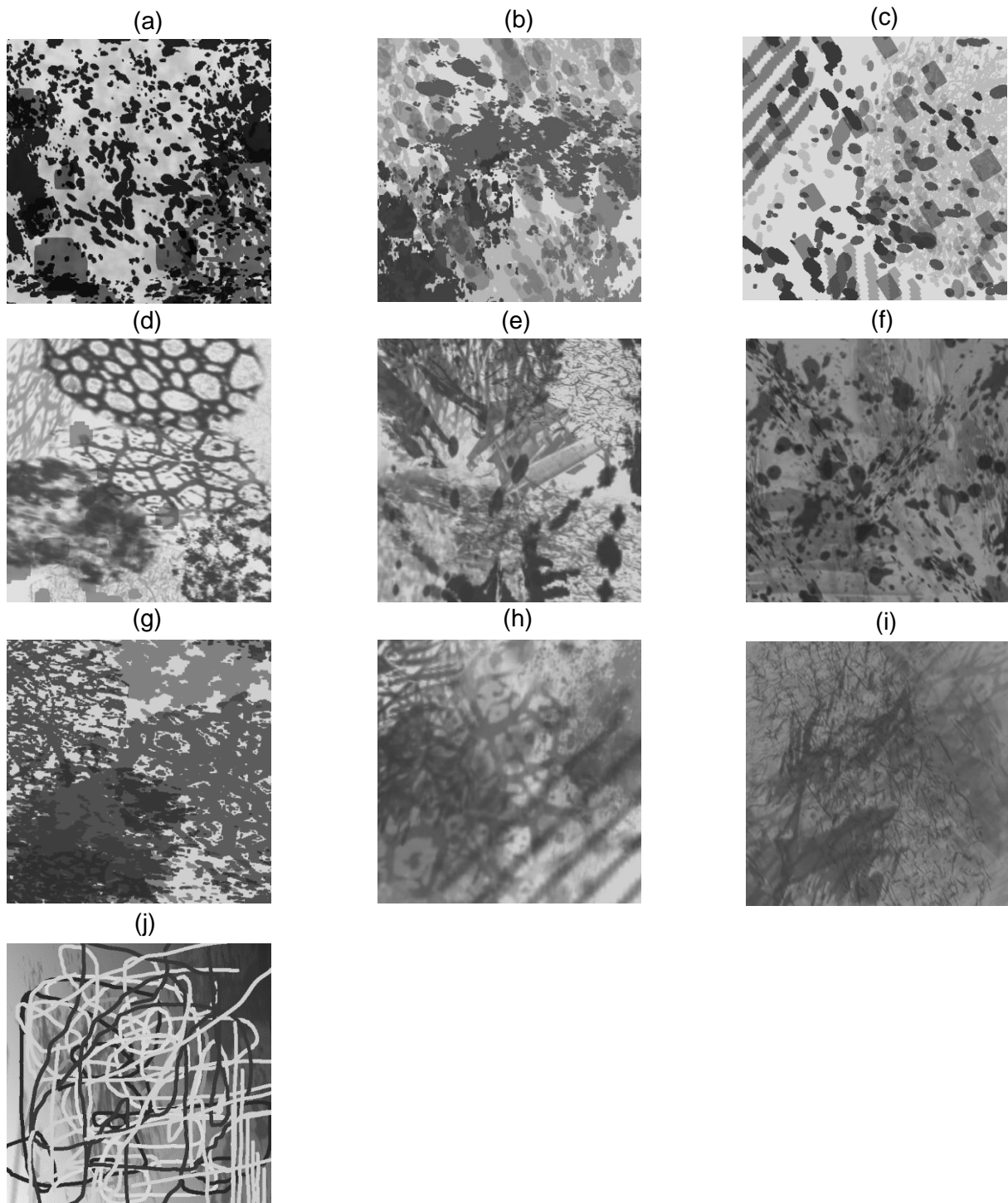


Abbildung VIII: Die verwendeten 500×500 Graustufenpixelgrafiken als Vorlage für semistochastische Erzeugung der Permeabilitätsfelder. Sie weisen alle sehr unterschiedliche Anisotropien bzgl. der Farbbreite sowie Farbkontrast auf. Dadurch lassen sich sehr unterschiedliche Permeabilitätsfelder erzeugen.

Anhang 2

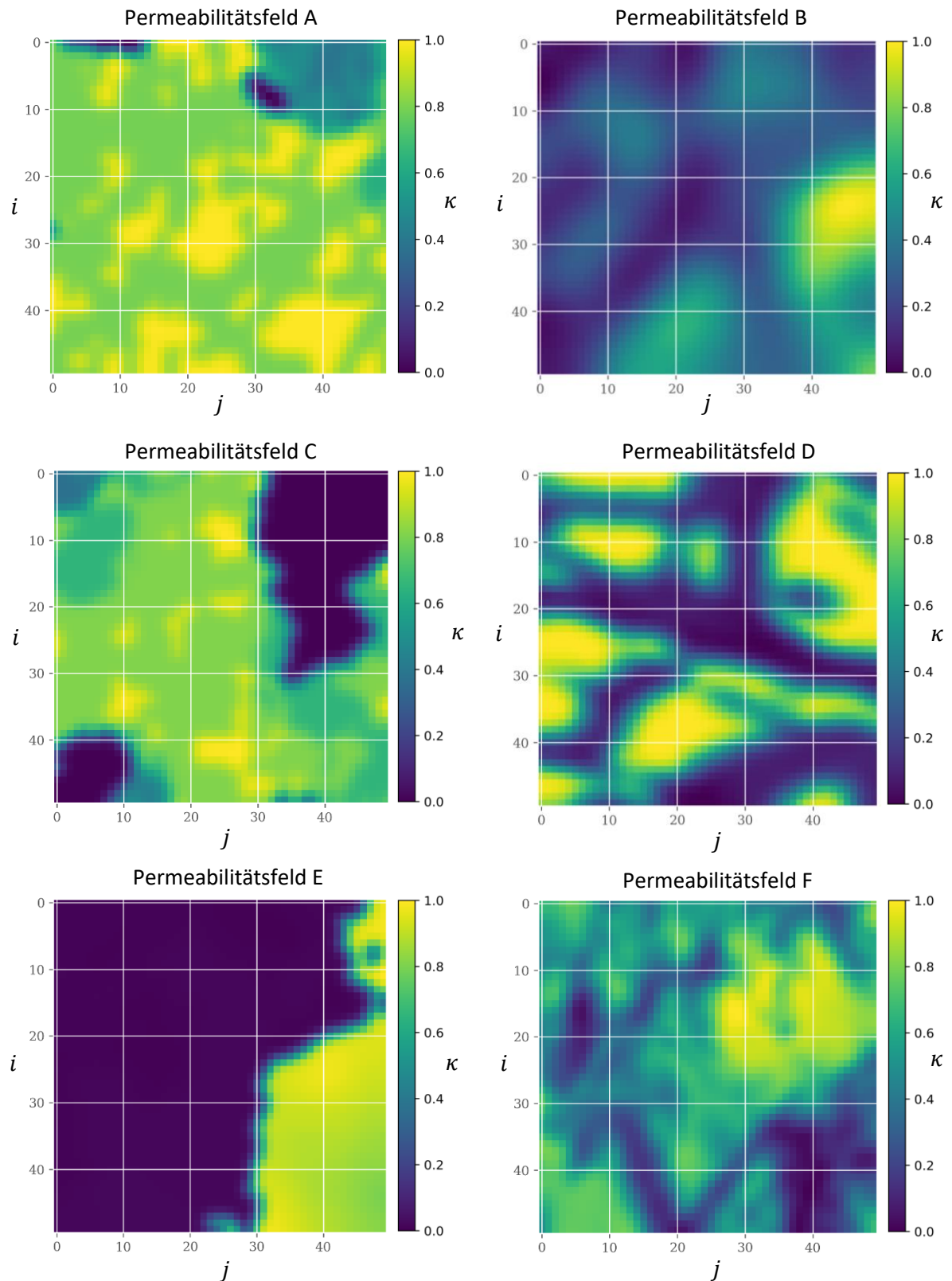


Abbildung IX: Verschiedene semistochastisch erzeugte Permeabilitätsfelder visualisiert als Heatmap. Es ist zu sehen, dass die randomisierte Permeabilitätsfelderzeugung auf Basis der 10 Eltern-Permeabilitätsmatrizen sehr unterschiedliche Permeabilitätsfelder erzeugt. Sie unterscheiden sich etwa durch den Kontrast bzw. Übergang zwischen verschiedenen permeablen Bereichen sowie auch über Spektrum der Permeabilität. Jeder Pixel der 50×50 Pixel-Grafik entspricht der Modellgitterzelle. i und j stellen die Indizes der Gitterzellen- bzw. Pixelkoordinaten dar. Zudem sind in einem Abstand von 10 Pixeln weiße Gitterlinien hinzugefügt wurden.